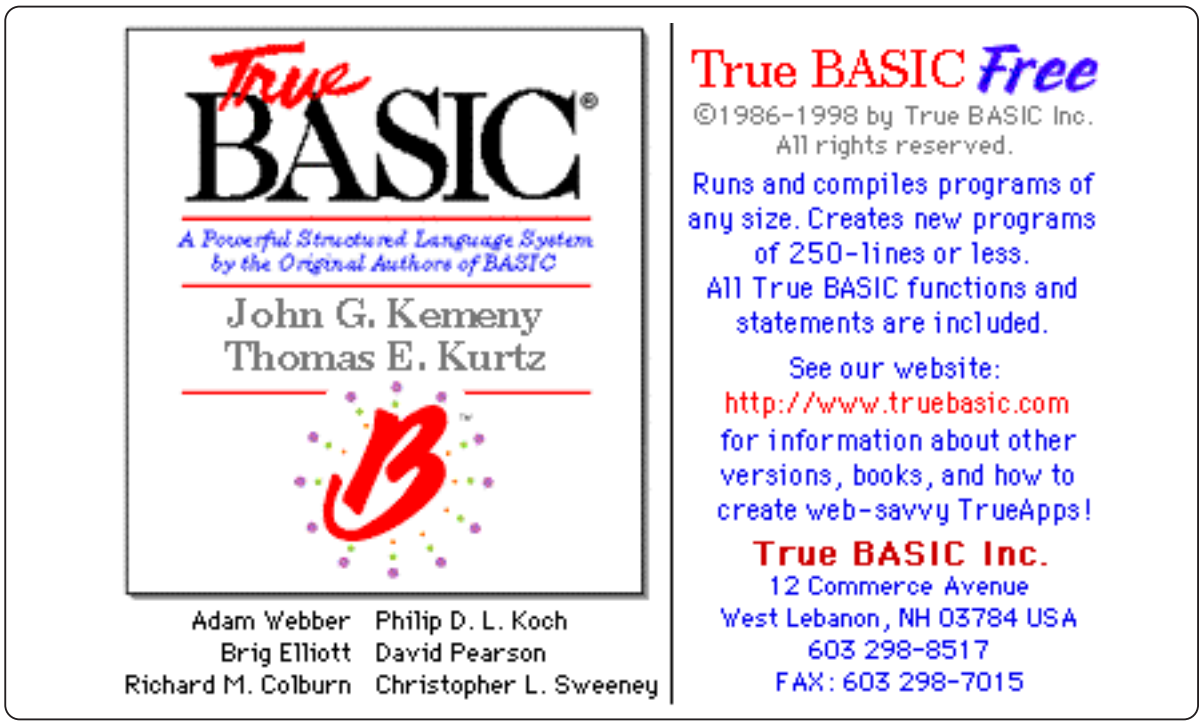


Introduction

Your **True BASIC *Free*** is identical to our powerful full-featured Language System and will run programs of *any* size. It only has two major limitations:

- New or modified programs are limited to 250 lines.
- The utilities used to create free-standing, double-click applications are not included.

Otherwise, your **True BASIC *Free*** is identical to our other Language System versions. You are able to run programs of any size, use libraries and modules, and invoke DO programs. All the True BASIC statements and functions are included for your use.

The current True BASIC Language System series consists of:

True BASIC *Free* - MacOS and DOS versions with line limit

True BASIC **Student** - MacOS and DOS with no line limits

True BASIC **Standard** - MacOS & DOS with developer tools

True BASIC **BRONZE** - MacOS, OS/2 and Windows versions

True BASIC **SILVER** - MacOS, OS/2 and Windows versions

The **www.truebasic.com** website lists features, specifications, and prices of each of these editions.

Many of the concepts and operations described in this guide will be new to you. To make it easier for you to understand, we use the following style conventions to make clear the many new concepts you will encounter:

Important new terms: **words in bold type**

Variable names: *words in italic*

True BASIC keywords: ALL CAPS

Program listings: `Code font`

Items to be typed by user: ***Code font***

Important concepts: ☑ **Bold type within lines**

Menus & menu commands: **MENU font**

Names of programs: ALL CAPS

Names of built-in functions: ALL CAPS

This guide frequently refers to the *True BASIC Bible*. This is an unabridged listing of all statements and functions found in True BASIC. It is available in two formats:

CD version: Item 75-001-*True BASIC Bible CD*

Book version: Item 70-BK31-*True BASIC Bible*

Both can be ordered from True BASIC Inc., via our website http:// www.truebasic.com or by phone at 603 298-8517.

Contents

1 • An Introduction to Programming

What is a computer program? What is a programming language? Why should you want to learn to write programs?

A **computer program** contains the instructions that tell the computer to do a certain task, such as play a game of football, format and print a letter, or predict the survival of lemmings over several generations. People who used the earliest computers had to know how to write their own programs. There were no stores down the block where they could buy a ready-to-use package that would track cash flow for their company.

Today, most people who use computers are not programmers. Instead, they use **application packages** such as word processors, spreadsheets, address organizers, or flight simulators. You can become a very sophisticated computer user and know nothing about writing programs.

Yet even if you have no intention of becoming a software developer or writing complex applications packages, you can still learn to program and enjoy solving your own problems in your own way. Why should people learn to program and why would you want to write your own programs?

There are several personal and practical reasons for learning to program:

• Acquire training and practice in logical thinking. Many business schools continued to teach programming to their students even after spreadsheets and database packages became widely available.

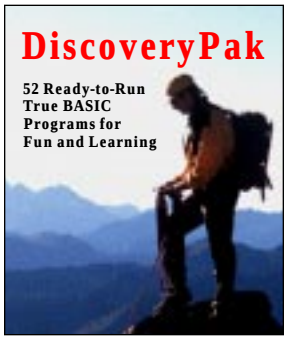
• Get a better understanding of how computers work. Everything a computer does boils down to programmed instructions.

• Create your own solutions to those little tasks that aren't easily handled by general-purpose applications. Calculate the results of a multi-race sailing regatta. Or combine judges' scores and distances for a ski jumping meet.

• Explore a new career field. Computer specialists have to start somewhere. And the computer industry needs "new blood" if we are to avoid becoming "hostage" again to those few who know how to program.

• Just have fun! Write a program to simulate a baseball game, or analyze a bridge hand, or solve a puzzle.

The **True BASIC *Free*** package introduces you to programming format and structures common to today's structured programming languages. The best way to learn is to sit down at a computer and do all the examples as you go through this book. This book does not cover all features in-depth, but it will give you a good start and hint at some of the additional power available with the True BASIC language.



As the next step in your programming adventure you might want to consider purchasing a copy of the True BASIC *DiscoveryPak* which contains more than fifty ready-to-run programs, covering subject areas from art to vocabulary.

The **DiscoveryPak** programs were designed to teach coding concepts as you use them in interesting programs. Creative changes are suggested for each program.

True BASIC DiscoveryPak, True BASIC, Inc., 406 pp. (ISBN 0-939553-06-6) Item: **25-DOS**&L or **25-MAC**&L

Other books available include:

Personal Math and Computing, Frank Wattenberg, MIT Press, 556 pp. (ISBN 0-262-23157-3) Item: 70-BK11

True BASIC by Problem Solving, Brian D. Hahn, VCH Publishers, 337 pp. (ISBN 3-527-26863-4) Item: 70-BK15

Programming in True BASIC: Problem Solving with Structure and Style, Stewart M. Venit & Sandra Schleiffers, West Publishing, 498 pp. (ISBN 0-314-78410-1) Item: 70-BK32

The above books are available directly from True BASIC, where all the listed titles are carried in stock. Our complete book list with descriptions and current prices is also found at www.truebasic.com/books.

2. Why True BASIC?

True BASIC is the ideal language for the beginning student and for the sophisticated programmer who wants to solve complex problems on several different computers. Two key phrases sum up the benefits of True BASIC over other languages: **powerful simplicity** and **portability**.

Simply Powerful!

• True BASIC is simple enough to let the beginning student write useful and interesting programs right from the start. True BASIC's screen editor makes it easy to read, write, and modify programs. New programmers can use the simpler features without knowing anything about the full complexity of the language.

• The same True BASIC language contains a full range of modern programming structures. The advanced programmer has access to such tools as graphics, sound, external libraries, modules, and full matrix algebra.

• You will never have to unlearn the logic and structures you learn in True BASIC. Because of its power, True BASIC may be the only language you ever need, but the skills learned here will also apply to object oriented or other modern languages.

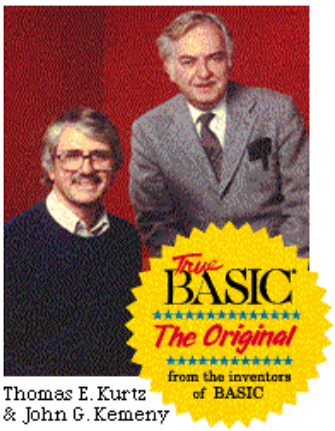
Portable and Compatible

• True BASIC programs run on any of today's major computer systems. (There are versions for IBM and compatible personal computers, Apple Macintosh, OS/2, Linux, and computers using the UNIX™ operating system.) You can easily move your programs from one computer to another. The only differences you'll see are in the editing features (and possibly in graphics and sound depending on the limitations of the computers.)

• True BASIC conforms to American and international programming standards. BASIC is the most widely used programming language in the world and is not limited by national boundaries. There is even a Japanese version of True BASIC that is designed to fully exploit the personal computers used in Japan.

As a **structured language**, True BASIC promotes good programming skills. True BASIC programs are easy to read. From the beginning, you'll learn modern looping and decision structures. You'll learn about using blank lines, comments, and indenting to make your programs easy to follow and modify later in the process.

You'll also learn how to use functions and subroutines to break your programs into small, manageable units. These units simplify your programming task. They let you concentrate on one problem at a time. They also let you create programs that are easy for humans to read and understand! (Users of other versions of BASIC may notice this book uses no line numbers or potentially confusing GOTO statements. True BASIC allows these holdovers from an older style of programming, but we do not recommend them.)

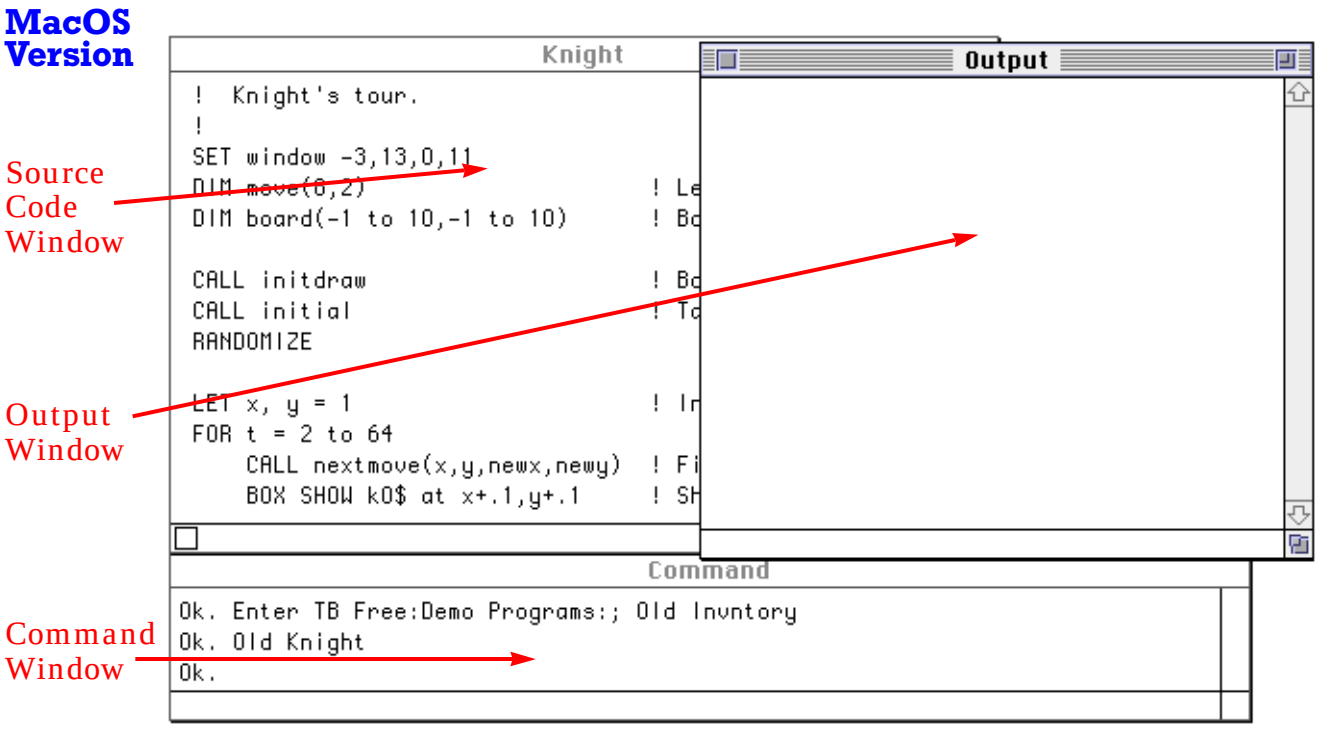


Thomas E. Kurtz
& John G. Kemeny

Dartmouth College Professors John G. Kemeny and Thomas E. Kurtz invented BASIC in the 1960s. The modern True BASIC language maintains their original philosophy. They designed a language that was easy for beginners, but provided power for advanced programmers. In the 1970s, graphics devices appeared and the concept of structured programming was widely accepted. At Dartmouth, BASIC continued to grow with these developments. Unfortunately, some of the earlier versions on the first personal computers were limited and did not benefit from new developments. Since 1985, True BASIC has provided an easy-to-use yet powerful, fully structured language for users of personal computers.

3. The Parts of True BASIC Free

MacOS Version



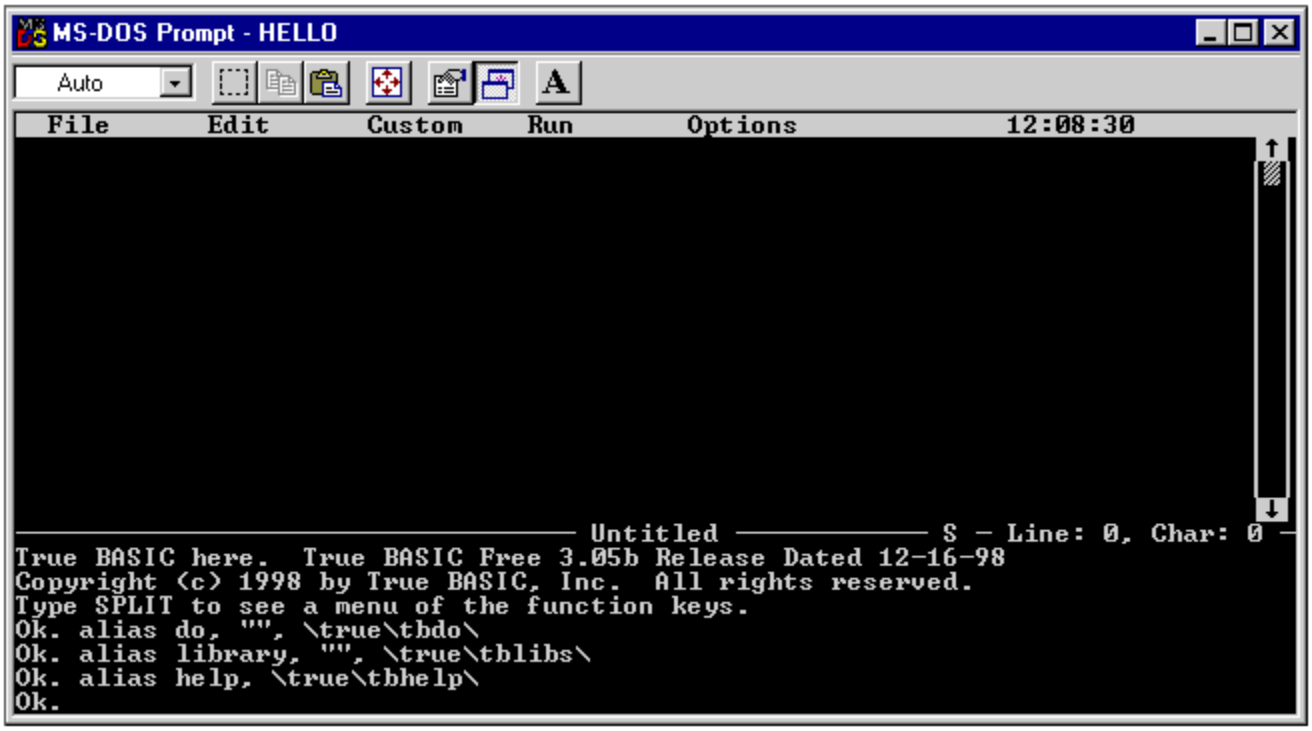
True BASIC *Free* gives you **three** working windows. If only one window is shown when you start True BASIC, you can open the other two windows by choosing them from the **Windows** menu.

Source Code Window: Where you write a program. It works much like a word processor. You can also **Open** previously written True BASIC program files and display the file contents in this window. You can edit previously written programs and overwrite the text by using the SAVE command, or keep the original text as it was, and create a new file of your modified code by using **SAVE AS...** in the **File** menu.

Output Window: This window display the results of a program, either as text or as graphics. After you write your source code you choose **RUN** from the **Run** menu and, if there are no errors in your code, the Output Window will open show the program.

Command Window: This window shows a running list of all the file operations you perform during a session and, if you like to use shortcuts, will allows you to enter commands that your would normally have to access through menus.

True BASIC *Free* only requires 1MB of memory for use. Also, we recommend that you create a TB folder that contains the items that come as part of this application as well as folders for the new program files you create.



PC Version

True BASIC *Free* for PCs is the DOS version of the True BASIC Language System. It operates much like the MacOS edition, but has a different appearance.

The main window contains both a **Source Code** section and a **Command Window** section. The top portion has a scroll bar to move up and down in your source code. On text that is too wide, click in the line and move to the right margin of the screen. The text will scroll horizontally. The **F2** function key will move you from the Source Code to Command Window.

Error messages and text output will appear in the Command Window. The depth of the Command Window can be modified by typing a command at the **OK>** prompt or by using the **Split at...** choice in the **Options** menu.

When you **Run** a program, the Output Window usually takes over the entire window. When the program is finished running the screen reverts its original view.

A listing of the PC Menus and Keyboard Shortcuts is shown on the next page.

The File Menu – Alt 1

Item	Keys	Function
New	Alt N	Create a new program file
Open	Alt O	Open a saved program file
Switch	Alt L	Switch to an open program file
Close	Alt W	Close an open program file
Save	Alt S	Save the current program file
Save As	Alt Z	Save the current program file under a different name
Unsave	Alt H	Delete a program file from the disk
Print	Alt P	List the current file to a printer
Quit	Alt Q	Exit True BASIC

The Edit Menu – Alt 2

Item	Keys	Function
Cut	Alt X	Remove the selected text to temporary storage
Copy	Alt C	Copy the selected text to temporary storage
Paste	Alt V	Insert the text from temporary storage at the current cursor position
Find	Alt F	Find a specific word or phrase
Find Again	Alt G	Find the next occurrence of the previously found word or phrase
Change	Alt E	Change a specific word or phrase to another
Keep	Alt K	Retain only a portion of the current program file
Include	none	Include a saved file at the current position of the text cursor
Edit	none	Edit a specific routine or block of text
Select	none	Mark a specific block of text
Move To	Alt Y	Move the text cursor to a specific line of text

The Custom Menu – Alt 3

Item	Keys	Function
Load	none	Load a library file into the workspace
Forget	none	Clear all non-essential memory
Script	none	Invoke a script file
Do	none	Invoke a preprocessor
Do Format	Alt D	Invoke the formatting preprocessor
Do Trace	none	Invoke the tracing preprocessor

The Run Menu – Alt 4

Item	Keys	Function
Run	Alt R	Compile and run the current program

Continue	none	Continue running the current program

4. Running Demo Programs

True BASIC *Free* comes with several demonstration programs. You don't need to understand how these programs work, but you can use them to learn how to run a True BASIC program.

To open an existing program, select **Open...** in the **File** menu (point to **Open...** in the menu bar and hold the mouse button as you drag down to **File**). In the dialog box that appears on your screen, select the Demo Programs folder by clicking on it with the mouse. Then click the Open button. You'll now see all the programs in the Demo Programs folder. Select the program GALTON by clicking on it and then clicking the Open button. (You can also click twice in rapid succession on the folder or program name to both select and open it.)

The File menu's Open... command tells True BASIC to get a document from the disk and make it the current program displayed in the editing window.

The program GALTON appears in the window. These are the instructions that tell the computer what to do. To have the computer carry out these instructions, choose **Run** from the **Run** menu. When the computer carries out programming instructions it is said to execute the program.

The Run menu's Run command tells True BASIC to execute the program (or carry out the instructions) currently shown in the editing window.

The GALTON program shows what happens when several balls drop into a box containing a series of pegs above several vertical chambers. Each ball begins in the center, but is randomly deflected by the pegs. The Galton box is often used in the study of probability.

When you run the program, the editing window disappears and the program results appear on the full screen. When the program run is complete, all the menu choices are gray or inactive.

To return to the editing window, click anywhere in the results. Run the program a few more times and you'll see randomly different distributions of the balls.

To stop the program while it is running, choose **Stop** in the **Run** menu. That returns you to the editing window with the message

"Program stopped" printed at the bottom. Some programs, such as the BOUNCE demo program, will run indefinitely until you use the **Stop** command.

The Run menu's Stop command stops a program run and returns to the editing window.

Quitting True BASIC

You've now learned how to start True BASIC and open and run existing programs. When you are finished using True BASIC, choose **Quit** in the **File** menu.

5. Writing and Running Your First Program

Start True BASIC, if you haven't already, as described in the preceding chapter. This time, instead of using an existing program, you'll create your own in the editing window. If you've just started **True BASIC *Free***, you'll have a blank editing window called "Untitled" because you haven't yet named your program. If you've been running an existing program, choose **New** in the **File** menu to get a blank "Untitled" window.

Creating a Program

Suppose you've driven 420 miles on 14.3 gallons of gas. To compute your gas mileage, you would divide 420 by 14.3. You can write a program to do this for you. Type the following into the Untitled window. Press the Return key at the end of each line.

```
LET miles = 420
LET gallons = 14.3
PRINT miles, gallons, miles/gallons
END
```

It doesn't matter whether you use uppercase or lowercase letters or more spaces than shown, but otherwise be sure you enter the program just as shown above. Don't forget that the digits one (1) and zero (0) and the letters "el" (l) and "oh" (O) are four distinct keys on a computer.

If you make a mistake while you are typing, you can use the Delete key to erase characters you have just typed (on older keyboards, this key may be labelled Backspace). Press Delete once to erase the preceding character; press it several times to

erase several characters. You can also use the arrow keys (if your keyboard has them) to move the cursor bar anywhere on the screen to make a correction. Move the I-beam cursor with the mouse and click at the point where you wish to make a correction. Or drag and highlight several characters that you may then delete.

Now let's see what the program does. Select **Run** in the **Run** menu. You should see the following "output":

```
420                14.3                29.3706
```

The result is a little more than 29 miles per gallon. (If you get different results or if the program doesn't run, check that you entered the numbers correctly in your program and that you spelled the words *miles* and *gallons* the same way throughout. LET, PRINT, and END must also be spelled correctly.)

Click anywhere on the screen (except the menu bar) to return to the editing window so you can look at how the program works.

Each line in the program is a **statement** in True BASIC. Like sentences in English, each statement contains an instruction that True BASIC can follow. Each statement begins with a **keyword**. Your program uses three types of statements: LET, PRINT, and END. You don't have to type keywords in uppercase, but we've done that throughout this manual to clearly distinguish them from the rest of the information in the statement. Keywords must end with a space unless there is nothing else on the same line.

The LET Statement

The keyword LET tells True BASIC to **assign** a value to something. LET statements are sometimes called **assignment statements**. The first line of the program assigns the value 420 to the word *miles*. When you again use *miles* in the PRINT statement, True BASIC knows to use the value 420.

In programs, values such as 420 are called **constants**, and a name such as *miles*, which could be assigned various values, is called a **variable**. More about constants and variables later.

The PRINT Statement

The PRINT statement shows the results of a program on your screen. Your program uses one PRINT statement to display three values: the values assigned to *miles* and *gallons*, and the value obtained by dividing the value of *miles* by the value of *gallons*.

You can use PRINT statements to print constants, variables, or **expressions** (formulas that combine constants and variables).

For example, the PRINT statement in your program could have been:

```
PRINT miles, 14.3, 420/gallons
```

and the results would have been exactly the same.

The END Statement

The last statement in your program is an END statement. It's the signal to True BASIC that there are no more instructions to carry out.

Every True BASIC program must finish with an END statement.

How True BASIC Runs a Program

When you ran your program, True BASIC carried out (executed) the statements one by one, from the first to the last — the same order in which you would read them. No statement was skipped or carried out more than once. This is called a straight-line **flow of control**. In later chapters, you'll learn about structures that create branches and loops in the flow of control.

Saving Your Program

To save your program, return to the editing window if necessary and select **Save** in the **File** menu. Since this is the first time you've saved this program, you'll be asked to enter a name. Call this program MPG and click in the Save button. You'll use MPG again in the next chapter where you'll learn how to make changes to an existing program.

In the previous section, you learned how to write a simple program and save it. Now, you'll make some modifications to that program and save those changes. In the process, you'll learn how to add comments to a program and how to have the program ask for information when it runs.

If it is not still in your editing window, open the MPG program you created and saved in the last chapter. You can use the **Open...** command in the **File** menu for any program that you saved, just as you did with GALTON.)

```
LET miles = 420
LET gallons = 14.3
PRINT miles, gallons, miles/gallons
END
```

Making Simple Changes

In the source window, a blinking vertical bar | indicates the **insertion point**. When you type something on the keyboard, that new text appears at the insertion point. If you want to change 420 to 420.6, you must first put the insertion point after the 0 in 420 and then type .6. You can move the insertion point with the mouse, the arrow keys, or the tab key.

The **mouse pointer** appears as an **I-beam** whenever it is in the "active" source window. Point and click with the I-beam at the desired point in the text and the insertion point moves to that spot.

The **arrow keys** move the insertion point a character or line at a time throughout the text. The **tab key** moves the insertion point word by word across a line.

There are two ways you can change existing text, such as replacing 14.3 with 15.7 in the second line:

• Move the insertion point to follow 14.3 and press the Delete (or Backspace) key four times. You may then type the new number.

• Highlight ("select") the number 14.3 by dragging across it with the mouse. Now when you begin to type, the highlighted text disappears and is replaced by what you type.

You can add new or blank lines by pressing the Return key at the beginning or end of an existing line. To remove a blank line, use the Delete key at the beginning of the line that follows it.

You can split or join lines in much the same way: split a line with the Return key at the split point; join two lines with the Delete key at the beginning of the second line.

Adding Comments to Your Program

Comments and blank lines have absolutely no effect on how your program runs, but they make programs much easier to read. From the very start, you should develop the habit of adding comments to your program.

In True BASIC, comments start with exclamation points (!). Everything from the exclamation point to the end of the line is part of the comment. You may put a comment on a line by itself or add one at the end of regular statement. Add some comments to your MPG program:

```
!  Compute miles per gallon
!
LET miles = 420         ! miles traveled
LET gallons = 15.7      ! gas used
PRINT miles, gallons, miles/gallons
END
```

To add the comments to an existing line, first move the insertion point to the end of the line and then use the space bar to move out to the right a bit before you type the comment. To align a comment with one above it, move to the end of the next line and use the Tab key — the insertion point jumps to the next word in the preceding line.

Saving Your Changes

You've now improved your MPG program by adding comments to it. The saved version doesn't have those changes, however, until you again save the program. To do that choose **Save** in the **File** menu. True BASIC replaces the old copy of MPG with a copy as it now appears in your source window.

If you've saved a program once and named it, the **Save** command doesn't ask for a file name for subsequent saves. It assumes you want to use the same name and **replace** the existing version. If you wanted to keep the old copy and save the new, edited one with a different name, you should use the **Save As...** command. First, let's make some more changes to the program.

The INPUT Statement – Getting Information From the User

The way the MPG program is written, you have to edit it in the source window whenever you want to compute miles per gallon for different numbers of miles or gallons. A program like this is more useful if you can enter values when the program runs.

Instead of LET statements, you can use INPUT statements to assign values while the program is running. Replace the LET statement lines in your program with INPUT statements as shown in the program below. To quickly replace one or more entire lines, point just to the left of the first line and drag down one to select. Next, press the Delete key once to erase the selection. You can then type new lines.

```
!  Compute miles per gallon
!
INPUT miles
INPUT gallons
PRINT miles, gallons, miles/gallons
END
```

When you're satisfied you've typed the changes correctly, run the program to see how the INPUT statement works.

When the program starts, it prints a "?", which is a signal that it is waiting for you to enter a number of miles. Type the number 100 and press the Return key. The program then prints another question mark, now looking for the number of gallons. Type the number 4 followed by the Return key. Next, the program prints the results and stops. Your output window should look like this:

```
? 100
? 4
 100              4                25
```

Whenever it sees an INPUT statement, True BASIC prints a question mark and waits for you to enter a response. Whatever you enter is assigned to the variable in the INPUT statement. True BASIC knows that you are finished entering your number when you press the Return key.

How will someone running your program know what they are supposed to enter when they see a question mark? The simplest way to fix this problem is to use PRINT statements with text for the program to print:

```
!  Compute miles per gallon
!
PRINT "How many miles";
INPUT miles
PRINT "How many gallons";
INPUT gallons
PRINT miles, gallons, miles/gallons
END
```

Notice that the text to be printed is in quotation marks. This is necessary so that True BASIC won't think the words are variables such as *miles* and *gallons*.

Later sections explain the semicolon (;) at the end of the PRINT statement — the semicolon is not necessary here, but it makes the question mark appear on the same line as the text, and close to it.

The *Formatting and Printing Your Program* section explains how you can PRINT to a printer.

Add the PRINT statements shown above to your program and run it again. You should see the following output:

```
How many miles? 100
How many gallons? 4
 100              4                25
```

Saving Your Program With a Different Name

You've now made additional changes to the MPG program since you last saved it. What if you want to save these additions but you also want to keep the version as it was when you last saved

it? In other words, you want two versions of the program — one with the data supplied by LET statements and one that requests the information with INPUT statements.

To save a copy of a program under a new name, use **Save As...** in the **File** menu. You'll get a dialog box with a space for entering a new name before you save. Save this version of your program with a name such as MPG2. The MPG program as you last saved it is not changed or replaced.

Opening or Quitting without Saving

If you have edited a program and then attempt to **Quit** True BASIC without saving the program, True BASIC asks if you want to save the file. You have three possible responses:

click **Yes** to save the program (replacing a version with the same name) and quit True BASIC

click **No** to quit True BASIC without saving your current program

click **Cancel** to get back to the program, where you could then use **Save As...** if you wish to save under a new name

True BASIC asks the same question if you use **New** or **Open...** in the **File** menu without first saving changes to your current program. True BASIC closes your existing program before opening or creating a new one; any changes to the existing program are lost if you do not save it first.

7. Constants, Variables and Expressions

True BASIC lets you work with two kinds of information — numbers and strings. By definition, strings are any combination of characters. Examples of string data include names, addresses, or phone numbers. Let's look first at numbers in True BASIC programs.

When you use numbers in a True BASIC program, they may be constants, variables, or expressions (expression is just another name for formula). Look again at the simple MPG program that you created earlier:

```
!  Compute miles per gallon
!
LET miles = 420          ! miles traveled
LET gallons = 15.7       ! gas used
PRINT miles, gallons, miles/gallons
END
```

Constants

The MPG program contains two numbers: 420 and 15.7. These are called **constants** or **numeric constants**.

Constants are quantities whose values can't change during a program run.

You can write constants as whole numbers, such as 420, or as decimals such as 15.7 Note, however, that you can't include any spaces or commas in numbers in True BASIC. Thus 10,000 must be written as 10000. The following table shows some rules for writing numeric constants:

Number Constants

Acceptable	Not Acceptable
6	VI
1002	1,002
321.33	1.2.3
0.003	1 000 000
.25	

Variables

In the MPG program, the **variables** are *miles* and *gallons*.

Variables are names for quantities whose values may change during the run of a program.

You could think of a variable as a box that can contain a value. A variable name (such as *miles* or *gallons*) identifies a box and that name remains the same throughout the program, but the value put into that box — assigned to that variable — can change each time the program runs or even during a program run.

The LET statement assigns a value to a variable. After the first line in the MPG program, the variable *miles* contains the value 420. The value of *miles* remains the same in this particular program, but you'll see later how values of variables can change within a program.

You can pick any names you want for variables in True BASIC as long as you follow certain "spelling" rules explained below. Although the computer doesn't care what names you use, it's usually a good idea to pick a name that somehow conveys what the variable means. For example, *miles* is a better choice than the letter *m* to represent miles traveled.

Variable names can be up to 31 characters long. You may use either capital or small letters, or any combination. True BASIC ignores the difference. The main rule is:

Variables names must begin with a letter, but subsequent characters can be letters, digits, or the underscore (_) character.

The underscore is the only punctuation mark allowed in variable names. You can't use spaces or hyphens because these mean something special to True BASIC. (A hyphen is the same as a minus sign.)

Variable Names

Acceptable	Not Acceptable
`miles`	`# of miles`
`miles_per_gallon`	`miles.per.gallon`
`profits`	`13`
`tax1040`	`1world`
`time_of_day`	`time-of-day`

Expressions and Formulas

Since computer keyboards don't have all the arithmetic symbols (or operators) on them, True BASIC has made a few substitutions. The symbols or **arithmetic operators** that True BASIC uses are:

Symbol	Meaning	Example
+	addition	a + b
–	subtraction	3 - 2
*	multiplication	length*width
/	division	miles/gallons
^	exponentiation (x^2)	x^2

You can use constants and variables to do arithmetic calculations. When you combine constants or variables using arithmetic symbols, you are writing an **expression**, which is just another name for a **formula**.

For example:

```
miles/gallons
```

is an expression that divides the value of *miles* by the value *gallons*.

True BASIC does not notice spaces in expressions. For example, "a+b" means the same thing as "a + b", and "miles/gallons" is equivalent to "miles / gallons". (Remember, however, that variable names cannot contain spaces.) You can also use parentheses in expressions to specify a certain order of calculation; the next section explains order of calculation and the use of parentheses.

Notice the symbols for multiplication and division. Computer keyboards don't always have the usual ÷ symbol. Similarly True BASIC wouldn't know if an X were a variable name or a multiplication symbol. Therefore, you must always use the multiplication symbol (*) when you want to multiply. In algebra, the expression "ab" means "a X b". True BASIC, however, would assume that "ab" is a variable name unless you specify "a*b". (The expression "a b" is "illegal" because variable names cannot contain spaces and expressions must contain an arithmetic operator.)

There is also a special symbol for exponentiation (raising to a power) because all computers can't write superscripts properly.

All expressions in True BASIC must contain appropriate arithmetic operators and they must be typed entirely on one line; that is, you must not press the Return key before you finish typing the expression.

In the MPG program, for example, the expression that computes miles per gallon must be written as:

```
miles/gallons
```

not

```
miles ÷ gallons
```

or

```
  miles
-------
gallons
```

True BASIC follows rules that decide the **order of calculation** in an expression.

• True BASIC performs multiplications and divisions before it performs additions and subtractions. Thus, if you type

```
6+10/2
```

the computer first divides 10 by 2 and then adds the 5 from that operation to the 6, getting 11. If you want to add 6 to 10 and then divide the sum by 2, you must use parentheses to force True BASIC to do that calculation first.

```
(6+10)/2
```

• If you have several multiplications and/or divisions in one expression, True BASIC computes them in order, from left to right. Thus, if you type

```
12/6*2
```

True BASIC first divides 12 by 6, and then multiplies the result (2) by 2 giving 4 as the final result. If you want to divide 12 by the result of 6 times 2 (giving 1 as the final result), you must again use parentheses to tell True BASIC to do that first:

```
12/(6*2)
```

• True BASIC computes exponents first, even before multiplications and divisions.

(A good way to remember how True BASIC does arithmetic is that it is the opposite of how you probably learned arithmetic: exponentiation first, then multiplication and division, and finally addition and subtraction. To be sure you get the results you want, use parentheses even if you think you don't need them.)

The following table shows some examples of the differences between writing regular mathematical formulas and expressions in True BASIC:

In Mathematics	In True BASIC
1 + 2 + 3	1 + 2 + 3
3 X (4 + 5)	3*(4 + 5)
$\frac{1 + 2}{4}$	(1 + 2)/4
$\frac{AB}{CD}$	(A*B)/(C*D)
x^2	x^2

Changing Values of Variables

The MPG program contains both constants and variables but it is a very simple program where each variable retains the same value throughout one program run.

Consider the following COST program that adds the cost of three items, computes a sales tax, and then gives the total purchase cost:

```
LET item1 = 250
LET item2 = 26
LET item3 = 1200
LET total = item1 + item2 + item3
LET tax = .04 * total
LET total = total + tax
PRINT total
END
```

Notice the variable *total*. In the fourth line, an arithmetic expression assigns a value to *total* (the sum of the three items, or 1476 in this case):

```
LET total = item1 + item2 + item3
```

The next line uses that value of *total* with the constant .04 to compute the value of *tax* (.04 X 1476 = 59.04). Now examine the next statement:

```
LET total = total + tax
```

This statement assigns a new value to *total* by adding the previous value of *total* (1476) to the value of *tax* (59.04). After this statement, *total* has this new value (1535.04), and thus the PRINT statement uses that value when you run the program.

You could rewrite the COST program to use a separate variable (such as *itemtotal* or *subtotal*) for the intermediate total. Indeed, using two different variables may often be the wisest choice. However, this ability to add to the value of a variable is important as you'll see when you begin to use loops in your programs.

An Introduction to Strings

True BASIC can process words as well as numbers. In computer terminology, anything that doesn't necessarily have a numeric value is called a **string**. Your age is a number, but your name or street address is a string. Strings can include any character your computer can display. Like numbers, strings can be constants, variables, or expressions.

In the Section 5, you used strings with PRINT statements to tell the user what to enter for the INPUT statements in your MPG2 program:

```
!  Compute miles per gallon
!
PRINT "How many miles";
INPUT miles
PRINT "How many gallons";
INPUT gallons
PRINT miles, gallons, miles/gallons
END
```

Another common use of strings in computer programs is to print text with the output, to make it clear what the numbers mean. You could add another PRINT statement near the end of the above program:

```
. . .
PRINT "Miles", "Gallons", "Miles per Gallon"
PRINT miles, gallons, miles/gallon
END
```

The pieces of text in all but the last of the PRINT statements are **string constants**; they cannot be changed when the program runs.

String constants (text) must be enclosed in double quote marks.

The double quotation marks keep True BASIC from treating those words as variable names.

Add the new PRINT statement shown above to your MPG2 program and run it. You should see a result similar to:

```
How many miles? 450
How many gallons? 13.6
Miles           Gallons          Miles per gallon
 450             13.6             33.0882
```

Save your MPG2 program again to keep the new PRINT statement.

Using String Constants and Variables

Just as you can have numeric constants and numeric variables, you can have string constants and string variables. **String variables** are names that represent strings, just as numeric variables are names that represent numbers. String variables may have different string values assigned to them during the run of a program.

String variable names must end in a dollar sign ($) to differentiate them from numeric variables.

Other than that, rules for string variable names are the same as those for numeric variables. That is, string variable names can consist of a letter followed by up to 30 letters, digits, or the underline character.

Programs often ask for your name and then use it again later. In a language lab, for example, a program that teaches Spanish might start by asking "Como te llamas?" and then PRINT good morning to you in Spanish. Your answer would be stored in a string variable; the Spanish phrases would be string constants.

The demo program SPANISH uses one string variable and three string constants to say hello in Spanish. (Open this program from the Demo Programs folder.)

```
!  Ask for a name, then say good morning.
!
PRINT "Como te llamas";                ! "What's your name"
INPUT name$                            ! Get the answer.
PRINT "Buenos dias, "; name$; "."      ! "Good morning..."
END
```

Run the program, and enter your name when it asks "Como te llamas?" For example:

```
Como te llamas? Sara
Buenos dias, Sara.
```

The next chapter gives more information on using strings with PRINT and INPUT statements.

A Brief Look at String Expressions

Just as there are numeric expressions, you can also use special **string expressions** in your programs.

You can combine, or **concatenate**, string constants or variables with the & (ampersand):

```
LET first$ = "Orville"
LET last$ = "Wright"
LET full$ = first$ & " " & last$
```

You can also use just part of a string — called a **substring**. The following statements create a code name from the first four characters of the last name plus the first three characters of the first name — similar to codes used on mailing labels.

```
LET first$ = "Orville"
LET last$ = "Wright"
LET code$ = last$[1:4] & first$[1:3]
PRINT code$
END
```

will print

```
WrigOrv
```

For more information on substrings and string expressions, consult the *True BASIC Bible*.

8. More on Input and Output

You've seen how INPUT and PRINT statements let you get information into and out of a program. This chapter explains these statements more fully and then introduces the LINE INPUT statement.

Printing Zones and the PRINT Statement

Look one more time at the MPG2 program and the output you get when you run the program:

```
!  Compute miles per gallon
!
PRINT "How many miles";
INPUT miles
PRINT "How many gallons";
INPUT gallons
PRINT "Miles", "Gallons", "Miles per Gallon"
PRINT miles, gallons, miles/gallons
END

How many miles? 450
How many gallons? 13.6
Miles            Gallons          Miles per gallon
 450              13.6             33.0882
```

Note that the text and the numbers in the last two lines of output line up neatly in columns. That's done by the commas in the PRINT statements.

The commas tell True BASIC that you want the items to be in print zones, or columns, that are 16 characters wide.

Change the commas to semicolons in those last two PRINT statements, and run the program again:

```
PRINT "Miles"; "Gallons"; "Miles per Gallon"
PRINT miles; gallons; miles/gallons
```

Your results should look something like this:

```
How many miles? 312
How many gallons? 8
MilesGallonsMiles per gallon
 212 8 39
```

The semicolons tell True BASIC to print the output items right next to each other.

True BASIC leaves a space on each side of a printed number, but none around strings. (True BASIC replaces the space in front of a negative number with the minus sign.)

When you write a PRINT statement to give several values, you'll probably want to use commas to separate those values into neat columns. The semicolon is useful when you are printing prompts for INPUT statements.

```
PRINT "How many miles";
INPUT miles
```

The semicolon tells True BASIC to print the ? for the INPUT statement in the space immediately following the text "How many miles".

```
How many miles?
```

With no punctuation after the PRINT statement, True BASIC would have put the ? on the next line, just as it usually puts the information from each PRINT statement on a new line.

Unless a PRINT statement ends with a comma or semi-colon, True BASIC prints the next item on a new line.

You can create blank lines in your output by using a blank PRINT statement. You can "tie" two or more PRINT statements together by ending the line with a comma or semicolon. Consider the following statements:

```
PRINT "Congratulations, "; name$; "!"
PRINT
PRINT "You have won";number_of_wins;"games out of";
PRINT number_of_attempts;"tries."
```

Can you figure out how True BASIC would print this? Make up values for the variables, but don't peek below!

Notice that the PRINT statements include **string constants** (the information in quotes), a **string variable** (*name$*), and two **numeric variables** (*number_of_wins* and *number_of_attempts*). Notice also, that the string constant "Congratulations, " includes a space so that there will be a space before the value of *name$*. But you don't need spaces in the strings that will print next to the numeric values. Remember that True BASIC puts strings right next to each other when you use semicolons, but it puts a space before and after any positive numeric value that it prints. (True BASIC puts a minus sign instead of the space before negative numbers.) Thus, True BASIC would print:

```
Congratulations, Chris!

You have won 12 games out of 25 tries.
```

More about Controlling Output

The comma and semicolon in PRINT statements let you control the appearance of your output. These two punctuation marks and the use of spaces in text constants should be adequate for most of your early ventures in programming.

The PRINT USING, SET MARGIN, and SET ZONEWIDTH statements and the TAB function let you control your True BASIC output even more precisely. PRINT USING is especially helpful if you want numeric output to follow a specific pattern. These statements are described in the *True BASIC Bible*.

You can also send your output to a printer or another file on your disk. As you've seen, the PRINT statement "prints" in the output window of your computer screen. (The reason for that is that, originally, computers didn't have screens; all output went to a printer.) Section 11 explains briefly how you can send output to a printer or a file. Section 13 has more information on sending output to a file. More information on these topics are also found in the *True BASIC Bible*.

More about the INPUT Statement

True BASIC provides a special form of the INPUT statement that lets you write your own **prompt** without a PRINT statement. For example, you could rewrite the MPG2 program to look like this:

```
!  Compute miles per gallon
!
INPUT PROMPT "How many miles?": miles
INPUT PROMPT "How many gallons?": gallons
PRINT "Miles", "Gallons", "Miles per Gallon"
PRINT miles, gallons, miles/gallons
END
```

(Don't forget the quotes and the colons.) The results will be exactly the same as before.

One last refinement of the MPG2 program: you can input both values with a single statement. You could combine the two INPUT PROMPT statements as follows:

```
INPUT PROMPT "Miles, gallons?": miles, gallons
```

When you run the program, you must now give two numbers, separated by a comma:

```
Miles, gallons? 429, 12
Miles           Gallons         Miles per gallon
 429             12              35.75
```

Save this version of MPG2 if you wish.

The LINE INPUT Statement

When you use a comma in response to an INPUT statement, True BASIC assumes you are entering another item. What happens if you want to enter a string that contains a comma?

Look again at the SPANISH demo program you saw in the last chapter:

```
!  Ask for a name, then say good morning.
!
PRINT "Como te llamas";                  ! "What's your name"
INPUT name$                              ! Get the answer.
PRINT "Buenos dias, "; name$; "."        ! "Good morning..."
END
```

If you use a comma when you give your name, you will get an error message:

```
Como te llamas? Ruy Diaz of San Antonio, Texas
Too many input items.  Excess ignored.
Buenos dias, Ruy Diaz of San Antonio.
```

One way to avoid this problem is to put quote marks around your reply:

```
Como te llamas? "Ruy Diaz of San Antonio, Texas"
Buenos dias, Ruy Diaz of San Antonio, Texas.
```

People who use your programs may not know they must use quotes, however. The LINE INPUT statement provides a better solution.

LINE INPUT tells True BASIC to take the entire line as a single item, no matter what it looks like.

Here's the SPANISH program written with a LINE INPUT statement:

```
!  Ask for a name, then say good morning.
!
PRINT "Como te llamas";                  ! "What's your name"
LINE INPUT name$                         ! Get the answer.
PRINT "Buenos dias, "; name$; "."        ! "Good morning..."
END
```

Now you can run the program and include commas in the input line:

```
Como te llamas? Ruy Diaz of San Antonio, Texas
Buenos dias, Ruy Diaz of San Antonio, Texas.
```

You can even enter no reply to a LINE INPUT by just pressing the Return key. (If you just press Return with an INPUT statement, True BASIC complains that you did not give enough input.)

9. Loop Structures

So far you've seen only "straight-line" programs. True BASIC starts at its top line, and goes straight through the program. Each statement is carried out in turn and only once. A **loop structure** lets you repeat a group of statements more than once. In a FOR-NEXT loop, you tell True BASIC exactly how many times you want to execute the statements in the loop. The DO loop lets the program decide how many times to repeat.

How a FOR-NEXT Loop Works

Let's start with the simple problem of printing the numbers from 1 to 10. Instead of a PRINT statement with ten items, or ten different PRINT statements, you can use a FOR-NEXT loop. Type in the following program and run it:

```
! Count from 1 to 10.
!
FOR i = 1 to 10                ! For each value from 1 to 10
    PRINT i;                   ! Print current value
NEXT i                         ! Increase i
END
```

Since the PRINT statement uses a semicolon, the results look like:

```
1  2  3  4  5  6  7  8  9  10
```

Let's look at what happens to *i*, the loop **index variable**. The first time True BASIC sees the FOR statement, it gives *i* the value 1. The PRINT statement uses that current value of *i*. Then, the NEXT statement increases the value of *i* by one and sends True BASIC back to the FOR statement. Now *i* equals 2.

This loop repeats ten times, until *i* reaches the value 11. At this point, *i* is greater than the high end (10) given in the FOR statement, and so True BASIC goes to the first statement after the NEXT statement, the END statement. Thus, this FOR-NEXT loop means "for each number from 1 to 10, print the number."

The FOR-NEXT loop is a **structure** in True BASIC, or a kind of framework that organizes other statements. The variable *i* in this program is called the **index variable**; it acquires a new value each time the loop runs.

The same index variable must appear in both the FOR statement and the NEXT statement.

The statement(s) between the FOR and the NEXT statements are carried out (or executed) as many times as the loop is repeated. In this Guide, the statements inside the loop (in this case, the

PRINT statement) are indented more than the FOR and NEXT statements. This is a matter of style; it's not required in True BASIC, but it makes the program much easier to read.

The loop alters the straight-line flow of control by repeating a group of statements. Such structures let you take advantage of the great power of computers.

Step Size in a Loop

The NEXT statement above added 1 to the index variable each time through the loop. You can make the NEXT statement add something other than 1 by putting your own **step size** in the FOR statement. For example, if you want a table of square roots in increments of one-tenth, you can use .1 as the step size.

Now open the demo program SQROOT from the Demo directory:

```
! Square roots.
!
PRINT "number", "Square Root"    ! Print labels
PRINT! Leave blank line
FOR number = 0 to 1 step .1      ! 0 to 1 in small steps
    PRINT number, Sqr(number)    ! Number & square root
NEXT number
END
```

and run it:

```
  Number          Square Root

   0               0
   .1              .316228
   .2              .447214
   .3              .547723
   .4              .632456
   .5              .707107
   .6              .774597
   .7              .83666
   .8              .894427
   .9              .948683
   1.              1
```

This program uses the built-in function SQR to obtain the square root of *number*. (Chapter 14 explains functions.)

If you want, you can have a negative number for a step size. This makes the loop count down instead of up. Change the FOR statement so that your program looks like this:

```
! Square roots.
!
PRINT "Number", "Square Root"    ! Print labels
PRINT                  ! Leave blank line
FOR number = 10 to 5 step -1     ! Go from 10 down to 5
    PRINT number, Sqr(number)    ! Number & square root
NEXT number
END
```

When the step size is negative, the starting and ending conditions for the loop must also be backwards — that is, they must go from large to small. In the first version of SQROOT, the loop stopped when the number became greater than one. In the version with a negative step size, the loop stops when *number* becomes less than five. (If you forget to change the step from .1 to -1, your loop won't execute at all, because *number* can't get from 10 to 5 without a negative step.)

```
Number          Square Root

 10              3.16228
 9               3
 8               2.82843
 7               2.64575
 6               2.44949
 5               2.23607
```

You can use the index variable (here, *number*) outside its loop. But what value will it have outside the loop? Add a PRINT statement to SQROOT so you can see what value *number* has after the loop stops:

```
! Square roots.
!
PRINT "Number", "Square Root"    ! Print labels
PRINT                            ! Leave blank line
FOR number = 10 to 5 step -1     ! Go from 10 down to 5
    PRINT number, Sqr(number)    ! Number & square root
NEXT number
PRINT number
END
```

and run it again:

```
Number          Square Root

 10              3.16228
 9               3
 8               2.82843
 7               2.64575
 6               2.44949
 5               2.23607
 4
```

As you can see, *number* equals 4 after the loop ends.

A FOR-NEXT loop always leaves the index variable with the first value that fails the end test.

Nested Loops

You may use any True BASIC statements inside a FOR-NEXT loop, even another loop. Some problems are best solved by using loops inside loops, that is, **nested loops**.

As an illustration, open the demo program EXES:

```
! Print pattern of x's.
!
FOR row = 1 to 6

    FOR xcount = 1 to row
        PRINT "x";
    NEXT xcount

    PRINT
NEXT row
END
```

This program prints a pattern of x's on the screen:

```
x
xx
xxx
xxxx
xxxxx
xxxxxx
```

Let's analyze this program. It has two loops: an outer loop with the variable *row* as the loop index, and within that an inner loop with the index variable *xcount*.

The inner or nested loop must be entirely inside the outer loop.

Each time the outer loop goes through one big cycle, the inner loop goes through as many cycles as the current value of *row*. This creates the triangle pattern. As you can see, the first row has one x, the second has two, and so on.

Note the empty PRINT statement just after the inner loop and just before the end of the outer loop. This second PRINT statement is carried out only at the end of a row. It tells True BASIC to start a new line. If it wasn't there, the program would just print 21 x's on one line.

If you want to print more than one triangle, you'll have to use three loops, not just two. Nest a new loop between the *row* and *xcount* loops. Notice how the indenting and blank lines help you keep track of which loop is which:

```
! Print pattern of x's.
!
FOR row = 1 to 6

    FOR triangle = 1 to 3   ! new loop starts here

        FOR xcount = 1 to row
            PRINT "x";
        NEXT xcount

        PRINT,          ! new PRINT with comma
    NEXT triangle       ! new loop ends here

    PRINT
NEXT row
END
```

Just as you need an empty PRINT statement to move to the next line before the NEXT row, you also need a PRINT statement with a comma before the NEXT triangle, to move to the next PRINT zone.

```
x               x               x
xx              xx              xx
xxx             xxx             xxx
xxxx            xxxx            xxxx
xxxxx           xxxxx           xxxxx
xxxxxx          xxxxxx          xxxxxx
```

An Introduction to Conditions

In the FOR-NEXT loop, you must specify how many times you want the loop to repeat. Computers, however, are quite capable of making decisions based on a condition that you specify. The DO loop, introduced in the next section, and the decision structures you'll see in the next chapter both use conditions.

A **condition** in True BASIC is a comparison of values. Conditions use **relational operators**:

Operator	Meaning
=	equal to
<> or ><	not equal to
<	less than
<= or =<	less than or equal to
>	greater than
>= or =>	greater than or equal to

Conditions themselves have either true or false values. For example:

Condition	Value
1 < 2	true
1 + 2 < 3	false
5 + 3 >= 8	true
“abc” <> “ABC	true
“yes” = “no”	false
“elephant” < “spider”	true
“elephant” < “Spider”	false
“moon” < “moonbeam”	true

Notice that you can compare strings as well as numbers. True BASIC orders string values containing letters alphabetically except that all uppercase letters come before (are less than) any lowercase letters. Shorter strings come before longer strings that begin with the same characters. Most other characters (such as !, “, #. and $) and numbers come before letters. The order for string characters is based on the ASCII character set, which is the standard code that most computers use to represent keyboard characters. (Appendix A lists the ASCII character set.)

The next section shows how you can use conditions in DO statements.

An Introduction to DO Loops and Counters

The DO loop lets you repeat a group of statements just like the FOR-NEXT loop except that you don’t specify number of repetitions. Instead, you specify a **condition** and True BASIC repeats the loop **until** the condition becomes true or **while** (as long as) the condition remains true.

Let’s say you have $10,000 in a savings account, and the bank gives 8.5% interest. At the end of the first year, the bank will give you $850. If you leave this money in the account, the next year you’ll earn interest on $10,850, which yields slightly more than another $850. And so forth. Each year you’ll make a little more in interest than the year before. How long will it take for your money to double?

Open the program INTEREST from the Demo Programs folder.

```
! Program to compute interest on a bank account.
! Stop when the money has doubled.
!
LET years = 0
LET money = 10000           ! Start with $10,000
LET original = money        ! Remember original amount
LET interest = 8.5/100      ! Interest is 8.5%

DO until money >= 2 * original ! Loop until money doubles
```

```
   PRINT years, money       ! Print year and money
   LET years = years + 1    ! Keep track of how long
   LET money = money + (interest * money) ! Add interest

LOOP
PRINT "In"; years ; "years, you'll have $"; money
END
```

Run the program:

```
 0              10000
 1              10850
 2              11772.2
 3              12772.9
 4              13858.6
 5              15036.6
 6              16314.7
 7              17701.4
 8              19206.
In 9 years, you'll have $ 20838.6
```

Let's analyze how this program works. It starts off with three LET statements assigning starting values to the variables *years*, *money*, *original*, and *interest*. (It's a good idea to treat *original* and *interest* as variables instead of constants, because then it'll be easier to change the program later on.)

The DO UNTIL statement means "repeat the following group of statements until *money* is greater than or equal to two times the original amount." The PRINT statement displays the current values of *years* and *money*, and the first LET statement inside the loop adds 1 to the value of *years*. The second LET statement in the loop takes the "old" value of *money*, computes the interest on that value, adds the interest to the "old" value, and puts that sum into the "new" value of *money*. The LOOP statement marks the end of the group of statements, and tells True BASIC to go back to the DO UNTIL statement.

True BASIC checks the condition (*money* > = 2 * *original*) each time before it executes the loop. If it had been true the very first time, True BASIC would never have executed the loop!

The second time around, *money* is 10850, still less than $20,000, so True BASIC repeats the loop. The third time it's 11772.20 so True BASIC repeats the loop, and so on. The last time through, *money* reaches the value 20838.60. Then, when True BASIC returns to the DO UNTIL statement, money is greater than 2 * *original*. So the loop ends.

True BASIC then continues with the next statement after LOOP, which is the last PRINT statement. Thus the loop finishes when *money* has doubled (or more).

Notice again the LET statement inside the loop that adds 1 to the value for *years*. The variable *years* is a **counter**. It is counting the number of times True BASIC goes through the loop, which in this case is the number of years the money has been in the bank.

Change the interest rate and see how that affects the DO loop. Edit the LET statement that assigns the initial value to *interest* and run the program again.

```
LET interest = 11.5/100! Interest is 11.5%
```

With 11.5% interest, you should find that the DO loop works only seven times instead of nine as it did before. However, the condition (*money* > = 2 * *original*) is still met.

Note: The INTEREST program doesn't format dollar amounts as you are used to seeing them:

```
In 9 years, you'll have $ 20838.6
```

True BASIC's PRINT USING statement lets you control the exact format of numeric (and string) output. For example, you could replace the last PRINT statement in INTEREST with the following two PRINT statements:

```
PRINT "In"; years ; "years, you'll have ";
PRINT USING "$##,###.##": money
```

With those statements, the final output line looks like:

```
In 9 years, you'll have $20,838.56
```

More information on this topic is found in the *True BASIC Bible*.

Variations on DO Loops, and Combining Conditions

With the UNTIL condition test on the DO statement, it is possible that the statements in the loop will never run. You can put the test on the LOOP statement instead of the DO statement. In that situation, the statements in the loop will always run at least once, because True BASIC won't check the condition until it reaches the end of the loop.

```
DO
   PRINT years, money                ! Print year and money
   LET years = years + 1             ! Keep track of how long
   LET money = money + (interest * money)  ! Add in interest
LOOP until money >= 2 * original  ! Loop until money doubles
```

Instead of repeating the loop until the condition becomes true, you can loop while the condition remains false. The two statements:

```
LOOP until money >= 2 * original
```

and

```
LOOP while money < 2 * original
```

are equivalent. "While" and "until" are opposites, just as >= and < are opposites.

As with UNTIL, you can use either DO WHILE or LOOP WHILE. A DO WHILE loop may never be used if the condition is false the first time; a LOOP UNTIL loop always runs at least once since the test is made at the end of the loop.

You can also combine conditions with True BASIC's logical operators: AND, OR, and NOT. You can use a combined condition anywhere a simple condition works. For example, the following statement would continue the loop until either the money doubles or 8 years go by:

```
LOOP until money >= 2 * original OR years >= 8
```

These variations on the DO loop and on using conditions are described more fully in the *True BASIC Bible*.

10. Decision Structures

So far, you've seen simple programs where every statement is carried out in turn straight through the program. You've also learned about using loops where a group of statements may be used several times or not at all. In this chapter, you'll write programs that can decide which of two sets of statements to use.

Simple IF-THEN Decisions

The IF-THEN statement in True BASIC forms a structure, or framework, for a decision. The IF part of the structure contains a condition that True BASIC uses to decide which parts of the structure to use.

IF statements use conditions just as the DO loop introduced in the last chapter. (If you need a quick review, refer to "An Introduction to Conditions" in the previous chapter.)

The simplest IF-THEN decision carries out a single statement if a certain condition is true. Call up the demo program COINS to see an example of a simple decision.

```
! Flip a coin five times.
!
FOR toss = 1 to 5
    IF Rnd<.5 then PRINT "Heads, you win"
NEXT toss

END
```

This program simulates tossing a coin by using the RND, or random number, built-in function. RND gives a different random number between 0 and 1 each time it's used. Half the time, the random number will be greater than 1/2, half the time it will be less. The COINS program prints "heads, you win" each time the random number is less than 1/2. The rest of the time, it doesn't print anything. (Chapter 14 explains built-in functions more fully.) For example:

```
Heads, you win
Heads, you win
```

Two out of the five times, the "coin" came up "heads" or less than 1/2. The other three times it was "tails" or greater than or equal to 1/2. You can't tell which tosses were heads or tails, however. When it was tails, True BASIC just ignored the PRINT statement and went on to the NEXT statement.

Single-line IF-THEN-ELSE Decisions

The ELSE keyword lets you write a statement that will be carried out only when the condition is false. To print a different message for tails, add an ELSE and another PRINT statement to the IF-THEN structure in the COINS program:

```
! Flip a coin five times.
!
FOR toss = 1 to 5
   IF Rnd<.5 then PRINT "Heads, you win" ELSE PRINT "Tails, you
lose"
NEXT toss

END
```

Remember that you must enclose text in double quotes (“). Run this new version:

```
Tails, you lose
Heads, you win
Tails, you lose
Heads, you win
Tails, you lose
```

Now you know that the second and fourth times were heads, and the first, third, and fifth were tails. Just as the THEN keyword precedes the statement to be executed when the condition is true, the ELSE keyword precedes the statement to be executed when the condition is false.

Multiple-Line Decisions

Quite often you want to execute more than one statement if a condition is true or false. In that case, you need to use more than one line for the IF-THEN or IF-THEN-ELSE structure. You also need an END IF keyword to mark the end of the structure.

Even though it has only one statement each for true or false conditions, you can change your COINS program to use a multiple-line IF-THEN-ELSE structure. Press the Return key to split the IF-THEN statement onto several lines, and add an END IF statement.

```
! Flip a coin five times.
!
FOR toss = 1 to 5
    IF Rnd<.5 THEN
       PRINT "Heads, you win"
    ELSE
       PRINT "Tails, you lose"
    END IF
NEXT toss

END
```

Run this program. You should see the same results as when it was a single-line IF-THEN-ELSE structure.

If you get an error message such as "Can't use this statement here", "Doesn't belong here", or "Ending doesn't match beginning", you probably haven't started the new lines in the right places.

In the multiple-line IF structures, the keyword THEN must be the last word in the IF statement. The two keywords ELSE and END IF must be on lines by themselves.

Each statement (such as a PRINT or LET) within the structure must also be on a line by itself.

When the condition is true, True BASIC executes the statements between the IF statement and the ELSE keyword, ignores the statements between the ELSE keyword and the END IF keyword, and jumps to the statement right after the END IF statement. When the condition is false, True BASIC ignores the statements between the IF Statement and the ELSE keyword, executes the statements between the ELSE keyword and the END IF keyword, and continues with the statement right after the END IF statement.

More About Counters

In the previous chapter, you saw how a variable can count the number of times something happens in a program run. The **counter** there was the variable *years*. The statement

```
LET years = years + 1
```

added 1 to the value stored in *years* each time the loop was run.

You can use variables such as *heads* and *tails* in the COINS program to count the number of times the toss comes up heads or tails. Add the two LET statements to the IF structure as shown below along with the two new PRINT statements after the FOR-NEXT loop.

```
! Flip a coin five times.
!
FOR toss = 1 to 5
   IF Rnd<.5 then
      PRINT "Heads, you win"
      LET heads = heads + 1      ! Count heads
   ELSE
```

```
      PRINT "Tails, you lose"
      LET tails = tails + 1        ! Count tails
   END IF
NEXT toss

PRINT
PRINT "You won"; heads; "times.  I won"; tails; "times."

END
```

Run this version of COINS. Each LET statement assigns the variable its "old" value plus one whenever its group of statements are used. (In True BASIC, every numeric variable starts with the value of zero.)

```
Tails, you lose
Heads, you win
Tails, you lose
Heads, you win
Tails, you lose

You won 2 times.  I won 3 times.
```

The RANDOMIZE Statement

You may notice that each time you run COINS, the tosses come out the same: tails, heads, tails, heads, tails. The "random number generator" for the RND function creates the same sequence of "random" numbers each time. This makes it easier for you to "debug" or check your programs for accuracy. Even if it uses random numbers, your program will work the same each time you run it. However, this feature also makes your programs less random.

To scramble the sequence of random numbers, add a RANDOMIZE statement to the start of your program. You only need one RANDOMIZE statement in a program to make the RND function unpredictable in that program. (In fact, using RANDOMIZE more than once can actually make your random numbers *less* random.) It's a good idea to put the RANDOMIZE statement after the comments at the very beginning of the program and before any other "executable statement".

```
! Flip a coin five times.
!
RANDOMIZE

FOR toss = 1 to 5
   IF Rnd<.5 then
      PRINT "Heads, you win"
      LET heads = heads + 1        ! Count heads
   ELSE
```

```
      PRINT "Tails, you lose"
      LET tails = tails + 1        ! Count tails
   END IF
NEXT toss

PRINT
PRINT "You won"; heads; "times.  I won"; tails; "times."

END
```

Run this version of COINS several times. You should get different results each time.

Save a copy of this version of the program if you wish — perhaps with a different name. You may want to use all or part of it in your own programs later on.

The STOP Statement

Many programs use IF structures to decide when to stop. The program could ask the user if they wish to continue and then make a decision based on the response, or the program could "decide" to stop when it completes its task.

Call up and look at the demo program GUESS. This program uses the built-in functions INT and RND to "think" of a number between 1 and 6. (The next section describes how that works.) You then have three chances to guess the number. A FOR-NEXT loop gives you the three guesses. If you guess correctly before you've used all three chances, a STOP statement in the IF structure ends the program at that point.

```
! Program to play a guessing game.
!
RANDOMIZE
LET answer = Int(Rnd*6) + 1          ! Choose number from 1-6

PRINT "I'm thinking of a number from 1 to 6."
PRINT "You have 3 chances to guess it."
PRINT
FOR chance = 1 to 3
    PRINT "Enter your guess";        ! Ask for number
    INPUT guess
    IF guess = answer THEN
       PRINT "Correct!!!"
       STOP                          ! Stop here, you guessed it
    END IF
NEXT chance
PRINT
PRINT "The number was"; answer
END
```

Run the program a few times to see how lucky you are. The output will be different each time, because the program has a RANDOMIZE statement.

Generating Random Whole Numbers

You've now seen two programs that use the RND built-in function to produce a number randomly. The RND function always gives a decimal value between 0 and 1 (but never exactly 1). In the COINS program, you didn't what care the number was, you just needed to split the numbers into halves — less than .5, or .5 or greater.

The GUESS program is a bit trickier:

```
LET answer = Int(Rnd*6) + 1
```

First the RND function gives a decimal value between 0 and 1 (but never exactly 1). That value is multiplied by 6 to create a value between 0 and 6 (but never exactly 6). As that value is very likely a decimal value (such as 4.327), the statement also uses the INT (Integer) function to take just the integer or whole number part: 0, 1, 2, 3, 4, or 5. Finally, 1 is added to give a whole number between 1 and 6.

Other Decision Structures

The IF-THEN-ELSE structure gives you two possible **branches** for your decisions. The program makes a decision and then carries out one of two sets of statements. You can **nest** an IF structure inside another if you wish to make additional decisions, but this can be awkward if you have several related decisions.

True BASIC includes two more decision structures that let you choose among three or more sets of statements. The *True BASIC Bible* describes these structures more fully. The programs shown below provide a quick introduction; these programs are in the Demo Programs folder that came with **True BASIC *Free***.

The **ELSE IF statement** expands the IF structure to allow for multiple decisions. Consider the guessing game played in the GUESS program. In that program there are just two things that might happen after you guess: the program says you are wrong, or it says you are correct and the game ends. The program GUESS2 can do one of five things based on your guess:

```
! Program to play a guessing game.
!
RANDOMIZE
LET answer = Int(Rnd*10) + 1    ! From 1 to 10

PRINT "I'm thinking of a number from 1 to 10."
PRINT "You have 3 chances to guess it."
PRINT

FOR chance = 1 TO 3
```

```
   PRINT "Enter your guess";     ! Ask for number
   INPUT guess! Get a guess

   IF guess < 1 THEN ! Check it out
      PRINT "Must be at least 1."
   ELSE IF guess > 10 then
      PRINT "Can't be more than 10."
   ELSE IF guess < answer then
      PRINT "Too low."
   ELSE IF guess > answer then
      PRINT "Too high."
   ELSE! Must be right
      PRINT "Correct!!!"
      STOP
   END IF
NEXT chance

PRINT "The number was"; answer; "."
END
```

The **SELECT CASE structure** lets you choose among several alternatives as does the IF-THEN-ELSE IF statement, but it handles the condition test a bit differently. The CRAPS program plays the dice game "Craps". The rules are simple. You play ten times. Each time you roll two dice. If you roll 2, 3, or 12, you lose; roll 7 or 11 and you win outright. Otherwise, you remember your "point" on that first roll, and keep rolling until you get either a 7 or your point again. If you get your point, you win; but if you get a 7, you lose. If you don't know the game, the True BASIC program might make the rules easier to follow:

```
! Craps game.
!
RANDOMIZE

FOR game = 1 to 10! Play 10 games

   LET die1 = Int(6*Rnd + 1)     ! Roll 1 die
   LET die2 = Int(6*Rnd + 1)     ! And the other
   LET dice = die1 + die2   ! Sum of two dice

   PRINT dice;         ! Print this roll

   SELECT CASE dice   ! Branch on roll

      CASE 2, 3, 12   ! dice = 2, 3, or 12
         PRINT "You lose."

      CASE 7, 11! dice = 7 or 11
         PRINT "You win."

      CASE ELSE         ! Anything else
         LET POINT = dice   ! Remember that roll
         DO
```

```
            LET die1 = Int(6*Rnd + 1) ! Roll again
            LET die2 = Int(6*Rnd + 1) ! Both dice
            LET dice = die1 + die2
            PRINT dice;     ! Print this roll
         LOOP until dice = 7 or dice = point

         IF dice=point then PRINT "You win" else PRINT "You lose"
   END SELECT

NEXT game

END
```

For more information about these structures see the *True BASIC Bible*.

11. Formatting and Printing Your Program

You've now learned the basic elements of programming. This is a good time to review and add to your knowledge of program format. First, a quick review of the "facts":

• True BASIC programs can contain comments, blank lines, or "executable" statements that give instructions to True BASIC.

• **Statements** always begin with a **keyword**. A space must separate the keyword from anything else on the same line.

• **Comments** begin with an exclamation point. They may be on a line by themselves or at the end of an executable statement. They have no effect on how the program runs, but they make it much easier for a person to understand what the program does.

• **Blank lines** have no effect on how the program runs, but like comments they make a program much easier to read.

• **Variable names** may be up to 31 characters long. They must begin with a letter, but may then contain any letters, digits, or underscore characters (_). String variable names must end with a dollar sign ($).

• All **string constants** (text) must be inside double quotation marks.

• All True BASIC programs must end with an **END statement**.

Guidelines for Good Programming

The program examples in this book illustrate some simple guidelines that can make your programs easier to read and lead you to good programming style:

• Use comments at the beginning of a program to tell what the program does. This is also a good place to add your name and information about the date and version of the program.

• Use comments throughout the program to explain what each segment or structure does.

• Use variable names that give some clue about what they are used for. *Miles*, *years*, *original*, *roll*, *toss*, *guess*, and *answer* say a lot more than *m*, *y*, *o*, *r*, *t*, *g*, or *a*.

• Indent multiple-line structures such as loops and decision structures to show more clearly the structure itself and the blocks of statements that are contained within the structure.

Indenting with Do Format

True BASIC comes with a formatting tool that can indent your program for you. The NOINDENT demo program in the Demo Programs folder is another version of the GUESS program with no blank or indented lines. This version has a nested IF structure. Open NOINDENT and try to follow the structures in the unindented format.

```
! Program to play a guessing game.
!
randomize
let answer = Int(Rnd*6) + 1 ! Choose number from 1-6
print "I'm thinking of a number from 1 to 6."
print "You have 3 chances to guess it."
print
for chance = 1 to 3
print "Enter your guess";  ! Ask for number
input guess
if guess = answer THEN
print "Correct!!!"
stop! Stop here, you guessed it
else! Analyze wrong answers
if guess > answer then
print "Too high.  Guess again."
else
print "Too low.  Guess again."
end if
end if
next chance
print
print "The number was"; answer
end
```

Now select the **Do Format** command in the **Custom** menu (or use the Command-D shortcut). This command indents the statements inside structures and puts all keywords into uppercase. You should find the structures much easier to follow. (In fact, Do Format is a good first step in debugging your program. Chapter 18 has more information on that.)

```
! Program to play a guessing game.
!
RANDOMIZE
LET answer = Int(Rnd*6) + 1 ! Choose number from 1-6
PRINT "I'm thinking of a number from 1 to 6."
PRINT "You have 3 chances to guess it."
PRINT
FOR chance = 1 to 3
    PRINT "Enter your guess";       ! Ask for number
    INPUT guess
    IF guess = answer THEN
```

```
         PRINT "Correct!!!"
         STOP                  ! Stop here, you guessed it
      ELSE! Analyze wrong answers
         IF guess > answer then
            PRINT "Too high.  Guess again."
         ELSE
            PRINT "Too low.  Guess again."
         END IF
      END IF
   NEXT chance
   PRINT
   PRINT "The number was"; answer
   END
```

You should now be able to easily see and follow the nested IF structure that is in the ELSE segment of the first IF structure.

To make this program even more readable, you could add some blank lines. Remember how to do this? Place the cursor (vertical bar) at the end or beginning of a line and press the Return key. Use the Delete key at the beginning of the line to remove undesired blank lines.

Indenting Blocks with > and < keys

You can, of course, indent single lines by adding spaces at the beginning of the line.

You can also easily indent a block of lines in True BASIC. First, select the lines you wish to indent by dragging across those lines with the mouse pointer. Then you can use the > or < keys to move all the selected lines to the right or left. Each time you press > the block moves one space to the right; each time you press <, it moves one space to the left. (Notice that you must hold the Shift key to get < or > instead of a comma or period.)

Listing Your Programs on a Printer

You can get a paper (or hard-copy) listing of your program by using the **Print...** command in the **File** menu (or press Command-P). To get a listing of the entire program, be sure that you haven't selected any lines when you give the Print command.

To print just part of your program, first use the mouse to select the desired lines and then choose the **Print...** command. Select multiple lines by dragging across them with the mouse. (The next chapter shows other ways of selecting multiple lines.)

If you have trouble printing, check the following:

- Be sure your printer is turned on.
- Check that the printer cable is firmly connected at both ends.

• Select the **Chooser** Desk Accessory in the Apple menu and make sure you have selected the proper printer.

• Use the **Page Setup...** command in the **File** menu to be sure you have the proper choices for paper and orientation.

See the next section in this chapter "Using the Command Window" for information on the LIST command that also prints all or part of your program.

The next section describes the LIST command, used in the command window, that also prints all or part of your program.

Using the Command Window

So far, you've told True BASIC what to do with menu choices (or Command key shortcuts). You can also give commands by typing them in a **command window**.

Open the command window by choosing **Command** in the **Windows** menu (or use the Command-J shortcut). True BASIC types 'Ok.' in this window to signal that you can type a command. You may type many commands that are also available in the menu, such as RUN, SAVE, OLD (to open an existing program), NEW (to create a new untitled window), or DO FORMAT. There are also several True BASIC commands that are not in the menu:

• The LIST command prints all or part of your program on your printer. To list part of your program indicate the lines to be printed, such as LIST 1-10 for the first 10 lines.

• The ECHO and RUN > > commands let you send copies of your output to a printer or a file; these are described in the next section.

• Other commands are helpful in debugging or correcting errors in your programs. Section 19 introduces some debugging commands. These and all True BASIC commands are described fully in the *True BASIC Bible*.

Echoing Output to a Printer or File

When you run your programs, the results are "printed" on the screen in the output window. You can also send those results to your printer or to a file (which you could later list on a printer). The ECHO command, which you use in a command window, is the easiest way to do this.

To send output to your printer, open the command window and then type ECHO at the 'Ok.' prompt. When you next use the RUN command, True BASIC sends a copy of your output to your printer; True BASIC also prints any commands you give. This stay in effect until you type ECHO OFF. You must be sure that your printer is

on and directly connected to your computer. This command does not work for a printer connected over a network.

If you have problems echoing to a printer or you do not have a direct connection to a printer, the best solution is to echo to a file and then print that file. To do that, use the command window to give the command ECHO TO followed by a file name. (If the file doesn't exist, True BASIC creates it for you; otherwise, it adds to the end of the existing file.) True BASIC sends a copy of all subsequent commands and program output to that file until you type ECHO OFF. You can then open the file and print it with the **Print...** command in the **File** menu or the LIST command in the command window.

As an example, open the MPG2 program you created earlier and then open the command window and type the following at the 'Ok.' prompt:

```
echo to mpgout
run
```

The program will run as usual. True BASIC will ask for number of miles and gallons and prints the results on your screen. You can then return to the command window and type:

```
echo off
old mpgout
list
```

The MPGOUT file will contain the following; the LIST command will send these lines to your printer.

```
Ok. run
How many miles? 450
How many gallons? 13.6
Miles            Gallons          Miles per Gallon
 450              13.6             33.0882
Ok. echo off
```

You can also echo output to a file by giving the file name with the RUN command in the command window as follows:

```
run > filename
```

Only the results of the run will go to the file, not the commands.

Sending Output from a Program to a Printer

Your True BASIC program can also send all or part of its output to a printer. You do this by "opening a channel" to the printer within the program. Here is a quick introduction:

```
OPEN #1: printer          ! Opens channel #1 for the printer
FOR i = 1 to 10
    PRINT #1: i           ! Print to #1 -- the printer
NEXT i
END
```

After the OPEN statement that identifies the printer, a plain PRINT statement will still "print" to the screen, but PRINT #1 will send output to the printer. You may want to print input prompts on the screen, but send the results of a calculation to the printer. If you want results to go to both the printer and the screen, you must have two print statements for each output line.

You can also open a channel to a file and PRINT output to that file. Chapter 13 tells more about using files with your programs.

Printing graphics output is a bit trickier. Chapter 17 in this book has a brief introduction to this process.

Using Line Numbers

True BASIC's structures and editing features make it unnecessary to use line numbers on your programs. Although True BASIC recognizes and allows statements that rely on line numbers (such as GOTO 1025), such statements are a holdover from the days before good structured programming languages were developed. You won't find them described in this manual. For information on using line numbers on True BASIC programs see the *True BASIC Bible*.

Some of the Edit menu commands, introduced in the next chapter, let you specify lines by sequence number. For example, 1-10 refers to the first 10 lines of an unnumbered program.

12. Editing Hints and Shortcust

You've already edited several small True BASIC programs, and you've seen in the previous chapter how you can improve the format of your programs. True BASIC has some special editing commands and shortcuts that you may find useful as you continue working with more and larger programs.

The **Edit** menu contains four sections of commands. This chapter explains the first two groups of command. The last two groups are introduced briefly; you'll find them more helpful later as you begin to work with larger programs.

Selecting, Deleting, Moving, and Copying

The **Cut**, **Copy**, and **Paste** commands in True BASIC's **Edit** menu work just like those commands in just about any application. Use these commands to move lines or parts of lines or to duplicate a part of the program.

If you are not familiar with **Cut**, **Copy**, and **Paste**, practice using them with the SMOKY demo program as described below. (Just don't save your changes without using **Save As...** to rename the program; you'll use SMOKY again in Chapter 18.)

Selecting Text. To delete, move, or copy something, you must first select or highlight the desired text. Use the mouse to:

• drag across the desired words or lines

• double-click on a word to select that word

• double-click at the beginning or end of a word to select the word along with adjacent space and punctuation

• triple-click to select an entire line

You can extend a selection by moving the mouse pointer and then holding the Shift key while you click with the mouse.

Open the demo program SMOKY and run it to see what it does. Now practice selecting the four lines of DATA statements. Try both methods of selecting lines:

• Triple-click to select the first DATA statement (move the pointer to that line and click three times quickly without moving the mouse). Next, move the pointer down to the last DATA statement and Shift-click (hold the Shift key while you click on the mouse).

• Drag across the four lines (hold the mouse button down while you move the pointer across the lines).

Deleting Lines. Once you've selected something, use the **Cut** command to remove the text.

Select the two comment lines in the SMOKY program and choose **Cut** in the **Edit** menu (or press Command-X). The lines will disappear and True BASIC will print the message "2 lines deleted" at the bottom of the window.

The **Cut** command puts these lines into the "clipboard" (usually invisible) so you can get them back. Choose **Paste** in the **Edit** menu (or press Command-V). True BASIC will put the lines back and print the message "2 lines put back" at the bottom of the window.

The Cut command removes selected text from your program and puts it in the clipboard.

Note that you can also use the Delete key to remove selected lines. But, unlike **Cut**, the Delete key does not put anything in the clipboard. You cannot Paste something that has been "deleted".

Moving Lines. Use **Cut** and **Paste** to remove selected lines and then insert them elsewhere in the file.

This time, select the four DATA lines in the SMOKY program. Use the **Cut** command to remove the lines (and put them in the clipboard). Then move the insertion point to the left of the DO statement. Now choose the **Paste** command. True BASIC puts the DATA lines before the DO loop and prints the message "4 lines put back".

The Paste command puts the current contents of the clipboard at the current insertion point in your program.

Run the program again. It still works, regardless of the location of the DATA lines. You'll learn more about this statement in the next section.

Notice that the two comment lines disappeared from the clipboard when you copied the four DATA lines. The clipboard holds only one selection at a time. It contains the last thing you cut or copied. Previous contents are lost each time you use **Cut** or **Copy**, but you may **Paste** the same text from the clipboard as many times as you wish.

Copying Lines. You can copy selected lines to another part of your program by using **Copy** and **Paste**. **Copy** puts the selected

lines into the clipboard without removing them from the program. You can then **Paste** a copy to another spot.

Make a second copy of the four DATA lines to follow the existing DATA lines. Select the four DATA lines in the SMOKY demo program and choose **Copy** in the **Edit** menu. The selection will go away and the insertion point will reappear.

The Copy command puts a copy of selected text in the clipboard without removing the text from your program.

Move the insertion point to the line below the last DATA statement, and choose **Paste** in the **Edit** menu . True BASIC inserts a new copy of the four DATA lines and prints the message "4 lines put back".

Run the program again. You'll hear the same lines twice. (Remember you can stop the program at any point by selecting **Stop** in the **Run** menu.)

Search and Replace

Finding Words. Put the insertion point at the beginning of the SMOKY program, and choose **Find...** from the **Edit** menu. True BASIC will present a box with buttons for "OK" and "Cancel". Type:

```
data
```

in either upper or lowercase. Press the Return or the Enter key, or click the OK button on the right side of the box. True BASIC will select the first occurrence of the word *data* in the program:

```
DO while more data
```

To find the next occurrence of the word *data*, choose **Find Again** in the **Edit** menu (or press Command-G). True BASIC will select the next occurrence of the word *data*, which is the first DATA statement.

When searching for whole words, True BASIC finds the word regardless of whether it is in lowercase or capital letters.

Finding Parts of Words. Choose the **Find...** command again. This time, type:

```
dat
```

and click OK. True BASIC will tell you, "Not found" and the insertion point will stay on the first DATA statement. Although *dat* is part of *data*, and there are three more occurrences of *data* in the program, **Find...** doesn't normally locate just part of a word.

If you want to find just part of a word, you must first remove the X from the "Find Whole Word" check box. Choose **Find...** from the **Edit** menu again. Retype *dat* so that it is in uppercase letters:.

```
DAT
```

Click the "Find Whole Word" check box so that the X disappears (this turns the option off), and click OK. This time True BASIC will move down to the next DATA statement. If the search fails, it's because your upper or lowercase letters didn't match those in the program.

With Find Whole Word turned off, the characters you type must exactly match the characters in the program.

Without moving your insertion point, choose **Find...** one more time. This time look for the word:

```
read
```

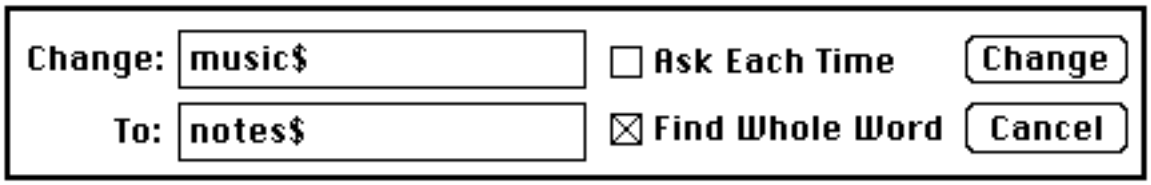
Click to put a check next to "Find Whole Word" again, and click OK. Even though the program SMOKY contains a READ statement, True Basic won't find it because the insertion point was below the READ statement when you used Find...

Summary: True BASIC always searches from the insertion point to the end of the program, and then stops.

Now move the insertion point to the very beginning of the program. Choose **Find Again**. True BASIC will find the READ statement now, because you started the search at the very beginning of the program.

Changing Text

The **Change...** command lets you change all occurrences of a word or number to a different word or number. Choose **Change...** from the **Edit** menu. On the Macintosh, you'll see a box with two places for you to type. In the rectangle marked "Change:", type the word *music$*. Press the Tab key or click to move your insertion point to the "To:" rectangle, and type the word *notes$*.



Change Box for MacOS

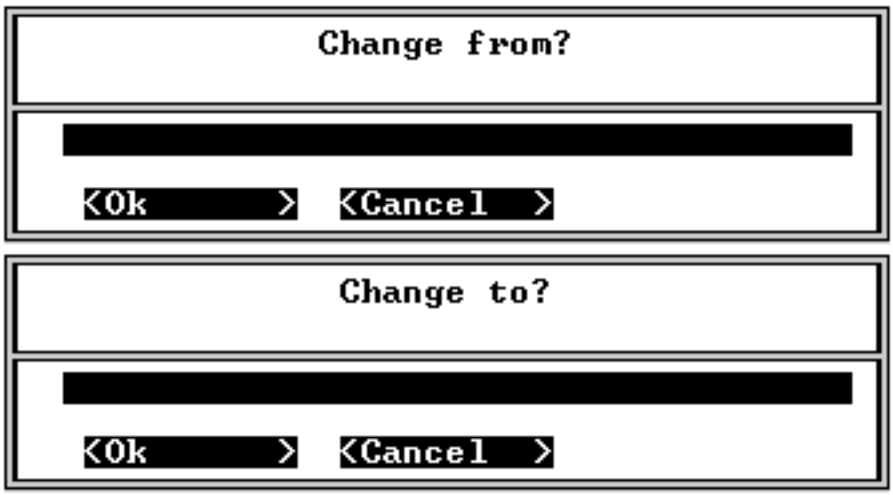
Leave the preset values of "Ask Each Time" and "Find Whole Word" as they are, and click the Change button at the right of the box.

The **Change...** command, like **Find...**, won't change just part of a word or number unless you ask it to do so. To change part of a word, click the "Find Whole Word" check box so that the X disappears. That will make the **Change...** command look for just part of a word.

The option "Ask Each Time" lets you decide whether to make the change for each occurrence. Choose **Change...**, and type:

Click the "Ask Each Time" box to put an X in it, then click the Change button. True BASIC will highlight the first occurrence of the word "the" in your program and present a new box with the question "Change?".

Click "No" and no change will be made at that place in your program. True BASIC then highlights the next occurrence of the word and you'll have the same chance to decide whether to make the change. Click "Yes" to make that change. You can stop at any point just by clicking "Cancel".



PC Version Change from? and Change to? box

On the PC version of **True BASIC *Free***, the **Change...** is performed by presenting two dialog boxes. In the first you enter the 'Change From' text, click OK and then you will see the Change To dialog box. Enter the text for the change and click OK.

On both the MacOS and the PC versions, True BASIC will confirm how many changes were made.

Extracting Parts or Merging Programs

The next two commands in the **Edit** menu will become useful as you begin to work with larger programs.

If you want to remove all but one section of a program, use the **Keep...**command. Select the part you want to keep and then choose **Keep...** from the **Edit**menu. True BASIC will delete everything in your program **except** the selected text.

The **Include...** command lets you add the contents of another file to your program. Put the insertion point at the place where you wish to add the new file and select **Include...** from the **Edit** menu. You'll get a dialog box where you can specify any existing file in any folder on any disk. True Basic will insert the contents of that file at the insertion point of your current program.

Edit..., Select..., and Move To...

The three remaining commands in the **Edit** menu are also useful as you work with larger programs.

The **Edit...** choice lets you restrict your editing (such as use of **Find...** or **Change...**) to a selected group of lines.

The **Select...** and **Move To...** commands let you select a group of lines or move to a specific place in the program by specifying line numbers or the name of a particular subroutine or function. For example, you can move to the beginning of your program by using **Move To...** and specifying line 1. You can select the first 10 lines of your program by using **Select...** and specifying 1-10. You'll learn more about subroutines and functions in section 15.

For more details on these last two groups of **Edit** menu commands, see the *True BASIC Bible*.

13. Using and Storing Data

So far, you've used the LET and INPUT statements to assign values to variables. These work fine if you have just a few values. The READ and DATA statements described in this chapter let you supply a list of numbers or strings in your program and assign them, one by one, to variables. They always go together: the DATA statement lists all the values, and the READ statement assigns them to variables.

You can also assign values to variables from data stored in files. The latter sections of this chapter describe how you can use text files to store data for use by your programs.

The DATA and READ Statements

Call up the demo program TRIVIA and look at how it uses READ and DATA statements.

```
! Trivia quiz.
!
READ num_quest                          ! Number of questions

FOR i = 1 to num_quest                  ! Read all questions

    READ question$, answer$

    PRINT question$;
    LINE INPUT reply$                   ! Get user's guess

    IF reply$ = answer$ THEN            ! If correct...
       LET right = right + 1            ! Count right replies
       PRINT "Correct."                 ! And say bravo
    ELSE
       PRINT "No, the correct answer is "; answer$; "."
    END IF

NEXT i

PRINT "You got"; 100 * right/num_quest; "% right."

DATA 5

DATA What is the capital of Austria, Vienna
DATA What year did Franklin Pierce take office, 1853
DATA "What is the capital of Manitoba, Canada", Winnipeg
DATA "How many years, on average, does a baboon live", 20
DATA How about a gray squirrel, 5

END
```

The first executable statement after the initial comment lines is a READ statement. This "reads" the first item in the first DATA

statement and assigns that value to the variable *num_quest*. The value of *num_quest* determines how many times the program goes through the FOR-NEXT loop.

The second READ statement is inside the FOR-NEXT loop. It gets the next two values from the list of DATA statements and assigns them to the two variables in the READ statement. *Question$* takes the value "What is the capital of Austria" and *answer$* gets the value "Vienna". The next time through the loop, *question$* and *answer$* take the next two values in the DATA statements, and so on.

Run the program to see how it works. You can give any answers you want; the dialog below is just a sample.

```
What is the capital of Austria? Salzburg
No, the correct answer is Vienna.
What year did Franklin Pierce take office? 1844
No, the correct answer is 1853.
What is the capital of Manitoba, Canada? Winnipeg
Correct.
How many years, on average, does a baboon live? 20
Correct.
How about a gray squirrel? 15
No, the correct answer is 5.
You got 40 % right.
```

DATA statements may be placed anywhere in your program.

You saw that the location of the DATA statements didn't matter when you moved them in the SMOKY program in the last chapter. Often they go at the very end of a program; sometimes it's more convenient to put them right after a READ statement. You may use a separate DATA statement for each item, or use commas to put several items on one statement. True BASIC lumps all the DATA statements in a program together, in order, into one long list of data items. Each time it executes a READ statement, True BASIC reads the next item in the DATA list, regardless of where it appeared in the program.

READ and DATA statements can use either numbers or strings.

You may freely mix strings and numbers in your DATA statements. Just be sure that the variable name type (numeric or string) is reading an appropriate type of data item. You can't

read a string data item into a numeric variable, but you can read a number into a string variable. The TRIVIA program reads some numbers for the string variable *answer$*. This is perfectly legal in True BASIC, as long as you don't try to use that variable to do arithmetic calculations.

You must put double quote marks around string data items that contain commas, or around items that begin or end with spaces.

If you don't use quote marks, True BASIC will assume that any commas are separating data items, and it will ignore any extra spaces before or after the data.

Checking for More Data

The TRIVIA program stores the number of questions in the first item in the DATA statements. The number of questions then controls the FOR-NEXT loop so that it reads the correct number of items. If the program tried to read more items than are contained in the DATA statements, True BASIC would give you an error message.

It is not always convenient to count the number of DATA statement item, however. True BASIC provides a way that you can use a DO loop to check whether there are any more data items available. The SMOKY Demo Program you edited in the last chapter illustrates this method. You haven't learned the PLAY statement yet for performing music, but you should be able to follow the logic of the program.

```
! Plays the beginning of "On Top of Old Smoky".

DO while more data

   READ music$    ! Get the string representations
   PLAY music$    ! And play the notes

LOOP

DATA O4 L4 C C E G O5 L2 C. O4 A.
DATA L4 A F G A L1 G
DATA L4 C C E G L2 G. D.
DATA L4 E F E D L2 C.

END
```

The DO WHILE MORE DATA statement means "keep looping while there are more data items to read". This is why the

program still worked even when you copied and pasted an extra set of the DATA statements.

MORE DATA is true as long as there are more items in the DATA list.

DO WHILE MORE DATA makes it easier to change the amount of data at the end of the program. You never have to count the data items, or remember to change the number saying how many data items there are. After all, the computer should do all this bookkeeping work!

(As a practice exercise, rewrite the TRIVIA program to use a DO WHILE MORE DATA statement instead of the FOR-NEXT loop.)

Besides the MORE DATA condition, True BASIC also has an END DATA condition, which works just the opposite way. END DATA is true if you've run out of data to read. It's probably easiest to use END DATA with a DO UNTIL or LOOP UNTIL statement. For example, you could rewrite the SMOKY program to use a plain DO statement with a LOOP UNTIL END DATA statement.

END DATA is true when there are no more items in the DATA list.

Reusing Data Values

So far, the TRIVIA and SMOKY programs have read each data item once and only once.

True BASIC's RESTORE statement lets you reuse data values that have already been assigned to variables.

After you use a RESTORE statement, True BASIC begins reading again at the first item in the list of DATA statements. The following version of SMOKY uses a RESTORE statement whenever the end of the data is reached. This program also illustrates the END DATA condition which is the opposite of MORE DATA.

```
! Plays the beginning of
! "On Top of Old Smoky".

DO while more data

   READ music$! Get the string representations
   PLAY music$! And play the notes

   IF end data then RESTORE

LOOP

DATA O4 L4 C C E G O5 L2 C. O4 A.
DATA L4 A F G A L1 G
DATA L4 C C E G L2 G. D.
DATA L4 E F E D L2 C.

END
```

Notice that this program now contains an **infinite loop**. The program will never end on its own. First, it will play through to the end of the data. When the last item is read, the IF END DATA condition will then be true and the RESTORE statement will "reset" True BASIC to the beginning of the DATA items. DO WHILE MORE DATA will therefore still be true. Thus, the data will play again, and again be restored after the last item. To stop such a program, you must use the **Stop** command in the **Run** menu.

Notice also, that you may use the END DATA or MORE DATA conditions anywhere that you can use a logical condition. Thus, you can use them in IF-THEN statements as well as on a DO WHILE or DO UNTIL.

You can also combine checks for END DATA or MORE DATA with other conditions using AND or OR. With AND, both conditions must be true. With OR, if just one condition is true then the test is true. Can you figure out how the following version of the TRIVIA program will work?

```
! Trivia quiz.
!
DO

   READ question$, answer$

   PRINT question$;
   LINE INPUT reply$                  ! Get user's guess

   IF reply$ = answer$ THEN           ! If correct...
      LET right = right + 1           ! Count correct replies
      PRINT "Correct."                ! And say bravo
   ELSE
      PRINT "No, the correct answer is "; answer$; "."
```

```
      END IF

      IF end data and right <3 then
         RESTORE
         LET right = 0
      END IF

   LOOP until end data or right >=3

   DATA What is the capital of Austria, Vienna
   DATA What year did Franklin Pierce take office, 1853
   DATA "What is the capital of Manitoba, Canada", Winnipeg
   DATA "How many years, on average, does a baboon live", 20
   DATA How about a gray squirrel, 5

   END
```

Storing Data in Files

True BASIC also lets you write and read data to and from other files. A **file** is a unit of information saved on a disk in your computer. Files may contain text, data, or programs; each of the True BASIC programs you've been creating are saved in separate files. Because files continue to exist after your program stops and even after you turn off your computer, they serve as long-term storage. There are several advantages to storing your data in one or more files separate from the file containing your program:

• It is easier to create and maintain a large amount of data in a separate file. You need no DATA statements, and your data takes no space in your program.

• You can run a program with several different sets of data (each stored in a different file), or have one set of data that can be used by several programs.

• A program can change or make additions to data stored in files. You can store results for use in later program runs.

True BASIC programs can read and write to three kinds of files: text, record, and byte files. Here, we'll look at just text files as these are the easiest to create and understand.

A **text file** contains lines that True BASIC can display on the screen. You can create text-file lines at the keyboard using True BASIC's screen editor or by printing output from a True BASIC program to a file. All of the True BASIC programs you've been looking at are actually text files.

Reading Data From Text Files

The demo program TRIVIA2 is a version of the Trivia quiz that gets its data from the text file TRIVDATA. Open the TRIVIA2 program and notice how it differs from the versions you've seen so far:

```
! Trivia quiz -- reads data from a file.
!
OPEN #1: name "TrivData"          ! Open file as channel #1

DO
    INPUT #1: question$, answer$! Get data from channel #1
    LET total = total + 1         ! Count the questions
    PRINT question$;
    LINE INPUT reply$             ! Get user's guess

    IF reply$ = answer$ THEN      ! If correct...
       LET right = right + 1      ! Count right replies
       PRINT "Correct."           ! And say bravo
    ELSE
       PRINT "No, the correct answer is "; answer$; "."
    END IF

LOOP until end #1

PRINT "All done.  You answered"; right; "out of"; total;
PRINT "questions correctly."

CLOSE #1                          ! Close the file

END
```

The OPEN statement "opens a channel" to the file TRIVDATA. This channel, #1 in this case, then serves as a shorthand name for the file you have opened. (This is similar to the way you "open a channel" to the printer as seen in section 11. The PRINTER and NAME keywords tell True BASIC what you want. By using different channel numbers, you can open a printer and one or more files at the same time.)

??? Difference in writing OPEN and CLOSE path names between MacOS and PC versions

The INPUT #1: statement looks at the opened file for input rather than asking for it at the keyboard. The LOOP UNTIL END #1 statement works as does LOOP UNTIL END DATA, but it looks for data in the opened file rather than in DATA statements within the program. You may also use MORE #1 wherever you might use a MORE DATA statement.

Similarly, if you add the statement:

```
IF end #1 then RESET #1: begin
```

just before the LOOP statement, the program will run continuously using the TRIVDATA questions over and over again. In that case, you would have to use the **Stop** command in the **Run** menu to stop the program.

The CLOSE #1 statement closes the channel to the file. Although True BASIC automatically closes any open files at the end of a program, it's a good idea to close a channel when you no longer need it.

The TRIVDATA file must contain the data just as you would type it on the keyboard in response to an INPUT statement. The INPUT #1 statement asks for two input items. Look at TRIVDATA and you'll see that each line (except the last two) contains two input items separated by a comma.

```
Which part of a lemon provides the zest, skin
What is a German motorway or freeway called, Autobahn
Which is the most populous country in the world, China
What year did the SS Titanic sink, 1912
What is the largest snake in South America, Anaconda
What shape does a honeybee make its cell, hexagonal
What is the main power source for orbiting research
satellites,solar
```

The data-file lines must exactly match the INPUT requests as the program cannot "re-ask" a file for input.

If there are too few or too many items, or the types do not match, your program will stop with an error. If you can't fit all required input items on one line (as with the last question), you can end a line with a comma to indicate that another input item follows on the next line.

Use the arrow keys to move to the end of the TRIVDATA file and you'll see that the last line of data is the last line of the file. There is no extra return or linefeed at the end of the file. (If a data file ends with a blank line, you may receive an error message such as "Too few input items" when True BASIC expects more data but finds no input items on the line.)

You may also use the LINE INPUT, MAT INPUT, and MAT LINE INPUT statements to read from text files. LINE INPUT is, in fact, the best statement to use with strings that might have commas or quotes in them; see the section "Using LINE INPUT with String Data in Text Files" below. Just be sure that the data in the file matches the appropriate format for the input statement or state-

ments in the program. (The MAT statements read from arrays and are explained in the next chapter.)

Creating Text Files

You may use True BASIC's screen editor to enter data into a text file. Create a new file as if you were creating a new program, and then type in your data in the proper format. Do not use any DATA statements — and of course no line numbers!

You can also create data files with any application (such as a word processor, spreadsheet, or database program) that lets you save text-only files. Check the instructions for your application to learn how to save such files and how to put commas between data items if necessary.

For practice, create an alternative set of questions for the TRIVIA2 program. You can then edit TRIVIA2 to open your new data file, or you can modify the program to ask you what file to use for input:

```
INPUT PROMPT "What file contains the questions? ": filename$
OPEN #1: name filename$
```

True BASIC programs can also create text files and put data in them, as described in the next section.

Printing String Data to Text Files

Just as you can open a channel to a printer and then PRINT to the printer instead of the screen (see section 11), you can open a channel to a file and PRINT to that file. You can easily adapt any program you've written so far to send output to a file rather than to the screen or printer:

```
OPEN #1: name "outfile"        ! Opens channel #1 to a file
ERASE #1! Make sure file is empty
FOR i = 1 to 10
    PRINT #1: i                ! Print to #1 file
NEXT i
CLOSE #1! Close the file
END
```

Simply opening a file and replacing your PRINT statements with PRINT #1 statements works fine if you merely want to save your output — perhaps for later listing on a printer. However, if you are storing data for future use by a program, you must plan ahead.

If you want to print data to a file for later use by a program, you must put the data into the file in a format appropriate for input.

Consider the following variation on TRIVIA2.

```
! Trivia quiz -- reads data from a file.
!
INPUT PROMPT "File containing the questions? ": filein$
OPEN #1: name filein$

INPUT PROMPT "File to store missed questions? ": fileout$
OPEN #2: name fileout$, create newold  ! Open or create 2nd file
RESET #2: end                          ! Move to end of 2nd file

DO
    INPUT #1: question$, answer$  ! Get data from channel #1
    LET total = total + 1         ! Count the questions
    PRINT question$;
    LINE INPUT reply$             ! Get user's guess

    IF reply$ = answer$ THEN      ! If correct...
       LET right = right + 1      ! Count correct replies
       PRINT "Correct."           ! And say bravo
    ELSE
       PRINT "No, the correct answer is "; answer$; "."
       PRINT #2: question$; ","; answer$   ! Save question
    END IF

LOOP until end #1

PRINT "All done.  You answered"; right; "out of"; total;
PRINT "questions correctly."

CLOSE #1                           ! Close the files
CLOSE #2

END
```

This program opens a second file and prints to it each missed question along with the correct response. Notice that the PRINT #2 statement also prints the comma that must separate these two items if you later wish to use the file for input.

The CREATE NEWOLD keywords on the second OPEN statement tell True BASIC to create a new file if it can't find one with the specified name.

The RESET #2: END statement tells True BASIC to move to the end of the second file. True BASIC is always "looking" at the beginning of a newly opened file, which is fine if you are using the file for input or if the file is empty. But True BASIC can print only to the end of existing text files, so you must either erase the file or move to the end before you can PRINT. (If the file is empty; the RESET statement has no effect.)

If you want to PRINT to a text file that is not empty, you must first ERASE the file or RESET to the END of the file.

Make these changes to the TRIVIA2 program and try it out.

Reusing Stored Data for Input

Each time you run the above program, it adds any missed questions to the end of the #2 file — your "output file". If you send output to the same file for several runs of the program, it may eventually contain a long list of questions.

You could later use those saved questions to quiz yourself again because the questions and answers were printed to the file in a proper format for input. For example, assume you ran the program with TRIVDATA as the source of the questions and a file called REQUIZ for the missed questions. You could then run the program again, naming REQUIZ as the source of questions and giving a new file name to receive the missed questions.

Note! Do not open the same file for both channel #1 and #2. This is rarely, if ever, desirable, and with the TRIVIA program as written above, you'll get an error message if you attempt to do so. This is because True BASIC normally opens a file with "permission" to read from it and write to it, and one file can give only one "write permission" at a time. For more information on file permissions, see the *True BASIC Bible*.

Using LINE INPUT with String Data in Text Files

Look at the following question, which you might want to add to a data file read by the TRIVIA2 program:

```
Who wrote 20,000 Leagues Beneath the Sea, Jules Verne
```

As written above, this line would produce the error message "Too many input items." True BASIC would interpret the comma in 20,000 as marking the end of the first input item. You can place such an input string in double quotes to indicate that the comma is part of the string:

```
"Who wrote 20,000 Leagues Beneath the Sea", Jules Verne
```

But what if you want to place the title "20,000 Leagues Beneath the Sea" in quotes? You would have to use single quotes for the title, or you could repeat the double quotes where you want True BASIC to see them as quotes and not as markers for the end of the string:

```
"Who wrote '20,000 Leagues Beneath the Sea'", Jules Verne
```

or

```
"Who wrote ""20,000 Leagues Beneath the Sea""", Jules Verne
```

Although you can add quotes as necessary if you create the data file yourself, you could easily make mistakes. And it becomes even more complex if you want your program to PRINT such strings to a file for later use as input!

The LINE INPUT statement provides a much "cleaner" way to use strings for input to text files. To "fix" the TRIVIA2 program, first place the questions and answers on different lines in your data file. For example:

```
Who wrote "20,000 Leagues Beneath the Sea"
Jules Verne
What name is given to burnt sugar used as flavoring
caramel
```

You can then easily change the TRIVIA2 program to read a complete input line for each variable, regardless of punctuation:

```
LINE INPUT #1: question$, answer$
```

And, you can very easily PRINT strings to a file that could later be used for input:

```
PRINT #2: question$
PRINT #2: answer$
```

These two PRINT statements put each string on a separate line in file #2.

Printing Numeric Data to Text Files

The demo program BALANCE shows how you can send both numeric and string data to a file and then reuse the data in that file when the program is run again:

```
! Check balance program - keeps current data in a text file

! Open the data file and get existing values, if any
OPEN #1: name "CHKDATA", create newold

! If file contains data, get it & report current amounts
IF more #1 then
   LINE INPUT #1: bal_date$
   INPUT #1: curbal, lastcheck_amt, lastdep_amt
   PRINT "As of "; bal_date$; ", your balance was $";curbal
   PRINT "Your last check was $";lastcheck_amt
   PRINT "Your last deposit was $";lastdep_amt
   PRINT
   PRINT "Input all checks and deposits since ";bal_date$
ELSE
   PRINT "Input all checks and deposits."
END IF
```

```
! Get new transactions

PRINT "Enter one per line: use - for checks, + for deposits"
PRINT "Enter 0 (zero) when done"

DO ! Get new transactions
   INPUT amount
   LET curbal = curbal + amount ! Update balance
   IF amount < 0 THEN
      LET lastcheck_amt = amount*(-1)
   ELSE IF amount > 0 THEN
      LET lastdep_amt = amount
   END IF
LOOP UNTIL amount = 0

LINE INPUT PROMPT "Date of last transaction: ": bal_date$
PRINT "Your current balance is $";curbal

! Clear data file and enter new amounts
ERASE #1! Remove any existing data
PRINT #1: bal_date$
PRINT #1: curbal; ","; lastcheck_amt; ","; lastdep_amt
CLOSE #1

END
```

This program uses a single data file CHKDATA. The program first reads the current values (if any) from the file to variables used by the program. After it calculates all new transactions, the program erases the data file and prints the new information to it. Thus, you could use CHKDATA again and again, and you will always be working with the most recent information about your bank balance.

If you run this program and then open the CHKDATA file, you'll see the data as follows:

```
July 4, 1993
460.93 , 436.5 , 1000
```

Notice that the program prints commas between the three numeric data items to match the INPUT statement. It prints the string *bal_date$* to a line by itself and uses a LINE INPUT statement to read that line. This avoids the problem that a comma within the date would cause with an INPUT statement.

More About File Input and Output

When a True BASIC program opens a text file, the program is normally "looking" at the beginning of the file. The first input statement reads the first line of data, the second input statement reads the second line, and so on. You can re-use the data in a text file by using a RESET statement:

```
RESET #1: begin
```

A True BASIC program can print only to the end of a text file. You must move to the end of the file by first reading all the data, by erasing the file, or by using a RESET statement:

```
RESET #1: end
```

Record files let you move around within a file more easily, and the True BASIC language provides additional statements for use with files. For more information about all three of True BASIC's data-file types and the statements used with them, see the *True BASIC Bible*.

14. Arrays and Matrices

Problems often arise that would require an unreasonable number of variables to solve. Open the demo program INVNTORY, which keeps the inventory of a hardware store:

```
! Inventory for 5 items.
!
READ item1$, number1
READ item2$, number2
READ item3$, number3
READ item4$, number4
READ item5$, number5

PRINT "You have these items:"
PRINT item1$, item2$, item3$, item4$, item5$
PRINT number1, number2, number3, number4, number5

DATA hammers, 4, umbrellas, 2, wood stoves, 1
DATA bags of salt, 4, pliers, 2
END
```

Imagine how much trouble it would be to change this program to handle thirteen items! Now consider that a large store might have thousands of different items in stock. Clearly, you need a better way of handling many similar values.

One-Dimensional Arrays

This problem calls for array variables. An **array** is a variable that can hold several different values at once. You could think of it as a list of items. You identify each item with the name of the list and the item's position in the list.

Rewrite the INVNTORY program to use two arrays, *item$* and *number* as shown below:

```
! Inventory with arrays
!
DIM item$(5), number(5)

FOR i = 1 to 5
    READ item$(i), number(i)
NEXT i

PRINT "You have these items:"
FOR i = 1 to 5
    PRINT item$(i), number(i)
NEXT i

DATA hammers, 4, umbrellas, 2, wood stoves, 1
DATA bags of salt, 4, pliers, 2
END
```

When you run this program, you get the following output:

```
You have these items:
hammers          4
umbrellas        2
wood stoves      1
bags of salt     4
pliers           2
```

The table below illustrates the two arrays *item$* and *number*. The DIM statement declares that the variables are arrays and sets their size; each array can hold five different values. (DIM is short for "dimension," as it fixes an array's dimensions.)

item$

hammers	umbrellas	wood stoves	bags of salt	pliers
item$(1)	item$(2)	item$(3)	item$(4)	item$(5)

number

4	2	1	4	2
number(1)	number(2)	number(3)	number(4)	number(5)

You must name every array in a DIM statement before you can use it in the program.

The ten individual values within *item$* and *number* are the **elements** of the arrays. The elements of *item$* are strings, and the elements of *number* are numbers. The name of a string array must end in a dollar sign, just like the name of a regular string variable. You cannot mix numbers and strings in a single array.

Array Subscripts

The numbers used to identify a particular element of an array are **subscripts**. Subscripts must be enclosed in parentheses () after the array name. The elements of *item$* and *number* automatically use subscripts from 1 to 5 because the DIM statement set the size of the arrays to 5.

Each time through the FOR-NEXT loops, True BASIC reads and prints different elements of *item$* and *number*. The first time through the loop, *i* equals 1, so the program reads and prints *item$(1)* and *number(1)*. The second time through, *i* equals 2, so the program reads and prints *item$(2)* and *number(2)*, and so on. (You describe elements in an array as "*item-dollar-sub-one*" or "*number-sub-two*.)

You can use the elements in an array in any order. For example, you could change the second FOR statement to print the elements in reverse order.

```
! Inventory with arrays
!
DIM item$(5), number(5)

FOR i = 1 to 5
    READ item$(i), number(i)
NEXT i

PRINT "You have these items:"
FOR i = 5 to 1 step -1
    PRINT item$(i), number(i)
NEXT i

DATA hammers, 4, umbrellas, 2, wood stoves, 1
DATA bags of salt, 4, pliers, 2
END
```

The program will print the items in reverse order:

```
You have these items:
pliers           2
bags of salt     4
wood stoves      1
umbrellas        2
hammers          4
```

Array Bounds

In the INVNTORY program, *item$* and *number* both have five elements, numbered from 1 to 5. In True BASIC, however, you can use any numbers as the **lower bound** and **upper bound** for the array. That is, instead of having a lower bound of 1, the array could have a lower bound of 1991. Instead of having an upper bound of 5, you might use 1995. You still have an array with five elements, but with different bounds.

You may want to adjust array bounds to make a particular problem easier to solve. The following program shows how you could read and compare census figures for a couple of towns:

```
! View census figures
!
DIM springfield(1985 to 1990), woodsville(1985 to 1990)

FOR y = 1985 to 1990
    READ springfield(y), woodsville(y)
NEXT y

INPUT PROMPT "What year are you interested in? ": year
IF springfield(year) > woodsville(year) then
```

```
   LET town$ = "Springfield"
ELSE
   LET town$ = "Woodsville"
END IF
PRINT "In"; year; town$; " had the largest population."

DATA 17635, 16413, 17986, 16920, 18022, 17489
DATA 18130, 17983, 18212, 18433, 18371, 18778
END
```

A sample run produces output such as:

```
What year are you interested in? 1987
In 1987 Springfield had the largest population.
```

The DIM statement declares bounds from 1985 to 1990 for the arrays *springfield* and *woodsville*, so each array has six elements.

You may use any numbers you wish for an array's upper and lower bounds. For example, to keep track of Centigrade temperatures in the northern United States or Canada, you might want to dimension an array such as *temp(-40 to 40)*. This array has 81 elements.

Naturally, as your arrays get bigger, they take more computer memory to store. True BASIC limits the size of your arrays only to what will fit in your computer's memory.

Arrays of Two or More Dimensions

So far, you've seen only "one-dimensional" arrays. These arrays require only one number as subscript. But True BASIC lets you have arrays with 2, 3, 4, or almost any number of dimensions. (The maximum number of dimensions is 255.)

Typically, you would use a **two-dimensional array** when you have two different sets of strongly related values. Open the Demo Program STATES, which plays a trivia quiz with state capitals, and run it.

```
! State capital quiz.
!
RANDOMIZE
DIM state$(50,2)                  ! 50 states, 2 items per state

FOR i = 1 to 50
    READ state$(i,1)               ! Read state name
    READ state$(i,2)               ! And capital
NEXT i

FOR i = 1 TO 10! Ask 10 questions
    LET n = Int(50*Rnd) + 1     ! Pick a number between 1 and 50
    PRINT "The capital of "; state$(n,1); " is";
    LINE INPUT capital$     ! Get the reply
```

```
    IF Lcase$(capital$) = Lcase$(state$(n,2)) THEN
       PRINT "RIGHT!"
    ELSE
       PRINT "Nope, it's "; state$(n,2); "."
    END IF
NEXT i

DATA Alabama,Montgomery, Alaska,Juneau, Arizona,Phoenix
DATA Arkansas,Little Rock, California,Sacramento
DATA Colorado,Denver, Connecticut,Hartford, Delaware,Dover
DATA Florida,Tallahassee, Georgia,Atlanta, Hawaii,Honolulu
DATA Idaho,Boise, Illinois,Springfield, Indiana,Indianapolis
DATA Iowa,Des Moines, Kansas,Topeka, Kentucky,Frankfort
DATA Louisiana,Baton Rouge, Maine,Augusta, Maryland,Annapolis
DATA Massachusetts,Boston, Michigan,Lansing
DATA Minnesota,St. Paul, Mississippi,Jackson
DATA Missouri,Jefferson City, Montana,Helena
DATA Nebraska,Lincoln, Nevada,Carson City
DATA New Hampshire,Concord, New Jersey,Trenton
DATA New Mexico,Santa Fe, New York,Albany
DATA North Carolina,Raleigh, North Dakota,Bismarck
DATA Ohio,Columbus, Oklahoma,Oklahoma City, Oregon,Salem
DATA Pennsylvania,Harrisburg, Rhode Island,Providence
DATA South Carolina,Columbia, South Dakota,Pierre
DATA Tennessee,Nashville, Texas,Austin, Utah,Salt Lake City
DATA Vermont,Montpelier, Virginia,Richmond, Washington,Olympia
DATA West Virginia,Charleston, Wisconsin,Madison
DATA Wyoming,Cheyenne
END
```

(Note: This program uses the LCASE$ built-in function to convert all answers to lowercase for comparison since upper and lowercase letters are not equal. The next chapter explains the use of functions.)

A good way to visualize a two-dimensional array is as a table with rows and columns. In the STATES program *state$(50,2)* has 50 rows corresponding to the 50 states, and 2 columns corresponding to the two items for each state. The state name is in the first column and the state capital is in the second column.

state$

state$(1,1)	Alabama	Montgomery	state$(1,2)
state$(2,1)	Alaska	Juneau	state$(2,2)
state$(3,1)	Arizona	Phoenix	state$(3,2)
state$(4,1)	Arkansas	Little Rock	state$(4,2)
state$(5,1)	California	Sacramento	state$(5,2)

A Two-dimensional Array

The MAT Statements

The sample programs you've seen so far have used FOR-NEXT loops to READ each value into an array or to PRINT each value of an array. True BASIC has several MAT statements that let you do something for a whole array in one statement. The keyword MAT is short for **matrix** which is another word for a two-dimensional array. However, you may use MAT statements with arrays of any dimension.

The **MAT READ statement** lets you read an entire array in one statement. For example, you could remove the FOR loop from the revised INVNTORY program and substitute a MAT READ statement. Notice that you must also edit the DATA statements!

```
! Inventory with arrays
!
DIM item$(5), number(5)

MAT READ item$, number

PRINT "You have these items:"
FOR i = 5 to 1 step -1
    PRINT item$(i), number(i)
NEXT i

DATA hammers, umbrellas, wood stoves, bags of
 salt, pliers
DATA 4, 2, 1, 4, 2
END
```

The MAT keyword tells True BASIC to read the entire array, so you don't put anything in parentheses after the array name.

MAT READ fills the first array named before reading to any other arrays named in the statement.

You must therefore edit the DATA statements to put all the values for *item$* first, followed by all the values for *number*. If you don't, you'll get the error message "Data item isn't a number" when the program tries to read a string item into an element of *number*. (Remember that True BASIC lets you read a number as a string, but cannot accept anything but numeric constants for numeric items.)

The **MAT PRINT statement** lets you print out the contents of an array with a single statement. You could replace the remaining FOR loop from the INVNTORY program:

```
! Inventory with arrays
!
DIM item$(5), number(5)

MAT READ item$, number

PRINT "You have these items:"
MAT PRINT item$, number

DATA hammers, umbrellas, wood stoves, bags of
  salt, pliers
DATA 4, 2, 1, 4, 2
END
```

The output will be different from the previous version, because MAT PRINT prints all the elements of *item$* and leaves a blank line before it prints the elements of *number*. Commas and semicolons in MAT PRINT statements have the same effect as in regular PRINT statements.

```
You have these items:
hammers        umbrellas     wood stoves   bags of salt   pliers

 4              2             1             4              2
```

True BASIC prints arrays of two or more dimensions in similar fashion, except that it moves to a new line for each new dimension printed. For example, a *MAT PRINT state$* statement in the STATES quiz would begin a new line after of each row of two items:

```
Alabama          Montgomery
Alaska           Juneau
Arizona          Phoenix
. . . (etc.)
```

MAT INPUT and **MAT LINE INPUT** let you input a whole array in one statement. For example:

```
DIM expense(1980 to 1989)
PRINT "Please enter the 10 expense items"
MAT INPUT expense
```

For more information on using these statements see the *True BASIC Bible*.

Advanced Work with Arrays and Matrices

As your programming skills increase, you may wish to explore further about how you can use arrays in True BASIC. This section gives you a quick introduction to some of these features. For more information, see the *True BASIC Bible*.

You can **redimension** arrays as a program is running. You can't actually change the number of dimensions, but you can change the bounds or sizes of the dimensions of an array. This lets you write flexible programs that can adjust array sizes to different sets of data. Both the MAT INPUT and MAT READ statements have versions that let you change the size of an array to fit the number of items available. You can also change the size of an array with the MAT REDIM statement. True BASIC also has built-in functions to let the program figure out the current size or upper and lower bounds of any array. (The next chapter introduces built-in functions; Appendix C lists most of True BASIC's built-in functions.)

You can make **matrix assignments** with the simple **MAT statement**. You can assign the same value to every element in an array:

```
MAT initial = 10
```

You can also assign one array to another as long as they have the same number of dimensions. The array being assigned to adjusts its size to match the other array. In the following statements, the question mark (?) with the MAT INPUT statement adjusts the size of the array *scores* to equal the number of items entered. The following MAT statement assigns the same values to the array *initial* and adjusts the size of *initial* so that it matches *scores*.

```
DIM initial(100), scores(100)
MAT INPUT scores(?)              ! input any number of items
MAT initial = scores             ! arrays are equal & same size
```

True BASIC's **matrix arithmetic** lets you add, subtract, and multiply arrays. For addition or subtraction, two arrays must have the same size and shape. To multiply two arrays, the number of columns in the first array must equal the number of rows in the second. You can also multiple an array by a single number.

For more information about using these array features, see the *True BASIC Bible*.

15. Functions and Subroutines

As your programs get bigger and bigger, you'll find them easier to read and "debug" if you have them segmented into smaller parts. True BASIC's subroutines and functions offer you ways to break down your programs into logical units.

Subroutines

Call up the demo program CRAPS, which introduced the SELECT CASE structure Chapter 10. Notice that the four lines that simulate the dice roll (three LETs and one PRINT) appear twice in the program. The first time is right after the FOR statement, and the second is right after the DO statement.

```
! Craps game.
!
RANDOMIZE

FOR game = 1 to 10              ! Play 10 games

   LET die1 = Int(6*Rnd + 1)    ! Roll 1 die
   LET die2 = Int(6*Rnd + 1)    ! And the other
   LET dice = die1 + die2       ! Sum of two dice

   PRINT dice;                  ! Print this roll

   SELECT CASE dice             ! Branch on roll
      CASE 2, 3, 12             ! dice = 2, 3, or 12
         PRINT "You lose."

      CASE 7, 11                ! dice = 7 or 11
         PRINT "You win."

      CASE ELSE                 ! Anything else
         LET POINT = dice       ! Remember that roll
         DO
            LET die1 = Int(6*Rnd + 1) ! Roll again
            LET die2 = Int(6*Rnd + 1) ! Both dice
            LET dice = die1 + die2
            PRINT dice;         ! Print this roll
         LOOP until dice = 7 or dice = point

         IF dice=point then PRINT "You win" else PRINT "You lose"
   END SELECT

NEXT game

END
```

You can rewrite this program to use a **subroutine**. Move one set of the dice-rolling lines (the three LETs and one PRINT) to the beginning of the program following RANDOMIZE, and remove the other set. Add SUB and END SUB statements to define the group of statements as a subroutine. Insert CALL statements where you want to use the subroutine:

```
! Craps game with subroutine for rolling the dice.
!
RANDOMIZE

SUB Rolldice
   LET die1 = Int(6*Rnd + 1)       ! Roll 1 die
   LET die2 = Int(6*Rnd + 1)       ! And the other
   LET dice = die1 + die2          ! Sum of two dice

   PRINT dice;                     ! Print this roll
END SUB
FOR game = 1 to 10                 ! Play 10 games

   CALL Rolldice                   ! Subroutine rolls dice

   SELECT CASE dice                ! Branch on roll

      CASE 2, 3, 12                ! dice = 2, 3, or 12
         PRINT "You lose."

      CASE 7, 11                   ! dice = 7 or 11
         PRINT "You win."
      CASE ELSE                    ! Anything else
         LET POINT = dice          ! Remember that roll
         DO
            CALL Rolldice          ! Roll again
         LOOP until dice = 7 or dice = point

         IF dice=point then PRINT "You win" else PRINT "You lose"
   END SELECT

NEXT game

END
```

True BASIC skips around the subroutine when you run the program. The statements in the subroutine are used only when a CALL statement in the main part of the program (the main program) "calls" that subroutine name. At the END SUB statement, True BASIC returns to the line following the CALL statement.

When True BASIC returns to the CALL statement in the main program in the above example, the variable *dice* has the new value assigned by the subroutine. Thus the SELECT CASE or

LOOP UNTIL statements **share** the variable *dice* with the subroutine in this program.

Run this edited version of CRAPS and you should find that it works just as before.

Subroutines with Parameters

Subroutines let you write general purpose "tools" that you can use anywhere in your programs. You can use the subroutine from CRAPS any time you want to simulate the rolling of two dice. However, in this version of the subroutine, you have to refer to the result by the same variable name that the subroutine uses (in this case, *dice*).

To make subroutines more general and more helpful to you, you can use **parameters** in your SUB statements and **arguments** in your corresponding CALL statements. To illustrate, rewrite the subroutine *Rolldice* so that it can simulate the rolling of any given number of dice:

```
SUB Rolldice (sum_dice, num_dice)
   LET sum_dice = 0                          ! Initialize
   FOR i = 1 to num_dice
       LET roll = Int(6*Rnd + 1)             ! Roll a die
       LET sum_dice = sum_dice + roll        ! Add to sum
   NEXT i
   PRINT sum_dice;                           ! Print this roll
END SUB
```

You're now using two parameters in the SUB statement above. *Sum_dice* represents the sum of the rolls, and *num_dice* gives the number of dice rolled. The subroutine doesn't change *num_dice* but it does change *sum_dice*.

To use this new subroutine, you must also use two arguments in the CALL statement. For example:

```
CALL Rolldice (dice, 2)
```

The first argument, *dice*, is the main program's name for the sum of dice rolls, and 2 is the number of dice to be thrown.

Arguments share values with their corresponding parameters when the subroutine runs.

Dice and *sum_dice* temporarily become equivalent so that when True BASIC returns to the main program *dice* has the value of *sum_dice*. Similarly, *num_dice* has the value of 2 during this call to the subroutine.

This subroutine illustrates two kinds of parameters:

• *Num_dice* is an **input parameter** that is only for sending information into a subroutine. Since an input parameter returns nothing, you may use constants for the corresponding argument on CALL statements as in the example above.

• **Output parameters** are variables whose values are changed by the subroutine. They send information out from the subroutine to the corresponding argument in the main part of the program. *Sum_dice* is an output parameter.

Built-in Functions

You've already seen several of True BASIC's **built-in functions**: RND, INT, SQR, and LCASE$, for example. Appendix C lists most of True BASIC's built-in functions.

To use a built-in function, all you do is refer to the function by name (perhaps giving it some information such as the number whose square root you want). True BASIC then "returns" a value to the program (such as the square root of the number you used with the function.) In the following short example, *answer* acquires the value 3.16228, which is returned by the function SQR.

```
LET answer = Sqr(10)
PRINT answer
END
```

You can think of a **function** as a machine that takes some numbers or strings as input, and produces one number or string as output. Functions differ from subroutines in that

• functions can return only one value and

• functions cannot change the values of any parameters sent to them.

Now you'll see how to define your own functions and use them to break your programs into logical units.

One-line Functions

One-line functions are the simplest kind of function. You can simulate the rolling of one die as a one-line function. Here's the CRAPS program again, rewritten to use a function *Rolldie*.

```
! Craps game with one-line function for rolling one die.
!
RANDOMIZE

DEF Rolldie = Int(6*Rnd + 1)        ! Roll 1 die
```

```
FOR game = 1 to 10! Play 10 games

   LET dice = Rolldie + Rolldie     ! Rolldie function twice

   SELECT CASE dice                 ! Branch on roll
      CASE 2, 3, 12                 ! dice = 2, 3, or 12
         PRINT "You lose."
      CASE 7, 11! dice = 7 or 11
         PRINT "You win."
      CASE ELSE! Anything else
         LET POINT = dice           ! Remember that roll
         DO
            LET dice = Rolldie + Rolldie  ! Roll again
         LOOP until dice = 7 or dice = point
         IF dice=point then PRINT "You win" else PRINT "You lose"
   END SELECT

NEXT game

END
```

Once you have defined a function in a **DEF statement**, you use that function simply by using its name where you would a variable. True BASIC carries out the instructions in the DEF statement and the resulting value is "returned" to the function name.

You must define a function before you use it in your program.

If you don't define it first, True BASIC won't know that you are referring to a function and not a variable when you use the function name.

Multi-line Functions

You can also write **multi-line functions** to solve problems that require several lines of True BASIC statements. DEF and END DEF statements define a multi-line function. As with one-line functions, you must define your multi-line functions before you use them.

The SGN function is a multi-line function already built into True BASIC. SGN returns the sign of a number. That is, you give it a single number as an argument, and it returns:

-1	if the number is negative
0	if the number equals 0
+1	if the number is positive

You could easily define a SGN function yourself and test it as follows:

```
! Define the Sgn function
!
DEF Sgn(x)
    SELECT CASE x
    CASE is < 0! If negative . . .
         LET Sgn = -1 ! . . .return -1
    CASE 0 ! If zero . . .
         LET Sgn = 0  ! . . .return a 0
    CASE else! Otherwise must be positive
         LET Sgn = +1 ! . . .return +1
    END SELECT
END DEF

INPUT n ! Input a number
PRINT Sgn(n)! Print its sign
PRINT Sgn(3-5*2)! And the sign of this formula
END
If you run this program and give 35 as input, you will see the
following results:
? 35
1
-1
```

Inside the definition of *Sgn*, the program selects one of three cases depending upon the sign of the parameter and assigns a value to *Sgn*. At the END DEF line, the function actually produces its output value, which is whatever value was assigned during the execution of the function. (If no value is assigned, then 0 is returned.)

Global Variables

You've seen how you can pass variables as parameters to subroutines and functions, but what about other variables used within a subroutine or function definition? They, too, are shared with the rest of the program. Such variables shared by two parts of a program are **global variables**.

Global variables are sometimes useful, but often they are a source of hard-to-spot program bugs. Consider the example in the Demo Programs folder – BUG:

```
! An insidious bug
!
DEF XXX$(n)                          ! Return a string of n X's
    LET s$ = ""                      ! Start with an empty string
    FOR i = 1 to n                   ! Loop. . .
        LET s$ = s$ & "x"            ! . . . adding an X each time
    NEXT i
    LET XXX$ = s$
END DEF

FOR i = 1 to 4                       ! Ask four times
    PRINT "How many X's";
    INPUT n
    PRINT XXX$(n)
```

```
NEXT i
END
```

When you run this program and give an input of 10, you would see the following:

```
How many X's? 10
xxxxxxxxxx
```

What happened? This program should ask for input four times and draw four sets of X's. The problem is that two different parts of the program are using the variable *i*, and one part is causing trouble for the other. Follow the program step by step:

• First, the function definition is created but not used.

• The FOR-NEXT loop that asks for input four times begins and *i* takes the value 1. The program asks "How many X's?" and you reply "10". The program calls the function XXX$ with 10 as its argument; in other words, *XXX$* should return a string of ten x's. So far, so good.

• Within the *XXX$* definition, *s$* starts as an empty string. Then a FOR-NEXT loop adds an "x" to the value of *s$* 10 times. After 10 times through the loop, *i* equals 11 so the loop stops. The program assigns the value of *s$* to *XXX$* and returns to the main program where it prints that returned value ("xxxxxxxxxx"). That looks OK.

• The program moves on to the NEXT i statement where it increases the value of *i* by one. Here is the problem! At the end of the function, *i* is 11 and that value is shared with the main program. After the NEXT i statement in the main program, *i* equals 12! The FOR-NEXT loop in the main program never runs again and the program ends.

The function uses two variables that are not parameters: *s$* and *i*. This is a dangerous situation, since some other part of the program might use either variable as happens in this example.

Bugs of this sort are very typical when you use **global**, or shared, variables within a function or subroutine. You may be more likely to avoid this kind of error if you keep all the statements that use a certain variable within a few lines of each other. In True BASIC, you may also escape these pitfalls by using **external subroutines** and **external functions** or by declaring variables in a LOCAL statement.

External Subroutines and Functions

External subroutines and functions are like internal ones, but with two important differences.

• They are all defined after the END statement. They are outside the **main program**.

• All their variables are **local** to the function or subroutine definition. Except for parameters, no variables share values with the main program, even if they have the same names.

To see how this works, you can rewrite the "buggy" example from the previous section.

```
! Using an external function

DECLARE DEF XXX$

FOR i = 1 to 4                    ! Ask four times
    PRINT "How many X's";
    INPUT n
    PRINT XXX$(n)
NEXT i
END

! XXX$ -- returns n x's

DEF XXX$(n)                       ! Return a string of n X's
    LET s$ = ""                   ! Start with an empty string
    FOR i = 1 to n! Loop. . .
        LET s$ = s$ & "x"         ! . . . adding an X each time
    NEXT i
    LET XXX$ = s$
END DEF
```

When you run this version of the program, you'll find that it now correctly asks for x's four times:

```
How many X's? 10
xxxxxxxxxx
How many X's? 4
xxxx
How many X's? 7
xxxxxxx
How many X's? 2
xx
```

You must add one new statement when you use an external function. The **DECLARE DEF statement** tells True BASIC that XXX$ is a function and not a variable or an array.

The DECLARE DEF statement must appear before an external function is used.

You need only give the function's name in a DECLARE DEF statement; you do not have to list parameters or even say how many there are.

External subroutines go after the END statement, just like external functions. However, because you use subroutine names only in a CALL statement, you do not have to declare them with a DECLARE SUB statement. True BASIC knows that anything in a CALL statement is a subroutine.

The LOCAL Statement

If you name variables in a LOCAL statement within a subroutine or function, those variables will not share values with the main program. Here is the XXX$ function from the BUG program written with a local statement:

```
DEF XXX$(n)                    ! Return a string of n X's
    LOCAL i, s$
    LET s$ = ""                ! Start with an empty string
    FOR i = 1 to n             ! Loop. . .
        LET s$ = s$ & "x"      ! . . . adding an X each time
    NEXT i
    LET XXX$ = s$
END DEF
```

Now, XXX$ can be an **internal function** and you could safely use the variable names *i* and *s$* in the main program. Those variables will not share values.

You can also use the LOCAL statement in main programs along with the OPTION TYPO statement to help catch misspelled variable names. Chapter 19 describes this programming technique.

16. Creating and Using Libraries

Subroutines and functions — sometimes called procedures — let you segment your True BASIC programs. They may be either internal or external. Internal procedures are part of the program that uses them. External procedures are outside the "calling" program. In the examples you've seen they appear after the END statement of the **main program**.

External functions and subroutines are even more useful when you put them into libraries and modules.

Libraries

A **library** is a file that has no main program. It is only a collection of external functions and subroutines. Any program can use these procedures. All you have to do is include a **LIBRARY statement** in the program to identify the library file. Thus, a library file acts as a "tool kit" of useful functions and subroutines.

Each library file must begin with an EXTERNAL statement, which indicates that the file has no main program in it.

The GAMESLIB file in the Demo Programs folder is a library file. It's a small library, with a subroutine that simulates rolling any number of dice, and a function that simulates flipping a coin:

```
EXTERNAL

SUB Rolldice (sum_dice, num_dice)

    LET sum_dice = 0
    FOR i = 1 to num_dice
        LET roll = Int(6*Rnd + 1)
        LET sum_dice = sum_dice + roll
    NEXT i

END SUB

DEF Coin$

    IF Rnd < .5 then
       LET Coin$ = "heads"
    ELSE
       LET Coin$ = "tails"
    END IF

END DEF
```

You can revise the CRAPS program to use this library:

```
! Craps game using Library file.
!
LIBRARY "gameslib"
RANDOMIZE

FOR game = 1 to 10                    ! Play 10 games

   CALL Rolldice(dice,2)              ! Subroutine rolls 2 dice

   SELECT CASE dice                   ! Branch on roll

      CASE 2, 3, 12                   ! dice = 2, 3, or 12
         PRINT "You lose."

      CASE 7, 11! dice = 7 or 11
         PRINT "You win."
      CASE ELSE                       ! Anything else
         LET POINT = dice             ! Remember that roll
         DO
            CALL Rolldice(dice,2)     ! Roll again
         LOOP until dice = 7 or dice = point

         IF dice=point then PRINT "You win" else PRINT "You lose"
   END SELECT

NEXT game

END
```

The above program doesn't use the function to flip a coin. You don't have to use everything in the library. But, you can expand CRAPS so that it flips a coin to decide which of two players goes first. Notice that you must use a DECLARE DEF statement before you use the function, just as you must with an external function in the same file.

```
! Craps game.
!
LIBRARY "gameslib"
DECLARE DEF Coin$
RANDOMIZE

INPUT PROMPT "Heads or tails? ": choice$
LET toss$ = Coin$                     ! Flip the coin

IF Lcase$(choice$) = toss$ then       ! Tell who won
   PRINT choice$; ", you go first"
   LET player$ = "You "
ELSE
   PRINT toss$; ", I go first"
   LET player$ = "I "
END IF

FOR game = 1 to 10                    ! Play 10 games

   CALL Rolldice(dice,2)              ! Subroutine rolls 2 dice

   SELECT CASE dice                   ! Branch on roll

      CASE 2, 3, 12                   ! dice = 2, 3, or 12
```

```
            PRINT player$; "lose."

         CASE 7, 11! dice = 7 or 11
            PRINT player$; "win."
         CASE ELSE! Anything else
            LET POINT = dice          ! Remember that roll
            DO
               CALL Rolldice(dice,2) ! Roll again
            LOOP until dice = 7 or dice = point

            PRINT player$;
            IF dice=point then PRINT "win" else PRINT "lose"
      END SELECT

      IF player$ = "You " then        ! Switch players
         LET player$ = "I "
      ELSE
         LET player$ = "You "
      END IF

   NEXT game

   END
```

Notice that this program has several new or revised statements. New statements include the group near the beginning that tells who won the coin toss, and the group at the end of the FOR loop that switches players after each game. Several PRINT statements now use the variable *player$* to indicate whose game it is.

The built-in function LCASE$ lets you enter answers in upper or lowercase when you run the program; LCASE$ translates all answers to lowercase. You do not declare LCASE$ because True BASIC already knows about all built-in functions.

Appendix C in this manual lists most of True BASIC's built-in functions. For complete details on built-in functions, see the *True BASIC Bible*.

If the file named on the LIBRARY statement is not in the same directory as the program you are running, or in the TB Library folder you copied from the original True BASIC diskette, you must give True BASIC information about where to find the file.

For Further Study – Modules

Modules are libraries of external procedures that give you extra control over which variables are shared or local. Modules also let you share variables among some procedures but not with others. The *True BASIC Bible* explains the format and use of modules.

17. Graphics

Using True BASIC, you can write programs to draw points, lines, curves, and filled regions. You can produce animation and color, you can easily mix text with your graphics, and you can supply graphical input while your program is running. True BASIC's Pictures let you create re-usable graphics procedures. This chapter introduces several aspects of True BASIC graphics. For full information, see the *True BASIC Bible*.

Drawing Points

The easiest kind of graphics is marking points or drawing lines on a coordinate grid. The **PLOT statement** lets you do this on your output screen.

For each point you plot, you must give **two coordinates**: the X-axis or horizontal coordinate, and the Y-axis or vertical coordinate. Unless you specify otherwise (you'll see how to do that in a bit), True BASIC assumes your output screen uses a horizontal (X) axis from 0 to 1 and a vertical (Y) axis from 0 to 1. A simple True BASIC program to draw this point on your screen has just two lines:

```
PLOT .2,.4
END
```

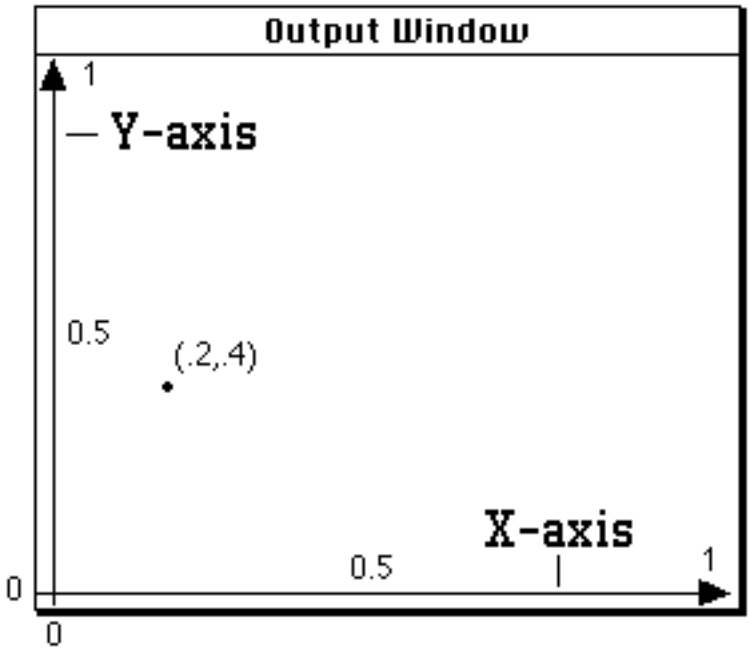


Figure 16.1: PLOT .2, .4

To plot additional points, you just add more PLOT statements. The following program puts four points on the screen. Create this program and run it.

```
PLOT .2,.4
PLOT .4,.4
PLOT .4,.6
PLOT .2,.6
END
```

Graphics Output

If your output window is active, the points appear there; otherwise, your output will occupy the full screen. You can force your graphics output to always occupy the full screen by adding the following statement to your program:

```
SET MODE "graphics"
```

You may find the output window especially useful for graphics, however, because you can adjust it to the shape that's best for showing off your output. The axes adjust to fit the window shape so the objects you draw may be distorted. If 0 to 1 on the X-axis is longer than 0 to 1 on the Y-axis, square objects will not appear square and circles will not be round.

True BASIC clears the output window at the start of each program run. Within a program, you can use the **CLEAR statement** to erase the contents of the output window.

You can also use just part of an output window or open several windows and put different displays in each. The *True BASIC Bible* explains the **OPEN** and **WINDOW** statements that let you control graphics output this way.

Drawing Lines

To draw lines, you use semicolons with your PLOT statements. Imagine that you are drawing with a light pen. A simple PLOT statement uses the pen's beam to draw a point and then turns the beam off. A semicolon at the end of a PLOT statement (or between two points in the PLOT statement) leaves the beam on. When True BASIC moves to the next point, it draws a line with the light pen. The beam stays on until a PLOT statement ends without a semicolon.

Add semicolons to the above program so that it connects points to draw two horizontal lines *(Figure 16.2)*:

```
PLOT .2,.4;    ! Draw a line to next point
PLOT .4,.4     ! Turn the "pen" off
PLOT .4,.6;    ! Draw a line to the next point
PLOT .2,.6
END
```

When drawing lines, you can combine several points on one PLOT statement. The following program connects all the points to draw

a box *(Figure 16.3)*. Notice that you must add another PLOT statement to close the box:

```
PLOT .2,.4; .4,.4; .4,.6; .2,.6;      ! Connect all points
PLOT .2,.4      ! Close the box and turn off the "pen"
END
```

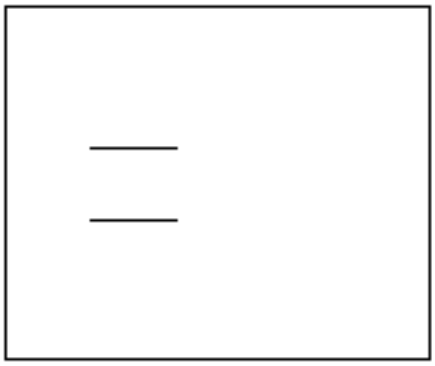


Figure 16.2: Horizontal Lines

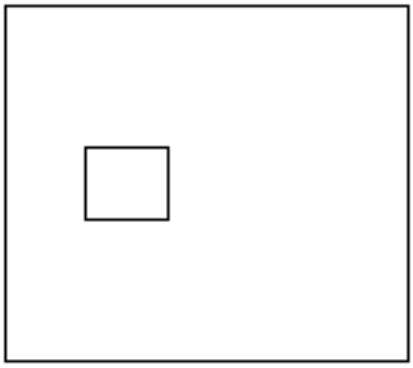


Figure 16.3: A Box

Changing the Coordinates

As you saw above, True BASIC assumes the output coordinates go from 0 to 1 in both the horizontal and vertical directions. However, you can use a **SET WINDOW statement** to set any boundaries you want.

For example, if you want the coordinates to go from 0 to 10 in both directions, you could include the following statement before you give any PLOT statements:

```
SET WINDOW 0, 10, 0, 10
```

The first two numbers give the start and end values for the horizontal axis, the second numbers give the start and end for the vertical axis.

Your coordinate system need not begin at zero, and the horizontal and vertical axes need not match. For example if you were plotting a graph to show production of cars over this century, you might set your coordinates as follows:

```
SET WINDOW 1900, 1990, 0, 10000000
```

The horizontal axis would show the range of years, and the vertical axis would let you plot production amounts from 0 to 10,000,000.

You can change the coordinates within a program. All PLOT statements use the coordinate system specified in the most recent SET WINDOW statement.

Drawing Shapes

True BASIC gives you two ways to draw empty or solid shapes. The **BOX statements** are the easiest and fastest method.

The **BOX LINES statement** draws the outline of a square or rectangle. You give the coordinates of the left, right, bottom, and top edges in the same way as in the SET WINDOW statement. The following program outlines a square as shown in Figure 16.4.

```
! Draw a square
!
SET WINDOW 0, 30, 0, 20          ! 15,10 is center of window
BOX LINES 10, 20, 5, 15          ! Draw box with sides = 10
END
```

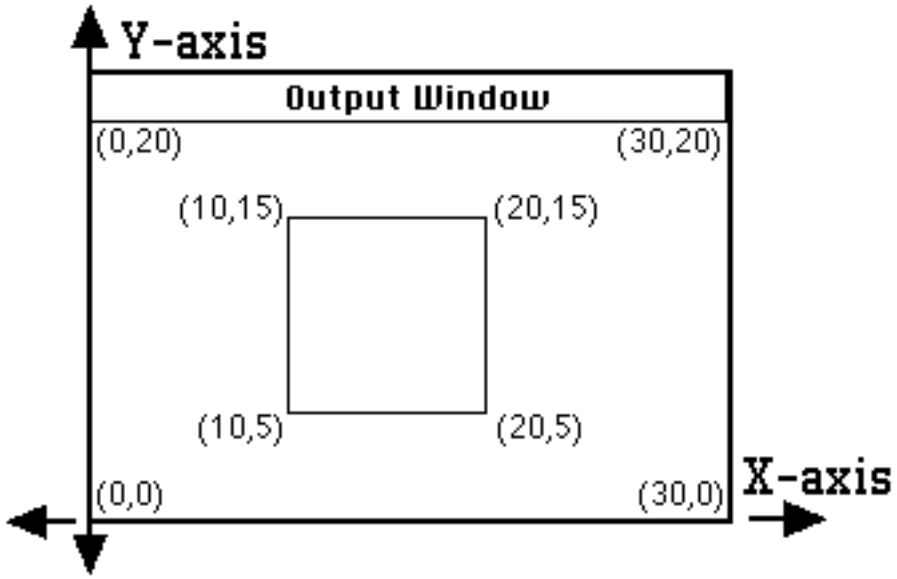


Figure 16.4: BOX LINES 10, 20, 5, 15

Similarly, the **BOX AREA statement** draws a solid square or rectangle using the coordinates you give in the statement:

```
! Draw a solid square
!
SET WINDOW -15, 15, -10, 10       ! 0,0 is center of
 window
BOX AREA -5, 5, -5, 5               ! 5*2 is length
 of each side
END
```

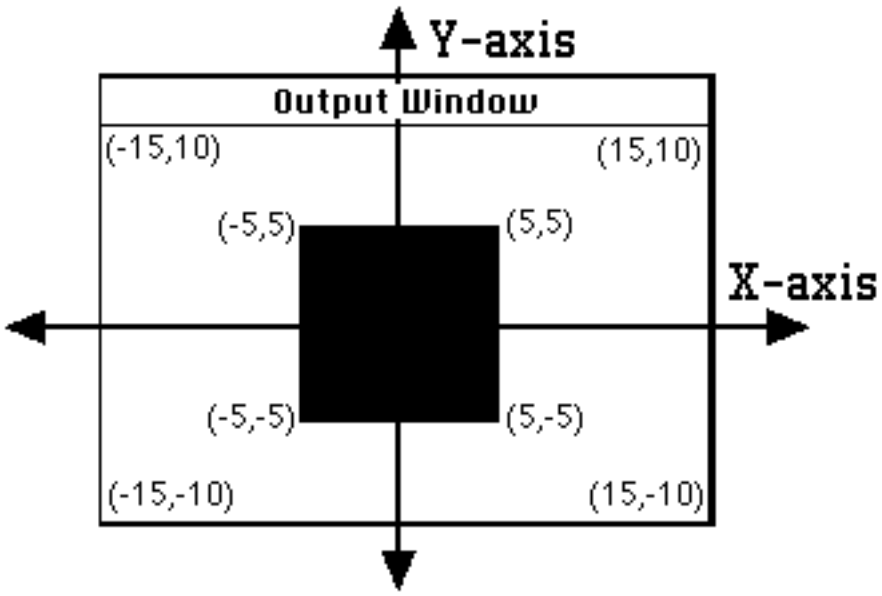


Figure 16.5: BOX AREA -5, 5, -5, 5

You can draw circles and ellipses using the **BOX CIRCLE** or **BOX ELLIPSE** statement. You give coordinates to these statements just as you do for BOX LINES and BOX AREA. True BASIC draws a circle or ellipse inside the border of the invisible box defined by the coordinates. It doesn't matter whether you use the CIRCLE or ELLIPSE keyword. If your coordinates define an invisible square, you get a circle; if the coordinates define a rectangle, you get an ellipse.

If you wish to draw a solid circle or ellipse, first draw the figure and then fill it in with the **FLOOD statement**. For the FLOOD statement, you give the coordinates for some point inside the object you want to fill. True BASIC fills the object from that point out to its boundaries. For example *(Figure 16.6)*:

```
SET WINDOW -10,10,-10,10
BOX CIRCLE -5, 5, -5, 5
FLOOD 0,0
END
```

You can draw more complex objects using a series of PLOT statements ending in semicolons. If you wish to fill the object you can then use a FLOOD statement. The following program outlines a knight from a chess set and then fills the object *(Figure 16.7)*:

```
! Draw a knight
!
PLOT .2,.1;.8,.1;.8,.2;                          ! Draw the outline
PLOT .7,.25;.7,.3;.8,.4;.65,.7;.6,.9;.55,.9;
PLOT .5,.82;.2,.75;.2,.6;.3,.6;.4,.55;
PLOT .25,.45;.2,.37;.3,.3;.3,.25;.2,.2;.2,.1
FLOOD .5,.5                                      ! Fill it in
END
```

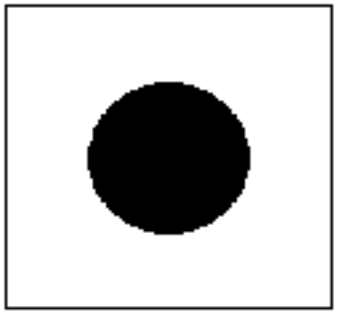


Figure 16.6:
BOX CIRCLE and FLOOD

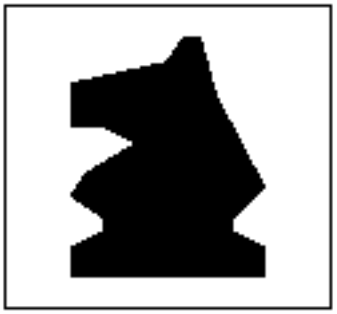


Figure 16.7:
PLOT and FLOOD

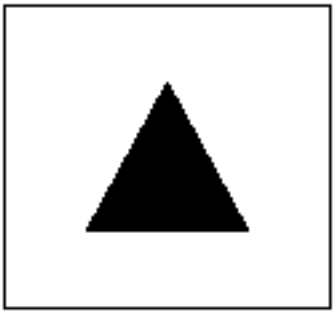


Figure 16.8:
PLOT AREA

The **PLOT AREA statement** connects a series of points **and** fills in the object. It works much as a series of PLOT statements except that PLOT AREA always connects the last point to the first. So you need not repeat the first point. The following statements draw and fill a triangle (Figure 16.8). Note that the PLOT AREA statement has a colon after the AREA keyword.

```
SET WINDOW -2, 2, -2, 2
PLOT AREA:  -1,-1; 1,-1; 0,1
END
```

Using Colors

In the examples used so far in this chapter, all solid objects are filled with black. You can also use different colors, or shades of gray if you have a black and white monitor. The **SET COLOR statement** lets you set a color or shade for succeeding PLOT statements. You can set colors by number or name:

```
SET COLOR "red"
SET COLOR 3
```

If you have a black and white monitor, you can use color names or numbers 1 to 5 to obtain equivalent shades of gray. The table shows the equivalent color names, numbers, and shades of gray for a black and white monitor.

Name	Number	Meaning
background	0	current background color
black	1	black
green	2	dark gray
blue	2	dark gray
magenta	3	medium gray
red	3	medium gray
cyan	4	light gray
yellow	4	light gray
brown	4	light gray
white	5	white

The following program (SQUARES in the Demo Programs folder) draws a series of solid squares in different colors or shades of gray:

```
! Draw six squares
!
SET WINDOW -10, 10, -10, 10
BOX AREA -6, 6, -6, 6              ! Draw outer square in black
FOR i = 5 to 1 step -1             ! From large to small
    SET COLOR i                    ! Change color
    BOX AREA -i, i, -i, i          ! Draw next square
NEXT i
END
```

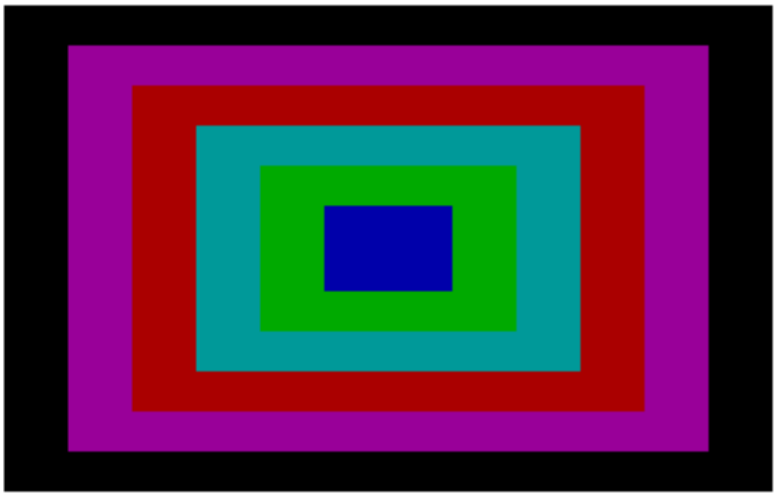


Figure 16.9: Six squares.

If you have a color monitor, you can use the nine True BASIC color names (listed in the table above). If your computer can produce more colors, you can use color numbers and the **SET COLOR MIX statement** for greater variety. The color numbers you can use depend on the color mode of your Macintosh. If your Macintosh has 256 colors, you can use color numbers 0 through 255. SET COLOR MIX lets you control the red, green, and blue elements producing a given color number. For more information on using colors, see the *True BASIC Bible*.

Animation

True BASIC's **BOX KEEP**, **BOX CLEAR**, and **BOX SHOW** statements let you simulate movement on the screen. The idea is to draw an image within a rectangular area on the screen, save that image as a string variable, and then redraw the image a slight distance away.

BOX KEEP saves the contents of a rectangular area on the screen in a string variable. You then erase the rectangular area on the screen with BOX CLEAR, and redraw the object somewhere else with BOX SHOW.

The ARROW program in the Demo Programs folder uses these statements to shoot an arrow across the screen. Open it and run it.

```
! Shoot an arrow across the screen
!
SET MODE "graphics"  ! Force use of full screen
SET WINDOW 0, 10, 0, 10

PLOT 0,5; 1,5! Draw arrow
PLOT .6,4.5; 1,5; .6,5.5
BOX KEEP 0, 1, 4, 6 in arrow$    ! Memorize arrow
PAUSE 1! Pause before shooting

LET x = 0
FOR move = 1 to 50! Move in small steps
    BOX CLEAR x, x+1, 4, 6 ! Erase old arrow
    LET x = x + .2! Advance x position
    BOX SHOW arrow$ at x,4 ! Draw at new position
NEXT move

END
```

Notice that the BOX KEEP and BOX CLEAR statements take coordinates to define a rectangular area just as the other BOX statements. For BOX SHOW you specify just the lower left corner where you want to draw the new image.

The **PAUSE statement** makes True BASIC wait before it erases and begins to move the arrow. The number tells how many seconds to pause. To slow the progress of the arrow across the window, you can add a PAUSE statement inside the FOR loop, just before the NEXT statement.

BOX CLEAR clears just the specified area so that other images can remain. If you wish to clear the entire screen, use the CLEAR statement.

For a more sophisticated program using animation, look at the Demo Program KNIGHT.

For more details on these and related statements, see the *True BASIC Bible*.

Pictures

Pictures are like subroutines for graphics. You can think of them as stencils. Define a picture and you can use it repeatedly to redraw an object at different locations.

As you will see, pictures are more flexible than stencils. You can draw the same picture repeatedly, but change its size or shape, or rotate it on the screen.

A picture is much like a subroutine. You name it and put the statements that plot it inside **PICTURE** and **END PICTURE** statements. When you want to use the picture, you "call" it with a **DRAW statement**. The following program uses a picture to draw a knight from a chess game. You'll notice that the picture contains the same statements used to draw a knight in the previous section on "Drawing Shapes". The following version is saved as PICTURE in the Demo Programs folder.

```
! Draw a knight using a picture
!
PICTURE Knight
   PLOT .2,.1;.8,.1;.8,.2;            ! Draw the outline
   PLOT .7,.25;.7,.3;.8,.4;.65,.7;.6,.9;.55,.9;
   PLOT .5,.82;.2,.75;.2,.6;.3,.6;.4,.55;
   PLOT .25,.45;.2,.37;.3,.3;.3,.25;.2,.2;.2,.1
   FLOOD .5,.5                        ! Fill it in
END PICTURE

DRAW Knight

END
```

Like subroutines and functions, pictures may be internal or external. External pictures may be stored in Library files.

Transformations

So far there doesn't seem to be any great benefit to defining a picture. The true power of pictures comes when you use them with transformations and parameters. **Transformations** let you move pictures or rotate, re-scale, or tilt them when you draw them. For example, you could replace the DRAW statement above with the following lines to draw lots of knights all over the screen.

```
SET WINDOW 0, 10, 0, 10
FOR x = 0 to 9
    FOR y = 0 to 9
        DRAW Knight with shift(x,y)
    NEXT y
NEXT x
```

The SHIFT transformation moves horizontal and vertical coordinates by the amounts you specify. The above statements use a larger coordinate system (SET WINDOW 0, 10, 0, 10) and then draw the knight 100 times within that window. Try it!

Similarly, you can double the size of the knight:

```
DRAW Knight with scale (2,2)
```

or make it twice as tall as wide:

```
DRAW Knight with scale (2,4)
```

SCALE multiplies the horizontal and vertical coordinates of your picture by the amounts you specify. Be aware that your scaled picture may become bigger than the window coordinates! Use a SET WINDOW statement to give enlarged coordinates if necessary.

Other transformations let you "shear" (or tilt) the picture or rotate the picture. You must give the amount of tilt or rotation in radians unless you include an **OPTION ANGLE DEGREES statement** first. You may then use degrees.

The SHEAR transformation leans vertical lines forward (clockwise) by the angle you specify. For example,

```
OPTION ANGLE DEGREES                ! Use degrees
DRAW Knight with shear (45)
```

makes the knight lean to the right by 45 degrees. Use a negative angle to lean a picture to the left. As with SCALE, you may have to use a SET WINDOW statement so that the picture doesn't lean out of the window.

ROTATE moves pictures counterclockwise (clockwise if you use a negative angle) around the (0,0) point in the window. Note that this is not the same as rotating a picture in place! You can easily rotate a picture out of coordinate window, unless you adjust coordinates with SET WINDOW or also shift the picture.

For example, if you rotate the knight 90 degrees, it would "fall on its face to the left" and be out of the standard coordinate system (0, 1, 0, 1). The upper right box of Figure 16.10 shows the knight drawn in the standard coordinate system with no transformations. The gray knight was rotated with the statements:

```
OPTION ANGLE DEGREES                ! Use degrees
DRAW Knight with rotate (90)
```

True BASIC rotates the knight about the point (0,0) and out of the standard coordinate window.

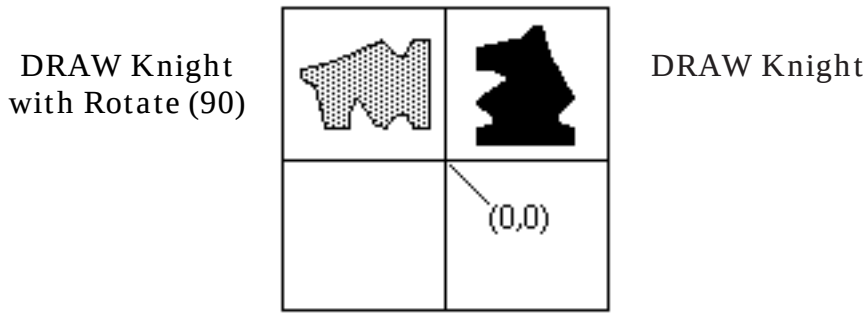


Figure 16.10: RotateTransformation

You can **combine transformations** on one DRAW statement by placing an asterisk (*) between transformations. For example, you could rotate the knight and then move it back into the (0, 1, 0, 1) window:

```
OPTION ANGLE DEGREES          ! Use degrees
DRAW Knight with rotate (90) * shift (1,0)
```

When you use more than one transformation, True BASIC performs them in order from left to right. Because of this, the order of transformations can make a difference. You're most likely to get the results you expect if you use SHIFT as the last transformation.

Much more information about pictures and transformations is found in the *True BASIC Bible*.

Creating Complex Pictures

With pictures and transformations you can create complex graphics. You can transform pictures and use them within other picture definitions. The HOUSES program in the Demo Programs folder combines simple pictures and transformations to provide a "neighborhood" of houses. Look at the program and run it. Try some variations of your own!

The GraphLib Library

Your **True BASIC *Free*** program also comes with a folder called TB Library. This folder contains libraries of subroutines that you can use in your True BASIC programs. The GRAPHLIB library provides the following routines:

`Frame`	frames the graphics window
`Axes`	draws X and Y axes
`Ticks`	draws X and Y axes with tick marks
`Polygon`	draws a polygon with any number of sides
`Bars`	draws a bar graph of data
`Fplot`	plots a function
`Arc`	draws the arc of a circle

These are all subroutines; use them with a CALL statement. You must give arguments for several of the subroutines. Open the GRAPHLIB file in the TB Library folder to see what each subroutine expects.

Remember that your program must include a LIBRARY statement to identify the GRAPHLIB file. Your program must either be in the same folder as GRAPHLIB, or you must give more information in the LIBRARY statement. For example, if you save

your program in the same location as (but not inside) the TB Library folder, you could use the following LIBRARY statement. This program draws coordinate axes with tick marks at every unit.

```
LIBRARY "TB Library:GraphLib"
SET WINDOW 0, 10, 0, 10
CALL Ticks(1,1)
END
```

Other Graphics Features

As you become more proficient, you might want to use some of True BASIC's other graphics statements. Several of these are briefly described below. For complete information, see the *True BASIC Bible*.

Text in Graphics Output. You can use the PRINT command in a graphics window, but it is hard to control the location and appearance of the text. The PLOT TEXT command lets you specify a coordinate location for the string you wish to print:

```
PLOT TEXT, at -1, 5 : "Test Results"
```

The coordinates designate the lower left corner of the text unless you control the location with a command such as SET TEXT JUSTIFY "center", "bottom".

Graphics Input. The GET POINT and GET MOUSE commands let you give coordinates to your program by "pointing to" a spot in the output window while the program is running. Using these commands, you could draw a figure by pointing to various places on the screen and having your program connect the points.

MAT PLOT Statements. If you are plotting many points, you could compute the coordinates and store them in a two-dimensional array with one row for each point (with X coordinates in the first column, and Y coordinates in the second). You can then use MAT PLOT POINTS, MAT PLOT LINES, and MAT PLOT AREA to plot the coordinates in the array.

Open the MATPLOT program in the Demo Programs folder to see the following example of a MAT PLOT AREA statement. (This uses the SIN, COS, and PI built-in functions; Appendix C lists most of True BASIC's built-in functions.)

```
!  MAT PLOT AREA example
!
DIM points (201,2)
SET WINDOW -1, 1, -1, 1

FOR t = 0 to 2 step .01    ! Compute points
    LET c = c+1! Count points
    LET points(C,1) = sin(3*t*pi)    ! x-coordinate
    LET points(c,2) = cos(5*t*pi)    ! y-coordinate
NEXT t

MAT PLOT AREA: points ! Draw and fill in

END
```

Printing a Graphic Image

True BASIC *Free* allows you to print graphics output by taking a screen "snapshot". The process for doing this will depend on the operating system you are using.

On a PC, press the Prt Sc or Print Screen key.

On the MacOS, press the Shift-Command-3 key combination.

Other editions of the True BASIC Language System (Bronze, Silver and Gold) allow you to print graphics directly from the program output window.

18. Sound and Music

You've already seen the demo program SMOKY that plays the first few lines of "On Top of Old Smoky". True BASIC's PLAY and SOUND statements let you produce melodies and general sound effects on your computer.

The PLAY Statement

The PLAY statement lets you play simple melodies on your computer. When you use a PLAY statement, you give it a string consisting of codes for notes, tempo, and how the notes should be played.

Open the SMOKY program, run it again, and then take a look at the music codes in the DATA statements.

```
! Plays the beginning of
! "On Top of Old Smoky".

DO while more data

   READ music$    ! Get the string representations
   PLAY music$    ! And play the notes

LOOP
DATA O4 L4 C C E G O5 L2 C. O4 A.
DATA L4 A F G A L1 G
DATA L4 C C E G L2 G. D.
DATA L4 E F E D L2 C.

END
```

Look at the first DATA statement, which represents the first six notes of "On Top of Old Smoky." The letters A through G represent the notes A through G. The other codes give True BASIC information about how to play the sequence of notes.

The letter **O followed by a digit** sets the current **octave**. The octaves start at C and go up to B, as on a piano keyboard. (Middle C is the first note in octave 5.) This song begins in the fourth octave, so the first string item is "O4".

Next, the letter **L followed by a digit** tells True BASIC the **length of the note or notes** to play. The larger the number with the code L, the shorter the length of the note. Therefore, "L4" means a quarter note, "L2" a half note, and "L1" a whole note. True BASIC plays all notes following an L code at that length until another L appears in the string expression.

After the first DATA statement sets the octave and the length of notes, "C C E G" tells True BASIC to play two C's, an E, and a G as quarter notes. The next note, however, is in the next octave, so you need another O code to set the octave to O5.

After O5, the next note is a 3/4 note C. This is done by changing the length to L2 (half note) and adding a dot after the letter C. The **dot multiplies the length of the note by 3/2**, just as it does in written music. The line ends by going back down to octave 4 and playing another 3/4 note, A.

The remaining string data use these codes to play the next three lines of the song. You may type the letters in the codes in upper or lowercase. Also, the spaces between the codes don't matter to True BASIC, but they do make the program easier to read!

True BASIC has other music codes that give you more control over the notes and the way they're played. The letter **T sets the tempo**, or speed, for the rest of the melody. The number given with T represents the number of quarter notes played in one minute. If you don't specify the tempo, True BASIC plays 120 quarter notes per minute. Add the code T180 to the first DATA statement, and run SMOKY again.

The **ML code plays music legato**, and **MS plays staccato**. (Legato means play the music smoothly with a connection between successive notes. Staccato means play the music briskly with no connection between notes.) Add some of these codes to Smoky, and run it again. You can use the **MN code to set the music style back to normal**.

You can include **sharps** and **flats** in your music by adding a "+" or "#" after the note to indicate a sharp, or "-" after the note to indicate a flat. You can also write lengths of single notes by putting the appropriate digit after the letter for that note. For example, the first two lines of "America" in the key of F would look like this:

```
F4 F4 G4 E4. F8 G4
A4 A4 B-4 A4. G8 F4
```

The letter **R stands for rest**. The number given with R has the same meaning as the numbers associated with the code L. That is, R4 means rest for the length of a quarter note, R2 means rest for the length of a half note, etc.

The following table summarizes the PLAY codes.

Code	Meaning
`A to G`	Play a note in current octave, at current tempo, etc.
`L n`	Set the length of subsequent notes.
`ML`	Play music legato, or smoothly.
`MN`	Play music normally (not legato or staccato).
`MS`	Play music staccato, or briskly.
`O n`	Set current octave. Middle C is the first note in octave 5.
`R n or P n`	Rest (pause) for length n.
`T n`	Set the tempo.
`# or +`	Sharp.
`-`	Flat.
`.`	Play dotted note.

The SOUND Statement

The SOUND statement makes your computer emit sounds that are not necessarily musical notes. You specify the frequency of the sound in Hertz (cycles per second) and the duration of the sound in seconds. For example, the statement:

```
SOUND 440, 10
```

plays concert A, which has a frequency of 440 Hertz, for 10 seconds.

Here's a familiar noise you can imitate. You can create a trilling sound with a fast alternation between two frequencies. The following program imitates the ringing of a telephone by alternating between the frequencies of 600 and 1500. Type it in and run it. Remember that you can stop a program with the **Run** menu!

```
! Imitate ringing of a telephone.
!
FOR ring = 1 to 8          ! Let ring 8 times
    FOR i = 1 to 30        ! Each ring = 30 alternations
        SOUND 600, .03     ! Freq 600 for .03 sec
        SOUND 1500, .03        ! Then 1500 for .03 sec
    NEXT i
    PAUSE 2               ! Pause 2 seconds
NEXT ring
END
```

The PAUSE statement tells True BASIC to pause for the given number of seconds.

19. Correcting Errors and Debugging Your Programs

There are three kinds of mistakes you might make when writing a program: (1) improperly used True BASIC statements, (2) errors that occur when a program runs, and (3) "bugs" that prevent your program from working as you intended. True BASIC can help you find many of these errors, and you can learn some tricks to help you find others.

Illegal Statements

One of the easiest things that True BASIC can find for you is a statement or structure you have used incorrectly. When you attempt to run a program with an **illegal statement**, True BASIC prints an error message at the bottom of the window and places the cursor at the offending spot in your program. You can then correct that error and run the program again. If there are more errors of this type, True BASIC finds them one at a time and prints appropriate messages.

Consider the following:

```
PIRNT  "You are about to toss a coin"
IF rnd<.5 PRINT "Heads, you win" else PRINT "Tails, you lose"
```

The first time you attempt to run this program, True BASIC prints the message "Illegal statement" and places the cursor before the misspelled keyword PIRNT. If you correct that and again run the program, you would see the message "Expected 'then'." and the cursor would appear just before the first PRINT in the second line. Finally, you would receive a message "Missing end statement." Your corrected program should look like this:

```
PRINT  "You are about to toss a coin"
IF rnd<.5 then PRINT "Heads, you win" else PRINT "Tails, you lose"
END
```

Appendix D lists and briefly explains the error messages you are likely to see as you write programs using the statements introduced in this book. If you are not sure of the corrections you need to make, reread the appropriate sections of the book.

If you use Do Format to indent your programs, you can often catch problems in multi-line structures such as IF-THEN-ELSE decisions or FOR-NEXT loops.

Errors During Program Runs — Exceptions

A program can sometimes cause errors when it is run (executed). For example, the statement

```
LET answer = a/b
```

is a "legal" statement. But if *b* equals 0 when this statement is carried out, the program would stop and you would get a "Division by zero" error. Errors that happen during program runs are called **exceptions**. The list of error messages in Appendix D includes exceptions.

True BASIC has a structure and three built-in functions that you can include in your programs to intercept this type of error and provide a remedy that can enable the program to keep running.

The WHEN structure lets you tell True BASIC what to do if an error occurs in a statement included in the structure. For example, consider a program such as those in section 13 that ask the user to input the name of a file containing data for the program to read. If the user types the file name incorrectly or names a file that True BASIC can't find, the program would stop with an "exception". The following WHEN structure protects against this error:

```
DO
   PRINT "File containing data";      ! Ask user for file
   INPUT filename$
   WHEN error in
        OPEN #1: name filename$
        LET success = 1               ! Set success "flag"
   USE
        PRINT "Cannot find that file."
   END WHEN
LOOP until success = 1                ! Repeat until successful
```

If the statements following WHEN ERROR IN cause no errors the program skips ahead to the END WHEN statement; the loop does not repeat because *success* now equals 1. If the input value causes an error in the OPEN statement, however, the skips to the USE part of the structure and the loop repeats, again asking the user for a file name.

This WHEN structure and the EXLINE$, EXTEXT$, and EXTYPE functions are explained in the *True BASIC Bible*.

Correcting Bugs in Your Programs

True BASIC cannot detect the third type of programming error. Your program may be "legal" and contain no "exceptions", but it still gives the "wrong" answers. Somehow, you've not written the program correctly to accomplish what you wanted to do.

True BASIC can't tell what you want your program to do, so it can't tell you where you've gone wrong, but there are some tools you can use to **debug** your programs.

• One of the first things to do is use DO FORMAT to make the program more readable (see section 11). Next, get a printed listing of your program and read it carefully.

• As you read, check your variable names. Have you spelled them correctly and consistently throughout the program? The OPTION TYPO and LOCAL statements described below can help you catch spelling errors in variable names.

• If you are not sure where your errors are, but suspect parts of the program, insert some extra PRINT statements to see what values your variables have at various points in your program.

If these simple methods don't help, try one of the following debugging features. These are described more completely in the *True BASIC Bible*.

OPTION TYPO and LOCAL. You can put an OPTION TYPO statement at the beginning of your program to request True BASIC to check all variables in that program. For this to work, all variable names must be **declared** in a LOCAL statement or appear as parameters in a SUB, DEF, FUNCTION, or PICTURE statement. (All arrays must be declared in DIM or LOCAL statements.) True BASIC gives an "Unknown variable" error for any undeclared variable that it sees. You have to do some extra typing to list all variables in a LOCAL statement, but it can save debugging time by finding misspelled variables. Section 15 introduces the LOCAL statement. (You may also declare variable names in PUBLIC, SHARE, or DECLARE PUBLIC statements as described in the *True BASIC Bible*.)

Breakpoints. You can insert **breakpoints** into your program. When you run the program, True BASIC halts at each breakpoint. You can then use PRINT statements in the command window to see the current value of any variables. Type the CONTINUE command to resume the program run. (For a review on using the command window, see Chapter 11.)

To insert a breakpoint, move the cursor to the desired line and choose **Breakpoint** from the **Run** menu. To remove a breakpoint, select the line and again choose **Breakpoint** from the **Run** menu.

DO TRACE. This is a powerful tool that lets you step through the program line by line seeing how values of selected variables change. Use this in the command window. As an example,

```
do trace, step (miles, first, name$)
```

will step through the program instruction by instruction (waiting for you to press the space bar between each step) showing every time the value of one of the three variables *miles*, *first*, or *name$* changes. You can trace up to 8 variables. For more information about this command and other options available with it, see the *True BASIC Bible*.

DO XREF prints your program with line numbers and then prints a table of every variable, number, and keyword in the program, with the numbers of the lines on which they occur. Examining this may help you spot a bug. To use DO XREF you must have a printer available to your computer. Then type `do xref` at the Ok prompt in the command window.

For more information about this command and other options available with it, see the *True BASIC Bible*.

Appendix A: The ASCII Character Set

This table lists the ASCII character set. The order of characters determines how string conditions are evaluated. The decimal and hexademinal equivalents given for each character are useful for advanced programmers.

The True BASIC Bible contains additional information how you can use the ASCII character set in various programming situations.

Decimal	Name	Hex	Decimal	Name	Hex
0	nul	0	27	esc	1B
1	soh	1	28	fs	1C
2	stx	2	29	gs	1D
3	etx	3	30	rs	1E
4	eot	4	31	us	1F
5	enq	5	32	space	20
6	ack	6	33	!	21
7	bel	7	34	“	22
8	bs	8	35	#	23
9	ht	9	36	$	24
10	lf	0A	37	%	25
11	vt	0B	38	&	26
12	ff	0C	39	’	27
13	cr	0D	40	(	28
14	so	0E	41	)	29
15	si	0F	42	*	2A
16	dle	10	43	+	2B
17	dc1	11	44	,	2C
18	dc2	12	45	-	2D
19	dc3	13	46	.	2E
20	dc4	14	47	/	2F
21	nak	15	48	0	30
22	syn	16	49	1	31
23	etb	17	50	2	32
24	can	18	51	3	33
25	em	19	52	4	34
26	sub	1A	53	5	35

Decimal	Name	Hex	Decimal	Name	Hex
54	6	36	91	[	5B
55	7	37	92	\	5C
56	8	38	93	]	5D
57	9	39	94	^	5E
58	:	3A	95	_	5F
59	;	3B	96	`	60
60	<	3C	97	a	61
61		3D	98	b	62
62	>	3E	99	c	63
63	?	3F	100	d	64
64	@	40	101	e	65
65	A	41	102	f	66
66	B	42	103	g	67
67	C	43	104	h	68
68	D	44	105	i	69
69	E	45	106	j	6A
70	F	46	107	k	6B
71	G	47	108	l	6C
72	H	48	109	m	6D
73	I	49	110	n	6E
74	J	4A	111	o	6F
75	K	4B	112	p	70
76	L	4C	113	q	71
77	M	4D	114	r	72
78	N	4E	115	s	73
79	O	4F	116	t	74
80	P	50	117	u	75
81	Q	51	118	v	76
82	R	52	119	w	77
83	S	53	120	x	78
84	T	54	121	y	79
85	U	55	122	z	7A
86	V	56	123	{	7B
87	W	57	124	|	7C
88	X	58	125	}	7D
89	Y	59	126	~	7E
90	Z	5A	127	del	7F

Appendix B: List of True BASIC Statements

This appendix lists all of the statements in True BASIC, and then lists an example or two of those statements that are discussed in this book. For a more complete explanation, see the *True BASIC Bible* which is available for sale from True BASIC Inc.

Ordinary Statements and Structures

These statements are fundamental to almost all programs.

PROGRAM
END
LET
DO Loop Structure
    EXIT DO
    LOOP
IF
IF Structure
    ELSEIF
    ELSE
    END IF
FOR Loop Structure
    EXIT FOR
    NEXT
SELECT CASE Structure
    CASE
    CASE ELSE
    END SELECT

These statements are of a miscellaneous type; some are discussed in this manual.

ASK FREE MEMORY
DIM
LOCAL
PAUSE
RANDOMIZE
    REM
STOP

These statements deal with line-number programs; they are not discussed in this guide but found in the *True BASIC Bible*.

GOSUB
GOTO
ON GOSUB
ON GOTO
RETURN

These statements allow setting various options; only the first and last are discussed in this manual.

OPTION ANGLE
OPTION BASE
OPTION NOLET
OPTION TYPO

Input and Output Statements

These are the main statements dealing with input and output that are discussed in this manual.

DATA
INPUT
LINE INPUT
MAT INPUT
MAT LINE INPUT
MAT PRINT
MAT READ
PRINT
READ
RESTORE

These input-output statements are not discussed in this book.

ASK MARGIN
ASK ZONEWIDTH
SET MARGIN
SET ZONEWIDTH

Functions and Subroutines

These statements are the heart and soul of organizing complicated programs.

CALL
DECLARE DEF
DEF
DEF Structure
    EXIT DEF
    END DEF
EXTERNAL
LIBRARY
LOCAL
SUB Structure
    EXIT SUB
    END SUB

The following statements are not discussed in this book.

FUNCTION
FUNCTION Structure
    EXIT FUNCTION
    END FUNCTION
DECLARE FUNCTION
DECLARE SUB
CHAIN

Graphics and Sound Statements

These graphics and sounds statements are discussed in this manual.

BOX AREA
BOX CIRCLE
BOX CLEAR
BOX ELLIPSE
BOX KEEP
BOX LINES
BOX SHOW
CLEAR
DRAW
FLOOD
PICTURE Structure
    EXIT PICTURE
    END PICTURE
PLAY
PLOT
PLOT AREA
PLOT LINES
PLOT POINTS
PLOT TEXT
SOUND
SET WINDOW

These graphics statements are not discussed in this manual.

ASK BACK
ASK COLOR
ASK COLOR MIX
ASK CURSOR
ASK MAX COLOR
ASK MAX CURSOR
ASK MODE
ASK PIXELS
ASK SCREEN
ASK TEXT JUSTIFY
ASK WINDOW
GET KEY
GET MOUSE
GET POINT
MAT PLOT
MAT PLOT AREA
MAT PLOT LINES
MAT PLOT POINTS
OPEN SCREEN
SET BACK
SET COLOR
SET COLOR MIX
SET CURSOR
SET MODE
SET TEXT JUSTIFY

MAT Statements

Several of these MAT statements are discussed in this book.

DIM
MAT INPUT
MAT LINE INPUT
MAT PRINT
MAT READ

Some of the MAT statements are not discussed in this book.

MAT REDIM
MAT WRITE
MAT PLOT AREA
MAT PLOT LINES
MAT PLOT POINTS

Files Statements

Four file statements are introduced in this manual. The following file statements are not discussed:

ASK #n: ACCESS
ASK #n: DATUM
ASK #n: DIRECTORY
ASK #n: ERASABLE
ASK #n: FILESIZE
ASK #n: FILETYPE
ASK #n: MARGIN
ASK #n: NAME
ASK #n: ORGANIZATION
ASK #n: POINTER
ASK #n: RECORD
ASK #n: RECSIZE
ASK #n: RECTYPE
ASK #n: SETTER
ASK #n: ZONEWIDTH
READ #n:
SET #n: DIRECTORY
SET #n: MARGIN
SET #n: NAME
SET #n: POINTER
SET #n: RECORD
SET #n: RECSIZE
SET #n: ZONEWIDTH
UNSAVE
WRITE #n:

Module Structures

These statements, that deal with modules, are not discussed in this book.

MODULE Structure
- PRIVATE
- PUBLIC
- SHARE
- DECLARE PUBLIC
- END MODULE

Exception Handling

Exception handling is not discussed in this book.

CAUSE
WHEN Structure
- USE
- END WHEN

Alphabetical Listing of Statements

This section gives examples and brief descriptions of the statements and structures discussed in this manual.

BOX AREA Statement

BOX AREA left, right, lower, upper

Draws the rectangle specified and fills it with the current foreground color.

BOX CIRCLE Statement

BOX CIRCLE left, right, lower, upper

Draws an ellipse (or circle) inscribed in the rectangle specified in the current foreground color.

BOX CLEAR Statement

BOX CLEAR left, right, lower, upper

Clears the rectangular region specified; that is, it fills that region with the current background color.

BOX ELLIPSE Statement

BOX ELLIPSE left, right, lower, upper

BOX ELLIPSE is the same as BOX CIRCLE.

BOX KEEP Statement

BOX KEEP left, right, lower, upper IN stringvar$

Stores the entire rectangular region specified into stringvar$.

BOX LINES Statement

BOX LINES left, right, lower, upper

Draws the outline of a rectangle specified in the current foreground color.

BOX SHOW Statement

BOX SHOW stringvar$ AT left, lower

BOX SHOW restores the image previously stored in stringvar$ to the rectangular position whose lower left corner is specified.

CALL Statement

CALL subroutine-name (arg1, arg2, ..., argn)

The CALL statement invokes the subroutine given by the SUB statement with the same name. The arguments in the CALL statement must match with the parameters in the SUB statement (in number, positions, type, and number of dimensions.)

Parameter passing **by reference**; that is, changes to them within the subroutine will cause simultaneous changes the arguments in the CALL statement.

CLEAR Statement

CLEAR

Clears the screen or output window and resets the text cursor to the row 1, column 1; also switches to the full screen for output.

CLOSE Statement

CLOSE #n

The CLOSE statement closes the channel (a file or printer) opened as #n. You must close a channel before you can OPEN a new channel with the same number.

DATA Statement

DATA: element, ..., element

The data elements can be quoted or unquoted strings.

At program startup, all the data in the collection of DATA statements in a program-unit are collected into a data list, in the order in which they are encountered.

(See also READ and RESTORE).

DECLARE DEF Statement

DECLARE DEF funcname, ..., funcname

DECLARE DEF statements must name all external functions used in the given program-unit before their first use. DECLARE DEF statements must name all internal functions used in the given program-unit whose definitions occur later in the program-unit than their first use.

DEF Statement

DEF identifier = numeric-expression

DEF identifier (parm1, ..., parm n) = numeric-expression

DEF identifier$ = string-expression

DEF identifier$ (parm1, ..., parm n) = string-expression

The DEF statement allows the programmer to define single-line functions.

The function is invoked by including its name, with suitable arguments, in an expression. The arguments must match the parameters in the DEF statement in number, position, type, and number of dimensions.

DEF Structure

```
DEF identifier (parm1, ..., parm n)
  ...
  EXIT DEF   [optional]
  ...
END DEF
```

The DEF structure may contain one or more EXIT DEF statements. The DEF structure allows the programmer to define new multi-item functions.

The function is invoked by including its name, with suitable arguments, in an expression. The arguments must match the parameters in the DEF structure in number, position, type, and number of dimensions. Parameter passing is **by value**; that is any changes to the parameters will *not* cause changes to the corresponding arguments.

The defined function can also contain DECLARE DEF and LOCAL statements.

DIM Statement

DIM array (bounds), ..., array (bounds)

Except for function or subroutine parameters, each array in a program-unit must be dimensioned in a DIM or LOCAL statement that occurs lexically before the first reference to that array.

DO Loop

DO { | WHILE *condition* | UNTIL *condition* | }

. . .

EXIT DO [optional]

. . .

LOOP { WHILE *condition* | UNTIL *condition* | }

The DO statement can contain either a WHILE or UNTIL part, or nothing, and the same for the LOOP statement. There can be any number of EXIT DO statements.

DRAW Statement

DRAW picture name (arg 1, ..., arg n)

DRAW picture name (arg 1, ..., arg n) WITH *trans* *... * *trans*

trans:: SCALE (size)
SCALE (xsize, ysize)
ROTATE (angle)
SHIFT (xshift, yshift)
SHEAR (angle)

The (argument-list) is optional. The DRAW statement causes the picture named to be drawn on the screen, just as if the DRAW statement were replaced by the code of the picture definition. The angles in ROTATE and SHEAR are measured in radians unless OPTION ANGLE DEGREES is in effect.

If the WITH clause is present, then the transformation applies applies to PLOT, FLOOD, and MAT PLOT statements (but not BOX statements) in the picture before drawing it. If a picture also contains DRAW statements with WITH clauses, then the final transformation is the “product” of the transformations along the way. The transformation consists of shifts, rotations, shears, or changes of scale, or any sequence thereof.

SCALE with one argument is the same as SCALE with two arguments with the same scale factor applied to both the x- and y-directions. That is, SCALE(a)= SCALE(a,a).

ROTATE causes the picture to be rotated counter-clockwise through the given angle.

SHIFT causes the picture to be shifted in the x-direction by an amount given by the first argument, and in the y-direction by an amount given by the second argument.

SHEAR causes the picture to be tilted clockwise through the specified angle. That is, it leaves horizontal lines horizontal, but tilts vertical lines through the given angle.

END Statement

The END statement must be the last statement of a program and is required. Only one END statement is allowed. The file that contains the program can also contain external procedures and modules following the END statement. Executing the END statement stops the program.

END DEF Statement

The END DEF statement must appear as the last statement of a DEF structure.

END IF Statement

The END IF statement must appear as the last statement of an IF structure.

END PICTURE Statement

The END PICTURE statement must appear as the last statement of a PICTURE structure.

END SELECT Statement

The END SELECT statement must appear as the last statement of a SELECT structure.

END SUB Statement

The END SUB statement must appear as the last statement of a SUB structure.

ERASE Statement

ERASE #n

The ERASE statement erases the contents of the file opened as channel #n. An error occurs if you attempt to erase a file opened access INPUT.

EXIT DEF Statement

EXIT DEF

The EXIT DEF statement jumps to just beyond the END DEF statement of the innermost function that contains it, and is optional.

EXIT DO Statement

EXIT DO

The EXIT DO statements jumps to just beyond the LOOP statement of the inner-most DO loop containing the EXIT DO, and is optional.

EXIT FOR Statement

EXIT FOR

The EXIT FOR statement jumps to just beyond the NEXT statement of the inner-most FOR loop containing the EXIT FOR, and is optional.

EXIT PICTURE Statement

EXIT PICTURE

The EXIT PICTURE statement jumps to just beyond the END PICTURE statement of the innermost picture that contains it, and is optional.

EXIT SUB Statement

EXIT SUB

The EXIT SUB statement jumps to just beyond the END SUB statement of the innermost subroutine that contains it, and is optional.

EXTERNAL Statement

EXTERNAL

The EXTERNAL statement must appear at the start of a LIBRARY file of external procedures.

FLOOD Statement

FLOOD xcoord, ycoord

FLOOD will fill, with the current foreground color, the closed graphical region containing the point whose x-coordinate is xcoord and whose y-coordinate is ycoord.

FOR Loop

FOR forvar = numeric-expression TO numeric-expression STEP numeric-expression

...

EXIT FOR [optional]

...

NEXT forvar

The simple numeric variable (not a numeric array element) in the NEXT statement must be the same as the numeric variable appearing in the FOR statement. The STEP part is optional. If missing, the increment is 1.

IF Statement

IF *condition* THEN *simple-statement* ELSE *simple-statement*

If the condition is "true," then the *simple-statement* following the keyword THEN will be executed, following which control will pass to the next line.

If the condition is "false," and the ELSE clause is present, its simple-statement will be executed, following which control will pass to the next line. If the ELSE clause is not present, then control will pass directly to the next line.

IF Structure

```
IF condition1 THEN
  ...
ELSEIF condition2 THEN
  ...
ELSEIF condition3 THEN
  ...
ELSE
  ...
END IF
```

The IF structure can have 0 or more ELSEIF parts and 0 or 1 ELSE. If ELSE is present, it must follow any ELSEIF part. The keyword ELSEIF can be spelled ELSE IF.

If *condition 1* is "true," the statements immediately following are executed, up to the first ELSEIF, ELSE, or END IF, following which control jumps to the statement following the END IF.

If *condition 1* is "false," control passes to the first ELSEIF part following the IF line. If *condition 2* is "true," the statements immediately following it are executed, up to the next ELSEIF, ELSE, or END IF, following which control passes to the statement following the END IF line. If *condition 2* is "false," this process is repeated.

If there are no more ELSEIF parts, then control is passed to the ELSE part, and the statements following the ELSE line are executed, up to the END IF line. If there is no ELSE part, control is passed to the statement following the END IF line.

INPUT Statement

INPUT variable, ..., variable

INPUT PROMPT *string-constant*: variable, ..., variable

When the INPUT statement is executed, the program awaits an input-response from the user. The input-response consists of quoted-strings and unquoted-strings, separated by commas.

The items in the input-response are assigned to the variables in the INPUT statement. String variables can receive any input-

item, but numeric variables can receive only input-items whose characters form a numeric-constant. The rules are similar to those for READ and DATA statements.

LET Statement

LET variable = *formula*

The LET statement computes the formula on the right of the equal sign and then assigns the value to the variable on the left of the equal sign.

LIBRARY Statement

LIBRARY *quoted-string ..., quoted-string*

The LIBRARY statement names the file or files containing external routines needed by the entire program.

LINE INPUT Statement

LINE INPUT stringvar$, ..., stringvar$

LINE INPUT PROMPT *string-constant*: strvarvar$, ..., strvarvar$

A LINE INPUT statement requests one or more lines of input from the user. The first line is supplied to the the first stringvar$, the second to the second, and so on. All characters in the response-line are supplied, including leading and trailing spaces, embedded commas, and quote marks.

LOCAL Statement

LOCAL variable, ..., variable

A LOCAL statement specifies that the variables named in it are local to the routine containing the statement. If an array is named in a LOCAL statement, it must also include its subscript bounds. The LOCAL statement is normally irrelevant in external routines, since all variables except parameters are automatically local, but it can be important in internal routines. The LOCAL statement can be used in conjunction with the OPTION TYPO statement to avoid typographical errors in variable names.

LOOP Statement

The LOOP statement may occur only as the last statement of a DO loop, and is required. (See the DO Loop.)

MAT INPUT Statement

MAT INPUT array, ..., array

MAT INPUT assigns values from the input-response to the elements of the arrays, in order. There must be a separate input-response for each array in the inputlist. For each array, the

elements are assigned values in “odometer” order. (That is, if A is a 2-by-2 array, odometer order is A(1,1), A(1,2), A(2,1), A(2,2).) The input-response must contain a sufficient number of values of the appropriate type (numeric or string), separated by commas, in a single input-response or in a collection of input-responses with all but the last ending with a comma. (See the INPUT statement for details of input-responses.)

MAT LINE INPUT Statement

MAT LINE INPUT strarray$ ..., strarray$

MAT LINE INPUT assigns response-lines to the elements of the arrays in the redimarraylist, in order from left to right, and within each array in odometer order. The entire line of input is assigned to an array element, including leading and trailing spaces and embedded commas.

MAT PRINT Statement

MAT PRINT array, ..., array

The MAT PRINT statement prints the elements of each array in its matprintlist to the screen. The values of each array are printed separately, with a blank line following the printed values for each array. For two-dimensional arrays, the values for each row start on a new line. This rule also applies to arrays of three or more dimensions.

Any command may be replaced by a semicolon, in which case the elements of that array are printed side by side.

MAT READ Statement

MAT READ array, ..., array

MAT READ assigns values from the DATA list to the elements of each of the arrays, in order. For each array in the readarraylist, the values are assigned in “odometer” order – that is, the last subscript changes most rapidly, then the next to last, and so on.

A strvar can receive any valid datum. A numvar can receive only a datum that happens to be a valid and unquoted numeric-constant.

NEXT Statement

The NEXT statement can be used only as part of a FOR loop and is required.

OPEN Statement

OPEN #n: PRINTER

OPEN #n: NAME filename

OPEN #n: NAME filename, CREATE new-or-old

The OPEN statement opens a channel to a printer or file. You subsequently refer to the printer or file by its channel number on appropriate statements such as PRINT, INPUT, ERASE, SET or CLOSE.

You may give the file name as a string constant (in double quotes) or a string variable name. Allowable CREATE keywords are OLD (the default), NEW, or NEWOLD.

OPTION ANGLE Statement

OPTION ANGLE DEGREES
OPTION ANGLE RADIANS

The OPTION ANGLE statement allows you to specify the type of angle measure to be used with trigonometric functions and graphics transforms. In the absence of an OPTION ANGLE statement, the default angle measure is RADIANS.

OPTION TYPO Statement

OPTION TYPO

The OPTION TYPO statement requires that all non-array variables that appear lexically after it be declared explicitly. They must be declared in a LOCAL statement, or by appearing as parameters in a SUB, DEF, or PICTURE statement.

An OPTION TYPO statement applies to the rest of the procedure containing it and to all subsequent procedures in the program or library file.

PAUSE Statement

PAUSE seconds

The PAUSE statement stops the program for a time (in seconds) and then continue.

PICTURE Structure

PICTURE picture-name (parameter-list)
  ...
  EXIT PICTURE [optional]
  ...
END PICTURE

A PICTURE structure may contain one or more EXIT PICTURE statements.

A PICTURE is drawn with a DRAW statement. Other than that, a PICTURE acts exactly like a subroutine. The parameter passing mechanism is that of subroutines.

If the PICTURE contains PLOT statements (PLOT, MAT PLOT, FLOOD, GET POINT, or GET MOUSE) or contains CALL or DRAW statements to other pictures or subroutines, then the final picture will reflect all the transforms applied through all the DRAW statements.

‘

PLAY Statement

PLAY string-expression

See the *True BASIC Bible* for a full explanation of all the options.

PLOT Statements

For convenience, the term **point** means two coordinates (x and y) separated by a comma, as in “xcoord, ycoord”.

All PLOT statements in pictures are subject to the effects of the current transform.

All PLOT statements, except for PLOT TEXT, are clipped at the edges of the current window. That is, the portion of the drawing that is inside the window is shown, while the portion outside the window is not.

PLOT POINTS Statement

PLOT POINTS: point; ...; point

PLOT point

PLOT POINTS plots the points as dots. PLOT x,y is an abbreviation for PLOT POINTS: x,y.

PLOT LINES Statement

PLOT LINES: point; ...; point

PLOT point; ...; point

PLOT LINES: point; ...; point;

PLOT point; ...; point;

PLOT LINES plots the line-segments that connect the points. A line is drawn from the previous point to the first point if and only if the beam was left on.

The following two statements are equivalent:

```
PLOT x1, y1; x2, y2; x3, y3
PLOT LINES:   x1, y1; x2, y2; x3, y3
```

If the PLOT LINES and PLOT statements end with a semicolon, the beam stays on so that subsequent PLOT LINES or PLOT statements will continue plotting the line without a break; otherwise, the beam is turned off.

PLOT AREA Statement

PLOT AREA: point; ...; point

PLOT AREA plots the polygon defined by connecting the points and fills it with the current foreground color. The last point need not repeat the first point, as the line segment needed to close the polygon is automatically supplied.

PLOT TEXT Statement

PLOT TEXT, AT point: textstring$

PLOT TEXT plots the text string in the current color at the point specified in the AT clause.

Vacuous PLOT Statement

PLOT

PLOT LINES

PLOT LINES:

These statements turn off the beam in case a previous PLOT or PLOT LINES statement ended with a semicolon. They have no effect if the beam is already off.

PRINT Statement

PRINT

PRINT print-list

PRINT USING string: *using-list*

print-list:: printitem ... separator printitem
printitem ... separator printitem separator

using-list:: usingitem ..., usingitem
usingitem ..., usingitem ;

separator:: , or ;

Items in a print-list can be separated by commas or semicolons, and be followed by a final comma or semicolon. Items in a using-list can be separated only by commas, and be followed only by a semicolon.

The printitems are printed on the screen. Numeric values are printed with a trailing space and, for positive numbers, a leading space. String values are printed as is, with no additional leading or trailing spaces. If the separator between two items is a semicolon, then the items are printed juxtaposed. If the separator is a comma, then the next item is printed in the next print zone.

If a USING clause is present, the values are then printed according to the format specified, without regard to print zones. The string following the word USING determines the format.

If the PRINT statement ends with a semicolon, subsequent printing will occur immediately following on the same line. If the

PRINT statement ends with a comma, then subsequent printing will occur on the same line but in the next print zone. Otherwise, subsequent printing will start on the next line.

PROGRAM Statement

PROGRAM *program-name*

The PROGRAM statement, if used, must be the first statement of the main program, other than comment lines. For ordinary programs it serves no purpose other than to provide a place for the program name.

RANDOMIZE Statement

RANDOMIZE

The RANDOMIZE statement produces a new seed for the random number generator. It should not be used more than once in the running of a program.

READ Statement

READ variable, ..., variable

The READ statement assigns to its variables the next datum from the DATA list.

A string variable can receive any valid datum. A numeric variable can receive only a datum that is unquoted and is a valid numeric-constant.

REM Statement

REM character ... character

The REM statement allows you to add comments to your program. You can use any characters you want in the REM statement. REM statements are ignored.

A REM statement is equivalent to a comment line that begins with an exclamation mark (!). In addition, a (!) can be used to place comments on the same lines as other True BASIC statements.

RESET Statement

RESET #n: BEGIN

RESET #n: END

The RESET statement resets the data pointer to the beginning or end of the file opened as channel #n.

RESTORE Statement

RESTORE

The RESTORE statement resets the data pointer to the start of the data-list, and thus lets you reuse the data-list.

SELECT CASE Structure

SELECT CASE *select-expression*
CASE *case-specifier*
. . .
CASE *case-specifier*
. . .
CASE ELSE
. . .
END SELECT

case-specifier:: *case-part, ..., case-part*

case-part:: constant
constant TO constant
IS *relational-operator* constant

The SELECT CASE structure may have zero or more CASE parts, and zero or one CASE ELSE parts, but must have at least one of either a CASE or CASE ELSE part. The constants in a *case-specifier* must be of the same type (numeric or string) as the *select-expression* in the SELECT CASE statement.

The *select-expression* in the SELECT CASE statement is first evaluated. The *case-specifier* in the first CASE part is then examined. If it satisfies any of the *case-parts*, then the statements following that CASE statement are executed and control passes to the first statement following END SELECT.

If no *case-part* in the first CASE statement is satisfied, then the second CASE statement is examined in a like manner, and so on.

If no CASE statement is satisfied, then the statements following the CASE ELSE statement are executed. If no CASE statement is satisfied and there is no CASE ELSE part, an exception occurs.

SET COLOR Statement

SET COLOR colornumber
SET COLOR colorname

The SET COLOR statement sets the foreground color used in subsequent PLOT statements to the color number or name used in the statement. True BASIC always assigns 0 to the current background color (usually white).

With a monochrome monitor, 0 is the white background, 1 is the black foreground, and the numbers 2 through 5 give gray patterns going from dark to white; numbers greater than 5 turn to white.

SET COLOR MIX Statement

SET COLOR MIX (colornumber) red-intensity, green-intensity, blue-intensity

The SET COLOR MIX statement lets you adjust the shade of any color number. The intensity values may be any decimal fraction from 0 (off) to 1 (on). This statement has no effect with a monochrome monitor.

SET MODE Statement

SET MODE “graphics”

The SET MODE “graphics” statement forces True BASIC to use the full screen for graphical output on a Macintosh, even if you have opened an output window. For other uses of this command see the *True BASIC Bible*.

SET WINDOW Statement

SET WINDOW left, right, lower, upper

Sets the window coordinates for graphics in the current window.

SOUND Statement

SOUND frequency, seconds

The SOUND statement sounds a note with the specified frequency and duration.

STOP Statement

STOP

Stops execution of the program.

SUB Structure

```
SUB identifier (parm 1, ... , param n)
  ...
  EXIT SUB  [optional]
  ...
END SUB
```

The subroutine may contain one or more EXIT SUB statements. A CALL statement invokes the subroutine; that is, starts it running. The arguments in the CALL must match the parameter in the SUB statement in number, position, type, and number of dimensions. Parameter passing is **by reference**; that is, changes to the parameter within the subroutine will cause simultaneous changes to the arguments in the CALL statement.

WHEN ERROR IN Structure

```
WHEN ERROR IN
when-part
USE
use-part
END WHEN
```

The WHEN structure lets you provide statements to be carried out if a runtime error (exception) occurs. True BASIC uses the statements in the use-part when an error occurs in the statements in the when-part; otherwise, it skips the use-part statements.

Appendix C: True BASIC Built-in Functions

This appendix lists most of True BASIC's functions. Functions are first grouped by general type, and then listed alphabetically with a brief explanation. For complete information and examples of all built-in functions, see the *True BASIC Bible*.

Mathematical Functions

Function	Result
ABS(x)	Absolute value
ANGLE(x,y)	Angle between x-axis and (x,y)
ATN(x)	Arctangent
COS(x)	Cosine
DEG(x)	Translates radians to degrees
EXP(x)	Exponential function
FP(x)	Fractional part of x
INT(x)	Integer part
IP(x)	Greatest integer <= x
LOG(x)	Natural logarithm
LOG10(x)	Common logarithm (base 10)
LOG2(x)	Logarithm to the base 2
MAX(x,y)	Larger of two numbers
MIN(x,y)	Smaller of two numbers
MOD(x,y)	Remainder when x is divided by y
PI	Value of pi
RAD(x)	Translates degrees to radians
REMAINDER(x,y)	Remainder of x divided by y
RND	Random number between 0 and 1
ROUND(x,n)	Rounds x to n decimal places
SGN(x)	Sign of x
SIN(x)	Sine
SQR(x)	Square root
TAN(x)	Tangent
TRUNCATE(x,n)	Truncates x to n decimal places

Date and Time Functions

Function	Result
DATE	Year and day of year as a number
DATE$	Year, month, and day of month as a string
TIME	Seconds since midnight
TIME$	24-hour clock time as a string

String to Number Functions

Function	Result
CHR$(x)	Character represented by ASCII number x
ORD(x$)	Ordinal position of x$ in ASCII character set
STR$(x)	Changes number to a string
VAL(x$)	Changes string containing digits to a number

String Transforming Functions

Function	Result
LCASE$(x$)	Change letters to lowercase
UCASE$(x$)	Change letters to uppercase
LTRIM$(x$)	Remove leading blanks
RTRIM$(x$)	Remove trailing blanks
TRIM$(x$)	Remove leading & trailing blanks
REPEAT$(x$,n)	x$ repeated n times

String Search Functions

Function	Result
LEN(x$)	Number of characters in x$
POS(x$,y$,n)	First occurrence of y$ in x$ after character n
POSR(x$,y$)	Ditto POSR but starting from the end
CPOS(x$,y$)	First occurrence in x$ of any character in y$
CPOSR(x$,y$)	Ditto CPOS but starting from the end
NCPOS(x$,y$)	First occurrence in x$ of any character not in y$
NCPOSR(x$,y$)	Ditto, but starting from the end

Array Functions

Function	Result
DET(a)	Determinant for the square matrix a
DOT(a,b)	Dot product of arrays a and b
LBOUND(a,n)	Lower bound of dimension n for array a
UBOUND(a,n)	Upper bound of dimension n for array a
SIZE(a,n)	Number of element in dimension n of array a

MAT Functions *that can appear only in MAT assignment statements*

Function	Result
CON	Array of ones
IDN	Identity matrix
INV(a)	Inverse of array a
NUL$	Array of empty strings
TRN(a)	Transpose of array a
ZER	Array of zeroes

The descriptions in the alphabetical list use the following terms:

numeric-expression	numeric expression
*r*numeric-expression	rounded numeric expression
string-expression	string expression
redim	array redimensioning expression
arrayarg	array argument (array name with optional parentheses)

ABS Function

ABS(numeric-expression)

Returns the absolute value of the argument.

ANGLE Function

ANGLE(numeric-expression, numeric-expression)

ANGLE(x,y) returns the counterclockwise angle between the positive x-axis and the point (x,y). Note that *x* and *y* cannot both be zero. The angle will be given in radians or degrees depending on whether the current OPTION ANGLE is RADIANS (default) or DEGREES. The angle will always be in the range -180 < ANGLE(x,y) <= 180 (assuming that the current OPTION ANGLE is DEGREES).

ATN Function

ATN(numeric-expression)

ATN(x) returns the arctangent of x, which is the angle whose tangent is x. The angle will be given in radians or degrees according to whether the current OPTION ANGLE is RADIANS (default) or DEGREES. The angle will always be in the range -90 < ATN(x) < 90 (assuming that the current OPTION ANGLE is DEGREES).

CHR$ Function

CHR$(*r*numeric-expression)

Returns the character whose ASCII decimal number is *r*numeric-expression (see Appendix A). If *r*numeric-expression is not in the range 0 to 255, inclusive, it is adjusted modulo 256 before the function is evaluated.

CON Array Constant

CON *redim*
CON

CON is an array constant that yields a numeric array consisting entirely of ones. CON can appear only in a MAT assignment statement.

COS Function

COS(numeric-expression)

Returns the value of the cosine function. The argument is assumed to be in radians or degrees depending on whether the current OPTION ANGLE is RADIANS (default) or DEGREES.

CPOS Function

CPOS(string-expression, string-expression)

CPOS(string-expression, string-expression, *r*numeric-expression)

Returns the position of the first occurrence in the first argument of any character in the second argument. If no character in the second argument appears in the first argument, or either string is empty, then CPOS returns 0.

If a third argument is present, then the search for the first occurrence starts at the character position in the first string given by that number and proceeds to the right. The first form of CPOS is equivalent to the second form with the third argument equal to one.

CPOSR Function

CPOSR(string-expression, string-expression)

CPOSR(string-expression, string-expression, *r*numeric-expression)

Returns the position of the last occurrence in the first argument of any character in the second argument. If no character in the second argument appears in the first argument, or either string is empty, then CPOSR returns 0.

If a third argument is present, then the search for the last occurrence starts at the character position in the first string given by that number and proceeds to the left (that is, backwards). The first form of CPOSR is equivalent to the second form with the third argument equal to the length of the first argument.

DATE Function

DATE

DATE, a no-argument function, returns the current date in the decimal numeric form YYDDD, where YY is the last two digits of the year and DDD is the day number in the year. If your computer cannot tell the date, DATE returns -1.

DATE$ Function

DATE$

DATE$, a no-argument string-valued function, returns the current date in the character string form “YYYYMMDD”. Here

YYYY is the year, MM is the month number, and DD is the day number. If your computer cannot tell the date, then DATE$ returns “00000000”.

DEG Function

DEG(numeric-expression)

Returns the number of degrees in numeric-expression radians. This function is not affected by the current OPTION ANGLE.

DET Function

DET (numarr)
DET

Returns the value of the determinant for the square numeric matrix named as its argument.

DOT Function

DOT(*arrayarg*, *arrayarg*)

DOT computes and returns the dot product of two arrays, which must be one-dimensional, numeric, and have the same number of elements. (The subscript ranges need not be the same, however.) If both arrays have no elements, then DOT returns 0.

EPS Function

EPS(numeric-expression)

EPS(x) returns the smallest positive number that can “make a difference” when added to or subtracted from x.

EXP Function

EXP(numeric-expression)

Returns the natural exponential of the argument. That is, EXP(x) calculates e^x, where e = 2.718281828..., the base of the natural logarithms.

FP Function

FP(numeric-expression)

Returns the fractional part of the argument.

IDN Array Constant

IDN *redim*
IDN

IDN is an array constant that yields an identity matrix, which is a square numeric matrix consisting of ones on the main diagonal and zeroes elsewhere. IDN can appear only in a MAT assignment statement.

INT Function

INT(numeric-expression)

Returns the greatest integer that is less than or equal to numeric-expression.

INV Array Function

INV(numarr)

Returns the inverse of its argument, which must be a square two-dimensional numeric matrix. INV can appear only in a MAT assignment statement.

IP Function

IP(numeric-expression)

Returns the greatest integer that is less than or equal to numeric-expression without regard to sign, that is, towards zero.

LBOUND Function

LBOUND(*arrayarg*, *r*numeric-expression)
LBOUND(*arrayarg*)

If there are two arguments, LBOUND returns the lowest value (lower bound) allowed for the subscript in the array and in the dimension specified by *r*numeric-expression. If there is no second argument, *arrayarg* must be one-dimensional array, and LBOUND returns the lowest value (lower bound) for its subscript.

LCASE$ Function

LCASE$(string-expression)

Returns the value of string-expression with all ASCII uppercase letters converted into lowercase. Characters outside the range of the ASCII uppercase letters are unchanged.

LEN Function

LEN(string-expression)

Returns the length (that is, the number of characters) of the argument string-expression. All characters count, including control characters and other nonprinting characters.

LOG Function

LOG(numeric-expression)

Returns the natural logarithm of numeric-expression, which must be greater than 0. The natural logarithm of *x* may be defined as that value *v* for which e^v = x, where e = 2.718281828....

LOG10 Function

LOG10(numeric-expression)

Returns the common logarithm of numeric-expression, which must be greater than 0. The common logarithm of *x* is defined as that value *v* for which 10^v = x.

LOG2 Function

LOG2(numeric-expression)

Returns the logarithm to the base 2 of numeric-expression, which must be greater than 0. The logarithm to the base 2 of *x* is defined as that value *v* for which 2^v = x.

LTRIM$ Function

LTRIM$(string-expression)

Returns the value of string-expression but with leading blank spaces removed. Trailing spaces, if any, are retained.

MAX Function

MAX (numeric-expression, numeric-expression)

Returns the larger of the values of the two arguments.

MAXNUM Function

MAXNUM

A no-argument function, MAXNUM returns the largest number that can be represented in your computer.

MIN Function

MIN (numeric-expression, numeric-expression)

Returns the smaller of the values of the two arguments. (Note: -2 is smaller than -1.)

MOD Function

MOD(numeric-expression, numeric-expression)

MOD(x,y) returns *x* modulo *y*, provided *y* is not equal to zero.

NCPOS Function

NCPOS(string-expression, string-expression)

NCPOS(string-expression, string-expression, numeric-expression)

Returns the position of the first occurrence in the first argument of any character that is not in the second argument. If all characters in the first argument appear in the second argument, or the

first argument is empty, then NCPOS returns 0. If the second argument is empty but not the first, then NCPOS returns 1.

If a third argument is present, then the search for the first non-occurrence starts at the character position in the first string given by that number and proceeds to the right. If the second argument is empty but not the first, then NCPOS returns the starting position.

The first form of NCPOS is equivalent to the second form with the third argument equal to one.

NCPOSR Function

NCPOSR(string-expression, string-expression)

NCPOSR(string-expression, string-expression, numeric-expression)

Returns the position of the last occurrence in the first argument of any character that is not in the second argument. If all characters in the first argument appear in the second argument, or if the first argument is empty, then NCPOSR returns 0. If the second argument is empty but not the first, then NCPOSR returns the length of the first string.

If a third argument is present, then the search for the last non-occurrence starts at the character position in the first string given by that number and proceeds to the left (that is, backwards). If the second argument is empty but not the first, then NCPOSR returns the starting value.

The first form of NCPOSR is equivalent to the second form with the third argument equal to the length of the first argument.

NUL$ Array Constant

NUL$ *redim*

NUL$

NUL$ is an array constant that yields a string array consisting entirely of empty strings. NUL$ can appear only in a MAT assignment statement.

ORD Function

ORD(string-expression)

Returns the ordinal position in the ASCII character set of the character given by string-expression, which must be either a single character or an allowable two- or three-character name of certain ASCII characters as described in Appendix A, except that ORD(“”) = -1. ORD is the opposite of the CHR$ function in that ORD(CHR$(n)) = n for all n in the range 0 to 255. However, CHR$(ORD(a$)) = a$ only if the value of a$ is a single ASCII character.

PI Function

PI

A no-argument function, PI returns the value of pi, the ratio of a circle's circumference to its diameter (approximately equal to 3.14159265). It gives as much accuracy as possible on your computer, but in any case at least ten decimal places.

POS Function

POS(string-expression, string-expression)

POS(string-expression, string-expression, numeric-expression)

Returns the position of the first character of the first occurrence of the entire second string in the first string. If the second string does not appear in the first string, or if the first string is empty while the second is not, then POS returns 0. If the second string is empty, then POS returns 1.

If a third argument is present, then the search for the second string starts at that character position in the first string given by that number and proceeds to the right. If the second string is empty, POS returns the starting position. The first form of POS is equivalent to the second form with the third argument equal to one.

POSR Function

POSR(string-expression, string-expression)

POSR(string-expression, string-expression, numeric-expression)

Returns the position of the first character of the last occurrence of the entire second string in the first string. If the second string does not appear in the first string, or if the first string is empty but the second is not, POSR returns 0. If the second string is empty, then POSR returns the length of the first string plus one.

If a third argument is present, then the search for the last occurrence starts at the character position in the first string given by that number and proceeds to the left (that is, backwards). If the second string is empty, POSR returns the starting position.

The first form of POSR is equivalent to the second form with the third argument equal to the length of the first argument plus one.

RAD Function

RAD(numeric-expression)

RAD(x) returns the number of radians in *x* degrees. This function is not affected by the current OPTION ANGLE.

REMAINDER Function

REMAINDER(numeric-expression, numeric-expression)

REMAINDER(x,y) returns the remainder obtained by dividing *x* by *y*; *y* must not be equal to 0.

REPEAT$ Function

REPEAT$(string-expression, numeric-expression)

Returns the string consisting of *r*numeric-expression copies of string-expression.

RND Function

RND

A no-argument function, RND returns the next "pseudo-random" number in the sequence. These numbers, which have no obvious pattern, fall in the range 0 < = RND < 1. If the program containing RND is rerun, True BASIC produces the same sequence of RND values. If you want your program to produce unpredictable results, use a RANDOMIZE statement before you use the RND function.

ROUND Function

ROUND(numeric-expression, numeric-expression)

ROUND(numeric-expression)

ROUND(x,n) returns the value of *x* rounded to *n* decimal places. Positive values of *n* round to the right of the decimal point; negative values round to the left. ROUND(x) is the same as ROUND(x,0).

RTRIM$ Function

RTRIM$(string-expression)

Returns the value of string-expression but with the trailing blank spaces removed. Leading spaces, if any, are retained.

SGN Function

SGN(numeric-expression)

SGN(x) returns the "sign" of *x*.

SIN Function

SIN(numeric-expression)

Returns the sine of the angle numeric-expression. The angle is measured in radians unless OPTION ANGLE DEGREES is in effect, in which case the angle is measured in degrees.

SIZE Function

SIZE(arrayarg, numeric-expression)

SIZE(arrayarg)

If there are two arguments, SIZE returns the number of elements in the array named in the first argument and in the dimension specified by *r*numeric-expression. If there is no second argument, then SIZE returns the total number of elements in the entire array.

SQR Function

SQR(numeric-expression)

SQR(x) returns the positive square root of *x*, where *x* must be greater than or equal to zero.

STR$ Function

STR$(numeric-expression)

Returns the number converted to a string.

TAB Function

TAB(numeric-expression)

TAB(numeric-expression, numeric-expression)

TAB can appear only in PRINT statements. Strictly speaking, TAB is not a function, as it does not return a value.

TAB(c) causes the printing cursor to "tab" over to print position (column) c*c* TAB(r,c) causes the printing cursor to be positioned on the screen at row *r* and column *c* of the current window.

TAN Function

TAN(numeric-expression)

TAN(x) returns the tangent of *x*. Here, *x* is assumed to be in degrees if OPTION ANGLE DEGREES is in effect, and in radians otherwise.

TIME Function

TIME

A no-argument function, TIME returns the number of seconds since midnight. At midnight, TIME returns 0. If your computer does not have a clock, then TIME returns -1.

TIME$ Function

TIME$

A no-argument function, TIME$ returns a string that contains the time as measured by the 24-hour clock.

TRIM$ Function

TRIM$(string-expression)

The value of the argument returned with leading and trailing blank spaces removed.

TRN Array Function

TRN(numarr)

Returns the transpose of its argument, which must be a two-dimensional numeric array. TRN can appear only in a MAT assignment statement.

TRUNCATE Function

TRUNCATE(numeric-expression, numeric-expression)

TRUNCATE(x,n) returns the value of *x* truncated to *n* decimal places. Positive values of *n* truncate to the right of the decimal point; negative values truncate to the left. TRUNCATE(x,0) is the same as IP(x).

UBOUND Function

UBOUND(arrayarg, numeric-expression)

UBOUND(arrayarg)

The two-argument form returns the largest value (upper bound) allowed for the subscript in the dimension specified by *r*numeric-expression in the array named. The one-argument form returns the largest value (upper bound) for the subscript in a one-dimensional array.

UCASE$ Function

UCASE$(string-expression)

Returns the value of string-expression with all lowercase letters in the ASCII code (see Appendix A) converted into their uppercase equivalents. Characters outside the range of the ASCII lowercase letters are unchanged.

USING$ Function

USING$(string-expression, expr ..., expr)

expr:: numeric-expression

string-expression

USING$ returns the string of characters that would be produced by a PRINT USING statement with string-expression as the format string and with the *exprs* as the numeric or string expressions to be printed.

VAL Function

VAL(string-expression)

Returns the numerical value given by I>string-expression, provided it represents a numerical constant in a form suitable for use with the INPUT or READ statement. The string can contain leading and trailing spaces, but not embedded ones.

ZER Array Constant

ZER *redim*

ZER

ZER is an array constant that yields a numeric array consisting entirely of zeros. ZER can appear only in a MAT assignment statement.

Appendix D: Error Messages

This appendix contains a partial list of True BASIC error messages, in alphabetic order. Error messages referring to statements or features not introduced in this book are omitted.

The number following some messages is the error number for error (exceptions) that occur when the program runs. These numbers can be used with the WHEN structure and EXTYPE function explained in *True BASIC Bible*.

Argument types don't match.

You're calling a routine with some arguments, but earlier in your program you defined or called the same routine with different arguments. Either you're giving a different number of arguments in the calls, or their types are different – that is, you're passing strings instead of numbers, or vice versa. Check this call against preceding calls, and against the routine's definition.

Bad FIND item; try using quotes.

When you're trying to find a string which contains a comma or quotation marks, you must enclose the entire string within quote marks. (These rules are the same as the rules for strings in INPUT replies or DATA statements.)

Badly formed input line. (8102)

Your reply to an INPUT statement (either from a file or from the keyboard) is badly formed. Most likely you have not properly matched up opening and closing quote marks.

Can't continue.

You've just given a CONTINUE command, to resume running a suspended program. However, True BASIC cannot continue the program. There are several possible reasons. You cannot continue a program that you haven't yet started running, or one which you've just changed. You cannot continue a program which stopped because an error occurred. And you cannot continue a suspended program after using a DO command. If you are trying to debug a program which stopped because of an error, try using the BREAK command to insert breakpoints before the erroneous line, and then run the program again.

Can't help with that. Try HELP TOPICS.

You've asked for help, with either the HELP command or the HELP key. True BASIC can't help you with the topic you've requested. Make sure that you've spelled the topic name

correctly, and in full. And make sure that you've got the True BASIC disk in your computer, or that the TB Help folder was copied to your hard disk.

Can't invert singular matrix. (3009)

You are using the matrix INV function, but the matrix you want to invert is singular. Singular matrices simply have no inverses.

Can't PRINT to middle of text file. (7350)

You can not overwrite data in a text file. Use the RESET statement to move to the end of the file, or ERASE the file, before printing to it.

Can't SET WINDOW in picture. (11004)

Pictures may not reset window or screen coordinates. Move the OPEN SCREEN or SET WINDOW statement to outside the picture.

Can't use #0 here. (7002)

You may not use channel #0 in OPEN or CLOSE statements, since #0 is always open.

Can't use ANGLE(0,0). (3008)

ANGLE(0,0) is not defined. Make sure that at least one of its arguments is nonzero.

Can't use this statement here.

You've used part of a True BASIC structure, but in the wrong place. For instance, you might have placed a CASE part outside of any SELECT CASE statement, or ELSE IF statement outside of any IF-THEN statement. True BASIC also prints this message if you add an extraneous statement between the SELECT CASE line and its first CASE part. Refer to the proper chapters of this manual to see how the structured statements are formed.

Channel is already open. (7003)

You are trying to open a channel that is already open. Make sure you are not already using that channel number somewhere else in your program. Also, remember to CLOSE a channel when you're done using it.

Channel isn't open. (7004)

You're trying to use a channel that you haven't yet opened. Be sure you've used the correct channel number and that you've used an OPEN statement for that channel before you attempt to use it. Also, be sure you haven't already closed that channel (with a

CLOSE statement). If the channel was opened in a different program unit, be sure that the channel was passed as an argument.

Channel number must be 1 to 1000. (7001)

Channel numbers must lie in the range 1 to 1000.

Constant too large: constant in routine.

The numeric constant displayed is too large for your computer to handle. Type PRINT MAXNUM in the command window to see the largest possible number on your computer, and then change your program to use a smaller number.

DET needs a square matrix. (6002)

The DET function can only be used on a square matrix, since the determinant is mathematically defined only for such matrices.

Disk full. (9006)

You are writing output to a file, and the disk has become full.

Diskette removed, or wrong diskette. (9005)

You had opened a file, but, while True BASIC was using it, you removed the diskette and inserted another one. Don't switch diskettes while they're in use!

Division by zero. (3001)

One of your expressions tried to divide some quantity by zero. If you want to substitute the largest possible number and continue (without an error), enclose the expression in a WHEN statement:

```
WHEN ERROR IN
     LET x = (1+2+3)/0
USE
     LET x = Maxnum
END WHEN
```

MAXNUM is a True BASIC function which gives the largest positive number available on your computer.

Do you want to save this file?

True BASIC gives you this reminder when you try to Open another file, start a New file, or Quit your True BASIC session without saving your current file. Answer "yes" if you do want to save the file (replacing the current saved copy), "no" if you want to discard your changes, or "cancel" if you want to do something else (for example, save the file with a different name).

Doesn't belong here.

The cursor points to some word in your program which doesn't make sense. Look to see what kind of statement you are using, and then look up the proper form of that statement in this book. Then correct your program and continue.

Ending doesn't match beginning.

You are using a structured statement, such as FOR-NEXT or IF-THEN-ELSE, and the ending statement doesn't properly match the beginning of the structure. Most likely you have forgotten the ending statement for some structure within this one. Or you may have begun a FOR loop using one index variable, but used another variable on the NEXT statement. Read the statements inside the structure carefully to see what you've left out.

Error in PLAY string. (4501)

The string given in your PLAY statement doesn't follow True BASIC's rules.

Expected "thing".

The cursor points to a spot where True BASIC expected some word or punctuation, but found something else. This message may jog your memory enough so that you can repair the statement. Otherwise, look up the statement in this manual, and then fix your program.

Expected relational operator.

The cursor points to a spot where you must put a relational operator, such as = or <. Finish writing out the comparison which must be there. (Note that True BASIC does not allow testing statements like IF A THEN ..., as some other BASICs do. Change such statements to IF A<>0 THEN ....)

File already exists. (9004)

Your are trying to create a file that already exists. Check to make sure that you've given the right name. If you want to use an existing file, change the CREATE NEW i the OPEN statement to CREATE NEWOLD.

File is read or write protected. (9001)

You are trying to input from or output to a file that has write- or read-protection. Quit True BASIC and check the file and diskette. If the write-protect tab on the diskette is open, close that. Also, select the file's icon and use **Get Info** in the Macintosh **File** menu to be sure the "Locked" box is not checked.

IDN must make a square matrix. (6004)

Identity matrices must be square. Therefore, when you use the IDN(x,y) function, you must make sure that x = y.

Illegal array bounds. (6005)

You've redimensioned an array in a MAT *REDIM* statement or with a *redim*-expression in a MAT statement where the upper bound is less than the lower bound minus one (e.g., MAT A = Zer(-5) or MAT *REDIM* X(10 to 5). True BASIC allows the lower bound to exceed the upper bound by one – thus defining an array with no elements.

Illegal array bounds for name in routine.

You've defined an array in a DIM, LOCAL, SHARE, or PUBLIC statement with an upper bound less than the lower bound minus one. (True BASIC allows the lower bound to exceed the upper bound by one, thus defining an array with no elements.)

Illegal data.

Your DATA statement is not properly written. Put commas between data items, but don't put a comma at the end of the list of items. Make sure that all quoted items are properly enclosed in quote marks: items such as "abc" def are not allowed.

Illegal expression.

The cursor points to something in an expression that doesn't follow True BASIC's rules. Check to make sure that you haven't given two operators in a row (such as "1++2"), that you haven't written down a number improperly (such as "1,000"), and that all your variable names follow True BASIC's rules.

Illegal keyword.

The cursor points to a word that doesn't make sense in that location. For instance, you may have forgotten to add LINES, AREA, or CLEAR in a BOX statement. Look up the statement in this book, and correct your program.

Illegal line number.

You might have a non-numbered line in a line-numbered program, or vice versa, or a GOTO or GOSUB to a nonexistent line number, or one in a control structure. You might have a badly formed line number (e.g., more than six digits). Or you might have a line with a number less than or equal to the previous line.

Illegal number.

The cursor points to some spot where a number is required, but you've given something else. `If you've written a number there, make sure that you've followed True BASIC's rules on numeric constants. Sometimes True BASIC is very finicky about what it will accept as a number: for instance, only integer constants are allowed as array bounds in DIM statements, and as line numbers.

Illegal option.

The only options supported by True BASIC are OPTION ANGLE, OPTION BASE, OPTION NOLET, and OPTION TYPO. Make sure you've spelled ANGLE, BASE, DEGREES, RADIANS, NOLET, or TYPO properly.

Illegal parameter.

You've written a SUB or DEF or PICTURE line defining a routine, but something is wrong with one of the parameters in the parameter list. You may have listed one parameter twice, or used something more complicated than a simple variable name.

Illegal statement.

Each statement must begin with some True BASIC keyword, such as LET or SELECT. Check to make sure that you've spelled the keyword properly.

Illegal statement: need LET for assignment, or try the NOLET command.

This is a wordier version of the "Illegal statement" error message if it looks like an assignment. Unless you use OPTION NOLET, True BASIC requires that you use the word LET when assigning to a variable.

Improper NUM string. (4020)

The string you've given to the NUM function doesn't represent an IEEE 64-bit floating point number. Check to make sure that you've correctly created, or read in, the string.

Improper ORD string. (4003)

The ORD function requires either a one-character string, or a string giving the official name of an ASCII character. No leading or trailing spaces are allowed. See Appendix A for a list of all the legal names for ASCII characters.

INV needs a square matrix. (6003)

Matrix inversion is defined only for square matrices. You are trying to use the INV function on a non-square matrix. See that matrix is two-dimensional, with the same size in each dimension.

LBOUND index out of range. (4008)

You are using a call such as Lbound(A,3) and the array A doesn't have three dimensions. Check to make sure that the dimension given lies between 1 and the number of dimensions in the array.

LOG of number <= 0. (3004)

Logarithms are only defined for positive numbers.

Mismatched array sizes. (6001)

You're using a MAT statement that requires arrays of the same size, but the arrays are different sizes. For example, matrix addition requires the two arrays added together to have the same sizes. Matrix multiplication has slightly more complicated rules. See the *True BASIC Bible* for more information about arrays.

Missing end statement.

Your program doesn't end with an END statement. All True BASIC programs must contain END statements. Add an END statement and try again.

MOD and REMAINDER can't have 0 as 2nd argument. (3006)

The MOD and REMAINDER functions do not allow zero as their second argument, since this is equivalent to dividing by zero. Check to make sure you're giving the arguments in the right order.

Must be a function name.

You've written a DEF or FUNCTION line, but no proper function name follows the DEF or FUNCTION.

Must be a number.

True BASIC allows numeric expressions almost anywhere that simple numbers are allowed, but there are a few exceptions. For instance, CASE tests may not use numeric expressions. Only numeric constants are allowed. If you must use an expression, rewrite the SELECT CASE structure as an IF-THEN-ELSE structure.

Must be a picture name.

Your DRAW statement names something other than a picture. Change the DRAW statement so it refers to a picture, and try again.

Must be a string constant.

True BASIC allows string expressions almost anywhere that string constants are legal, but there are a few exceptions. For

instance, CASE tests may not use string expressions. If you must use a string expression, rewrite the SELECT CASE structure as an IF-THEN-ELSEIF structure.

Must be a subroutine name.

The CALL statement can only be used to call subroutines. Change the statement so it uses a subroutine name.

Must be a variable.

You've used an expression, or a routine name, where only a variable will do. For example, you must use variables in LET and INPUT statements. Look up the statement in this book to make sure you are using it properly. Also make sure that the variable you're using isn't already used as a subroutine, picture, function, or array.

Must be an array.

There are many places in True BASIC where you must give an array's name, instead of an ordinary variable. For instance, the MAT statements work only on arrays. Various functions, such as Lbound and Size, also work only on arrays. Make sure that you're spelling the array's name correctly and that you've named the array in a DIM statement.

Name can't be redefined.

You can't use the same name for two different things. Thus, if you have a variable named X, you cannot also have a subroutine or array named X. Rename one of the things, so everything has its own unique name. True BASIC also prints this message when you try to use a "reserved word" as a variable.

Negative number to non-integral power. (3002)

You're trying to compute n^x, but n is negative and x is not an integer. The results are mathematically meaningless.

No CASE selected, but no CASE ELSE. (10004)

You have executed a SELECT CASE statement, but no CASE test has succeeded. Since you didn't have a CASE ELSE part to catch this problem, True BASIC prints this error message. Check to make sure that the expression you've selected is reasonable. Add a CASE ELSE part to handle all cases other than ones caught by the tests. If you want to ignore anything besides those things tested for, add a CASE ELSE part with no statements in it.

No main program.

Your current file contains functions, pictures, and/or subroutines, but doesn't contain a main program. Create a main program!

No such color. (11008)

You're using the SET COLOR statement with some color name that True BASIC doesn't recognize. You may give color names in upper- or lowercase, but may not use extra spaces in the names.

No such file. (9003)

You're trying to use a file which doesn't exist. You can get this error message from various commands (such as OLD), or from within a program. Check to make sure you spelled the program's name properly, and to make sure you have inserted the correct disk in your computer. Use the FILES command to see if that file exists on a disk. See also Appendix E.

No such file. Do you want to create it?

You have tried to REPLACE a file which doesn't yet exist. This gives you the chance to create a file with the name you specified. Answer "yes" to create the file, or "no" or "cancel" to cancel this command. If you're typing the reply, you can abbreviate it to one letter.

No such function or subroutine.

You've named a function, subprogram, or picture in some command, but this routine doesn't exist. For instance, you may have typed LIST MYFUNC. Check to make sure you spelled the name properly.

No such line numbers.

You've given a range of line numbers in a command, but no lines have those numbers.

Not found.

You've used the FIND menu item to find some word or phrase, but it wasn't found. Reread the section in this book on FIND to be sure you have given the search phrase properly. Have you asked to find whole word? Is this an instance when case much match? Remember that True BASIC searches from where you are in the program to the end and then stops.

Out of memory. (5000)

Your problem requires more memory than is attached to your computer. Since True BASIC will use all the memory supplied with your computer, you may be able to fix this problem by buying more memory. Otherwise, you must try to use less memory. Here are a few suggestions.

Use smaller arrays. Arrays can take up a surprising amount of space, especially if they have more than one dimension. If you

have big arrays, see if you can solve your problem using smaller arrays.

Compile your program, and use the compiled version. See the True BASIC Bible for information on compiling.

Check for "run-away" calls. You may have accidentally written a procedure that calls itself. This is perfectly legal, and often useful. But each call requires some amount of space, and such an accident can cause this error.

Overflow. (1000)

You've computed a number bigger than the one your computer can handle. PRINT MAXNUM to see the largest number that your computer can use. If you wish to have overflows silently turned into the largest possible number, enclose your computation in a WHEN structure:

```
WHEN ERROR IN
     LET x = 10^(10^10)
USE
     LET x = Maxnum
END WHEN
```

Please try "CHANGE old, new".

When changing a phrase in the command window, you must give both the old phrase and its replacement. If either phrase contains a comma or quote mark, enclose that entire phrase in quote marks.

Please try "DO filename".

You must give a filename when using the DO command in the command window. Give the command again, specifying the name of the file to execute.

Please try "ECHO" or "ECHO TO filename" or "ECHO OFF".

You probably gave the ECHO command without the keyword TO.

Please try "INCLUDE filename".

You must give a filename when using the INCLUDE command. Retype the command, giving the name of the file to include.

Please try "LOCATE item".

When trying to locate a phrase in the command window, you must give the phrase to be located.

Please try "OLD filename".

You must give a file name when using the OLD command in the command window. Retype the command, giving the name of the file to call up.

Please try "RENAME new" or "RENAME old, new".

You gave the RENAME command in the command window without specifying a filename. Give one name to change the current program name. Or give two names (old and new) to change a saved file's name.

Please try "SAVE filename" or "REPLACE filename".

You must give a filename when saving a file in the command window. Retype the command, giving a filename.

Please try "UNSAVE filename".

You must give a filename when trying to unsave a file in the command window. Retype the command, giving the name of the file to unsave.

Please type line numbers as 100 or 100-150.

You've given a command such as DELETE, with a line number or block of line numbers. But True BASIC can't understand what you said. Type a command such as DELETE 100 to delete line 100, or DELETE 100-120 to delete lines 100 through 120.

Program stopped.

You pressed the ⌘-. key or selected **Stop** from the main menu. The program stopped. There is no way your program can try to intercept a "stop".

Reading past end of data. (8001)

You've executed a READ statement, but have run out of DATA items to read. Did you remember to include a DATA statement? Check to make sure that you have as many data items as you expect. You may find the MORE DATA test handy for dealing with variable amounts of data.

Reading past end of file (8011)

You're trying to input more than exits in the file. Check to see if the file contains everything you think it should. You may find the MORE or END tests useful.

REPEAT$ count < 0. (4010)

You're using the REPEAT$(s$,n) function, but n is less than zero. Check to make sure that you've typed the right variable name.

Screen bounds must be 0 to 1. (11003)

The bounds given on an OPEN SCREEN statement must lie in the range 0 to 1 (inclusive). No matter how big your screen is, the left and bottom edges are defined to be 0; the right and top edges

are defined to be 1. See Chapter 12 for a description of how to open windows on the screen.

SIZE index out of range. (4004)

You're trying to take Size(A,3), for instance, when the array A has fewer than three dimensions. Check the relevant DIM statement to see how many dimensions the array has. The second argument must lie between 1 and this number.

Sorry, can't find HELP files.

The help files are not where True BASIC expects to find them. Either they are missing, or they are in another folder.

SQR of negative number. (3005)

You are trying to take the square root of a negative number. This is not possible.

Statement outside of program.

The cursor points to a statement outside of your main program, and not included within any external routine. Check to make sure you haven't accidentally moved the END statement so that it is no longer at the end of your program.

String given instead a number. (8103)

You've executed an INPUT statement which is trying to input a number. However, the reply given isn't a number – it only makes sense as a string. If you're inputting from the keyboard, and want to avoid this message, you should convert your input statement so it reads a string, and then use the VAL function to convert the result to a number. (You can enclose the call to VAL within an error handler to suppress the error message.) If this exception occurs, you will be requested to reenter the entire input line.

String too long. (1051)

You've created a string longer than the maximum size allowed on your computer.

Subscript out of bounds. (2001)

You've given an array subscript which lies outside the array's bounds. Try printing the subscript and then using Lbound and Ubound to find the array's bounds.

The BYE command is just "BYE".

When you want to leave True BASIC in the command window, just type BYE. Don't add anything else.

The CONTINUE command is just "CONTINUE".

When you want to continue running a suspended program, just type CONTINUE. Don't add anything else.

The FORGET command is just "FORGET".

When you want to "forget" the history or recent commands, delete loaded routines, and recover as much memory as you can, just type FORGET. Don't add anything else.

The NOLET command is just "NOLET".

When you want to allow the keyword LET to be omitted from LET statements, just type "NOLET". Don't add anything else.

The RUN command is just RUN.

When you want to run your program from the command window, just type "RUN". Don't add anything else.

This must first appear in a DIM or DECLARE DEF.

The cursor points to something that is evidently an array or a function. But True BASIC can't tell which it is. Be sure to add a DIM or DECLARE DEF line before this line, so True BASIC will know what it is.

Too few input items. (8002)

You've executed an INPUT statement, and the input reply doesn't contain as many items as the INPUT statement requested. If input is coming from the keyboard, the "?" is repeated, and you should just add more items. For file input, check the file to see if you omitted any items from the input line. If you want to spread out input items over several lines, be sure to end all lines but the last with a comma.

Too many input items. (8003)

You've executed an INPUT statement, and the input reply line contains more items than the INPUT statement requested. If the reply came from the keyboard, the excess items are ignored. If you are reading input from a file, check the file to make sure it contains the right number of items on a line. If any input items contain commas, be sure to enclose those items in quote marks.

Trouble using disk or printer. (9002)

True BASIC is having trouble using one of your disks or your printer. This message is given for various reasons on different computers. Check to make sure that the power is turned on, that a diskette is inserted in your disk drive, that your printer has sufficient paper and that it's not jammed, that the connecting cables are securely attached, and so forth.

Try "LOAD lib, lib, ...".

You have probably used incorrect punctuation in a LOAD command.

Type is wrong for name in routine.

You've tried calling a routine named *name* within another routine named *routine*. However, you got the arguments wrong in this call. They don't match the parameter list. You must give the same number of arguments as parameters, and they must be given in the same order. Check for passing numbers to strings, or vice versa. Also make sure that you're not trying to use a function as a subroutine, or vice versa.

UBOUND index out of range. (4009)

You've tried calling something like Ubound(A,3), where A is an array with less than 3 dimensions. Check the DIM statement for A to see how many dimensions it has, or if you might have used UBOUND without specifying a dim.

Undefined routine name in routine.

The routine named *name* has tried to use a function, subprogram, or picture named *name*. Unfortunately, this function, subprogram, or picture is nowhere defined. Check to see that you spelled the name correctly, and that you included a LIBRARY statement for the file which contains this routine.

True BASIC says "in MAIN program" if the error occurred in your main program.

Unknown OPEN option. (7101)

The option you gave after CREATE in the OPEN statement doesn't exist. Check that you've spelled the option correctly. Although you can write the option in any mixture of upper- and lowercase letters, you may not abbreviate options or include extra spaces.

Unknown variable.

You are using OPTION TYPO to check for spelling mistakes, and it has found a variable name that you haven't declared anywhere. If True BASIC has found a typing mistake, just correct the spelling. Otherwise, add a LOCAL statement that lists this variable, or include the variable in its correct DECLARE PUBLIC or SHARE statement.

VAL string isn't a proper number. (4001)

You've called the VAL function, but the string you gave doesn't properly represent a number.

What? (Please type HELP.)

You've typed a command that True BASIC doesn't understand. If you want further help from the computer, type HELP in the command window or use the Help menu for more instructions. You may abbreviate commands to three letters, but not fewer.

Window minimum = maximum. (11001)

You've executed a SET WINDOW statement that sets the vertical or horizontal window maximum equal to the minimum. True BASIC doesn't allow this, as it wouldn't let you see anything in that window. Remember that the order of edges for the SET WINDOW command is left, right, bottom, top.

Wrong number of arguments.

A function, subprogram, or picture was called with the wrong number of arguments.

Wrong number of dimensions.

You're trying to use an array, but have given the wrong number of dimensions. Check this use against the array's DIM statement, and make sure that both have the same number of subscripts. If you're passing an array to a routine, check the routine's parameters. Remember that a two-dimensional array must be indicated as A(,) in the parameter list, a three-dimensional array by A(,,) and so forth.

Wrong type.

You're trying to use a string where a number is needed, or a number where a string is needed. Check to make sure you're not trying to assign a number to a string variable, or vice versa. Remember, too, that string concatenation is written using an ampersand (&) in True BASIC, and not a plus sign (+).

You have two routines called name in routine.

In the routine named *routine*, you've defined two different routines named *name*. Since different things must have different names, you must change the name of one of them. Be sure to go through all calls to that routine, and change those names too.

True BASIC says "in MAIN program" if the error occurred in your main program (before the END statement).

Zero to negative power. (3003)

You are trying to compute 0^n, where n < 0. This is mathematically undefined, and so True BASIC gives an error.

This is just the beginning . . .

This **True BASIC *Free*** edition is provided by True BASIC Inc. in the hopes that it will introduce you to the exciting and wide-ranging possibilities of computer programming.

If you find it useful and exciting, please preview the commerical versions of the True BASIC Language System and a variety of instructional books that are listed at our website, http://www.truebasic.com.

The site also offers memberships in the **True BASIC Institute**. As an Institute member you can download how-to information and sample programs that will improve your skills and understanding of programming.