## Contents

1. **An Introduction to Programming** ................................................................. 9

2. **Why True BASIC?** .......................................................................................... 11

3. **Installing True BASIC and Running Demo Programs** ................................. 13
   - Installing the BRONZE Edition on Windows .............................................. 13
   - Installing the BRONZE Edition on the MacOS ......................................... 17
   - Using True BASIC BRONZE Edition ......................................................... 18
   - Menu Access ............................................................................................. 19
   - Running a Demonstration Program ......................................................... 20

4. **Writing and Running Your First Program** ................................................. 23
   - Creating a Program .................................................................................. 23
   - The LET Statement .................................................................................. 25
   - The PRINT Statement ............................................................................ 26
   - The END Statement .............................................................................. 26
   - How True BASIC Runs a Program .......................................................... 26
   - Saving Your Program ............................................................................. 26

5. **Modifying and Saving Programs** ............................................................... 27
   - Using Source and Output Windows .......................................................... 27
   - Making Simple Changes ......................................................................... 28
   - Adding Comments to Your Program ....................................................... 28
   - Saving Your Changes ............................................................................. 29
   - The INPUT Statement ............................................................................ 29
   - Saving Your Program With A Different Name ......................................... 31
   - Opening or Quitting without Saving ...................................................... 31

6. **Constants, Variables, and Expressions** .................................................... 33
   - Constants ............................................................................................... 33
   - Variables ............................................................................................... 34
   - Expressions and Formulas ...................................................................... 35
   - Changing Values of Variables ............................................................... 37
   - An Introduction to Strings ...................................................................... 38
   - Using String Constants and Variables .................................................... 39
   - A Brief Look at String Expressions ...................................................... 40
7. More on INPUT and OUTPUT ................................................................. 41
   Printing Zones and the PRINT Statement ............................................... 41
   More about Controlling Output ............................................................... 43
   More about the INPUT Statement ........................................................... 43
   The LINE INPUT Statement .................................................................. 44
   The TD_LineInput Subroutine ................................................................. 45

8. Loop Structures .................................................................................... 47
   How a FOR-NEXT Loop Works ................................................................. 47
   Step Size in a Loop .................................................................................. 48
   Nested Loops ............................................................................................ 50
   An Introduction to Conditions ................................................................. 52
   An Introduction to DO Loops and Counters ............................................ 53
   Variations on DO Loops, and Combining Conditions............................... 55

9. Decision Structures ................................................................................ 57
   Simple IF-THEN Decisions .................................................................... 58
   Single-line IF-THEN-ELSE Decisions .................................................. 58
   Multiple-line Decisions .......................................................................... 58
   More About Counters ............................................................................ 60
   The RANDOMIZE Statement .................................................................. 61
   The STOP Statement .............................................................................. 61
   Generating Random Whole Numbers .................................................... 62
   Other Decision Structures ....................................................................... 63

10. Formatting and Printing Your Program ............................................... 65
    Guidelines for Good Programming ........................................................ 65
    Indenting with DO Format ...................................................................... 66
    Indenting Blocks with > and < ............................................................. 67
    Listing Your Programs on a Printer .................................................... 68
    Listing Output from your Programs .................................................... 68
    Using Line Numbers ............................................................................ 69
    Using the Command Window .................................................................. 69

11. Editing Hints and Shortcuts ................................................................. 71
    Undoing ................................................................................................. 71
    Selecting, Deleting, Moving, and Copying .......................................... 77
    Find and Change .................................................................................. 73
    Keep and Include .................................................................................. 75
    Select All and Move To... ...................................................................... 76
12. Using and Storing Data ................................................................. 77
   The DATA and READ Statements ..................................................... 77
   Checking for More Data ................................................................. 79
   Reusing Data Values ......................................................................... 81
   Storing Data in Files ........................................................................ 82
   Reading Data from Text Files ............................................................. 83
   Creating Text Files ........................................................................... 85
   Printing String Data to Text Files ....................................................... 86
   Reusing Stored Data for Input ............................................................ 88
   Printing Numeric Data to Text Files .................................................... 89
   More about File Input and Output ....................................................... 91

13. Arrays and Matrices ................................................................. 93
   One-dimensional Arrays ................................................................... 93
   Array Subscripts ............................................................................. 95
   Array Bounds .................................................................................. 96
   Arrays of Two or More Dimensions .................................................. 97
   The MAT Statements ....................................................................... 99
   Advanced Work with Arrays and Matrices ....................................... 101

14. Functions and Subroutines .......................................................... 103
   Subroutines .................................................................................... 103
   Subroutines with Parameters ............................................................ 105
   Built-in Functions ........................................................................... 106
   One-line Functions .......................................................................... 107
   Multi-line Functions ....................................................................... 108
   Global Variables ............................................................................ 109
   External Subroutines and Functions ............................................... 111
   The LOCAL Statement ..................................................................... 112

15. Libraries ..................................................................................... 113
   Libraries ....................................................................................... 113
   Aliases ............................................................................................. 116
   Compiling ....................................................................................... 116

16. Graphics .................................................................................... 117
   Drawing Points ................................................................................. 117
   Drawing Lines .................................................................................. 118
   Changing the Coordinates ............................................................... 119
   Drawing Shapes ............................................................................... 120
   Using Colors ................................................................................... 122
   Animation ....................................................................................... 124
Using This Guide

The BRONZE Edition of the True BASIC Language System is an ideal way to start using this unique and powerful programming language created by the original inventors of BASIC. You are able to write or run programs of any size, use libraries and modules, and invoke DO programs. All the powerful True BASIC statements and functions are included in this inexpensive starter edition.

The functionality to create independent free-standing double-click applications is not included in the BRONZE Edition. For this the SILVER Edition of True BASIC is required. A GOLD Edition of True BASIC will be of special interest to advanced developers, corporate or academic multi-user sites. Specifications and prices of all True BASIC books and products can be found at the True BASIC website: http://www.truebasic.com. Be sure to note the special Annual-CD’s also offered there. These Annual-CD’s contain the latest version of the True BASIC SILVER editions and a wealth of how-to material at a very low price.

This BRONZE EDITION has been enhanced with an expanded HELP utility. Be sure to read Appendix F (page 207) for an introduction on how to use this powerful tool. The contents of an extensive reference manual are included in the HELP utility. Sample code for many routines are also included. You can copy and paste code from HELP to your program. Appendix B gives you a quick overview of the primary True BASIC statements and functions. The HELP utility provides more information about each statement and function.

Many of the concepts and operations described in this manual 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
- **Specific operating system info:** M = MacOS; W = Windows
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 BRONZE Edition package introduces you to programming using statements 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. If you wish to explore beyond the scope of this Bronze Edition, we suggest the following books:


The above books are available directly from True BASIC (where all the listed titles are carried in stock) or from the individual publishers.

Visit our Web Site at [http://www.truebasic.com](http://www.truebasic.com) or telephone our sales department at 800 436-2111 (international: 802 296-2711) to receive information on any new editions that might have been published since the printing of this guide and the current prices of all books.
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 beginner 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. Versions are available for Windows3™, Windows95™, Windows95™, Windows98™, Windows Windows2000, WindowsME™, WindowsXP™, LINUX and MacOS™ operating systems. You can easily move your programs from one computer to another. True BASIC has been designed with portability in mind and allows a program that is written once in True BASIC to run on many different operating systems with a minimum of modification. Some of the differences in operating systems that are beyond...
the control of True BASIC are noted in this Guide. For example, the appearance of
Windows and dialog boxes varies from operating system to operating system. We
use illustrations from more than one operating system in this Guide to give you a
sense of the subtle differences you will see from system to system.

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. Spanish and Japanese versions of True BASIC exist, and the lan-
guage system has been designed so that it can be localized as required.

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 indent-
ing to make your programs easy to follow and modify later on.

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.)

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 com-
puters 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. Dr. Kurtz remains active in True BASIC affairs and has taken a
leading role in insuring that this latest BRONZE Edition combines the traditional sim-
plicity of BASIC with a wealth of powerful new features.
Installing True BASIC
and Running Demo Programs

This chapter takes you from turning on your computer to running a program with the True BASIC BRONZE Edition and then quitting the application. If you are an experienced user, you may want to skim or skip the first two sections and begin with “Running a Demonstration Program”. Wherever you start, we recommend that you work at a computer and try all the sample programs as you go through this book.

Installing True BASIC from the CD-ROM

The True BASIC BRONZE Edition runs on:
- Windows95, 98, 2000, ME & XP – Memory requirement: 16MB memory.
- Windows 3.1 – Memory requirement: 8MB memory.
- MacOS – Memory Requirement: System 7.0 or later, 16MB memory.

Start up your computer. View the True BASIC BRONZE Edition CD-ROM from your CD drive. Install the software from the CD to your computer's hard disk. After you have installed the BRONZE Edition software, store the CD-ROM so it will be available if you need to re-install the software at a future time. Please note that making more than one copy of this True BASIC BRONZE Edition software for backup is a violation of our licensing agreement with you.

On the next pages you will find step-by-step installation procedures for the most popular operating systems. The illustrations will be similar to what you see on your computer and are subject to change as systems and installation scripts evolve.

If you need to install True BASIC on a Windows 3.1 operating system, please see:
Windows Installation from a CD-ROM

1. Insert CD-ROM in your CD drive (E:, F: to M:).
2. Click the CD-ROM Disc icon on your Windows desktop.

When you see the contents of the True BASIC CD-ROM, double-click on the `Setup.exe` icon. The SETUP program will begin to run and you will see this initial screen:

Notice the three buttons at the bottom of the screen. They allow you to move to the next installation step, return to the previous step, or cancel the installation altogether. Most times you will click on the `Next>` button after you have entered the information requested or made a selection from the options offered.
When we click **Next** on the initial screen, you will see:

The Installer is set to write the files on the **C:** drive with the title of **TB Bronze**. You can overwrite the drive and name of directory. Then, click again on the **Next** button.

The next screen shows the contents of your **Program Folder**, the default destination of most Windows applications. The text input box allows you to edit the default choice of “True BASIC Bronze Edition”. The **Next** button moves you to the next installation step.
Now the install programs begins to place files and creates the appropriate directories and shortcut icons. A progress bar will show you the installation status and announce when the Windows operating system installation is complete.

**MacOS Installation from Diskette or from CD-ROM**

1. Insert the CD-ROM in your CD drive. You will see the items shown below.
2. Click on the TB Bronze Installer. It will launch and display the start-up scheen shown below. Click on the Continue button.
What You Should See After Installation
When the installation is complete, look in the folder/directory containing the True BASIC BRONZE Edition. You should see the following:

Before installation begins, you have the opportunity to select the hard drive and folder location you prefer. When you have selected the location, click on the Install button to continue or the Quit button to stop the installation process.

Note: A “folder” on the MacOS is the same as a “directory” on Windows. Throughout this Guide we use the term “directory”.
Using the True BASIC BRONZE Edition

Double click on the True BASIC BRONZE icon and the language system will launch. A dialog box will appear asking if you want to retrieve an existing program or want to start a new program. If you choose **New**, the **Editing Window** and the **Command Window** will appear on your computer.

The **Editing Window** is where you edit your programs, and the **Command Window** is where you can communicate with True BASIC by typing commands as an alternative to using the mouse. Across the top edge of the Editing Window lies the Menu Bar. Selection from the Menus can be made using using the mouse.

When you **Run** your source code, the results are shown in the **Output Window**.

Along the right edge of the Editing Window lies the Scroll Bar. The Scroll Bar allows you to use the mouse to move through the text in the Editing Window a “page” at a time (a page is the number of lines that are visible in the Editing Window). With the mouse, click on the up or down arrows to move the text in the screen up or down one line at a time. If you hold the mouse button, this action will be repeated until you release the button.

True BASIC’s editor can also handle long lines, but of course you can’t see the whole line at the same time. Instead, True BASIC shows only part of a long line and highlights the right-most character to show that there is more. You can use the cursor keys to move to the right and view the unseen portions or drag the mouse to the right of the line on the screen, or you can use the horizontal Scroll Bar at the bottom of the Editing Window.
The Editing Window is where you edit your programs, and the Command Window is where you can communicate with True BASIC with typing.

You may only work in the “active” window. The active window is the one that currently contains the flashing cursor. To switch active windows with a mouse, simply click in the non-active window, and it will become the active one.

Menu Access

Using the keyboard:

On Windows, if you prefer, you can also access the menus using the keyboard. Hold down the ALT-key and type the number of the menu you wish to display. The menus are numbered from left to right beginning with one. Thus, ALT-1 displays the File menu, ALT-2 displays the Edit menu. When the menu containing the item you wish to select is visible, highlight the item using the up and down arrow keys and then press the RETURN or ENTER key. Using the <— and —> keys will close the current menu and open the right or left menu. Use the Esc key to leave a menu without making a selection.

To leave a menu without making a selection use the Esc key.

To navigate through your text with the keyboard, you can use the standard navigation keys, such as the arrow keys, Page Up, Page Down, Home, and End to move the cursor and the contents of the screen.

See Appendix E for a complete listing of command-key shortcuts for both the MacOS and Windows versions.

Using a Mouse:

With a mouse, simply place the mouse cursor over the menu title, click and hold the mouse button while you drag the cursor towards the bottom of the screen. To select a menu item continue dragging until the item you want is highlighted and then release the mouse button. To exit a menu without making a selection, move the mouse cursor back to the menu bar or outside the menu before releasing the button.
Running a Demonstration Program

The True BASIC BRONZE Edition disk has several demonstration programs on it. 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. Using the mouse, place the cursor on File, click the mouse and drag to the Open selection. You will be presented with a dialog box that contains a list of all files and subdirectories contained in your current directory or folder. You can “scroll” through this window using the up and down arrow keys or holding the mouse button down while you drag it through the list.

All True BASIC program names have the .tru extension. This is necessary for Windows, and is included on the MacOS for consistency. For ease in reading, the “.tru” extension is omitted in this Guide.

We want to select a program that is stored in the TBDEMOS directory so we will first have to select this directory by moving to it until it is highlighted. Once you’ve selected the TBDEMOS directory, a new dialog box will appear with a list of all programs available. Select the program GALTON by highlighting it with the mouse or the arrow keys.

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, which shows graphics, the editing window disappears and the program results appear on the full screen. When the program run is complete, the graphics remain on the screen.

To return to the editing window, press any key or click the mouse 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, select the Stop command from the File menu at the top of the Output Window. That returns you to the editing window. Some programs, such as the BOUNCE demo program, will run indefinitely until you stop them manually.

✓ If you need to stop a program during a run, select the Stop command from the Output Window File menu.
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 BRONZE Edition and chosen “New”, you’ll have a blank editing window called “Untitled 1” 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 window which is automatically named “Untitled #”.

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 editing 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 capital or lowercase letters or more spaces than shown. It only matters that you enter the program in a fashion similar to what is shown on the previous page. 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 BACKSPACE or Delete key to erase characters you have just typed. Press BACKSPACE once to erase the preceding character; press it several times to erase several characters. You can also use the arrow keys to move the cursor bar anywhere on the screen to make a correction. Move the mouse 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. (The next chapter tells how to make simple corrections to your program; Chapter 11 gives more details on editing.)

Now let’s see what the program does. Select Run in the Run menu. You should see the following “output”:
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.)

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*. You’ll learn more about constants and variables in Chapter 6.
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:

\[
\text{PRINT miles, 14.3, 420/gallons}
\]

and the results would have been exactly the same.

Chapter 7 describes the PRINT statement in more detail; Chapter 6 introduces expressions.

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 have saved this program, you will be presented with a dialog box which allows you to choose the directory where your file will be saved. Call this program MPG and press Return or click with the mouse.

You will again use this file in the next chapter where you’ll learn how to make changes to an existing program.
In the previous chapter, 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
```

**Using Source and Output Windows**

So far, we have been looking at the Editing Window which contains the program statements. As you begin to modify and test programs, you will be able to see both your output window and your editing or “source window” on the screen. When you run your program, the results appear in the Output Window.

True BASIC uses an Error window to report errors, while actual output is sent to the Output window. When your program has finished, True BASIC will wait for you to press a key or click the mouse. Then the output screen will be erased and you will be returned to the Source and Command Windows again.

(You can also keep the Output Window visible by selecting Output Window in the Window menu.)
Making Simple Changes

Before you can edit your source program, you must learn how to move the text cursor. First, make sure that the text cursor is in the desired window.

In the source window, a blinking vertical bar indicates the insertion point. When you type something on the keyboard, the 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 or the arrow keys.

The arrow keys move the insertion point a character or line at a time throughout the text.

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 on MacOS) 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 also select a word by double-clicking on that word.)

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, place the cursor at its beginning and press the Delete (or Backspace) key.

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 by moving the cursor to in front of the first word of the second line and press Delete or Backspace. The second line will be joined to the end of the line above.

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 = 14.3 ! 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.

**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. We will do that a bit later.

**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.

! 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. Chapter 6 explains this more fully. Chapter 7 explains the semicolon (;) at the end of the PRINT statement — the semicolon makes the question mark appear on the same line as the text, and close to it.

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. 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 Save** to save the program (or replace a version with the same name) and quit True BASIC
- **Click Discard** 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 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 = 14.3 ! gas used
PRINT miles, gallons, miles/gallons
END
```

**Constants**

The MPG program contains two numbers: 420 and 14.3. 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 14.3. 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:

<table>
<thead>
<tr>
<th>Number Constants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Acceptable</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>1002</td>
</tr>
<tr>
<td>321.33</td>
</tr>
<tr>
<td>0.003</td>
</tr>
<tr>
<td>.25</td>
</tr>
</tbody>
</table>

**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

<table>
<thead>
<tr>
<th>Acceptable</th>
<th>Not Acceptable</th>
</tr>
</thead>
<tbody>
<tr>
<td>miles</td>
<td># of miles</td>
</tr>
<tr>
<td>miles_per_gallon</td>
<td>miles.per.gallon</td>
</tr>
<tr>
<td>profits</td>
<td>13</td>
</tr>
<tr>
<td>tax1040</td>
<td>1world</td>
</tr>
<tr>
<td>time_of_day</td>
<td>time-of-day</td>
</tr>
</tbody>
</table>

### 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:

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>addition</td>
<td>a + b</td>
</tr>
<tr>
<td>−</td>
<td>subtraction</td>
<td>3 - 2</td>
</tr>
<tr>
<td>*</td>
<td>multiplication</td>
<td>length*width</td>
</tr>
<tr>
<td>/</td>
<td>division</td>
<td>miles/gallons</td>
</tr>
<tr>
<td>^</td>
<td>exponentiation (x^2)</td>
<td>x^2</td>
</tr>
</tbody>
</table>

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:

\[
\text{miles/gallons}
\]

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.
Notice the symbols for multiplication and division. Computer keyboards don’t usually contain the \( \div \) 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 most computers cannot write superscripts properly.

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

\[
\text{miles/gallons} \\
\text{not} \\
\text{miles} \div \text{gallons} \\
or \\
\frac{\text{miles}}{\text{gallons}}
\]

True BASIC follows rules that decide the order of calculation in an expression. You can also control the order of calculation with parentheses.

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

\[
6 + \frac{10}{2}
\]

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

\[
\frac{12}{6*2}
\]

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:

\[
\frac{12}{(6*2)}
\]

- True BASIC computes exponents first, even before multiplications and divisions.
True BASIC does arithmetic as follows: 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:

<table>
<thead>
<tr>
<th>In Mathematics</th>
<th>In True BASIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(1 + 2 + 3)</td>
<td>(1 + 2 + 3)</td>
</tr>
<tr>
<td>(3 \times (4 + 5))</td>
<td>(3 \times (4 + 5))</td>
</tr>
<tr>
<td>(\frac{1 + 2}{4})</td>
<td>(\frac{1 + 2}{4})</td>
</tr>
<tr>
<td>(\frac{AB}{CD})</td>
<td>(\frac{A \times B}{C \times D})</td>
</tr>
<tr>
<td>(x^2)</td>
<td>(x^2)</td>
</tr>
</tbody>
</table>

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

If a line is too long to fit on a single line of the screen, you can use the True BASIC line continuation feature. To continue a line in this way, type an “&” at the point you want the line to be broken and then press Return. At the beginning of the next line, type another “&” and then the rest of the line.

**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:

```basic
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 * 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 (see Chapter 8).

### An Introduction to Strings

True BASIC processes words as well as numbers. In computer terminology, anything that doesn't 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 Chapter 5, you used strings with PRINT statements to tell the user what to enter for the INPUT statements in your MPG2 program:

```basic
! 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:

```basic
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 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 TBDEMOS Directory.)
! 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

(See Appendix C for a complete list of string functions.)
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 again at the MPG2 program and the output you get when you run the program:

```plaintext
! 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:

```plaintext
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
Miles Gallons Miles per gallon
312  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.

```plaintext
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 semicolon, 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:

```plaintext
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 (see Appendix G) is especially helpful if you want numeric output to follow a specific pattern.

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. Chapter 10 explains briefly how you can send output to a printer or a file.

**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. Please Reenter input line.

Como te llamas ? Ruy Diaz of San Antonio
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:

```vbnet
! 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:

```plaintext
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.)

### The TD_LineInput Subroutine

An alternative to the LINE INPUT statement is the TD_LineInput dialog box. To use it you must include a library statement in your program to tell True BASIC which library file contains the subroutine. Then use a CALL statement. Both are shown below.

```vbnet
! Ask for a name, then say good morning.
!
LIBRARY "TrueDial.trc"
CALL TD_LineInput ("Como te llamas", name$)
PRINT "Buenos dias, "; name$; "."
END
```

The CALL TD_LineInput statement displays a dialog box on the screen; it looks something like this:

![Dialog Box](image)

You can then type your name into the small box.
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:

```basic
! 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 book, 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.

Open the demo program SQROOT from your True BASIC BRONZE Edition disk:

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

and run it:

<table>
<thead>
<tr>
<th>Number</th>
<th>Square Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.1</td>
<td>.31622777</td>
</tr>
<tr>
<td>.2</td>
<td>.4472136</td>
</tr>
<tr>
<td>.3</td>
<td>.547722256</td>
</tr>
<tr>
<td>.4</td>
<td>.63245553</td>
</tr>
</tbody>
</table>
This program uses the built-in function SQR to obtain the square root of \textit{number}. (Chapter 14 explains built-in 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:

\begin{verbatim}
! 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) ! Print number & square root
NEXT number
END
\end{verbatim}

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 \textit{number} becomes less than five. (If you forget to change the step from .1 to -1, your loop won't execute at all, because \textit{number} can't get from 10 to 5 without a negative step.)

\begin{center}
\begin{tabular}{|c|c|}
\hline
\textbf{Number} & \textbf{Square Root} \\
10 & 3.1622777 \\
9 & 3 \\
8 & 2.8284271 \\
7 & 2.6457513 \\
6 & 2.4494897 \\
5 & 2.236068 \\
\hline
\end{tabular}
\end{center}

You can use the index variable (here, \textit{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 \textit{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) ! Print number & square root
NEXT number
PRINT number
END

and run it again:

<table>
<thead>
<tr>
<th>Number</th>
<th>Square Root</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>3.1622777</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>2.8284271</td>
</tr>
<tr>
<td>7</td>
<td>2.6457513</td>
</tr>
<tr>
<td>6</td>
<td>2.4494897</td>
</tr>
<tr>
<td>5</td>
<td>2.236068</td>
</tr>
<tr>
<td>4</td>
<td></td>
</tr>
</tbody>
</table>

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 \textit{row} as the loop index, and within that an inner loop with the index variable \textit{xcount}.

\textbf{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 \textit{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 \textit{row} and \textit{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 xxxxx xxxxx
 xxxxxx xxxxxxx xxxxxxx
```

**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 an arbitrary 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**:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>equal to</td>
</tr>
<tr>
<td>&lt;&gt; or &gt;&lt;</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;= or =&lt;</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>greater than</td>
</tr>
<tr>
<td>&gt;= or =&gt;</td>
<td>greater than or equal to</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 &lt; 2</td>
<td>true</td>
</tr>
<tr>
<td>1 + 2 &lt; 3</td>
<td>false</td>
</tr>
<tr>
<td>5 + 3 &gt;= 8</td>
<td>true</td>
</tr>
<tr>
<td>“abc” &lt;&gt; “ABC”</td>
<td>true</td>
</tr>
<tr>
<td>“yes” = “no”</td>
<td>false</td>
</tr>
<tr>
<td>“elephant” &lt; “spider”</td>
<td>true</td>
</tr>
<tr>
<td>“elephant” &lt; “Spider”</td>
<td>false</td>
</tr>
<tr>
<td>“moon” &lt; “moonbeam”</td>
<td>true</td>
</tr>
</tbody>
</table>
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 of this book 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 5.5% interest. At the end of the first year, the bank will pay you $550. If you leave this money in the account, the next year you’ll earn interest on $10,550, which yields slightly more than another $580, 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 TBDEMOS Directory on your True BASIC BRONZE Edition disk:

```basic
! 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 = 5.5/100 ! Interest is 5.5%

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

8. Loop Structures 53
Run the program:

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>10000</td>
</tr>
<tr>
<td>1</td>
<td>10550</td>
</tr>
<tr>
<td>2</td>
<td>11130.25</td>
</tr>
<tr>
<td>3</td>
<td>11742.414</td>
</tr>
<tr>
<td>4</td>
<td>12388.247</td>
</tr>
<tr>
<td>5</td>
<td>13069.6</td>
</tr>
<tr>
<td>6</td>
<td>13788.428</td>
</tr>
<tr>
<td>7</td>
<td>14546.792</td>
</tr>
<tr>
<td>8</td>
<td>15346.865</td>
</tr>
<tr>
<td>9</td>
<td>16190.943</td>
</tr>
<tr>
<td>10</td>
<td>17081.445</td>
</tr>
<tr>
<td>11</td>
<td>18020.924</td>
</tr>
<tr>
<td>12</td>
<td>19012.075</td>
</tr>
</tbody>
</table>

In 13 years, you'll have $20057.739

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 10550, still less than $20,000, so True BASIC repeats the loop. The third time it's 11130.25 so True BASIC repeats the loop, and so on. The last time through, money reaches the value 20057.739. 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.
8. Loop Structures

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 = 8.5/100           ! Interest is 8.5%
```

With 8.5% interest, you should find that the DO loop works only nine times instead of thirteen as it did before. However, the condition \((money \geq 2 \times original)\) 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 $ 20838.557
```

True BASIC’s PRINT USING (see Appendix G) 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
```

**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
```
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.

```basic
! 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.

```basic
! 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

\[ \text{LET } \textit{years} = \textit{years} + 1 \]

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”.

```basic
! 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 "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 to 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 "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 care what 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 programs shown below provide a quick introduction; these programs are in the TBDEMOS Directory.

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:

```basic
! 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";
    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:

```basic
! 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
          END SELECT
          IF dice=point then PRINT "You win" else PRINT "You lose"
    END SELECT
    NEXT game

END
```
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 TBDEMOS subdirectory is another version of the GUESS program with no blank or indented lines. This version has a nested IF structure. Open this program and try to follow the structures in the unindented format.

```basic
! Program to play a guessing game.
!
randomize
let answer = Int(Rnd*6) + 1 ! Choose number from 1 to 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 **Run** menu. 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.)

```basic
! Program to play a guessing game.
!
RANDOMIZE
LET answer = Int(Rnd*6) + 1 ! Choose number from 1 to 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";
    INPUT guess
    IF guess = answer THEN
        PRINT "Correct!!!"
        STOP ! Stop here, you guessed it
    ELSE
        IF guess > answer then
            PRINT "Too high. Guess again."
        ELSE
            PRINT "Too low. Guess again."
        END IF
    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 (horizontal blinking 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 cursor. (Make sure that the entire
lines are selected, not just the first part.) 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 choosing *Print ...* in the *File* menu of the editing window.

To print just part of your program, first use the mouse to select the desired lines and then choose *Print Selection ...* in the *File* menu. Select multiple lines by dragging across them with the mouse.

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.

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

**Listing Output from Your Programs**

When you run your programs, the results are “printed” on the screen in the output window. If you wish to send those results to your printer, you must “open a channel” to the printer. 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.
The ECHO command, which you use in a command window, also lets you send program output to a printer. The last section in this chapter describes how to use the command window and the ECHO command.

Printing graphics output is even easier. Just choose Print in the menu of the output window.

**Using Line Numbers**

True BASIC's structures and editing features make it unnecessary to use line numbers in 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 structured programming languages were developed. You won't find them described in this manual. However, we do include a very useful True BASIC utility, the *Basic to True BASIC Converter* which will translate many earlier Basic programs into useful True BASIC code. The *Converter* is described in Appendix H.

**Using the Command Window**

So far, you've told True BASIC what to do with menu choices. You can also give commands by typing them in a *command window*. This window has two parts. The actual command part is limited to a single line at the bottom. The rest of the command window is actually a "history" window containing all the commands you have typed recently.

Click in the command window to make it active and allow you to 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. Some of these let you print copies of your program or output:

- **LIST**
  Prints all or part of your program on your printer (indicate lines to print just part of the program, such as LIST 1-10 for the first ten lines).

- **ECHO**
  Sends a copy of your output to a printer when you next use the RUN command. This stays in effect until you use ECHO OFF. (You can send output to a file with ECHO TO filename.)

- **ECHO OFF**
  Stops echo of subsequent output to a printer or file

- **RUN >> filename**
  Sends a copy of your output to the named file.

Other commands are helpful in debugging or correcting errors in your programs. Chapter 18 introduces some debugging commands.
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 five sections of commands. This chapter explains the first three groups of commands. The last two groups are introduced briefly; you’ll find them more helpful later as you begin to work with larger programs.

Most of the editing commands have keystroke equivalents. For example, to Cut text on the Macintosh, you could use command-X. On Windows you could use Alt followed by E followed by T. The details of these keystroke equivalents are not included here as they differ between operating systems. They are listed in detail in Appendix E.

Undoing
The Undo command in the Edit menu helps you recover from an editing mistake. For instance, if you have just deleted text (rather than cutting it to the clipboard,) simply select Undo.

This command will “undo” the effects of the most recent Cut, Delete, or Paste operation (see the next section.) And it will undo all typing since the last Cut, Delete, or Paste operation, or mouse click.

Selecting, Cutting, Copying, and Pasting
Deleting, Cutting, Copying, and Pasting text are important editing tools. The Cut, Copy, and Paste commands in True BASIC’s Edit menu work just like those commands in most other applications. They all depend on selecting text – selecting single words, parts or all of lines, or blocks of lines.
Selecting Text. To delete, move, or copy something, you must first select or highlight the desired text using the mouse in one of the following ways:

• drag across the desired words or lines
• double-click on a word to select that word

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

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 17.)

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

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. The lines will disappear.

The **Cut** command puts these lines into the “clipboard” so you can get them back later. Choose **Paste** in the **Edit** menu. True BASIC will put the lines back where they were originally.

---

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.
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 Chapter 12.

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 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.

Run the program again. You’ll hear the same lines twice.

Find and Change

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 Find dialog. Type:

```
Data
```

in either upper or lowercase. Press the Return or the Enter key, or click the Find button on the lower part of the box. True BASIC will select (display in reverse video) the first occurrence of the word data in the program:

```
DO while more data
```
To find the next occurrence of the word \textit{data}, choose \textbf{Find Again} in the \textbf{Edit} menu. True BASIC will select the next occurrence of the word \textit{data}, which is the first DATA statement.

\textbf{Finding Parts of Words.} Choose the \textbf{Find} command again. This time, type:
\begin{verbatim}
dat
\end{verbatim}

and press the RETURN key or click Find. True BASIC will select the next occurrence of \textit{dat}, which is the first part of the next occurrence of the word \textit{data}.

If you want to find just part of a word and distinguish between upper or lower case, click in the box “Case Sensitive” in the Find dialog box. If you want to find the exact word, click in the box “Entire Word” in the Find dialog box.

Without moving your insertion point, choose \textbf{Find} one more time. This time look for the word:
\begin{verbatim}
read
\end{verbatim}

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. Instead, you’ll be told

“read” not found

in the message line at the bottom of the Editing Window.

If you want to go back to the beginning of the file to continue the search, click in the box “Wrap” in the Find dialog box.

\begin{itemize}
\item \textbf{Check:} True BASIC always searches from the insertion point to the end of the program, and then stops, unless Wrap has been selected.
\end{itemize}
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. Type the word `music$` in the first line, and the word `notes$` in the second line. Now click on the Replace All button.

Look at the READ and PLAY statements, and you’ll see that the variable names have changed.

The **Change** command works over the entire contents of the Source Window. Otherwise, the **Change** command works like the **Find** command. You can make it case sensitive. And you can have it apply only to entire words and not parts of words.

### Keep and Include

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. True BASIC will also change the name of what is left to "Untitled ?" to prevent your accidentally saving it over the original file.

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 directory
on any disk. True Basic will insert the contents of that file at the insertion point of your current program.

**Select All and Move To**

The **Select All** command will select the entire contents of the Editing Window, whether visible or not. This can be useful if you want to move the entire file to another Editing Window.

The **Move To** commands lets you 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.

As another example, if you want to work with your subroutine MakeImage, just type its name in response to the Move To dialog box.
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.

**The DATA and READ Statements**

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

```plaintext
! 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."
    ELSE
        PRINT "No, the correct answer is "; answer$; "."
    END IF
NEXT i
```

77
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; some-
times it’s more convenient to put them right after a READ statement. You may use a sep-
perate 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 arith-
metic 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 state-
ments. The number of questions then controls the FOR-NEXT loop so that it reads the cor-
rect 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 items, 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.

```plaintext
! 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 04 L4 C C E G 05 L2 C. 04 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.

Summary: True BASIC’s RESTORE statements 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.

```basic
! Plays the beginning of
! "On Top of Old Smoky".
PRINT "Now playing '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 04 L4 C C E G 05 L2 C. 04 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. (Click in the close box of the output window to stop the program.)

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 a wide variety of files. A file is a collection 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 don’t need 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 five kinds of files: text, record, random, stream, 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:

```basic
! Trivia quiz -- reads data from a file.
! OPEN #1: name "TrivData.tru" ! 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 correct replies
    PRINT "Correct."
  ELSE
    PRINT "No, the correct answer is "; answer$; "."
  END IF
END IF
```

12. Using and Storing Data 83
LOOP until end #1

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

CLOSE #1

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 Chapter 10. 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.)

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 File menu of the Output Window, or click the close box of the Output Window 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 statements asks for two input items. Look at TrivData.tru and you’ll see that each line contains two input items separated by a comma.

```plaintext
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 are no extra CR or CR-LF sequences 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 statements in the program. (The MAT statements read into 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; 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 you new data file, or you can modify the program to ask you what file to use for input:

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

True BASIC programs can also create text files and put data into 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 Chapter 10), 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.tru", CREATE NEWOLD
! Opens channel #1 to a file
ERASE #1 ! Make sure file is empty
FOR i = 1 to 10
   PRINT #1: i ! Print to file #1
NEXT i
CLOSE #1 ! Close the file
END
```

Simply opening the 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.

The CREATE NEWOLD phrase that is part of the OPEN statement will create the output file if necessary.

---

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
RESET #2: end

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$
END IF

LOOP until end #1

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

CLOSE #1
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 call REQUIZ for the missed questions. You could then run the program again, naming REQUIZ as the source of questions and a new file name to received 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.

Reusing Stored Data For Input

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

Who wrote 20,000 Leagues Under 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 Under the Sea", Jules Verne

But what if you want to place the title "20,000 Leagues Under 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 Under the Sea'", Jules Verne

or

"Who wrote ""20,000 Leagues Under 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 Under 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$; ", your balance was "; curbal  
        PRINT "Your last check was "; lastcheck_amt  
        PRINT "Your last deposit was "; lastdep_amt  
        PRINT  
    END IF
    ELSE  
        PRINT "Input all checks and deposits since "; bal_date$  
    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, 1998
   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, listed below, for use with these and other kinds of files. Go to the online HELP facility and select these statements for information and examples.

Additional File Related Statements:

- ASK #n: ACCESS
- ASK #n: DATUM
- ASK #n: ERASABLE
- ASK #n: FILESIZE
- ASK #n: FILETYPE
- 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
- MAT INPUT #n:
- MAT LINE INPUT #n:
- MAT PRINT #n:
- READ #n:
- SET #n: MARGIN
- SET #n: POINTER
- SET #n: RECORD
- SET #n: RECSIZE
- SET #n: ZONEWIDTH
- UNSAVE
- WRITE #n:

Also, in Appendix I you can read more about the various file structures, text, stream, random, record, and byte, that are part of the True BASIC Language System and how each is typically used.
Problems often arise that would require an unreasonable number of variables to solve. Open the demo program `INVENTORY`, which keeps the inventory of a hardware store:

```basic
! 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 a one-dimensional array 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 **INVENTORY** program to use two arrays, *item*$ and *number* as shown below:

```vbnet
! 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
```

Figure 13.1 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.)

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

The five individual values within each of *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.

```basic
! 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
```

<table>
<thead>
<tr>
<th>item$</th>
<th>item$1</th>
<th>item$2</th>
<th>item$3</th>
<th>item$4</th>
<th>item$5</th>
</tr>
</thead>
<tbody>
<tr>
<td>hammers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>umbrellas</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>wood stoves</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>bags of salt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>pliers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>number</th>
<th>number1</th>
<th>number2</th>
<th>number3</th>
<th>number4</th>
<th>number5</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*Figure 13.1 – Items in Arrays*
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:

```basic
! 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 \texttt{temp(-40 to 40)}. This array has 81 elements.

Naturally, as your arrays get bigger, they take more computer memory to store. True BASIC places no limits on the size of your arrays except for what will fit in your computer's available memory.

\textbf{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 \textbf{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.

\begin{verbatim}
! 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
\end{verbatim}
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.)

<table>
<thead>
<tr>
<th>state$</th>
<th></th>
<th></th>
<th>state$</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>state$(1,1)</td>
<td>Alabama</td>
<td>Montgomery</td>
<td>state$(1,2)</td>
<td></td>
</tr>
<tr>
<td>state$(2,1)</td>
<td>Alaska</td>
<td>Juneau</td>
<td>state$(2,2)</td>
<td></td>
</tr>
<tr>
<td>state$(3,1)</td>
<td>Arizona</td>
<td>Phoenix</td>
<td>state$(3,2)</td>
<td></td>
</tr>
<tr>
<td>state$(4,1)</td>
<td>Arkansas</td>
<td>Little Rock</td>
<td>state$(4,2)</td>
<td></td>
</tr>
<tr>
<td>state$(5,1)</td>
<td>California</td>
<td>Sacramento</td>
<td>state$(5,2)</td>
<td></td>
</tr>
</tbody>
</table>

*Figure 13.2 – A Two-dimensional Array*

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. Figure 13.2 shows the first five rows.
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!

```basic
! 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 state-
ments.

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 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

You must respond with ten numeric constants separated by commas, entered in the form of
a single input-reply.
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.

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; the Help Utility and 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 multiply an array by a single number.
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 from Chapter 9. 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."
    END SELECT
```

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."
  END SELECT
NEXT game
CASE ELSE
LET POINT = dice
DO
CALL Rolldice
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.

- True BASIC does not distinguish between input and output parameters; it's only in the way you use them.

### 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, \( \text{answer} \) acquires the value 3.1622777, which is returned by the function SQR.

\[
\begin{align*}
\text{LET} \quad \text{answer} &= \text{Sqr}(10) \\
\text{PRINT} \quad \text{answer} \\
\text{END}
\end{align*}
\]

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 \( \text{Rolldie} \).

\[
! \text{Craps game with one-line function for rolling one die.}
!
\text{RANDOMIZE}
\text{DEF Rolldie = Int(6*Rnd + 1) ! Roll 1 die}
\text{FOR game = 1 to 10 ! Play 10 games}
\quad \text{LET dice = Rolldie + Rolldie ! Rolldie function twice}
\quad \text{SELECT CASE dice ! Branch on roll}
\quad \quad \text{CASE 2, 3, 12 ! dice = 2, 3, or 12}
\quad \quad \quad \text{PRINT "You lose."}
\quad \quad \text{CASE 7, 11 ! dice = 7 or 11}
\quad \quad \quad \text{PRINT "You win."}
\quad \quad \text{CASE ELSE ! Anything else}
\quad \quad \quad \text{LET POINT = dice ! Remember that roll}
\quad \quad \quad \text{DO}
\quad \quad \quad \quad \text{LET dice = Rolldie + Rolldie ! Roll again}
\quad \quad \quad \quad \text{LOOP until dice = 7 or dice = point}
\quad \quad \quad \quad \text{IF dice=point then PRINT "You win" else PRINT "You lose"}
\quad \quad \text{END SELECT}
\quad \text{END SELECT}
\text{NEXT game}
\text{END}
\]

14. Functions and Subroutines 107
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 or array 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:

```basic
! 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 \textit{Sgn}, the program selects one of three cases depending upon the sign of the parameter and assigns a value to \textit{Sgn}. At the \textsc{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.)

\textbf{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 \textit{global variables}.

Global variables are sometimes useful, but often they are a source of hard-to-spot program bugs. Consider the example in the TBDEMONS folder/directory – \textsc{BUG}:

\begin{verbatim}
! 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
\end{verbatim}

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

\begin{verbatim}
How many X's? 10
xxxxxxxxxxxx
\end{verbatim}
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 variable 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.

Try using debug mode and breakpoints with this program (see Chapter 18.) You will see that there is only one variable $i$ in your program; you may deduce from that that you are attempting to use it for two purposes.
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.

```basic
! 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$(n)  ! Return a string of n X's
    LET s$ = ""
    FOR i = 1 to n  ! Loop...
        LET s$ = s$ & "x"
    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
xxxxxxxxxxx
How many X's? 4
xxxx
How many X's? 7
xxxxxxxx
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 are no longer global and 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 18 describes this programming technique.
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.

**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 TBDEMOS folder/directory 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:

```basic
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.tru"
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
DECLARE DEF statement before you use the function, just as you must with an external function in the same file.

! Craps game.
!
LIBRARY "gameslib.tru"
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
            END DO
            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
END FOR

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. All of them are included in the HELP files. Type help on the command line, or select the menu “Help for True BASIC”. See Appendix F for more details.

**Aliases**

When you use a LIBRARY statement, True BASIC makes an effort to look for your library file. It looks first in your current directory or folder. Then it looks in the directory named “TBLibs”. Thus, when you use any of the True BASIC libraries that are included with the Bronze Edition, True BASIC will find them in the “TBLibs” directory, regardless of your current directory.

You can see the entire list of aliases by typing the command “alias” on the command line. Besides the aliases for libraries, there are aliases for “Do” programs and for “Help” files.

You probably will have no need to change these aliases, but you can do so by selecting “Set Alias” in the “Settings menu”. But be careful! If you accidentally mess them up, just quit or exit True BASIC and start again.

**Compiling**

Most of the LIBRARY files used with the Bronze Edition are text files — which are also known as “source code” files. They can also be compiled files. It doesn’t make any difference. (Source files usually have the extension .tru while compiled files have the extension .trc). You may notice that your program’s startup time is slightly less when the library files have been compiled, but it makes a real difference only with very large programs.

You can easily make a source file into a compiled file by selecting “Compile” in the “Run” menu. But first, be sure that your source file has been properly saved.
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.

**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 the output window that is currently active.

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. The point with the coordinates (.2, .4) is shown below.

![Figure 16.1 – PLOT .2, .4](image-url)
A simple True BASIC program to draw this point on your screen has just two lines:

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

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.

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

**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):

```basic
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, that is, to draw a line from the last point to the first point:

```basic
PLOT .2,.4; .4,.4; .4,.6; .2,.6; ! Connect all points
PLOT .2,.4 ! Close box; turn off "pen"
END
```
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.

```basic
! 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
```

Similarly, the BOX AREA statement draws a solid square or rectangle using the coordinates you give in the statement:

```basic
! 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 16.4 - BOX LINES 10, 20, 5, 15*
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. The result is shown in Figure 16.7 on the next page.
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.

```plaintext
SET WINDOW -2, 2, -2, 2
PLOT AREA: -1,-1; 1,-1; 0,1
END
```

**Using Colors**

In the examples used so far, all solid objects are filled with a color that is dependant on your monitor and default graphics mode for your computer. 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:

```plaintext
SET COLOR "red"
SET COLOR 3
```
The table shows the equivalent color names, numbers, and meanings for colors supported for most graphics modes with color monitors. In addition, there are two default colors: -1 (black) for the foreground, and -2 (white) for the background. When opened the first time, all windows have these default colors.

<table>
<thead>
<tr>
<th>Name</th>
<th>Number</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>black</td>
<td>0</td>
<td>black</td>
</tr>
<tr>
<td>blue</td>
<td>1</td>
<td>blue</td>
</tr>
<tr>
<td>green</td>
<td>2</td>
<td>green</td>
</tr>
<tr>
<td>cyan</td>
<td>3</td>
<td>cyan</td>
</tr>
<tr>
<td>red</td>
<td>4</td>
<td>red</td>
</tr>
<tr>
<td>magenta</td>
<td>5</td>
<td>magenta</td>
</tr>
<tr>
<td>brown</td>
<td>6</td>
<td>brown (yellow on some monitors)</td>
</tr>
<tr>
<td>white</td>
<td>7</td>
<td>white</td>
</tr>
<tr>
<td>8</td>
<td>gray</td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>bright blue</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>bright green</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>bright cyan</td>
<td></td>
</tr>
<tr>
<td>12</td>
<td>bright red</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>bright magenta</td>
<td></td>
</tr>
<tr>
<td>yellow</td>
<td>14</td>
<td>yellow (brown on some monitors)</td>
</tr>
<tr>
<td>15</td>
<td>bright white</td>
<td></td>
</tr>
</tbody>
</table>

The following program (SQUARES in the TBDEMOS directory) draws a series of solid squares in different colors or shades of gray:

```plaintext
! 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
```

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 graphics mode of your computer. SET COLOR MIX lets you control the red, green, and blue elements producing a given color number.

**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 TBDEMOS directory uses these statements to shoot an arrow across the screen. Open it and run it.

```
! Shoot an arrow across the 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.

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 TBDEMOS folder/directory.

! 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;

DRAW Knight
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.

```
DRAW Knight with Rotate (90)
```

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.

**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 TBDEMOS folder/directory 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**

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

These are all subroutines; use them with a CALL statement. You must give arguments for several of the subroutines. Open the GRAPHLIB file 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 directory 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 TBLIBS folder/directory, you could use the following LIBRARY statement. This program draws coordinate axes with tick marks at every unit.
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 described briefly below.

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 TBDEMOS folder/directory 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 Graphical Displays
You can print the contents of any physical window by selecting Print in the window's menu. If you have a color printer, the results will be printed in color. If you do not have a color printer, the colors will be shown as different shades of gray.

Displaying Sensitive Graphics Objects
You can add a graphical object, such as a button, to your program. And you can find out if anyone has "pushed" the button (i.e., clicked on it.) You must make or get the image of the graphical object ahead of time. It can be anyone of several types, such as MS BMP for Windows 95 or PICT for the Macintosh. You can have several objects at the same time.

You must add a LIBRARY statement to your program, like this:

LIBRARY "GOC.trc"

This is a compiled file that contains all the necessary routines. You should also include

DECLARE DEF G_Click
just after the LIBRARY statement.

Early in the program you must "start up" the graphics class mechanism by calling

CALL G_StartUp
Just before your program terminates, you should “close down” the graphics class mechanism by calling

```c
CALL G_ShutDown
```

Or, you can just click on the Close Box of the window and the graphics class mechanism will shut down your program automatically and cleanly.

### Creating a Live Graphics Object

Assume the image you want to display is in a file called “image”. Of course, it can have any name you want. Then call

```c
CALL G_Create (thisone, "image", left, bottom)
```

The first argument, ‘thisone’ is your name for the live graphics object. The third and fourth arguments give the left edge and the bottom edge of the object. (The object is assumed to be rectangular in shape. The coordinates are your coordinates, i.e., user coordinates.)

### Detecting a Click in the Object

Somewhere in your program you will decide that you want to find out if the user has clicked in the object. Use the defined function G_Click like this:

```c
IF G_Click(thisone) = 1 then
    ! The user has clicked since the last time you checked
ELSE
    ! The user has not clicked since the last time you checked
END IF
```

The name of the object you wish to ask about appears within parentheses.

If the user has clicked several times since the last time you checked, your program will “see” only one of the clicks; the graphics object mechanism absorbs the others.

### Moving and Other Things

Even after you have created the graphics object, you can move it around. Just use

```c
CALL G_Move (thisone, deltax, deltay)
```

The second and third arguments give the amounts to move the object in the x (horizontal) direction and the y (vertical) direction, respectively.
You can hide a graphics object by using
   CALL G_Hide (thisone)
and can make it reappear by using
   CALL G_Show (thisone)
If the graphics object is hidden, it cannot generate user clicks.

If you want to find the location of a graphics object, including its size, use
   CALL G_GetRect (thisone, rect())

The vector rect() gives the coordinates of the object’s rectangle, in your user coordinate system:

   rect(1)    left edge
   rect(2)    right edge
   rect(3)    bottom edge
   rect(4)    top edge

The vector rect() must, of course, appear in a DIM or similar statement.

**Freeing**

If you want to get rid of a graphics object (and not just hide it,) use
   CALL G_Free (thisone)

**Demo Programs**

There are graphics object class demo programs in the TBDemos folder. The simplest is GOCDemo0.tru.

```
! GOCDemo0.tru
! Demonstrates a sensitive graphical object

LIBRARY "GOC.trc"

DECLARE DEF G_Click

PRINT "Test of graphics object class."
PRINT "Click on Close Box to QUIT or EXIT."
```
CALL G_Startup

CALL G_Create (gid1, "button1.BMP", .4, .4)

DO
    IF G_Click (gid1) = 1 then PRINT "Clicked 1"
LOOP

CALL G_ShutDown

END

This program contains all the essentials for using the Graphics Object Class. You must start with a LIBRARY statement that names the GOC.trc library file. You must include a DECLARE DEF statement to tell True BASIC that G_Click is a defined function (it is actually contained within the GOC.trc library file.) And you must include a CALL to the G_Startup subroutine.

You can now define one or more graphics objects. The CALL to the G_Create subroutine assigns an identification (id) number to the object, and names the file containing the image. The last two arguments are the (x,y) coordinates, in your user coordinates system, of the lower-left corner of the image. (All images are rectangular, even though they may appear otherwise on the screen.)

Next, you will include an event loop (DO loop) that continuously looks for clicks on the graphics objects by examining the value of the defined function G_Click; its argument specifies which object is to be examined. If G_Click = 1, then a click has been made on the graphics object since the last time G_Click was called for that object. If G_Click = 0, then no such click has occurred. If multiple clicks have been made in the interim, they are compressed into a single click.

This program terminates cleanly when the user clicks on the Close Box of the Output Window. If you use other means of exiting from the event loop, then be sure to make a CALL to G_Cleanup.

The remaining demo programs expand on the general theme. GOCDemo1.tru uses two objects to show that clicks are “stored” until asked for, and are then compressed into a single click. Clicks for the second object are not sought until there is a click on the first object.
GOCDemo2.tru uses five objects, two or which are walking feet. This example illustrates (a) determining the location of an object using a CALL to the G_GetRect subroutine, and (b) moving an object with a CALL to the G_Move subroutine.

Finally, GOCDemo3.tru expands on the theme of GOCDemo2.tru. It portrays sets of walking feet, and illustrates how to combine the use of the Graphics Object Class GOC.trc with a non-trivial set of subsequent actions.

The graphics images for all these demo programs are found in the folder Images.

**Errors**

The graphics class mechanism reports several errors; all with error number 800.

- Can’t call G_StartUp during module startup.
- Image file not available: filename
- Can’t call G_ShutDown during module startup.
- Object already there.
- Can’t move object not there.
- Invalid object id.
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 04 L4 C C E G 05 L2 C. 04 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:
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.

<table>
<thead>
<tr>
<th>Code</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>A to G</strong></td>
<td>Play a note in current octave, at current tempo, etc.</td>
</tr>
<tr>
<td><strong>L n</strong></td>
<td>Set the length of subsequent notes.</td>
</tr>
<tr>
<td><strong>ML</strong></td>
<td>Play music legato, or smoothly.</td>
</tr>
<tr>
<td><strong>MN</strong></td>
<td>Play music normally (not legato or staccato).</td>
</tr>
<tr>
<td><strong>MS</strong></td>
<td>Play music staccato, or briskly.</td>
</tr>
<tr>
<td><strong>O n</strong></td>
<td>Set current octave. Middle C is the first note in octave 5.</td>
</tr>
<tr>
<td><strong>R n or P n</strong></td>
<td>Rest (pause) for length n.</td>
</tr>
<tr>
<td><strong>T n</strong></td>
<td>Set the tempo.</td>
</tr>
<tr>
<td><strong># or +</strong></td>
<td>Sharp.</td>
</tr>
<tr>
<td><strong>-</strong></td>
<td>Flat.</td>
</tr>
<tr>
<td><strong>.</strong></td>
<td>Play dotted note.</td>
</tr>
</tbody>
</table>

**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.
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 opens an error window and displays an error message that gives the line and character numbers at which the error was detected. If you double-click on one of the error messages, True BASIC will place the cursor at the offending spot in your program. You can then correct that error and run the program again. Repeat if there are more than one error in the error window.

Consider the following program "WRONG":

```basic
PRINT "You are about to toss a coin"
IF rnd<.5 PRINT "Heads; win" else PRINT "Tails; lose"
```

When you run this program, True BASIC opens an "Errors" window with contents like this:

```
Errors

Untitled 1:2:1:Expected "then".
Untitled 1:3:1:Missing end statement.
```
The first error shows that an "illegal statement" was encountered at line 1, character 1. A missing "then" keyword was detected in line 2, character 11. Finally, it was seen that there is no "end" statement.

If you now double-click on the first line, True BASIC places the editing window cursor at line 1, character 1, or just in front of the word PRINT. You can now correct this word by double-clicking on it and then retyping it correctly, PRINT.

Repeat with the second and third lines in the "Errors" window.

```
PRINT  "You are about to toss a coin"
IF rnd<.5 then PRINT "Heads; win" else PRINT "Tails; 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 this Guide.

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 four 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 is mentioned in Appendix B, and the EXLINE, EXLINE$, EXTEXT$, and EXTYPE functions are explained in Appendix C.

**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.
Correcting Errors and Debugging

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 Chapter 10).
- Next, get a printed listing of your program and read it carefully (see Chapter 10).
- 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.

**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. Chapter 14 introduces the LOCAL statement.

- 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.
- Go into debug mode and insert breakpoints into your program.

**Breakpoints.** You can insert breakpoints into your program. When you run the program, True BASIC halts at each breakpoint and displays a list of variable names and their current values. Most of the time you can actually change the value of one or more of these variables. Type the CONTINUE command or select the menu item Continue to resume the program run. (For a review on using the command window, see Chapter 10.)

The first step is to turn debugging on by selecting the third item in the Settings menu.

To insert a breakpoint, move the cursor to the desired line and select **Break** in the Run menu, or type **Break** on the command line. You can insert as many breakpoints as you like. To remove a breakpoint, select the line and again type **Break** on the command line.

Now run your program. When True BASIC reaches a breakpoint, it opens a Variable window that displays all the variables in your program and their current values. You can actually change the values of some of them, but this must be done carefully! To continue running the program, select Continue in the Variable window menu, or type Continue on the command line. If you want to stop your program, select Stop from the Variable window menu.
If you accidentally close the Variable window, you can reopen it by selecting it from the Windows menu of Editing window.

**Debugging - A Case Study**

Let's take a very simple problem, adding up the numbers from 1 to some positive whole number which we will call \( n \). A program to do this might be:

```basic
! Sum of numbers from 1 to n
INPUT n
FOR i = 1 to n
   LET sum = sum + i
NEXT i
PRINT sum
END
```

When you run this program and enter 5, it will print 15 (the correct answer.) When you run the program again and enter 3, it will print 6 (again, the correct answer.)

Since you want to use this program more than once, you might have the brilliant idea of including it in a loop, so you can enter several numbers without having to Run the program from scratch each time. Here is one possible solution (notice that you have added an IF statement to allow the program to stop!)
When you run this program and enter 5, it prints 15 as it should. But when you now enter 3, it prints not 6, but 21, which is a wrong answer.

You might be able to see the problem, and the solution, immediately. But let’s see how we can use Debugging Mode, Breakpoints, and the Variable Window to help us.

Make sure Debug Mode is checked in the Settings menu. Now place the cursor in front of the line ‘LET sum = sum + 1’, which is the workhorse line in the program. Now choose Run from the Run menu. The program will stop almost immediately at the breakpoint. The Variable Window will look like this:

Everything looks okay. Continue the program by selecting Continue from the Run of the Variable Window, or by typing ‘continue’ in the command line, until it prints the result 15, in the Output Window.
Now, enter 3 when the ‘?’ appears. Notice the current status of the Variable Window.

Once you see this, you may figure out the solution; In this case add this line to your program:

```plaintext
LET sum = 0
```

just after the IF statement and just in front of the FOR statement. The program will now run correctly.

True BASIC always initialized numeric variables to 0. But if you reuse a variable in your program, you’ll have to set it to 0 yourself!
This table lists the ASCII character set. The order of characters determines how string conditions are evaluated. The decimal and hexadecimal equivalents given for each character are useful for advanced programmers.

<table>
<thead>
<tr>
<th>Decimal</th>
<th>Name</th>
<th>Hex</th>
<th>Decimal</th>
<th>Name</th>
<th>Hex</th>
</tr>
</thead>
<tbody>
<tr>
<td>000</td>
<td>nul</td>
<td>00</td>
<td>029</td>
<td>gs</td>
<td>1D</td>
</tr>
<tr>
<td>001</td>
<td>soh</td>
<td>01</td>
<td>030</td>
<td>rs</td>
<td>1E</td>
</tr>
<tr>
<td>002</td>
<td>stx</td>
<td>02</td>
<td>031</td>
<td>us</td>
<td>1F</td>
</tr>
<tr>
<td>003</td>
<td>etx</td>
<td>03</td>
<td>032</td>
<td>space</td>
<td>20</td>
</tr>
<tr>
<td>004</td>
<td>eot</td>
<td>04</td>
<td>033</td>
<td>!</td>
<td>21</td>
</tr>
<tr>
<td>005</td>
<td>enq</td>
<td>05</td>
<td>034</td>
<td>&quot;</td>
<td>22</td>
</tr>
<tr>
<td>006</td>
<td>ack</td>
<td>06</td>
<td>035</td>
<td>#</td>
<td>23</td>
</tr>
<tr>
<td>007</td>
<td>bel</td>
<td>07</td>
<td>036</td>
<td>$</td>
<td>24</td>
</tr>
<tr>
<td>008</td>
<td>bs</td>
<td>08</td>
<td>037</td>
<td>%</td>
<td>25</td>
</tr>
<tr>
<td>009</td>
<td>ht</td>
<td>09</td>
<td>038</td>
<td>&amp;</td>
<td>26</td>
</tr>
<tr>
<td>010</td>
<td>lf</td>
<td>0A</td>
<td>039</td>
<td>'</td>
<td>27</td>
</tr>
<tr>
<td>011</td>
<td>vt</td>
<td>0B</td>
<td>040</td>
<td>(</td>
<td>28</td>
</tr>
<tr>
<td>012</td>
<td>ff</td>
<td>0C</td>
<td>041</td>
<td>)</td>
<td>29</td>
</tr>
<tr>
<td>013</td>
<td>cr</td>
<td>0D</td>
<td>042</td>
<td>*</td>
<td>2A</td>
</tr>
<tr>
<td>014</td>
<td>so</td>
<td>0E</td>
<td>043</td>
<td>+</td>
<td>2B</td>
</tr>
<tr>
<td>015</td>
<td>si</td>
<td>0F</td>
<td>044</td>
<td>,</td>
<td>2C</td>
</tr>
<tr>
<td>016</td>
<td>dle</td>
<td>10</td>
<td>045</td>
<td>-</td>
<td>2D</td>
</tr>
<tr>
<td>017</td>
<td>dc1</td>
<td>11</td>
<td>046</td>
<td>.</td>
<td>2E</td>
</tr>
<tr>
<td>018</td>
<td>dc2</td>
<td>12</td>
<td>047</td>
<td>/</td>
<td>2F</td>
</tr>
<tr>
<td>019</td>
<td>dc3</td>
<td>13</td>
<td>048</td>
<td>0</td>
<td>30</td>
</tr>
<tr>
<td>020</td>
<td>dc4</td>
<td>14</td>
<td>049</td>
<td>1</td>
<td>31</td>
</tr>
<tr>
<td>021</td>
<td>nak</td>
<td>15</td>
<td>050</td>
<td>2</td>
<td>32</td>
</tr>
<tr>
<td>022</td>
<td>syn</td>
<td>16</td>
<td>051</td>
<td>3</td>
<td>33</td>
</tr>
<tr>
<td>023</td>
<td>etb</td>
<td>17</td>
<td>052</td>
<td>4</td>
<td>34</td>
</tr>
<tr>
<td>024</td>
<td>can</td>
<td>18</td>
<td>053</td>
<td>5</td>
<td>35</td>
</tr>
<tr>
<td>025</td>
<td>em</td>
<td>19</td>
<td>054</td>
<td>6</td>
<td>36</td>
</tr>
<tr>
<td>026</td>
<td>sub</td>
<td>1A</td>
<td>055</td>
<td>7</td>
<td>37</td>
</tr>
<tr>
<td>027</td>
<td>esc</td>
<td>1B</td>
<td>056</td>
<td>8</td>
<td>38</td>
</tr>
<tr>
<td>028</td>
<td>fs</td>
<td>1C</td>
<td>057</td>
<td>9</td>
<td>39</td>
</tr>
</tbody>
</table>
Below are three short True BASIC programs that can help you determine or verify character numbers from your keyboard.

**Program A** displays the printable character when you enter a `chr$` value.

**Program B** shows the character number when you press a keyboard key.

**Program C** asks you to enter the character abbreviation to verify the character number.

**Program A:**
```
PRINT "Shows the printable character"
PRINT "for a given character number"
DO
    INPUT n
    PRINT chr$(n)
LOOP
```

**Program B:**
```
PRINT "Shows the character number for a given key"
PRINT "Press a key"
DO
    IF key input then
        GET KEY k
        PRINT k
    END IF
LOOP
```

**Program C:**
```
PRINT "Shows the character number for a"
PRINT "given character or character abbreviation"
PRINT "Enter an abbreviation"
DO
    INPUT abb$
    WHEN error in
        LET n = ord(abb$)
        PRINT n
    USE
    PRINT "Invalid abbreviation"
END WHEN
LOOP
```

Below are three short True BASIC programs that can help you determine or verify character numbers from your keyboard.

**Program A** displays the printable character when you enter a `chr$` value.

**Program B** shows the character number when you press a keyboard key.

**Program C** asks you to enter the character abbreviation to verify the character number.
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 Guide. Information may also be found in the Help facility; type HELP or select the menu item HELP that appears at the top of the screen. Choose STATEMENTS from the list of topics displayed. (See Appendix F)

**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**
- **PAUSE**
- **RANDOMIZE**
- **REM**
- **STOP**

These statements deal with line-number programs; they are not discussed in this Guide, but can be found in the online HELP facility.

- **GOSUB**
- **GOTO**
- **ON GOTO**
- **RETURN**
- **ON GOSUB**
These statements allow setting various options; only OPTION ANGLE AND OPTION TYPO are discussed in this manual.

- OPTION ANGLE
- OPTION ARITHMETIC
- OPTION BASE
- OPTION COLLATE
- OPTION NOLET
- OPTION TYPO
- OPTION USING

**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 but appear in the HELP facility.

- ASK MARGIN
- ASK ZONEWIDTH
- SET MARGIN
- SET ZONEWIDTH

**File Statements**

The following file statements are discussed in this manual:

- CLOSE #n
- ERASE #n
- INPUT #n:
- LINE INPUT #n:
- OPEN #n:
- RESET #n:
- PRINT #n:

**Functions and Subroutines**

These statements are the heart and soul of organizing complicated programs.

- CALL
- DECLARE DEF (FUNCTION)
- 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 but appear in the HELP facility.

FUNCTION DECLARE NUMERIC
FUNCTION Structure DECLARE STRING
EXIT FUNCTION DECLARE SUB
END FUNCTION CHAIN

Graphics and Sound Statements
These graphics and sounds statements are discussed in this manual.

BOX AREA PICTURE Structure
BOX CIRCLE EXIT PICTURE
BOX CLEAR END PICTURE
BOX DISK PLAY
BOX ELLIPSE PLOT
BOX KEEP PLOT AREA
BOX LINES PLOT LINES
BOX SHOW PLOT POINTS
CLEAR PLOT POINTS
DRAW PLOT POINTS
FLOOD SET WINDOW
SET TEXT JUSTIFY

These graphics statements are not discussed in this manual but appear in the HELP facility.

ASK BACK GET POINT
ASK COLOR MAT PLOT
ASK COLOR MIX MAT PLOT AREA
ASK CURSOR MAT PLOT LINES
ASK DIRECTORY MAT PLOT POINTS
ASK MAX COLOR MAT PLOT LINES
ASK MAX CURSOR OPEN SCREEN
ASK MODE SET BACK
ASK NAME
ASK PIXELS SET COLOR
ASK SCREEN SET COLOR MIX
ASK TEXT JUSTIFY SET CURSOR
ASK WINDOW SET DIRECTORY
BOX DISK SET NAME
GET KEY SET NAME
GET MOUSE WINDOW
**MAT Statements**

Several of these MAT statements are discussed in this book.

- MAT PRINT
- MAT Assignment
- MAT INPUT
- MAT READ
- MAT LINE INPUT

Some of the MAT statements are not discussed in this book, but are found in the HELP facility.

- MAT REDIM
- MAT WRITE
- MAT PLOT AREA
- MAT PLOT LINES
- MAT PLOT POINTS

**Files Statements**

Several file statements are discussed in this manual. Additional statements, listed below, are described in the HELP facility.

- ASK #n: ACCESS
- ASK #n: DATUM
- 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

**Module Structures**

These statements, which deal with modules, are not discussed in this book but are described in the HELP facility.

- MODULE Structure
  - PRIVATE
  - DECLARE PUBLIC
  - PUBLIC
  - END MODULE
  - SHARE
**Exception Handling**

Exception handling is not discussed in this book, but these statements are described in the HELP facility:

- CAUSE ERROR or CAUSE EXCEPTION
- CONTINUE
- HANDLER
  - END HANDLER
  - EXIT HANDLER
- RETRY
- WHEN Structure
  - USE
  - END WHEN

**Debugging Statements**

Certain debugging statements required by ANSI are not discussed in this book. Instead, it is recommended that you use the Breakpoint feature discussed in Chapter 18.

- BREAK
- DEBUG
- TRACE

**Builtin Subroutines**

While not, strictly speaking, statements, True BASIC includes several builtin subroutines. They are not discussed in this Manual but are contained in the HELP facility.

- Clipboard
- ComLib
- ComOpen
- Divide
- Object
- Packb
- Read_Image
- System
- Sys_Event
- TBD
- Unpackb (a function, not a subroutine)
- Write_Image
Alphabetical Listing of Statements

This section gives examples and brief descriptions of the statements and structures discussed in this Guide. A wealth of additional information about True BASIC statements can be found in the HELP facility which is part of your True BASIC Bronze Edition. Select HELP from the main menu at the top of your screen.

**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 is 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.

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 input order allows the programmer to define new multi-line functions. The DEF structure may contain one or more EXIT DEF statements. 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 occurslexically 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 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.
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 state-
ments 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: stringvar$, ..., stringvar$
```

A LINE INPUT statement requests one or more lines of input from the user. The first line is supplied to 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 named. 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 named, 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 named 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 named, the values are assigned in “odometer” order – that is, the last subscript changes most rapidly, then the next to last, and so on.
A string variable can receive any valid datum. A numeric variable can receive only a datum that happens to be an unquoted string and a valid numeric-constant.

**NEXT Statement**
The NEXT statement can be used only as part of a FOR loop and is required.

**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, or FLOOD), 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 Plays the notes in the string. (See Chapter 17 for details.)

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;
PLOT point; ...
PLOT lines; ...
PLOT point;
PLOT 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; ...
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  (see Appendix H for more information)

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.

**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

```sql
SELECT CASE select-expression
CASE case-specifier
    ... 
CASE case-specifier
    ... 
CASE ELSE
    ... 
END SELECT
```

case-specifier:: case-part, ..., case-part

```sql
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, then an exception occurs.

SET BACK Statement

```sql
SET BACK colornumber
SET BACK colorname$
```

SET BACK is an abbreviation for SET BACKGROUND COLOR. SET BACK with color-number sets the background to the color that has that number. SET BACK with colorname$ sets the background to the color named; see the SET COLOR statement for a list of allowed color names.
**SET COLOR Statement**

SET COLOR colornumber

SET COLOR colorname$

SET COLOR with colornumber sets the foreground color to the color that has that number. Numbers outside this range will have effects that are dependent on the particular machine. If your machine does not support color, True BASIC may supply a suitable pattern.

SET COLOR with colorname$ sets the foreground color to the color named, which must be one of the following:

<table>
<thead>
<tr>
<th>Color</th>
<th>Color</th>
<th>Color</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAGENTA</td>
<td>CYAN</td>
<td>WHITE</td>
</tr>
<tr>
<td>RED</td>
<td>BLUE</td>
<td>GREEN</td>
</tr>
<tr>
<td>YELLOW</td>
<td>BROWN</td>
<td>BLACK</td>
</tr>
<tr>
<td>BACKGROUND</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**SET MODE Statement**

SET MODE mode$

Changes the current screen mode to that specified. If it is a legal but unavailable mode, True BASIC will set the nearest available mode. If it is not a legal mode, that is, it is not the name of any mode, True BASIC will set the default mode for that machine.

**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 EXITSUB 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 Structure

    WHEN EXCEPTION IN
    ...
    ! Protected part
    USE
    ...
    ! Executed if an exception is in a protected part
END WHEN

This subroutine may be used to “trap” run-time errors called exceptions. Examples might be division by 0 or attempting to open a file that doesn’t exist.

If an exception of any type occurs in the protected portion, the recovery statements between the USE statement and the END WHEN statement are executed. If no exception occurs in the protected part, the recovery statements are ignored.

The functions EXLINE, EXLINE$, EXTEXT$, and EXTYPE can be used to determine the exact nature of an exception.
This appendix lists most of True BASIC’s functions. Complete explanations may also be found in the Help facility; type **HELP** or select the menu item **HELP** that appears at the top of the screen. Choose **FUNCTIONS** from the list of topics displayed. *(See Appendix F)*

### Mathematical Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(x)</td>
<td>Absolute value</td>
</tr>
<tr>
<td>ACOS(x)</td>
<td>Arcosine</td>
</tr>
<tr>
<td>ANGLE(x,y)</td>
<td>Angle between x-axis and (x,y)</td>
</tr>
<tr>
<td>ASIN(x)</td>
<td>Arcsine</td>
</tr>
<tr>
<td>ATN(x)</td>
<td>Arctangent</td>
</tr>
<tr>
<td>CEIL(x)</td>
<td>Ceiling (-INT(-x))</td>
</tr>
<tr>
<td>COS(x)</td>
<td>Cosine</td>
</tr>
<tr>
<td>COSH(x)</td>
<td>Hyperbolic cosine</td>
</tr>
<tr>
<td>COT(x)</td>
<td>Cotangent</td>
</tr>
<tr>
<td>CSC(x)</td>
<td>Cosecant</td>
</tr>
<tr>
<td>DEG(x)</td>
<td>Translates radians to degrees</td>
</tr>
<tr>
<td>EPS</td>
<td>Smallest nonzero positive number</td>
</tr>
<tr>
<td>EXP(x)</td>
<td>Exponential function</td>
</tr>
<tr>
<td>FP(x)</td>
<td>Fractional part of x</td>
</tr>
<tr>
<td>INT(x)</td>
<td>Integer part</td>
</tr>
<tr>
<td>IP(x)</td>
<td>Greatest integer &lt;= x</td>
</tr>
<tr>
<td>LOG(x)</td>
<td>Natural logarithm</td>
</tr>
<tr>
<td>LOG10(x)</td>
<td>Common logarithm (base 10)</td>
</tr>
<tr>
<td>LOG2(x)</td>
<td>Logarithm to the base 2</td>
</tr>
<tr>
<td>MAX(x,y)</td>
<td>Larger of two numbers</td>
</tr>
<tr>
<td>MAXNUM</td>
<td>Largest positive number</td>
</tr>
</tbody>
</table>
**Mathematical Functions (continued)**

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>MIN(x,y)</td>
<td>Smaller of two numbers</td>
</tr>
<tr>
<td>MOD(x,y)</td>
<td>Remainder when x is divided by y</td>
</tr>
<tr>
<td>PI</td>
<td>Value of pi</td>
</tr>
<tr>
<td>RAD(x)</td>
<td>Translates degrees to radians</td>
</tr>
<tr>
<td>REMAINDER(x,y)</td>
<td>Remainder of x divided by y</td>
</tr>
<tr>
<td>RND</td>
<td>Random number between 0 and 1</td>
</tr>
<tr>
<td>ROUND(x,n)</td>
<td>Rounds x to n decimal places</td>
</tr>
<tr>
<td>SEC(x)</td>
<td>Secant</td>
</tr>
<tr>
<td>SGN(x)</td>
<td>Sign of x</td>
</tr>
<tr>
<td>SIN(x)</td>
<td>Sine</td>
</tr>
<tr>
<td>SINH(x)</td>
<td>Hyperbolic sine</td>
</tr>
<tr>
<td>SQR(x)</td>
<td>Square root</td>
</tr>
<tr>
<td>TAN(x)</td>
<td>Tangent</td>
</tr>
<tr>
<td>TANH(x)</td>
<td>Hyperbolic tangent</td>
</tr>
<tr>
<td>TRUNCATE(x,n)</td>
<td>Truncates x to n decimal places</td>
</tr>
</tbody>
</table>

**Date and Time Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DATE</td>
<td>Year and day of year as a number</td>
</tr>
<tr>
<td>DATE$</td>
<td>Year, month, and day of month as a string</td>
</tr>
<tr>
<td>TIME</td>
<td>Seconds since midnight</td>
</tr>
<tr>
<td>TIME$</td>
<td>24-hour clock time as a string</td>
</tr>
</tbody>
</table>

**String to Number Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CHR$(x)$</td>
<td>Character represented by ASCII number x</td>
</tr>
<tr>
<td>ORD(x$)</td>
<td>Ordinal position of x$ in ASCII character set</td>
</tr>
<tr>
<td>NUM(x$)</td>
<td>Numeric value of IEEE 8-byte string</td>
</tr>
<tr>
<td>NUM$(x)$</td>
<td>IEEE 8-byte equivalent of numeric value</td>
</tr>
<tr>
<td>STR$(x)$</td>
<td>Changes number to a string</td>
</tr>
<tr>
<td>VAL(x$)</td>
<td>Changes string containing digits to a number</td>
</tr>
</tbody>
</table>
### String Transforming Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LCASE$(x$)$</td>
<td>Change letters to lowercase</td>
</tr>
<tr>
<td>UCASE$(x$)$</td>
<td>Change letters to uppercase</td>
</tr>
<tr>
<td>LTRIM$(x$)$</td>
<td>Remove leading blanks</td>
</tr>
<tr>
<td>RTRIM$(x$)$</td>
<td>Remove trailing blanks</td>
</tr>
<tr>
<td>TRIM$(x$)$</td>
<td>Remove leading &amp; trailing blanks</td>
</tr>
<tr>
<td>REPEAT$(x$,n)$</td>
<td>$x$ repeated n times</td>
</tr>
</tbody>
</table>

### String Search Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>LEN$(x$)$</td>
<td>Number of characters in $x$</td>
</tr>
<tr>
<td>POS$(x$,y$,n)$</td>
<td>First occurrence of $y$ in $x$ after character $n$</td>
</tr>
<tr>
<td>POSR$(x$,y$)</td>
<td>Ditto POS but starting from the end</td>
</tr>
<tr>
<td>CPOS$(x$,y$)</td>
<td>First occurrence in $x$ of any character in $y$</td>
</tr>
<tr>
<td>CPOSR$(x$,y$)</td>
<td>Ditto CPOS but starting from the end</td>
</tr>
<tr>
<td>NCPOS$(x$,y$)</td>
<td>First occurrence in $x$ of any character not in $y$</td>
</tr>
<tr>
<td>NCPOSR$(x$,y$)</td>
<td>Ditto NCPOS but starting from the end</td>
</tr>
</tbody>
</table>

### Array Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>DET$(a)$</td>
<td>Determinant of the square matrix $a$</td>
</tr>
<tr>
<td>DOT$(a,b)$</td>
<td>Dot product of vectors $a$ and $b$</td>
</tr>
<tr>
<td>LBOUND$(a,n)$</td>
<td>Lower bound of dimension $n$ for array $a$</td>
</tr>
<tr>
<td>UBOUND$(a,n)$</td>
<td>Upper bound of dimension $n$ for array $a$</td>
</tr>
<tr>
<td>SIZE$(a,n)$</td>
<td>Number of element in dimension $n$ of array $a$</td>
</tr>
</tbody>
</table>

### MAT Functions \( \text{that can appear only in MAT assignment statements} \)

<table>
<thead>
<tr>
<th>Function</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>CON</td>
<td>Array of ones</td>
</tr>
<tr>
<td>IDN</td>
<td>Identity matrix</td>
</tr>
<tr>
<td>INV$(a)$</td>
<td>Inverse of array $a$</td>
</tr>
<tr>
<td>NUL$</td>
<td>Array of empty strings</td>
</tr>
<tr>
<td>TRN$(a)$</td>
<td>Transpose of array $a$</td>
</tr>
<tr>
<td>ZER</td>
<td>Array of zeroes</td>
</tr>
</tbody>
</table>
The descriptions in the alphabetical list use the following terms:

- (numeric-expression) numeric expression
- (rnumeric-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.

**ACOS Function**

ACOS(numeric-expression)

Returns the value of the arccosine function. The result is given in radians or degrees depending on whether the current OPTION ANGLE is RADIANS (default) or DEGREES.

**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).

**ASIN Function**

ASIN(numeric-expression)

Returns the value of the arcsine function. The result is given in radians or degrees depending on whether the current OPTION ANGLE is RADIANS (default) or 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 RADI-
ANS (default) or DEGREES. The angle will always be in the range -90 < ATN(x) < 90 (assuming that the current OPTION ANGLE is DEGREES).

**CEIL Function**

\[ \text{CEIL}(\text{numeric-expression}) \]

Returns the least integer that is greater than or equal to numeric-expression. For example, CEIL(1.9) = 2, CEIL(13) = 13, and CEIL(-2.1) = -2.

**CHR$ Function**

\[ \text{CHR}$(\text{rnumeric-expression}) \]

Returns the character whose ASCII decimal number is rnumeric-expression (see Appendix A). If rnumeric-expression is not in the range 0 to 255, inclusive, exception 4002 occurs.

**CON Array Constant**

\[ \text{CON redim} \]

\[ \text{CON} \]

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**

\[ \text{COS}(\text{numeric-expression}) \]

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.

**COSH Function**

\[ \text{COSH}(\text{numeric-expression}) \]

Returns the value of the hyperbolic cosine function.

**COT Function**

\[ \text{COT}(\text{numeric-expression}) \]

Returns the value of the cotangent 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, rnumeric-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, rnumeric-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.

CSC Function
CSC(numeric-expression)
Returns the value of the cosecant function. The argument is assumed to be in radians or degrees depending on whether the current OPTION ANGLE is RADIANS (default) or DEGREES.

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.

**EXLINE Function**

EXLINE

EXLINE returns the line number in your program where the most recent error occurred. If your program does not have line numbers, EXLINE returns the ordinal number of the line in the file.

**EXLINE$ Function**

EXLINE$

EXLINE$ returns a string that gives the location in your program where the most recent error occurred. It gives the number of the line and the routine in which the error occurred. If the error occurred deeply in nested subroutine calls, EXLINE$ returns the genealogy of the error; i.e., it includes the names of the intervening subroutines and the line numbers of the CALL statements.
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.

EXTEXT$ Function

    EXTEXT$

EXTEXT$ returns the error message associated with the most recent error, if any, provided that the error was trapped in an error handler (see Chapter 18.) If an error is not trapped, True BASIC prints the error message and stops the program.

EXTYPE Function

    EXTYPE

EXTYPE returns the error number of the most recent error, provided that the error was trapped by an error handler (see Chapter 18.) Some of the error numbers are given in Appendix D, along with the associated error messages.

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, rnumeric-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 rnumeric-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 \(e = 2.718281828\ldots\).

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.

MAXLEN Function

MAXLEN (strvar)

Returns the maximum length (maximum number of characters) for the string variable or, if \( strvar \) refers to an array, the maximum length for each string in the array. If there is no determinable maximum length, MAXLEN returns MAXNUM.

MAXNUM Function

MAXNUM

A no-argument function, MAXNUM returns the largest number that can be represented in your computer.

MAXSIZE Function

MAXSIZE (arrayarg)

MAXSIZE always returns \( 2^{31} \).

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)
Returns x modulo y, provided y is not equal to zero.

NCPOS Function
NCPOS(string-expression, string-expression)
NCPOS(string-expression, string-expression, rnumeric-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, rnumeric-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.
NUM Function

NUM (strex)

NUM returns the numerical value that is stored as a string, which must contain exactly eight characters, using the IEEE eight-byte format. Normally, the string will have been previously constructed with the NUM$ function.

NUM$ Function

NUM$ (numex)

NUM$ returns a string of length eight that contains the numerical value using the IEEE eight-byte format. Normally, the NUM function must be used to convert the string back to a number.

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 ("" denotes the null string.) 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).

POS Function

POS(string-expression, string-expression)

POS(string-expression, string-expression, rnumeric-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, rnumeric-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, rnumeric-expression)

Returns the string consisting of rnumeric-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, include a RANDOMIZE statement early in the program.
ROUND Function
ROUND(numeric-expression, rnumeric-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.

RUNTIME Function
RUNTIME
A no-argument function, RUNTIME returns the number of seconds of processor time used since the start of execution. It may not return a meaningful value on some computers.

SEC Function
SEC(numeric-expression)
Returns the value of the secant function. The argument is assumed to be in radians or degrees depending on whether the current OPTION ANGLE is RADIANS (default) or DEGREES.

SGN Function
SGN(numeric-expression)
SGN(x) returns the “sign” of x, which will be –1, 0, or +1.

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.

SINH Function
SINH(numeric-expression)
Returns the value of the hyperbolic sine function.
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 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 or might be produced by the PRINT statement.

STRWIDTH Function
STRWIDTH$(numeric-expression, string-expression)
Returns the length of the string, in pixels, with reference to the current font, font-style, and font-size in the current physical window. If the value of the first argument is not the ID number of a physical window, an error occurs.

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 the start of the print position (column) 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.
TANH Function
TANH (numeric-expression)
Returns the value of the hyperbolic tangent function.

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 and is displayed in the form “HH:MM:SS”.

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, rnumeric-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, rnumeric-expression)
UBOUND(arrayarg)
The two-argument form returns the largest value (upper bound) allowed for the subscript in the dimension specified by rnumeric-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 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.
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 errors (exceptions) that occur when the program runs. These numbers can be used with the WHEN structure and EXTYPE function.

**Argument for SIN, COS, or TAN too large.** (-3050)
The argument for the sine, cosine, or tangent function is so large that range reduction results in almost complete loss of precision.

**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.

**Array too large** (5001)
You’ve tried to redimension an array to a size larger than the original DIM statement. Change the DIM statement, or use MAT REDIM.

**ASIN or ACOS argument must be between 1 and -1.** (3007)
The arcsine and arccosine functions are not defined for arguments larger than one in absolute value.
Badly formed using string. (8201)
The using string in PRINT USING statement is incorrectly formed.

Badly formed input line (nonfatal). (8102)
Your reply to an INPUT statement is badly formed. Most likely you have not properly matched up opening and closing quote marks. You will be requested to reenter the entire input line.

Badly formed input line from file. (8105)
The reply to an INPUT statement from a file is badly formed. Most likely you have not properly matched up opening and closing quote marks.

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 open PRINTER (9101)
You have tried to open the printer but True BASIC has been informed that the attempt has failed, either because the printer isn’t attached or has not been turned on. (This condition cannot be detected on all machines.)

Can’t output to INPUT file. (7302)
You may not write data to a file which was opened with ACCESS INPUT. If you must output to this file, change the OPEN statement to use ACCESS OUTIN.

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 ANGLE(0,0). (3008)
ANGLE(0,0) is not defined. Make sure that at least one of its arguments is nonzero.

Can’t use #0 here. (nonfatal) (7002)
You’ve tried to use #0 as a channel number for a file or window other than the default output window.

Can’t use READ or WRITE for TEXT file. (-8503)
The file is a text file; the allowed commands are PRINT, INPUT, and LINE INPUT.
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 guide to see how the structured statements are formed.

Channel is already open. (7003)
You’ve tried to open a file or window using a channel number currently in use.

Channel isn’t a window. (-11005)
You’ve used a window instruction with a channel number that refers to a file.

Channel isn’t open. (7004)
You’ve tried to use a channel number (for a file or window) without using the OPEN statement.

Channel number must be 1 to 1000. (7001)
All channel numbers must be in the range 1 to 1000, except for #0, which refers to the output window.

Constant too large: constant in routine.
The numeric constant displayed is too large for your computer to handle. Type PRINT MAXNUM to see the largest possible number on your computer, and then change your program to use a smaller number.

Data item isn’t a number. (8101)
You’ve used a numeric variable in a READ statement but the matching DATA item is not a 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:

```truebasic
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 close an Editing window or Quit your True BASIC session without saving your current file. Choose “Save” if you do want to save the file (replacing the current saved copy), “Discard” 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 a 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 ....)

**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. (True BASIC also supports OPTION ARITHMETIC, OPTION COLLATE, and OPTION USING; the first two are ignored.)

**Illegal parameter.**
You’ve written a SUB or DEF or PICTURE line, defining a routine. 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. Make sure that your 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 requires that the second dimension of the first matrix must equal the first dimension of the second matrix.

Mismatched string array sizes. (6101)
You’re using a MAT statement with concatenation of string arrays, and the arrays are not the same size.

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. (True BASIC “reserves” very few names. In addition to all no-argument function names, True BASIC reserves only ELSE, NOT, PRINT and REM.)
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. Go back and write 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.

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. 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.
Out of memory. (5000)
Your problem requires more memory than is attached to your computer. On some platforms, you may be able to increase the memory allocated to True BASIC or you might be able to turn on “virtual memory.” If these simple measures fail, you may need to purchase additional memory (RAM).
If this is not an option, here are a few suggestions for memory conservation:
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 by using smaller arrays.
Compile your program, and use the compiled version.
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. (1002)
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
```

Overflow in DET or DOT. (1009)
You have generated an overflow in the course of evaluating the DET or DOT function.

Overflow in INPUT (nonfatal). (1007)
You have entered as input a number that is too large. You will be required to reenter the entire input line.

Overflow in MAT operation. (1005)
You have generated an overflow in the course of evaluating a MAT operation.

Overflow in numeric constant. (1001)
You have used a numeric constant that is just too large, as in LET x = 1e1000.
**Overflow in numeric function. (1003)**
You have generated an overflow in the course of evaluating a function, such as EXP or TAN.

**Overflow in READ. (1006)**
An overflow was generated in the course of reading a number from a data statement.

**Overflow in VAL. (1004)**
You have generated an overflow in the course of evaluating the VAL function.

**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 “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 have selected Stop from one of the menus. The program has stopped.

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.

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.

SIZE index out of range. (4004)
You’re trying to take Size(A,3), for instance, when the array A has fewer than three dimen-
sions. Check the relevant DIM statement to see how many dimensions the array has. The
second argument must lie between 1 and this number.

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 (nonfatal). (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.

Subscript out of bounds. (2001)
You’ve given an array subscript which lies outside the array’s bounds. Try printing the sub-
script and then using Lbound and Ubound to find the array’s bounds.

System error.
An error has occurred in the True BASIC system itself. Record the system error and con-
tact customer support by FAX or e-mail. Thank you.

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 breakpointed 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 (nonfatal). (8002)
You’ve executed an INPUT statement, and the input reply doesn’t contain as many items as the INPUT statement requested. You will be requested to reenter the entire 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 (nonfatal). (8003)
You’ve executed an INPUT statement, and the input reply line contains more items than the INPUT statement requested. You will be requested to reenter the entire input line.

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 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 or select the menu item: HELP for True BASIC)
You've typed a command that True BASIC doesn't understand. If you want further help from the computer, just type HELP in the command window or use the Help menu for more instructions. When the HELP window appears, choose COMMANDS from the topics list. (Also, see Appendix F for more about the HELP facility.)

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 manual illustrates the operation True BASIC using menu selections. Many menu items can be selected using one or more keystrokes. This appendix lists the keystroke alternatives used on the Macintosh and Windows versions of True BASIC.

The accelerator and hot key equivalents are quite different on the two platforms.

**M** On the Macintosh, the hot key consists of holding down the command key [⌘] and pressing one of the twenty-six letters. For example, to Quit a Macintosh program, one can usually use command-Q or command-q (upper or lowercase doesn't matter.)

On the Macintosh, for any particular window there can be no duplicate hot keys. That is command-A cannot mean two things. On Windows, there can be no duplicates within a single menu, but different menus can use the same letter.

**W** On Windows, such keystroke equivalents are called Accelerators and consist of pressing the Alt key, followed by the first letter of the menu header, following by the underlined letter in the menu item. For example, to Exit a Windows application, one can usually use Alt, followed by F or f (since the Exit item is usually in the File menu,) followed by X or x (since that is the underlined letter in the Exit item.

Note that either uppercase or lowercase letters may be used in all instances. On the Windows side, if the menu items are displayed, then only the final letter is required. Also note that Alt - 1 will display the File Menu (the first menu,) Alt - 2 will display the Edit Menu (the second menu,) and so on.

A listing of the keystroke equivalents currently implemented in the BRONZE Edition of True BASIC follows:
<table>
<thead>
<tr>
<th><strong>Macintosh</strong></th>
<th><strong>Windows</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>File Menu</strong></td>
<td><strong>File menu</strong></td>
</tr>
<tr>
<td>New ⌘-N</td>
<td>New Alt - F - N</td>
</tr>
<tr>
<td>Open ⌘-O</td>
<td>Open Alt - F - O</td>
</tr>
<tr>
<td>Close ⌘-W</td>
<td>Close Alt - F - C</td>
</tr>
<tr>
<td>Save ⌘-S</td>
<td>Save Alt - F - S</td>
</tr>
<tr>
<td>Save As</td>
<td>Save As Alt - F - A</td>
</tr>
<tr>
<td>Uns &lt;s&gt;ave&lt;/s&gt;</td>
<td>Uns &lt;s&gt;ave&lt;/s&gt; Alt - F - U</td>
</tr>
<tr>
<td>Page Setup</td>
<td>Page Setup</td>
</tr>
<tr>
<td>Print ⌘-P</td>
<td>Print Alt - F - P</td>
</tr>
<tr>
<td>Print Selection ⌘-Q</td>
<td>Print Selection Alt - F - L</td>
</tr>
<tr>
<td>Quit</td>
<td>Quit Alt - F - X</td>
</tr>
<tr>
<td><strong>Edit Menu</strong></td>
<td><strong>Edit menu</strong></td>
</tr>
<tr>
<td>Undo ⌘-Z</td>
<td>Undo Alt - E - T</td>
</tr>
<tr>
<td>Cut ⌘-X</td>
<td>Cut ⌘-E - T</td>
</tr>
<tr>
<td>Copy ⌘-C</td>
<td>Copy Alt - E - C</td>
</tr>
<tr>
<td>Paste ⌘-V</td>
<td>Paste Alt - E - P</td>
</tr>
<tr>
<td>Find ⌘-F</td>
<td>Find Alt - E - F</td>
</tr>
<tr>
<td>Find Again ⌘-G</td>
<td>Find Again Alt - E - G</td>
</tr>
<tr>
<td>Change</td>
<td>Change Alt - E - N</td>
</tr>
<tr>
<td>Keep ⌘-K</td>
<td>Keep Alt - E - K</td>
</tr>
<tr>
<td>Include ⌘-I</td>
<td>Include Alt - E - I</td>
</tr>
<tr>
<td>Select All ⌘-A</td>
<td>Select All Alt - E - A</td>
</tr>
<tr>
<td>Move To ⌘-Y</td>
<td>Move To Alt - E - M</td>
</tr>
<tr>
<td><strong>Run Menu</strong></td>
<td><strong>Run Menu</strong></td>
</tr>
<tr>
<td>Run ⌘-R</td>
<td>Run Alt - R - R</td>
</tr>
<tr>
<td>Breakpoint</td>
<td>Breakpoint Alt - R - P</td>
</tr>
<tr>
<td>Compile ⌘-T</td>
<td>Compile Alt - R - C</td>
</tr>
<tr>
<td>Bind ⌘-B</td>
<td>Bind Alt - R - B</td>
</tr>
<tr>
<td>Trace</td>
<td>Trace Alt - R - T</td>
</tr>
<tr>
<td>Do ...</td>
<td>Do ...</td>
</tr>
</tbody>
</table>
Macintosh

Run Menu (continued)
- Do Format Alt - R - D
- Do Upper Alt - R - U
- Do Lower Alt - R - L
(additional items)

Windows

Run Menu (continued)
- Do Format Alt - R - D
- Do Upper Alt - R - U
- Do Lower Alt - R - L
(additional items)

Note: The Bind item is not active in the BRONZE Edition.
The tools required to create standalone applications is part of the SILVER Edition.
The Trace item will continue to be enhanced in future editions.

Window Menu
- Open Error Window
- Command Window Alt - J
- Variable Window
(additional items)

Settings Menu
- Set Font Alt - S - T
- Backup on Save
- Debug Mode
- Confirm Quit
- Save Configuration

Help Menu
- About Help
- About True BASIC

For the Macintosh, notice that there are no duplications. But for the Windows side, there are duplications. For example, P stands for a Breakpoint in the Run Menu and for Print in the File Menu.

You will also notice menus on the Command Window, the Errors Window (if you have errors in your program!) and the default Output Window.
Macintosh

COMMAND Window

File Menu
Quit ⌘-Q

Edit Menu
Copy ⌘-C
Paste ⌘-V

ERROR MESSAGE Window

Errors Menu
Locate Error ⌘-F
Close Error Window ⌘-W

OUTPUT Window

File Menu
Print ⌘-P
Stop ⌘-. Stop

And in Debug mode, a Variable Window is shown at appropriate times. Its menus are:

VARIABLE Window

File Menu
Close ⌘-W

Edit Menu
Copy ⌘-C
Paste ⌘-V

Run Menu
Continue ⌘-R
Stop ⌘-. Stop

Windows

File Menu
Exit Alt - F - X

Edit Menu
Copy Alt - E - C
Paste Alt - E - P

Errors Menu
Locate Error Alt - E - W

File Menu
Print Alt - F - P
Stop Alt - F - T

File Menu
Close Alt - F - C

Edit Menu
Copy Alt - E - C
Paste Alt - E - P

Run Menu
Continue Alt - R - C
Stop Alt - R - T
The Enter Keys
On most keyboards there may be two keys labelled “Enter”. One is located on the right
side of alphabetic portion of the keyboard. It is sometimes labelled “Enter” or “<-|”
Enter”, and sometimes “Return”. This key gives you a new line if pressed while in the
Editing window, and causes a command to be carried out if in the Command window.
If a dialog box is being displayed, the “Return” key can often be used as the equivalent
of pressing the outlined button.

There is another “Enter” on some keyboards. It is located as part of the numeric key-
pad on the extreme right of most keyboards. Pressing it does different things on dif-
ferent platforms. On Windows machines, it is the equivalent of the Return or Enter key
of the alphabetic portion of the keyboard. On the Macintosh, pressing it will cause the
Command window to appear (if it is invisible) or to become active. Thus, pressing this
Enter key on most Macintoshes will move the cursor from one of the Editing windows
to the command line of the Command window.

You should also be aware that the ASCII value of the “Return” and “Enter” keys differ
between the Windows and Mac operating systems. On page xx you will find a short pro-
gram that can help you determine the ASCII value of any key you press.

The Delete Keys
There are several keys that can be used to delete characters or selected text. On the
Macintosh, there is the "Delete" key located on the top right of the alphabetic portion
of the keyboard. Pressing it will delete the character just in front of the cursor. On
Windows machines, this key is called "<- Backspace", or simply "<-".

If you wish to delete the character just to the right of the cursor, press the "box-x del"
key located just above the cursor keys and just to the right of the alphabetic portion of
the Macintosh keyboard. On Windows machines, use the "Delete Next" or just "Delete"
key located in roughly the same place. Also, on Windows machines, you can use the
"Del" key that is part of the Numeric Keypad, provided that the Num Lock option is
turned off.
Online Help

All of the True BASIC statements are functions, and most of the menu selections and typed commands, are explained in the online help facility. To access this facility, select “Help...” from the “Help for True BASIC” menu in any editing window:

```
or type the command “help” on the command line. A HELP facility window will now appear.
```

HELP is divided into four major categories or topics: **Commands**, **Functions**, **Menus**, and **Statements**. Move to the “Topics” select box. When you select “Topics” you will see the four available categories.
Choose the one you want by clicking on “Topics”, moving the mouse to the topic you want, and releasing the mouse. The “Items” button just below will now contain all the items under that topic.

Choose the particular item of interest by clicking on the button just below, moving the mouse up or down the list until you seen the chosen item. In this example we have selected “Statements”. In the “Items” select list you will see all the True BASIC statements in alphabetic order.
In this example we have selected the BOX KEEP statement.

As soon as you make your selection, the information about the selected statement is displayed in the larger text section of the window. Long explanations can be read by using the scroll bar.

Using the “Edit” menu in the Help Window allows you to “Copy” and “Paste” information from the Online Help. This is especially handy if you wish to copy example code from the Online Help window and place it in your program for enhancement or customization.
If you don’t immediately see what you want, but can think of a key word that might be contained in an explanation, just type in the key word, or partial word, in the text input box above the “Find” button, and click on the “Find” button.

In this example we have entered “degree” in the FIND box. A list of all the items that contain that key word or phrase will now be listed in the “Items” section. Choose one of these items and you will see the expanded text.

To return to a main Topic, just select that Topic again.

The Topic “Functions” contains descriptions of all of True BASIC’s functions, and is taken directly from the reference manual, Chapter 18.

The Topic “Statements” contains descriptions of all the True BASIC statements, and is taken directly from the reference manual, Chapter 18.

The Topic “Commands” contains brief descriptions of the commands that you can type on the command line.

The Topic “Menus” contains brief descriptions of the menu items that you can select. For menu shortcuts, see Appendix E.
Making Your Own DO Programs

You may have noticed the directory “TBDo”, which contains several so-called “DO programs.” Actually, they are not regular programs, but are subroutines. They are designed to operate on the text file in the current editing window, but can be made to do just about anything.

You can make your own DO programs. Follow these simple steps:

1. Create a library file, carefully choosing its name.
2. On the first lines of the library file, enter

   \texttt{EXTERNAL}

   \texttt{SUB \ xxxxx (current\$(\), options\$)}

3. Now write what you want to do, which may involve modifying the lines or the current file.
4. At the end of the file, enter

   \texttt{END \ SUB}

\textit{Note: the actual name of the subroutine is irrelevant! A DO program is always identified by the name of the file containing it!}

Now save this file in the directory TBDo. When True BASIC starts up, the name or your new do program file will appear in the Run menu along with the names of all the other do programs.

You can invoke a do program in two ways. You can select the menu item “Do ...” in the “Run” menu, or you can type the command “do filename” on the command line. (Of course, you’ll actually type the file name you have selected.)

If you use the menu selection method, you may have to navigate the file system to find the directory TBDo. Then you’ll also be asked for the the command line parameters. Whatever you enter will then be assigned to the second argument in the calling
sequence, `options$`. If you use typed commands, anything you type following the `do` command itself and a comma will be similarly assigned. (For the typed command, you’ll automatically use the TBDo directory; see the discussion of aliases in Chapter 15.)

Here is a simple example: Suppose you want a do program that will change all uppercase letters into lowercase, or all lower case letters into uppercase. Write the following subroutine:

```plaintext
EXTERNAL
SUB xxxxx (current$(), options$)
    LET options$ = ltrim$(lcase$(options$))[1:1]
    FOR i = 1 to ubound(current$())
        IF options$ = "u" then
            LET current$(i) = ucase$(current$(i))
        ELSE IF options$ = "l" then
            LET current$(i) = lcase$(current$(i))
        ELSE
            PRINT "Use either 'upper' or 'lower'"
            EXIT SUB
        END IF
    NEXT i
END SUB
```

Now save it with the name “ChangeCase” in the directory TBDo.

To use your new DO program to change all uppercase letters to lowercase in your current program, type the command

```
do changecase, lower
```

Conversely, if you want to change to all uppercase, type the command

```
do changecase, upper
```

That’s all there is to it.

If you put your very own DO program in the directory (folder) TBDo, its name will appear in the Run menu the next time you start True BASIC. If you put it into a different directory, you can access it by selecting “Do...” from the Run menu, using the Finder to locate it, and then clicking on “Open”. In any case, you will be asked if there are any command-line parameters; whatever you enter will be supplied as the value of `options$` in the call to the DO program.
Several DO programs are already in the directory (folder) TBDo. There are three built-in ones that exist outside the TBDo folder is empty. They are:

- **Do Format**
- **Do Upper**
- **Do Lower**

**Do Format** formats your program by capitalizing some key words, and indenting the insides of loops and other structures.

**Do Upper** and **Do Lower** operate only on text that has been selected, and changes all letters in the selected region to uppercase (**Do Upper**) or lowercase (**Do Lower**).

The remaining DO programs are found in the TBDo directory. Three of them deal with adding line numbers to your program (**DoNumber**), removing them (**DoUnNum**), or changing them (**DoReNum**). The parameters for **DoNum** allow you to specify the starting line number, and the line number spacing. If you leave the parameters blank, you’ll get 1000 as the starting line number, and 10 as the spacing. If you would prefer to start with, say, 10000 and have a spacing of 100, you could use

10000, 100

as the parameter values.

The parameters for **DoReNum** are the same as those for **DoNum**.

**DoSort** will sort your current file using the ASCII sorting sequence (all uppercase letters come before all lowercase letters!) You would never want to do this with a real program, but this might be useful if your current file happens to be a list of names.

**DoSaveText** allows you to take the text in your current Source Code window and convert the line-endings for use on different operating systems. The line-ending marks for the most popular operating systems are:

- Windows, DOS, OS/2: Carriage Return + Line Feed
- Macintosh: Carriage Return
- Unix, Linux: Line Feed

You can select one of the following parameters to specify the platform:

- DOS
- Windows
- OS\2
- Unix
- Macintosh

For Linux, use Unix.
DoXRef will produce a cross-reference of your current program file. All keywords will be indentified, and located by giving the line numbers of the line in which they appear. Try it on a program of your own, but start with a small program as the DoXRef output is lengthy. The output will be sent to the printer unless you specify a file name as a parameter.

DoJoin and DoMakeApp have to do with preparing TrueApps, subjects and procedures that are discussed in How-To files that can be downloaded from the True BASIC website or found on the True BASIC Annual-CD’s.

The DO programs currently in the directory TBDo happen to be compiled, although they need not be. The source code for all except DoMakeApp can be found in the subdirectory sources. (The names of the source files are slightly different; for example, the source code for DoNum is called NUMBER.TRU.) You can change the source code as you see fit, compile it, and re-save it in TBDo, renaming it if desired. Thus, you can custom-fit any of the DO programs to suit your own purposes.
True BASIC normally prints numbers in a form convenient for most purposes. But on occasion you may prefer a more elaborate form. For example, you may want to print financial quantities with two decimal places (for cents) and, possibly, with commas inserted every three digits to the left of the decimal point. PRINT USING provides a way to print numbers in this and almost any other form.

Here is an example of the PRINT USING statement.

```
PRINT USING format$: x, y, z
```

*Format*$ is a string of characters that contains the instructions to PRINT USING for “formatting” the printing of $x, y,$ and $z$. This string is called a *format string*. It may be a string variable (as shown above), a *quoted-string*, or a more general string expression.

PRINT USING also allows one to print strings centered or right-justified, as well as left-justified. (The normal PRINT statement prints both strings and numbers left-justified within each print zone.)

The function USING$ duplicates the PRINT USING statement almost exactly but returns the result as a string rather than printing it on the screen. For example, the following two statements yield the same output as the preceding PRINT USING statement.

```
LET outstring$ = using$(format$, x, y, z)
PRINT outstring$
```

The USING$ function allows you to modify or save the string *outstring*$ before printing it. You can also use this function with WRITE and PLOT TEXT statements.
We will first examine how to format numerical output.

**Formatting Numbers**

The basic idea of a format string is that the symbol “#” stands for a digit position. For example, let us compare the output resulting from two similar PRINT statements, the first a normal PRINT statement and the second employing USING.

```
PRINT x
PRINT USING "###": x
```

In the following table, the symbol “|” is used to denote the left margin and does not actually appear on the screen.

<table>
<thead>
<tr>
<th>x</th>
<th>PRINT x</th>
<th>PRINT USING &quot;###&quot;: x</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>12</td>
<td>12</td>
</tr>
<tr>
<td>12</td>
<td>123</td>
<td>123</td>
</tr>
<tr>
<td>123</td>
<td>1234</td>
<td>***</td>
</tr>
</tbody>
</table>

We notice several things. Without USING, the number is printed left-justified with a leading space for a possible minus sign, and occupying only as much space as needed. With USING, the format string “###” specifies a field length of exactly three characters. The number is printed right-justified in this field. If the field is not long enough to print the number properly, asterisks are printed instead, the unformatted value (here, of x) is printed on the next line and printing continues on the following line. If all you need to do is to print integer numbers in a column but with right-justification, then the preceding example will suffice.

Printing financial quantities so that the decimal points are aligned is important. Also, you may want to print two decimal places (for the cents) even when they are “0”. The following example shows how to do this. (In order to print negative numbers, the format string must start with a minus sign.)

```
x PRINT x PRINT USING "-##.##": x
--- -------- ------------------
1 | 1 | 1.00
1.2 | 1.2 | 1.20
-3.57 | -3.57 | -3.57
1.238 | 1.238 | 1.24
123 | 123 | *****
0 | 0 | .00
-123 | -123 | *****
```
Notice that two decimal places are always printed, even when they consist of zeroes. 
Also, the result is first rounded to two decimals. If the number is negative, the minus sign occupies the leading digit position. If the number is too long to be printed properly (possibly because of a minus sign), asterisks are printed instead, the unformatted value is printed on the next line, and printing continues on the following line.

Financial quantities are often printed with a leading dollar sign ($), and with commas forming three-digit groups to the left of the decimal point. The following example shows how to do this with PRINT USING.

```
x PRINT USING "$#,###,###.##": x
```

```
0 | $   .00
1 | $   1.00
1234 | $ 1,234.00
1234567.89 | $1,234,567.89
1e6 | $1,000,000.00
1e7 | *************
```

Notice that the dollar sign is always printed and is in the same position (first) in the field. Also, the separating commas are printed only when needed.

You might sometimes want the dollar sign ($) to float to the right, so that it appears next to the number, avoiding all those blank spaces between the dollar sign and the first digit in the preceding example. The following example shows how to do this.

```
x PRINT USING "$$$$$.##": x
```

```
0 | $ .00
1 | $1.00
1234 | $1234.00
1234567.89 | $1234567.89
```

Digit positions represented by “$” instead of “#” cannot surround or be next to commas.

In the previous two examples, no negative amounts can be printed since the format string does not start with or contain a minus sign.
The format string can also allow leading zeroes to be printed, or to be replaced by asterisks (*). You might find the latter useful if you are preparing a check-writing program.

\[
x \text{ PRINT USING } "$\%,\%,\%,\%.##" : x
-- ------------------------------
0 | $0,000,000.00
1 | $0,001,000.00
1234 | $0,001,234.00
1234567.89 | $1,234,567.89
\]

\[
x \text{ PRINT USING } "$*\*,***,***.##" : x
-- ------------------------------
0 | $*********.00
1 | $********1.00
1234 | $****1,234.00
1234567.89 | $1,234,567.89
\]

You can also format numbers using scientific notation. Because scientific notation has two parts, the decimal-part and the exponent-part, the format string must also have two parts. The decimal-part follows the rules already illustrated. The exponent-part consists of from three to five caret (^) symbols that must immediately follow the decimal-part. The following example shows how.

\[
x \text{ PRINT USING } "+\.#####^^^^" : x
-- -----------------------------
0 | +0.00000e+00
123.456 | +1.23456e+02
-.001324379 | -1.32438e-03
7e30 | +7.00000e+30
.5e100 | +5.00000e+99
5e100 | ************
\]

Notice that a leading plus sign (+) in the format string guarantees that the sign of the number will be printed, even when the number is positive. Notice also that the last number cannot be formatted because the exponent part would have been 100, which requires an exponent field of five carets. Notice also that if there are more carets than needed for the exponent, leading zeroes are inserted. Finally, notice that trailing zeroes in the decimal part are printed.
Floating Characters

You’ll notice that one of the previous examples includes several “$”, but that only one of them is actually printed. It is printed just to the left of the left-most non-zero digit, but always within the positions given by the sequence of “$”. We say that the sequence of “$” defines a floating region and that the spot where the “$” is printed floats within this region.

Besides the “$”, the plus sign (+) and the minus sign (-) can also define floating regions.

The rules are:

1. You can use either zero, one, or two different floating characters (“+” and “.” cannot both appear, and neither can commas.)
2. You can repeat the first (or only) floating character an arbitrary number of times, but not the second.
3. Zero to two different floating characters generate a sequence of zero to two characters called a header, as follows:

<table>
<thead>
<tr>
<th>First</th>
<th>Second</th>
<th>Positive</th>
<th>Negative</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>+</td>
<td>&quot;$+$&quot;</td>
<td>&quot;$–$&quot;</td>
</tr>
<tr>
<td>$</td>
<td>–</td>
<td>&quot;$−$&quot;</td>
<td>&quot;$–$&quot;</td>
</tr>
<tr>
<td>$</td>
<td>none</td>
<td>&quot;$&quot;</td>
<td>error</td>
</tr>
<tr>
<td>+</td>
<td>$</td>
<td>&quot;+$&quot;</td>
<td>&quot;–$&quot;</td>
</tr>
<tr>
<td>+</td>
<td>none</td>
<td>&quot;+$&quot;</td>
<td>&quot;–&quot;</td>
</tr>
<tr>
<td>–</td>
<td>$</td>
<td>&quot;$−$&quot;</td>
<td>&quot;–$&quot;</td>
</tr>
<tr>
<td>–</td>
<td>none</td>
<td>&quot;−&quot;</td>
<td>&quot;−&quot;</td>
</tr>
<tr>
<td>none</td>
<td>none</td>
<td>&quot;&quot;</td>
<td>error</td>
</tr>
</tbody>
</table>

Notice that the header contains the same number of characters as the number of different floating characters.

4. The zero to two character header will be printed as far to the right as possible within the floating region.

5. The numerical value’s leading digits can overflow into the floating region, thereby “pushing” the header to the left.

6. If the numerical value exceeds the total space provided, the entire space is filled with asterisks.
The following example illustrates some of these rules.

<table>
<thead>
<tr>
<th>PRINT x</th>
<th>PRINT USING &quot;$$$$$$-#,###.##&quot;: x</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>$ 0.00</td>
</tr>
<tr>
<td>1</td>
<td>$ 1.00</td>
</tr>
<tr>
<td>-1</td>
<td>$-1.00</td>
</tr>
<tr>
<td>4321.5</td>
<td>$ 4,321.50</td>
</tr>
<tr>
<td>-4321.5</td>
<td>$-4,321.50</td>
</tr>
<tr>
<td>1.23456789e+7</td>
<td>$ 12345,678.90</td>
</tr>
<tr>
<td>-1.23456789e7</td>
<td>$-12345,678.90</td>
</tr>
<tr>
<td>1000000000</td>
<td>$ 1000000,000.00</td>
</tr>
<tr>
<td>-1000000000</td>
<td>$-1000000,000.00</td>
</tr>
</tbody>
</table>

Notice that the “$” is never printed outside the floating region. A place is allocated for the minus sign. The leading digits of the numerical value can overflow into the floating region, which does not (and cannot) contain commas.

### Formatting Strings

Strings can also be formatted through PRINT USING or the function USING$, although there are fewer options for strings than for numbers. Strings can be printed in the formatted field either left-justified, centered, or right-justified. As with numbers, if the string is too long to fit, then asterisks are printed, the actual string is printed on the next line, and printing continues on the following line. The following example shows several cases.

<table>
<thead>
<tr>
<th>USING string</th>
<th>String to be Printed</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;&lt;####&quot;</td>
<td>&quot;Ok&quot;</td>
</tr>
<tr>
<td>&quot;#####&quot;</td>
<td>&quot;Hello&quot;</td>
</tr>
<tr>
<td>&quot;&gt;####&quot;</td>
<td>&quot;Goodbye&quot;</td>
</tr>
</tbody>
</table>

Notice that if centering cannot be exact, the extra space is placed to the right.

Any numeric field can be used to format a string, in which case the string is centered. This is especially valuable for printing headers for a numeric table. The following example shows how you can format headers using the same format string we used earlier for numbers.
Multiple Fields and Other Rules

A PRINT USING format string can contain several format items. For example, to print a table of sines and cosines, we may want to use:

```
LET format$ = "-#.###  -#.######  -#.######"
PRINT USING format$: x, sin(x), cos(x)
```

The value of \(x\) will then be printed to three decimals, while the values of the sine and cosine will be printed to six decimals. Notice also that spaces between the format items will give equal spaces between the columns in the printed result.

If there are more format items than there are values (numbers or strings) to be printed, the rest of the format string starting with the first unused format item is ignored. If there are fewer format items than values to be printed, the format string is reused, but starting on the next line. Thus,

```
PRINT USING "  -#.#####": 1.2, 2.3, 3.4
```

will yield:

```
1.20000
2.30000
3.40000
```

**Literals in Format Strings**

We have just seen that spaces between format items in a format string are printed. That is, if there are four spaces, the four spaces are printed. The same is true for more general characters that may appear between format items. The rule is simple: you can use any sequence of characters between format items except the special formatting characters. The characters you use will then be printed.

The special formatting characters are:

```
#  %  *  <  >  ^  .  +  -  ,  
```
The following example illustrates this use.

```
PRINT USING ".## plus ##.## equals ##.##": 1.2, 2.3, 1.2+2.3
```

will yield:

```
1.20 plus 2.30 equals 3.50
```

If there are fewer values than format items, the unused format items are ignored, but
the last intervening literal string is printed. Thus,

```
PRINT USING ".## plus ##.## equals ##.##": 1.2, 2.3
```

will yield

```
1.20 plus 2.30 equals
```

If you need to have one of the special formatting characters appear in the output – for
example, if you want to have a final period, as in the last example – you can simply add
a one-character field to the format string and add the quoted-string "." to the PRINT
statement. Thus,

```
LET x = 1.2
LET y = 2.3
PRINT USING ".## plus ##.## equals ##.## ": x, y, x+y, ".
```

will yield

```
1.20 plus 2.30 equals 3.50 .
```

The PRINT USING statements found in True BASIC allow you to format your cal-
culated results or data in many easy-to-understand formats. Investing time in learning
the many capabilities of these statements will pay rich dividends.
TRUE BASIC File Types

Text Files

A text file consists of lines that you can create on the keyboard and display on the screen using the True BASIC Editor (or any other application that can create and read “text-only” files). You can also create a text file entirely from within your program. True BASIC puts information into text files in the same way it displays information on the screen or printer, and it gets information from them just as it gets input from the keyboard. Thus, you use the same PRINT and INPUT statements — along with an appropriate channel number — with text files.

Text files are easy to understand and use. In fact, the PRINT and INPUT statements work just as they normally do when used with the screen and the keyboard — all the same rules apply. Because you can create and view text files with any screen editor, you can see the file structure and understand how it interacts with your programs. Text files often provide input data to a program or store output for later display or printing.

Text files, however, are not as efficient as the other types of files for large amounts of data. It is often hard to output information (such as strings or arrays) to a text file in a format that programs can easily read. Also, you may lose some numeric precision when you store numeric information in text files.

To understand the loss of numeric precision within text files (and the major difference between text files and internal files), let’s take a brief look at what happens when a program takes input from the keyboard and displays it on the screen. At the keyboard, you type characters that True BASIC interprets based on a standard character set. If you input a string value, True BASIC stores the actual characters you type (less leading and trailing spaces) in internal memory; each character occupies one byte of memory. When you use a PRINT statement to display a string value, you get exactly what is stored in memory.

If you input a numeric value, however, True BASIC converts the characters you type into the number they represent and stores that value in an internal format. In that internal format, numeric values have a precision of at least 14 significant digits, and each value occupies eight bytes of memory. True BASIC performs all calculations using the full precision of the internal numeric format.
When a PRINT statement displays a numeric value, however, you may not see the value to its full precision. Unless you specify otherwise with a PRINT USING statement, the PRINT statement displays characters representing the numeric value according to the rules described in Chapter 3 “Output Statements.” For example, the program:

```
LET x = 296445886        ! Population
LET y = 1.37             ! Growth rate
PRINT x * y              ! New population
END
```
displays the value:

```
4.0613086e+8
```
even though the internal value is calculated to be 406130863.82.

If you use a PRINT statement to store this value in a text file, the same series of characters that represent the value on the screen would be used to represent it in the file. A subsequent INPUT statement would retrieve the value with its reduced precision. While this may not be a problem for many applications, you should be aware of it.

Let’s look now at a simple example that gets information from one text file and prints some of that information to another file. The INPUT and PRINT statements work just as they normally do except that you specify a channel number to indicate the file to be used:

```
OPEN #1: NAME "WAGES", ORG TEXT, ACCESS INPUT
OPEN #2: NAME "NAMES", ORG TEXT, CREATE NEWOLD
RESET #2: END

DO WHILE MORE #1          ! While there is more to read
    INPUT #1: name$, age, salary
    PRINT #2: name$, "Age:"; age
LOOP
END
```

Each time the INPUT statement in this example is executed, it reads a line from the first file, treating it as if it had been typed at the keyboard. The line must have just the right number of items, of the right type (i.e., using numbers for numeric variables), separated by commas. If the value to be assigned to the name$ variable contains a comma, the string must be enclosed in double quotes. For example, the following line in the file would be legal:

```
"Williams, Pat", 34, 28500
```
while this one would cause an error:

```
Williams, Pat, 34, 28500
```
because True BASIC would interpret Williams as the value of name$, and attempt to assign the string value Pat to the numeric variable age.
Likewise, if a line in the file contains too few or too many items or the types do not match, an error occurs, since there is no way of “re-asking” the file for input.

Lines being input from a file may end with a comma to indicate that there is more input on the next line. Along with the INPUT statement, you may use the LINE INPUT, MAT INPUT, and MAT LINE INPUT statements with text files. However, the various forms of the INPUT PROMPT statement are not allowed, since a file cannot be prompted.

If you attempt to use the INPUT statement with a file opened with the ACCESS OUTPUT option, an error occurs. You’ll also get an error if the file pointer is at the end of the file (i.e., if there is no more information to input). Remember that you can use the SET POINTER or RESET statements to move the pointer to the beginning of the file, and you can use the MORE or END logical clauses to test for more data in the file (see earlier section).

The PRINT statement in the example above:

\[
\text{PRINT} \#2: \text{name}$, \text{"Age:";} \text{age}
\]

also follows all the conventions for a PRINT statement used to display values on the screen, including commas and semicolons. The file has a margin and a zonewidth, whose default values are 80 and 16, respectively, as they are for logical windows on the screen. You may change these settings with the SET MARGIN and SET ZONEWIDTH statements as follows:

\[
\text{SET} \#3: \text{MARGIN 70}
\]

\[
\text{SET} \#3: \text{ZONEWIDTH 10}
\]

Similarly, your program can find out the current margin and zonewidth of a file with the ASK MARGIN and ASK ZONEWIDTH statements:

\[
\text{ASK} \#2: \text{MARGIN m}
\]

\[
\text{ASK} \#2: \text{ZONEWIDTH z}
\]

Since there is no cursor in a file, the SET CURSOR statement does not make any sense when applied to a file. Similarly the two-argument version of the TAB function is forbidden with text files. You may, however, use the TAB function with a single argument:

\[
\text{PRINT} \#2: \text{name}$; \text{Tab(45); "Age:";} \text{age}
\]

You may also use the MAT PRINT or PRINT USING statements to print to a text file. Here’s an example of the PRINT USING statement used with a text file:

\[
\text{LET form}$ = \text{"########################################> Age: \#\#"}
\]

\[
\text{PRINT} \#2, \text{USING form$; name$, age}
\]

If you attempt to use the PRINT statement with a file that has been opened with the ACCESS INPUT option, an error occurs. You’ll also get an error if you attempt to overwrite the existing contents of a text file. To avoid attempts to overwrite, erase the contents of a file with the ERASE statement or reset the pointer to the end of the file with a SET POINTER or RESET statement before printing to it.

As shown in the above example, it is easy to copy all or part of one file to another.
Here's another example that changes all letters in a file to lowercase:

```vbnet
DIM line$(1000)
OPEN #3: NAME "Program5.Tru"
LET i = 0
DO WHILE MORE #3       ! Read lines into array
  LET i = i + 1
  LINE INPUT #3: line$(i)
LOOP
ERASE #3               ! Erase the file
FOR j = 1 to i         ! Rewrite in lowercase
  PRINT #3: Lcase$(line$(j))
NEXT j
END
```

The program reads the file into an array, erases the file, and then writes lowercase versions of the lines back into the file.

A word of caution about using the MAT PRINT and MAT INPUT statements with text files: while both work with text files, the MAT PRINT statement does not write information in a format that will work with the MAT INPUT statement. The MAT INPUT statement expects items of a row to be separated by commas, but the MAT PRINT statement separates the items of a row by spaces. There are two ways to solve this problem:

1. Create the file's contents by printing individual elements, putting a comma after each item except the last:

   ```vbnet
   FOR i = 1 to Ubound(array) - 1
     PRINT #7: array(i);", ";
   NEXT i
   PRINT array(Ubound(array))
   ... 
   ```

2. Use the LINE INPUT statement to input an entire line from the file and then “parse” the line into its component items using the ExplodeN subroutine provided in the StrLib library.

   ```vbnet
   LIBRARY "C:\TBSilver\TBLIBS\STRLIB.TRC"  ! Use appropriate path name
   ... 
   LINE INPUT #4: line$
   CALL ExplodeN(line$, array(),"")
   ... 
   ```

You should also be cautious when printing strings to text files for later input. Remember that the INPUT statement requires double quotes around strings containing commas or leading
or trailing spaces. To overcome this problem you could print such strings with enclosing quotes or, better yet, print just one string value per line and then use the **LINE INPUT** statement to read the entire line. The latter solution is the best if your strings contain double-quote marks, as you would have to repeat the double quotes within the string for the **INPUT** statement to read the string correctly!

### Internal Files — Stream, Random, Record, & Byte

The important differences between text files and the other types of data files are the statements you use to get data to and from the files and the way in which the files store numeric values.

Within text files, both numeric and string values are stored as series of characters. Numeric values are converted to strings of digits that represent the value (with possible loss of full precision). Any application that can read text can print or display such files. Because the format of text files is the same as for keyboard input or displays to the screen, text files use the normal **INPUT** and **PRINT** statements with the addition of channel numbers.

The remaining file types are all **internal** files — numeric and string values are stored in the same internal format used by the computer’s memory when it runs your programs. String values are stored internally as characters just as they are displayed, with one byte per character. Numeric values, however, are stored in the standard IEEE eight-byte format that cannot be displayed. Because of the storage format, internal files require **READ** and **WRITE** statements to input and output data. While internal files cannot usually be displayed directly on the screen or printer, they do have several advantages:

- The numeric values retrieved from an internal file are read with exactly the same precision as the values written to the file. With a text file, numeric values may lose precision when the **PRINT** statement converts them from the computer’s internal format to a sequence of characters; any greater precision is lost and cannot be retrieved when that sequence of characters is input from the file.

- Reading and writing operations are faster with internal files, because there is no need to convert numeric values between internal and display formats.

- True BASIC internal files may be used with programs on any computer type. The internal format is the same no matter where you run your programs. Also, the ability to read a file as a byte file lets you read any file created by any application on any computer. Text files, however, must often be translated when they are moved between operating systems because of the variations in how operating systems view end-of-line characters within text files.

- Three types of internal files — random, record, and byte — permit the more efficient random access of records within the files. With random access you can jump directly to any part of the file, rather than having to work through the file from start to finish. Text
and stream files permit only sequential access — the items in the file must be retrieved
in exactly the same order in which they were stored.

Internal files come in four types: stream, random, record, and byte files, all of which are
explained below. Random and record files are organized by records. A record is a storage
location of fixed-length within a file. All the records within a file are numbered so that you
can move easily to any record in the file with a set record statement. The exact struc-
tures of random and record files are explained below.

As noted above, you use write and read statements with internal files. The exact usage
of these statements varies depending on the type of file, as described below.

The open, close, erase, and unsave statements work for internal files just as they
do for text files. Remember, however, that the default organization for a newly created file
is text, so you must specify the type of file when you are creating a new internal file. The set
and ask statements have several additional forms that are described with the different file
types below.

Stream Files

A stream file is simply a sequence of values. These values must be read back in the same
order in which they were written to the file. For example:

```
open #1: name "VALUES.STR", create new, org stream
write #1: pi, exp(1), "this is a string.", 3.14
...
set #1: pointer begin
read #1: a, b, c$
read #1: d
!
```

At this point, a is exactly equal to PI
!

b is exactly equal to EXP(1)
!

c$ is the string "This is a string."
!

d is exactly equal to 3.14

Notice that the write and read statements need not have the same number of variables
— there is no concept of a line of data as in text files or a record as in random and record files.
The one requirement is that the type (numeric or string) of a variable in the read state-
ment must match the type of the next value in the file. If the type is wrong, an error occurs.

Although it is up to the programmer to keep track of the type and purpose of the values in a
stream file, you can “peek” at the next value’s type with an ask datum statement. For
example:

```
ask #1: datum type$
select case type$
case "numeric"
    read #1: n
```
Random Files

Random files are composed of records. All the records within a single file have the same maximum length which is called the **record size** of that file.

Each record in a random file may contain any number of string and/or numeric values, provided that the cumulative length of the items (and their associated “bookkeeping” as explained below) does not exceed the file’s record size. In fact, different records within the same file may contain different numbers and types of items.

Any record whose actual length is less than the record size of the file will be automatically “padded” to the proper record size before being written to the file. This padding will be ignored when the values are subsequently retrieved from the file. Thus, you need not worry about padding records yourself.

Although True BASIC will automatically move the file pointer to the next record each time a record is read, allowing you to easily process a random file from beginning to end, you can also move the file pointer to any existing record within the file arbitrarily. The record to which the file pointer currently points may be retrieved and/or overwritten as necessary.

Before you can write records to a new or empty random file, you must first set the file’s record size. You may do this using a RECSIZE option in the **OPEN** statement, as in:

```
OPEN #1: NAME "NEWDATA.RDM", ORG RANDOM, RECSIZE 50, CREATE NEW
```

or by using a **SET RECSIZE** statement after the file has been opened, as in:

```
OPEN #1: NAME "NEWDATA.RDM", ORG RANDOM, CREATE NEW
SET #1: RECSIZE 50
```

Note, however, that you may set or change the record size only for a new or empty file — if the file contains any records you must erase it (with the **ERASE** statement) before you can change the record size.

If a file already exists and contains one or more records, it already has a record size which you cannot change without first erasing the file. You may use the **ASK RECSIZE** statement to find out the record size of a file as follows:

```
OPEN #1: NAME "DATA", ORG RANDOM, CREATE OLD
ASK #1: RECSIZE rsize
```

Here, the record size of the file named DATA would be assigned to `rsize`. 
If you attempt to write more bytes to a random file record than its defined record size, an error results. The record size must be large enough to hold both the data that will be stored in each record and some additional “bookkeeping” information.

This bookkeeping information keeps track of the kinds of information in each record (remember that random files allow an arbitrary number of values of arbitrary types within each record) and indicates the end of the record. Although you need not worry about this information when using the file, it does require storage space, and you must account for it when you set the record size for a new random file (or if you need to figure out how much you can write to new records in an existing random file).

A string item stored in a random file record will occupy one byte for each character in the string plus four bytes of bookkeeping information. On the other hand, a numeric value stored in a random file record will always occupy exactly nine bytes — eight bytes for the internal representation of the number and one byte for bookkeeping. In addition, you must always allow one byte in the record size for the end-of-record marker.

As an example, consider a situation in which you plan on storing two strings and three numbers in each record. First, you need to know the maximum length of the strings that you will store. Let’s assume that the first string will never be longer than 30 characters and the second string will never exceed 14 characters. Thus, you need to reserve 30 + 4 bytes for the first string and its bookkeeping information and 14 + 4 bytes for the second string and its bookkeeping information. Each of the three numeric values will occupy 8 + 1 bytes with its bookkeeping information. And don’t forget to reserve 1 byte for the end-of-record marker. By adding all of these requirements together, you know the proper record size for this random file is 34 + 18 + 9 + 9 + 9 + 1 = 90.

If the records in the random file will contain varying numbers and types of items, calculate the length based on the longest record you will need. If you attempt to write more bytes to a random file record than its defined record size, an error results.

Note: True BASIC does not know how you arrived at a random file’s record size; it simply checks to be sure total size of the record does not exceed the established record size. You might exceed a record size because you attempted to write more items than you had planned on, or because a string in the record is longer than you planned. True BASIC won’t know the difference; it will simply report that the record size was exceeded. You may want to use the DECLARE STRING statement to define a maximum length for string variables used in random file records. This lets True BASIC provide more specific diagnostics should a problem arise.
Each **READ** and **WRITE** statement reads or writes one complete record in a random file. Because individual records may contain different numbers and types of values, the pattern of the **READ** statement must mirror the pattern of the **WRITE** statement that produced the record; otherwise, an error will occur. In the following example, each record contains three values: a string value, a numeric value, and another string value:

```basic
! A new RANDOM file
OPEN #1: NAME "STUFF", CREATE NEW, ORG RANDOM, RECSIZE 100
...
WRITE #1: name$, age, occupation$
```

Later on, perhaps in a different program, you can retrieve that information, as follows:

```basic
! File already exists
OPEN #1: NAME "STUFF", ORG RANDOM
...
! True BASIC figures out the RECSIZE by looking at the file.
! CREATE option not needed, or use CREATE old.
...
! The READ statement must mirror the earlier WRITE
READ #1: person$, a, occ$
```

The **READ** statement typically reads all the values in the record, and the variable types must match the value types in the record. However, if the record contains many items and you want only the first few, you may use a **SKIP REST** clause in the **READ** statement as follows:

```basic
READ #1: person$, a, SKIP REST
```

The **SKIP REST** clause instructs True BASIC to ignore the remaining values in the record.

Remember that the records within a random file need not have the same shape — they may have different numbers and types of values of varying lengths (as long as they don’t exceed the record size). For example, a random file that contains a student’s grade record might contain different information in the first few records:

```basic
OPEN #5: NAME "SMYTHE", ORG RANDOM, ACCESS INPUT
READ #5: last$, first$, middle$, class ! First record
READ #5: street_address$ ! Second record
READ #5: city$, state$, zip$ ! Third record

PRINT "Grade Report for "; first$ & last$; ". Class of"; class
DO WHILE MORE #5
    READ #5: course$, grade, credits ! Remaining records
    PRINT course$; tab(20); grade, credits; "credits"
LOOP
...
```
Random files are so called because they permit random access. That is, you can access any particular record regardless of the order in which records were created. The records are automatically numbered starting at 1. The file pointer normally moves to the next record after a record has been read or written — remember that each READ or WRITE statement reads or writes an entire record in a random file. But you may also jump around to arbitrary records within a file using the SET POINTER and SET RECORD statements:

```
SET #3: POINTER SAME ! Go back to the record just read or written
SET #3: POINTER NEXT ! Skip the current record
SET #3: RECORD r ! Go to record number r
```

You may also use the keyword RESET as follows:

```
RESET #3: SAME ! Go back to the record just read or written
RESET #3: NEXT ! Skip the current record
RESET #3: RECORD r ! Go to record number r
```

Clearly, the last option is the most powerful one. You may find the current file pointer position, or the number of the current record, with the ASK RECORD statement as follows:

```
ASK #3: RECORD r
```

As an example, consider a simple computer-based dictionary. Suppose that one random file contains a list of words and another random file contains the corresponding definitions in the same order. If you open these two files as #1 and #2, respectively, you could look up words as follows:

```
DO
  INPUT PROMPT "Word: ": w$
  CALL Find (#1, w$, n)           ! Word in record n
  IF n = 0 then
    PRINT "Word not found"
  ELSE
    SET #2: RECORD n             ! Find definition
    READ #2: def$
    PRINT def$
  END IF
LOOP
```

The program-defined subroutine Find searches file #1 for the word and returns its record number (or 0 if it finds no word).

```
SUB Find (#9, word$, rec)
  RESET #9: 1                    ! Start at beginning of file
  ASK #9: FILESIZE last_rec      ! How many records?
  FOR r = 1 to last_rec
    IF Find (#9, word$, r) then
      RETURN r
    END IF
  NEXT
END
```
If the word is found, the program jumps to the same record number in file #2 and reads the definition. This is not possible with text files.

Changing an existing record in a random file is just as easy. Simply jump to the record and use a WRITE statement. You can add to the end of the file by first using:

```
SET #3: POINTER END
```

You may also use the MAT READ and MAT WRITE statements to read or write an entire array from or to a random file. With random files, the MAT WRITE statement puts all the array elements in the same record, provided the record is long enough. You may then recover the elements with a MAT READ statement — or with a READ statement that includes a variable for each element.

**Record Files**

Record files are like random files, except that you can place only one value — numeric or string — in a record. Although you will often find that a random file is better suited for a particular task, record files may be used if you are storing a single item per record.

When used with a record file, a WRITE statement stores each value in a separate record. And a MAT WRITE statement will use as many records as there are elements in the array. For example, the WRITE statement in:

```
! A new RECORD file
OPEN #2: NAME "STUFF1", CREATE NEW, ORG RECORD, RECSIZE 50
... WRITE #2: name$, age, occupation$
```

will use three records to store the three quantities. Later, you may retrieve these values with:

```
READ #2: person$, a, occ$
```

or with:

```
READ #2: person$
READ #2: a
READ #2: occ$
```

The READ statement need not mirror the WRITE statement, but the variable type — numeric or string — must be correct.

In contrast to a random file, calculating the proper record size for a record file is easy. Each record in a record file contains four bytes of bookkeeping information. However, since the

```
size of this information is the same for all records, you do not need to account for it in the record size (as you would for a random file). Thus, the record size of a record file need only reflect the length of a number (which is 8 bytes) or the length of the longest string value you expect to store in a single record. Remember that you may freely mix numeric and string values in a single record file, so the record size must reflect the length of the longest value you plan to store in a record.

---

**Note:** The bytes actually included in the record size are different for random and record files. For random files, the record size must include the extra, bookkeeping bytes along with the data bytes. For record files, however, the record size need include only the length of the data item to be stored. The bookkeeping bytes are there, but you don’t need to account for them.

---

In all other respects, record files are like random files. They permit random access, and you may use the same `SET` and `ASK` statements to move around and find out information about them.

### Byte Files

A **byte** file is not a special kind of file but rather a way of looking at a file. When a file is viewed as a byte file, it is considered simply as a sequence of bytes with no special format. That is, True BASIC does not make any assumptions about a byte file, and it will not perform any of the “housekeeping” tasks that it performs for other files (other than advancing the file pointer).

You may view any True BASIC file as a byte file by specifying the ORG BYTE option in the `OPEN` statement used to open that file. Indeed, you may view any file as a byte file, including compiled True BASIC programs, files created by other applications, or files created on another type of computer or under a different operating system.

As with other internal files, you use `READ` and `WRITE` statements to access byte files. The number of bytes read by a single `READ` statement depends on the type of variable being read.

A `READ` statement used to access a byte file may have only one variable, which is normally a string variable, since the contents of the file may be any sequence of bytes. Although byte files do not recognize records, True BASIC uses the current record size to decide how many bytes to read to a string variable.

You may set the record size using a `RECSIZE` clause in the `OPEN` statement, as you would for random or record files, or you may use a `SET RECSIZE` statement. Similarly, you may use an `ASK RECSIZE` statement to find the current record size of a byte file, as you would for random or record files. Because byte files are reading an arbitrary number of bytes, not
actual records, you may use the `SET RECSIZE` statement to change the record size of a byte file as many times as necessary.

Alternatively, you may specify the number of bytes to be read to a specific string variable by including a `BYTES` clause in the `READ` statement. For example:

```
READ #7, BYTES 32: y$
```

would read the next 32 bytes in the file associated with channel #7 into the string variable `y$`.

This method of overruling the file’s record size within an individual `READ` statement is commonly used with byte files, since you may need to read strings of different lengths from a single file. Often, you might want to read an entire file to a single string, as follows:

```
ASK #7: FILESIZE fs
READ #7, BYTES fs: y$
```

If you use a `READ` statement with a numeric variable, the next eight bytes in the file will be read as a numeric value stored in the IEEE eight-byte format. When a numeric value is read, the file’s record size is ignored. Likewise, the `BYTES` clause is not allowed in a `READ` statement that specifies a numeric variable.

If the file pointer is near the end of the file and the number of bytes remaining is less than the current record size, a `READ` statement simply reads all the remaining bytes. If the pointer is at the end of the file, however, a `READ` statement causes an error.

The `WRITE` statement may also be used with string or numeric values. With a string value, it writes as many bytes as there are characters in the string. Numeric values are written to byte files in the IEEE eight-byte format.

---

**Note:** The IEEE eight-byte representation used to store numeric values in a byte, random, or record file is identical to the IEEE eight-byte representation produced by the `NUM$` built-in function (see Chapter 18). This means that numbers may be read from a byte file as eight-byte string values and converted to numeric values using the `NUM` function. This may be a useful alternative to reading those values directly into numeric variables.

Within a byte file, each byte is numbered as if it were a separate record (regardless of the current “record size”) beginning with 1 at the first byte. Thus, the `SET` and `ASK` statements that require or return a record number actually refer to a byte number. For example, the statement:

```
SET #3: RECORD 120
```

when applied to a byte file, moves the file pointer to byte number 120. A program may read any consecutive sequence of bytes, and it may overwrite any such portion of the file. You may also use the `WRITE` statement to add to the end of the file, provided that the file pointer is at the end of the file.
The following examples illustrate some instances when byte files are helpful. The first is a routine that will copy any file, no matter what its format or content:

```basic
SUB FileCopy(from$, to$) ! Copy any file
OPEN #3: NAME from$, ORG BYTE  ! Open two files
OPEN #4: NAME to$, CREATE NEWOLD, ORG BYTE
ERASE #4

SET #3: RECSIZE 1024 ! Copy in 1K pieces
DO WHILE MORE #3
    READ #3: x$
    WRITE #4: x$
LOOP
END SUB
```

This procedure uses 1024 bytes (1K) as a convenient unit to read and write at one time. (A record size that is a power of two may allow your program to run faster.) If the file length is not a multiple of this, the last READ will result in a shorter string x$, but it will cause no error. The new file will have precisely the same content as the old one.

You may also use byte files to search a file for non-printing characters. Since True BASIC reads all bytes, including those such as a line feed, each byte can be identified by its character code. (See the ORD and CHR$ functions in Chapter 8 “Built-in Functions.”) You could therefore extract the text from any type of file by examining each byte and keeping only the printing characters, as follows:

```basic
SUB Text_extract (from$, to$)
OPEN #3: NAME from$, ORG BYTE  ! Open two files
OPEN #4: NAME to$, CREATE NEWOLD, ORG TEXT
ERASE #4

SET #3: RECSIZE 1 ! One byte at a time
DO WHILE MORE #3
    READ #3: x$
    IF 32<= Ord(x$) and Ord (x$) <=127 then ! Standard printing characters
        PRINT #4: x$;
    END IF
LOOP
END SUB
```

Note that this example is presented in the simplest form possible. There is plenty of room for improvement. For instance, you might read larger sequences of bytes and build up an output string in memory, sending it to the file only when it reaches a certain length. Each file access takes time, and the fewer times your program accesses a file, the more quickly it will run.
As an illustration of how byte files can store any type of information, consider how you might store a screen image, such as a complex diagram. The BOX KEEP statement stores the image displayed within a specified area on the screen into a string variable, which you can later display with the BOX SHOW statement (as described in Chapter 13 “Graphics”). If you need to save these strings for later display, you can store them in byte files, as in the following program fragment:

```truebasic
SET WINDOW 0,1,0,1
BOX KEEP 0,1,0,1 in keep$
OPEN #5: NAME "Image", CREATE NEW, ORG BYTE
WRITE #5: keep$
```

Another program fragment may then retrieve and display the image as follows:

```truebasic
OPEN #5: NAME "Image", ORG BYTE
ASK #5: FILESIZE fs ! Number of bytes in file?
READ #5, BYTES fs: keep$ ! Read entire file to string
SET WINDOW 0,1,0,1
BOX SHOW keep$ at 0,0
```

Byte files in combination with the built-in PACKB subroutine and the built-in UNPACKB function provide an efficient means of packing information to conserve storage space. As you have seen, numeric values stored in internal files always occupy eight bytes — whether the value is 0 or 3.7836126523e287. Often, however, your programs need to store only integers within a specific range. Eight bytes is generally much more storage than is necessary for integers, so storing many integers into an internal file can use much more disk space than would otherwise be required.

One way to eliminate this waste is to “pack” the integer values into string values, using the PACKB subroutine, before storing them to the file.

The PACKB subroutine allows you to represent an integer value as a specific series of bits within a string variable. For instance, the following program fragment writes a list of integers into a byte file.

It assumes that each integer fits into 16 bits (integers from 0 to 65,535) and there are n of them in the array list:

```truebasic
LET x$ = ""
LET j = 1
FOR i = 1 to n
    CALL Packb(x$,j,16,list(i))
    LET j = j+16
NEXT i
WRITE #1: x$
```
Each integer is packed into x$ using the **PACKB** subroutine. Once all the numbers have been packed into x$, x$ is written to the byte file.

Rather than maintaining the variable j as the starting bit position within the string x$, you may find it simpler to use the following trick:

\[
\text{CALL PACKB}(x$, \text{Maxnum}, 16, \text{list}(i))
\]

If the starting bit position provided to the **PACKB** subroutine is beyond the end of the string value, the resulting series of bits will begin next to the last bit in the current string value. In other words, by specifying a ridiculously large value as the starting bit position, you pack the integer value in \(\text{list}(i)\) into the 16 bits immediately following the end of the current value of x$. This eliminates the need for the variable j to keep track of the bit position.

You could recover the resulting list from the byte file using the **UNPACKB** function as follows:

\[
\text{ASK} \#1: \text{FILESIZE} \; \text{fs} \\
\text{READ} \#1, \text{BYTES} \; \text{fs}: \; x$ \\
\text{LET} \; j = 1 \\
\text{FOR} \; i = 1 \; \text{to} \; \text{Len}(x$)/2 \\
\quad \text{LET} \; \text{list}(i) = \text{UNPACKB}(x$, j, 16) \\
\quad \text{LET} \; j = j+16 \\
\text{NEXT} \; i
\]

The first two lines are the standard way of reading an entire byte file into the string. The first statement discovers how many bytes are in the file, and the second reads them all with a single **READ** statement.

You would save storage and gain speed by packing each number into two bytes (16 bits). Such packing is particularly important for storing large amounts of information. For example, if you have one million “yes/no” replies, they can be packed into one million bits, or 125,000 bytes. A byte file is the only reasonable way of storing such information.
Basic to True BASIC Converter

Introduction

The BASIC to True BASIC Converter (BtoTB) helps you convert programs written in other versions of BASIC into True BASIC. Other versions of BASIC include BASICA for the IBM PC and compatibles, Microsoft Compiled BASIC, GWBASIC, several versions of Microsoft QuickBASIC, Macintosh QuickBasic, and Microsoft Visual Basic. We use the word “Basic” to refer to any Basic-like languages other than True BASIC.

For simple programs as much as 85% of the original code will be converted to equivalent True BASIC code. For that code not directly convertible, the expanded PDF manual that is found in the same directory with the BtoTB Converter suggests other ways to rewrite your original code into True BASIC and achieve your original purpose.

Start the BtoTB Converter by double-clicking on its icon. An application with two windows and three buttons appears.
Click on the CONVERT button and a file selection dialog box will appear.

When you have selected the file you wish to translate, click Open and you will be presented with a file saving dialog box in which you can specify a new title. The BtoTB Converter will suggest a default name based on the original file name.

As soon as the original file and the results file have been created, the Converter will begin the translation. You will see the original code in the left window and the True BASIC code in the right window.

A status message area is below the two windows and tells when the conversion is finished. The Cancel button and Quit button allow you to interrupt or stop any procedure.

**Versions of Basic**

Early versions of Basic, such as BASICA on the IBM PC, contained many statements for working with that particular hardware. Most of these are no longer used. Furthermore, many of the syntax rules have evolved toward the ANSI Standard for BASIC, upon which True BASIC is based. As an example, early versions of Basic used WHILE and WEND to contain a loop structure. While those keywords are retained for historical reasons, most Basic programs are now written using the DO and LOOP keywords, just as in True BASIC. Modern versions of Basic allow creating Graphical User Interface (GUI) elements, such as push buttons and scroll bars. True BASIC also allows these but through use of a subroutine library and
subroutine calls, rather than through statements in the language. For such statements, BtoTB merely suggests the subroutine to be called, but does not attempt to develop the actual calling sequence.

**Line Numbers**

BtoTB accepts Basic programs that are line-numbered or not. The resulting True BASIC program has line numbers. If the original program is line-numbered, the resulting True BASIC program will have line numbers that correspond, and GOTO and similar statements will be left untouched. (The original line numbers must be spaced far enough apart to permit inserting additional statements.) For a non-line-numbered program, the GOTO and similar statements will be converted to GOTO a line number in the converted program. BtoTB does not attempt to convert the possibly “spaghetti code” of an old-fashioned line-numbered Basic program into the more modern “structured” form.

While it is theoretically possible to do this, the result is actually more difficult to understand. Therefore, all GOTO and similar statements are left as is. This manual describes several simple cases for which manual conversion from GOTOs to structured constructs is easily done and is recommended. BtoTB does not convert graphical user interface elements (buttons, windows, scroll bars, etc.) since the logic used to manage these elements in True BASIC is entirely different from the approach of other versions of Basic. Neither does it attempt to convert use of record files, as the logic used by True BASIC (and ANSI BASIC) is altogether different from most other versions of Basic. Finally, the BtoTB cannot convert data structures as there is no equivalent capability in True BASIC.

BtoTB works by reading a Basic program in a text file one line at a time, making the necessary syntactical changes and rewriting the line to an intermediate file. Statements that are the same in Basic and True BASIC are rewritten without change. Some Basic statements that do not exist in True BASIC are modified to work equivalently. In some cases this will involve invoking functions or calling subroutines located in the supporting libraries. Other Basic statements, for which there is no direct or simple True BASIC equivalent, are not converted, but are marked so that they will generate a True BASIC compiler error.

A second pass copies the converted program from the intermediate file to the file you named, filling in the forward jump references as it goes. The second pass also inserts any needed special internal function definitions and fills in the DECLARE DEF statements. (The right hand window shows the results of the first pass only.)

Whether the original Basic program is line-numbered or not, line numbers are included in the result if there are GOTO or similar statements present.
General Caveats

Although many features of the various versions of Basic are found also in True BASIC, sometimes in a slightly different form, many others are not found in True BASIC. One reason is that most versions of Basic allow access to the specialized features of a particular machine. In contrast, True BASIC has, as one of its valuable features, cross-machine portability.

BtoTB handles some of the machine-specific features through subroutines located in the library files DEFLIB.TRU. A direct conversion of a particular feature from Basic to True BASIC, through possible, may not be desirable. For example, many versions of Basic determine the graphics mode on DOS machines by “peeking” at a certain byte. True BASIC does this with the ASK MODE statement.

The smart user will use BtoTB to make the mechanical conversion of from 80% to 90% of the program. The result should then be scrutinized carefully and parts recoded by hand. A knowledge of the features of True BASIC and its libraries is essential to an efficient and correct conversion.

The BtoTB Converter is designed so that it can be updated to include other conversion routines that users find helpful. We welcome your ideas and suggestions. Direct your messages to support@truebasic.com.

Reference Materials

The process of program conversion is not trivial and it is important that you have proper reference material for the task.

You will find a major portion of the True BASIC Reference information in your online HELP facility. Our website shows other available texts.

Translating a file

To minimize confusion, create a BtoTB directory or folder. In it place the original Basic file you wish to translate to True BASIC and the BtoTB application program. As noted earlier, the original source code file is a text file. Text files on the DOS/Windows operating system and those on the MacOS operating system are slightly different. A DOS file ends each line with a return and a linefeed character.

MacOS files end with only a return. If you are using a MacOS computer, the BBEdit Lite text editor (also included in your MacOS Bronze Edition Utility directory) makes it very easy to convert and save files from DOS to MacOS, by adding or removing the linefeed character at the end of each line.
Testing the Converted Program

When you have finished your conversion, start up your copy the True BASIC Language System by clicking on the True BASIC icon.

When True BASIC is up and running, open the converted program. You should also make sure that the file DEFLIB.TRC is in the same directory, as the converted program may need one or more subroutines in it.

You can now modify the converted program using the True BASIC screen editor.

When you select Run from the Run menu, the converted program should run and give the same results as the old Basic program.

If you are not so lucky, the True BASIC may discover syntax errors, displaying them in its Error Window. Double-clicking on a particular error will place the editing window cursor at the offending code. BtoTB inserts "***" for some statements it cannot convert, which will lead to a compiler error.

General Considerations

The purpose of BtoTB is to produce a True BASIC program that is as functionally equivalent to the original Basic program as practical, but may not be the most concise or most efficient. For example, it makes no attempt to convert GOSUB statements to CALL statements. Some features not directly available in True BASIC are provided by subroutines. For example, the SCREEN and COLOR statements in BASICA are converted to calls to subroutines in True BASIC, since the treatment of color is different. Still other features of Basic have no counterpart in True BASIC and are left as is. For example, there is no True BASIC equivalent to the STRIG ON statement.

Line Numbers

Basic programs are assumed to be either line-numbered or not. BtoTB makes the distinction between the two by examining the first line of the file. If it starts with a line-number, BtoTB assumes that all lines (except line continuations) start with line-numbers. In this case, the line-numbers must be in order and must contain sufficient room between line-numbers to permit inserting True BASIC statements. (True BASIC does not allow multiple statements on a line and must put them on separate lines.) The first line number must be large enough to allow for the Preamble, which is about 30 lines long.

If the first line does not contain a line number, BtoTB makes the assumption that line-numbers, if present, are treated merely as statement labels. In this case there is no restriction that line numbers be in order. It generates its own line numbers that will bear no relation to the line number labels. Line number labels and other statement labels are converted to ordinary line numbers.
BtoTB has two passes to handle forward references. The first pass does most of the conversion, and places the result in a temporary file. The second pass fills in the forward label references, if any, and places the result in the output file you named.

Finally, if there is no use of GOTO or similar statements in the entire file, BtoTB removes the line numbers.

---

**IMPORTANT NOTE:** You may get the cryptic message: Illegal line number at line -1. This message is generated when the compiler can not determine if a program has line numbers. The error occurs when a blank line exists in a line numbered program. ALL lines must have line numbers, even if they are blank.

---

**Preamble**

BtoTB places a preamble at the beginning of each file of True BASIC programs. The preamble may contain a LIBRARY statement that names DEFLIB.TRC as the file containing subroutines needed by the converted program. It also places there, and at the beginning of each program unit, a DECLARE DEF statement containing the names of the actual functions needed, if any, and an OPTION BASE 0 statement. If no functions from DEFLIB.TRC are needed, the DECLARE DEF statement is not added. The second pass also inserts the actual code, if needed, of the several internal functions in True BASIC that refer to files. (The reason is that True BASIC does not allow file reference numbers to be passed to external functions; subroutines are needed instead.) Because there is often no way for BtoTB to determine the scope of such functions, however, you may find it necessary to move or copy these routines before the program will run correctly. The names of these functions are: LOF, LOC, EOF, and FREEFILE. (See Section 6 of the BtoTB PDF manual for alternate ways to code these in True BASIC.)

**Numeric Accuracy**

BtoTB properly handles the conversions between quantities of type integer and quantities of type single- or double-precision. All arithmetic in True BASIC is performed using double-precision floating point numbers with about fifteen decimal digits of accuracy. BtoTB treats both single- and double-precision numbers in Basic as equivalent. The corresponding conversion functions (CSNG, CDBL) are thus omitted.

BtoTB treats both long and short integer types in Basic as double precision in True BASIC. It properly inserts a ROUND function whenever a non-integer quantity is
assigned to an integer variable, or whenever an intermediate calculation, such as integer divide or mod, requires that the result be an integer. For example,

\[
\text{LET } a\% = b!
\]

is converted to the True BASIC

\[
\text{LET } a_i = \text{round}(b)
\]

If you know that \(b\) always has an integer value, you should remove the round function.

BtoTB properly deals with two or more different variables having the same name. For example, \(x, x!, x\%, x\#, \) and \(x\&\) are all different in a Basic program. They are changed to True BASIC variables as follows:

<table>
<thead>
<tr>
<th>Basic</th>
<th>True BASIC</th>
</tr>
</thead>
<tbody>
<tr>
<td>(x)</td>
<td>(x)</td>
</tr>
<tr>
<td>(x%)</td>
<td>(x_i)</td>
</tr>
<tr>
<td>(x!)</td>
<td>(x_s)</td>
</tr>
<tr>
<td>(x#)</td>
<td>(x_d)</td>
</tr>
<tr>
<td>(x&amp;)</td>
<td>(x_{li})</td>
</tr>
</tbody>
</table>

The type of \(x\) without a suffix is determined by the DEF type statements; the default type is single precision.

For string variables, when the DEF type statements specify that variable names starting with “a” are string type, \(a\) and \(a\$\) are treated by Basic as the same. Consequently, BtoTB merely adds the “\$” to the former.

Hexadecimal and Octal constants are converted to decimal integers.

**Booleans**

BtoTB properly handles Boolean expressions, including the IMP and EQV operators. For the NOT operator, parentheses surround the entire clause since \(\text{NOT NOT}\) is allowed in Basic but not in True BASIC.

If a logical expressions lack a relational operator, BtoTB adds a “\(<>0\)”. It does not convert Boolean-valued expressions that appear in arithmetic statements. That is,

\[
\text{LET } x = y < z
\]

is legal in Basic (\(x\) is assigned 0 or -1, according as \(y < z\) is true or false), but not in True BASIC. Instead, it is converted to:

\[
\text{LET } x = y *** z
\]
The straightforward representation in True BASIC might be:

```
IF y < z then LET x = 0 else LET x = 1
```

This is an example of a change that must be done by hand.

BtoTB does not handle logical expressions involving logical operators and numbers, or bit-by-bit logical operations. For example,

```
IF (PEEK(123) AND &H30) <> &H30 THEN ... 
```

will generate an error message and be left in its original form.

A determination must be made as to the purpose of the IF statement. In the example above, it is designed to determine the type of graphics card.

```
330 DEF SEG=0
340 IF (PEEK(&H410) AND &H30) <> &H30 THEN COLS = 3:GOTO 360
350 WIDTH 80:COLS=8
360 DEF SEG
```

An alternative way in True BASIC might be:

```
ASK MODE mode$
IF mode$ = "MONO" then
SET MARGIN 80
LET cols = 8
ELSE
SET MARGIN 40
LET cols = 3
END IF
```

which is slightly longer but more understandable.

**Arrays**

BtoTB expects an array dimension statement to occur before (i.e., in a lower-numbered line) any reference to it. Some versions of Basic allow you to dimension arrays at a higher-numbered line than a reference to it, as long as the DIM statement is executed first. Other versions of Basic allow automatic dimensioning of arrays. True BASIC requires that all arrays be dimensioned, and that the DIM statement appears in a lower-numbered line than any reference. BtoTB does not insert DIM statements where needed; they must be inserted later by hand. Or, you can insert a complete set of DIM statements into the original Basic program. In any event, the True BASIC compiler will provide a suggestive error message when an attempt is made to use an undimensioned array.

Some versions of Basic allow you to use the same variable name for a numeric value and an array, but True BASIC does not. If the variable name is the same as a previously dimensioned array name, BtoTB will attach ".t" to the variable name. The variable name will not be changed if there is an undimensioned array having the same name. Instead, the error will be caught by the True BASIC compiler.
More detailed documentation, in the Adobe Acrobat® PDF format, is included with your copy of the BtoTB Converter. In it, you will find additional information on how to translate functions and statements such as:

- Defined Functions
- GOTO Statements
- IF-THEN Statements
- INKEY$ Function
- Line Continuations
- INPUT & INPUT$
- Variable & Array Types
- KEY Statements
- Global and Local Variables
- LOC Function
- Program Units
- LOF Function
- FILE Input & Output
- PEEK and POKE
- Text Files
- Share and Static
- Record Files
- TYPE Structures
- Binary Files
- VAL Function
- Statements
- VIEW Statement
- Functions
- Windows
- GOSUB & RETURN
- Buttons
- CLOSE
- Edit Fields
- COMMAND$
- Menus
- Relative Graphics
- CSRLIN & POS
- DRAW Statement
- Event Handling
- EOF Function
- File Dialogs
- Error Handling

The BtoTB Converter is included with the Bronze Edition so that you can quickly convert other BASIC programs that you might have used in the past into True BASIC code that will continue to be useful and functional in the future.

The BtoTB Converter and accompanying documentation is in the UTILITY directory that is part of your original True BASIC CD.
You are now using the BRONZE Edition of the True BASIC Language System. Two more advanced versions are also offered. If you wish to upgrade to the SILVER or GOLD editions of this series, as a current True BASIC user, you will be entitled to special upgrade considerations.

True BASIC SILVER Edition

- All the features found in the BRONZE Edition, plus:
- Inexpensive single-user licenses and updates with Annual-CD program.
- A comprehensive series of TrueControl™ objects for the creation of advanced interfaces and controls for applications that will benefit from sophisticated event-loop programming techniques.
- Tools to compile & bind your source code into independent applications for enterprise use or commercial sale free of royalty fees.

True BASIC GOLD Edition

- All the features found in the BRONZE and SILVER Editions, plus:
- Licensing for multiple users at one site, plus annual update program.
- Advanced interface design and building tools, specialized developer tools, more than 2,000 pre-written library subroutines for graphics, statistics, text sorting, string handling, and advanced mathematical functions in both compiled and source code versions.
- Qwiklinks to compiled C libraries
- Professional development environment with no royalty fees, ever!

You will always find the latest product information and programs at: http://www.truebasic.com or by calling 800 436-2111
INDEX

ABS function, 170
Absolute value (See ABS function)
ACCESS clause (See Open-clause)
Accuracy (See Numbers)
ACOS function, 167, 170, 185
Addition (See Arithmetic operators)
Aliases, 116
Alt keys, 201
Ampersand (See Concatenation)
AND (See Logical operators)
ANGLE function, 170
Animation, 117, 124-125
Applications
creating, 204-205, 210, 214
launching, 18, 213
Arccosine function (See ACOS)
Arcsine function (See ASIN function)
Arctangent function (See ATN function)
Arguments
arrays as, 170, 173-175, 177, 181-182, 197
to functions, 110, 133, 153-154, 170-182, 185, 192, 200
numbers as, 106, 108, 153-154, 170-177, 179-182, 185, 197, 199-200
and parameters, 105-106, 153-154, 166, 199
strings as, 110, 170, 172, 175, 177-179, 182
to subroutines, 105-106, 128, 153, 166
Arithmetic operators
division (÷), 35-37, 140, 166, 168, 179, 188
exponentiation (^), 35-37
multiplication (×), 35-37, 62, 101, 126, 136, 191
subtraction (-), 35-37, 101
Arrays
as arguments, 170-177, 181, 197
arithmetic, 101
assignments, 101, 150, 159, 169, 171, 174, 177, 182-183
dimensioning, 94, 97, 99-101, 154, 159, 169, 175, 181-182, 191, 194, 197, 199-200
elements of, 94-97, 100-101, 157, 159, 169, 173, 181, 189
inputing, 85, 90, 100-101, 148, 150, 159, 195
functions, 168, 174, 182
as parameters, 154, 200
printing, 90, 95, 99-100, 148, 150, 159
redimensioning, 101, 170-171, 174, 177, 183, 186, 189
shape of, 101
shared, 189
variables, 93, 108, 112, 157
ASCII characters
codes, 168, 171, 183
order of, 53, 145
set of, 53, 145, 168, 178
ASIN function, 167, 170, 185
ASK statements
ACCESS statement, 150
BACK statement, 149
BACKGROUND COLOR statement (See ASK BACK statement)
COLOR MIX statement, 149
COLOR, 149
CURSOR, 149
DIRECTORY, 149
ERASABLE, 150
FILESIZE, 150
FILETYPE, 150
FREE MEMORY, 147
MARGIN, 148, 150
MAX COLOR, 149
MAX CURSOR, 149
MODE, 149
NAME, 29, 39-40, 44-45, 149-150
ORG, 150
ORGANIZATION (See ASK ORG statement)
PIXELS, 149
POINTER, 150
RECORD, 150
RECSIZE, 150
RECTYPE, 150
SCREEN, 149
SENDER, 150
TEXT JUSTIFY, 149
WINDOW, 149
ZONEWIDTH, 148, 150
Assignment statements, 25, 169, 171, 174, 177, 182-183
Asterisk operator (See Arithmetic operators)
ATN function, 170
Background color
current, 152
determining, 149
setting, 149, 164
Backspace key, 24, 28
Binary files (See Files)
Binding your program (See also Bound programs)
Blank lines, 12, 28, 42, 51, 65, 67
Blinking cursor, 67
Boolean expressions (See Expressions)
Bound programs, 7
BOX KEEP string format (See Image string formats)
BOX statements
AREA, 120-121, 123, 149, 152
CIRCLE, 121-122, 149, 152
CLEAR, 124-125, 149, 152
DISK, 149
ELLIPSE, 121, 149, 152
KEEP, 124-125, 149, 152
LINES, 120-121, 149, 152
SHOW, 124-125, 149, 152
Breakpoints
BREAK command, 186
continuing after, 141, 186, 198
creating, 141-143, 151, 186, 204-205
executing, 141
finding, 204
BYE command, 198
Byte file (See Files)
CALL statement, 45, 105-106, 112, 128, 153, 166, 192
Caret operator (See Arithmetic operators)
CASE statement, 164, 187, 193
CAUSE ERROR statement, 151
CAUSE EXCEPTION statement, 151
CD command, 16-17, 37
CEIL function, 171
CGA (See Color graphics adapter)
Chaining programs, 149
CHANGE command, 75
Channel numbers
   closing, 84
   with files, 83-84, 86, 88, 187
   with printers, 68, 84, 86
   with windows, 187
Characters
   ASCII values, 145-146
   nonprinting, 175
   number in a string (See LEN function)
CHECKED, 131, 143
CHR$ function, 168, 171, 178
CLEAR statement, 125, 152-153
Clicking, 18-21, 24, 26-27, 71-72
Clipboard, 71-73, 151
Clipping, 161
Clock, 168, 182
CLOSE BOX, 84, 118, 131-133
Colors
   background (See Background color)
   customizing (See SET COLOR MIX statement)
   foreground (See Foreground color)
   names, 123-124, 164-165, 194
   numbers, 124
   simulated, 165
Comma (See Print separators)
Command keys, 201
Command line arguments (See Bound programs)
Command window, 18-19, 68-69, 141, 196, 198, 200, 205-207
Commands (See also individual commands)
   using, 18, 69, 141, 196, 200
Comments, 12, 27-29, 61, 65-66, 163
Communications ports (in SILVER)
<table>
<thead>
<tr>
<th>Term</th>
<th>Page Numbers</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dragging</td>
<td>19, 28, 67-68</td>
<td></td>
</tr>
<tr>
<td>DRAW statement</td>
<td>125-126, 128, 154-155, 160, 192</td>
<td></td>
</tr>
<tr>
<td>ECHO command</td>
<td>69, 196</td>
<td></td>
</tr>
<tr>
<td>Editing</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Edit menu</td>
<td>19, 71-75, 201-202, 204</td>
<td></td>
</tr>
<tr>
<td>find and change</td>
<td>74-75</td>
<td></td>
</tr>
<tr>
<td>keep and include</td>
<td>75, 196</td>
<td></td>
</tr>
<tr>
<td>select all</td>
<td>76, 204</td>
<td></td>
</tr>
<tr>
<td>source window</td>
<td>18-21, 23, 26-27, 29, 68, 74, 76, 140, 142, 188, 205</td>
<td></td>
</tr>
<tr>
<td>Editing Window</td>
<td>18-21, 23, 26-27, 68, 74, 76, 140, 142, 188, 205</td>
<td></td>
</tr>
<tr>
<td>EGA (See Enhanced graphics array)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ELSE statement</td>
<td>164</td>
<td></td>
</tr>
<tr>
<td>Empty PLOT statement (See PLOT statement)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Empty string (See Null string)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>End of data</td>
<td>197</td>
<td></td>
</tr>
<tr>
<td>END statement</td>
<td>26, 47, 65, 87, 111-113, 140, 155, 192, 197, 201</td>
<td></td>
</tr>
<tr>
<td>END statements</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DEF, 156</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DO, 156</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR, 156</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PICTURE, 156</td>
<td></td>
<td></td>
</tr>
<tr>
<td>SUB, 156</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXLINE function</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>EXLINE$ function</td>
<td>173</td>
<td></td>
</tr>
<tr>
<td>EXP function</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>Exponentiation (Arithmetic operators)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Expressions</td>
<td>153-154, 170</td>
<td></td>
</tr>
<tr>
<td>array redimensioning</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>numeric, 40, 170, 183, 192</td>
<td></td>
<td></td>
</tr>
<tr>
<td>rounded numeric</td>
<td>170</td>
<td></td>
</tr>
<tr>
<td>string, 40, 136, 170, 183, 192</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Extended memory (See Memory)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>External procedures</td>
<td>113, 116, 155-156</td>
<td></td>
</tr>
<tr>
<td>EXTENSION statement</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTENSION$ function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>EXTENSION function</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Files</td>
<td>14, 19-21, 23, 26-27, 29, 31, 68, 84, 201-204</td>
<td></td>
</tr>
<tr>
<td>File selectors (See Files)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Files</td>
<td></td>
<td></td>
</tr>
<tr>
<td>byte, 83, 227</td>
<td></td>
<td></td>
</tr>
<tr>
<td>channels (See Channel numbers)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>closing, 84, 88</td>
<td></td>
<td></td>
</tr>
<tr>
<td>compiled, 130, 195</td>
<td></td>
<td></td>
</tr>
<tr>
<td>creating, 7, 9, 12, 15, 23, 27, 33, 69, 82-83, 85-89, 118, 131, 194</td>
<td></td>
<td></td>
</tr>
<tr>
<td>deleting, 150, 196, 204</td>
<td></td>
<td></td>
</tr>
<tr>
<td>erasing, 86-87, 90-91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>input, 86, 91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>lengths, 150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>libraries, 45, 113-114, 126, 133, 156, 158, 160, 199, 210</td>
<td></td>
<td></td>
</tr>
<tr>
<td>listing names of (See FILES command)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>margin, 148, 150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>moving, 76, 91</td>
<td></td>
<td></td>
</tr>
<tr>
<td>multiple, 68</td>
<td></td>
<td></td>
</tr>
<tr>
<td>naming, 7, 20, 23, 29, 69, 85-88, 106, 133, 145, 158, 163, 194-196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numbers (See Channel numbers)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>opening, 20, 39, 53, 66, 83-89, 91, 166, 187-188</td>
<td></td>
<td></td>
</tr>
<tr>
<td>organizing, 148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>output, 21, 69, 84, 86, 88, 91, 186, 188, 206</td>
<td></td>
<td></td>
</tr>
<tr>
<td>printing, 7, 30, 38, 42-43, 50-51, 58, 65, 67-69, 86-91, 96, 206</td>
<td></td>
<td></td>
</tr>
<tr>
<td>random, 229</td>
<td></td>
<td></td>
</tr>
<tr>
<td>record, 91, 233</td>
<td></td>
<td></td>
</tr>
<tr>
<td>recsize, 150</td>
<td></td>
<td></td>
</tr>
<tr>
<td>renames, 31, 34, 72, 75, 196, 201</td>
<td></td>
<td></td>
</tr>
<tr>
<td>saving, 26-27, 29, 31, 39, 72, 75, 82, 85, 128, 188, 196</td>
<td></td>
<td></td>
</tr>
<tr>
<td>stream, 228</td>
<td></td>
<td></td>
</tr>
<tr>
<td>text, 83, 85-87, 89, 91, 187, 223</td>
<td></td>
<td></td>
</tr>
<tr>
<td>zonewidth, 148</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FILES command</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>Finding, 73-74, 141</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Fixing program errors (See Debugging)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FLOOD statement</td>
<td>121, 156</td>
<td></td>
</tr>
<tr>
<td>Floor function (See INT function)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FOR statement</td>
<td>47-49, 95, 103, 144, 156-157</td>
<td></td>
</tr>
<tr>
<td>FOR...NEXT structure</td>
<td>47-50, 52-53, 60, 62, 78-89, 95, 99, 110, 116, 125, 140, 147, 154, 156-157, 160, 189</td>
<td></td>
</tr>
<tr>
<td>Foreground color</td>
<td></td>
<td></td>
</tr>
<tr>
<td>current, 152, 156, 161</td>
<td></td>
<td></td>
</tr>
<tr>
<td>determining, 149</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FORGET command</td>
<td>198</td>
<td></td>
</tr>
<tr>
<td>Format string</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>FORMAT utility</td>
<td>66-69, 140-141, 205</td>
<td></td>
</tr>
<tr>
<td>Formatting (See FORMAT utility)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FP function</td>
<td>174</td>
<td></td>
</tr>
<tr>
<td>Free memory (See Memory)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>FUNCTION structure</td>
<td>149</td>
<td></td>
</tr>
<tr>
<td>Functions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>external, 110-113, 115-116, 126, 148, 153</td>
<td></td>
<td></td>
</tr>
<tr>
<td>internal, 111-112, 126, 153</td>
<td></td>
<td></td>
</tr>
<tr>
<td>invoking, 7, 153-154</td>
<td></td>
<td></td>
</tr>
<tr>
<td>multi-line, 168, 154</td>
<td></td>
<td></td>
</tr>
<tr>
<td>numeric, 149, 195</td>
<td></td>
<td></td>
</tr>
<tr>
<td>one-line, 107-108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>string, 40</td>
<td></td>
<td></td>
</tr>
<tr>
<td>as structures, 61, 140, 148-149, 154, 185</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
user-defined, 131, 133, 154
G_Click, 130-133
G_Create, 131, 133
G_Free, 132
G_Hide, 132
G_Move, 131, 134
G_Show, 132
G_ShutDown, 131, 133-134
G_StartUp, 130, 133-134
GET statements
GET_MOUSE subroutine, 129, 149
Global variables (See Variables)
Graphics
animation, 117, 124-125
drawing lines, 117-118, 120-121,
152, 155, 190
drawing shapes, 120-121, 124-125,
152, 155, 190
pictures, 141, 155-156, 160, 192
transformations, 126, 128
GRAPHICS mode, 122, 124
Graphics modes (See Screen modes)
GraphLib library, 128-129
Help
files, 197
HELP command, 200
Help menu, 200, 205
Highlighting text (See Text)
Hyperbolic cosine function (See COH function)
Hyperbolic sine function (See SINH function)
Hyperbolic tangent function (See TANH function)
Hyperbolic functions, 167-168, 171,
180, 182
Identifiers, 153-154, 166
IDN array constant, 169, 174, 189
IEEE 8-byte format, 178
IF statement, 59, 63-64, 142, 144,
155, 157, 187
IF structure, 60, 62-63, 66-67, 147,
155, 157
INCLUDE command, 75, 196
Increments, 48
Indentation, 12, 48, 51, 66-67, 140
Index variable (See Variables)
Infinite loops (See Loops)
INPUT PROMPT (See Dot product)
INSERTION point, 28-29, 72-76
INT function, 174
INV array function, 174
IP function, 175
Input
from files, 43, 83-91, 186
formatted, 85-86, 88
graphical, 20, 37, 64, 68, 72, 75,
121, 129, 149
items, 44, 84-85, 88, 199
key, 84, 87, 149, 197
line, 29, 41, 44-45, 77, 82-86, 88-91,
97, 100, 148, 150, 158-159, 186-187,
195, 198-199
matrix, 85, 90, 100-101, 148, 150,
159, 195
prompt, 42-44, 68, 85-86, 90, 96,
115, 189
INPUT statement, 29-31, 38, 40-45,
77, 84-85, 90, 101, 148, 158-159,
183, 186, 193, 197-199
INPUT PROMPT statement, 44
Insertion point, 28-29, 72-76
Installation, 13-17, 19, 210, 213
INT function, 174
Integer division (See DIVIDE)
Integer part function (See INT function and IP function)
Internal color numbers (See Colors)
Internal files (See Files)
Internal format
Internal procedures (See Functions and Subroutines)
INV array function, 169, 174, 186,
191
IP function, 175
Joining lines, 28
KEEP command, 75
Keyboard
equivalents, 71, 201
Keywords, 7, 25, 58-59, 65, 67, 84, 87,
99, 121-122, 140, 157, 190-191,
196, 198
LBOUND function, 175
LCASE$ function, 175
LEN function, 175
LET statement, 25, 29, 31, 34, 54-55,
60, 158, 169, 171, 174, 177, 182-183, 198
Libraries
accessing, 45, 113, 128, 130, 133,
158, 199, 210
creating, 210, 214
files (See Files)
LIBRARY statement, 45, 113, 128,
130, 133, 158, 199
Limits, 97
LINE INPUT statement, 41, 44-45,
77, 82-83, 85-86, 89-90, 97, 100,
148, 150, 158-159, 187
LINE INPUT PROMPT statement, 90,
158
Line numbers (See also NUM,
UNNUM, and RENUM utilities),
12, 69, 76, 85, 173, 190, 194, 196
Lines
blank, 12, 28, 42, 48-51, 65-67, 85,
100, 159
blocks of, 67, 71, 196
commenting, 28-29, 65, 72-73, 78,
163, 214
continuations, 37
copying, 73
deleting, 28, 72, 196
indenting, 12, 51, 66-67
joining, 28
marking, 30, 117
moving, 52, 68, 72, 76, 100
selecting, 67-68, 72-73, 141, 187
splitting, 28
LIST command, 68
LOAD command, 199
LOCAL statement, 110, 112, 141,
154, 158-160, 200
Local variables (See Variables)
LOG function, 175
LOG10 function, 175
LOG2 function, 176
Logical expressions (See Expressions)
Logical operators
AND, 56
NOT, 56
OR, 56
Output Window, 18, 21, 27, 30, 43, 68-69, 84, 117, 129, 133, 143, 153, 187, 206

PACKB subroutine, 151
Parameters and arguments, 105-106, 153-154, 166, 199
arrays as, 141, 154, 200
expressions as, 153-154
to functions, 105, 107, 109-111, 141, 154
numbers as, 106, 153-154, 166, 199
passing by reference, 153, 166
passing by value, 154
passing mechanism, 160
to subroutines, 105-106, 109, 111-112, 154, 160, 166
Pasting text (See Text)
Paths, 210
PAUSE statement, 125, 160
PC-DOS (See DOS)
PEEK function, 43
PI function, 178
PICT file (See Files)
Pictures
PICTURE statement, 141, 155-156, 160
PICTURE structure, 125-126, 149, 155, 160, 193
transformations on (See Transformations)
Pixels
in current window, 149
determining color of, 149
PLAY statement, 79, 135, 161, 189
PLOT statements (See also MAT statements)
abbreviated form, 117-118, 161-162
AREA, 122, 129, 161
LINES, 161-162
POINTS, 161
TEXT, 162
Plus operator (See Arithmetic operators)
Pointer (See Mouse)
Points (See PLOT statements)
POS function, 178
POSR function, 179
Precision (See Numbers)

Preface bytes (See Assembly language)
Preprocessors (See DO utilities)
Print separators, 30, 41-43, 47, 51-52, 68, 87, 90, 100, 159, 162
PRINT USING statement, 43, 55, 162, 183, 186
Print zones, 41, 52, 162
Printers
opening a channel to, 68, 84, 86, 186
ports (See Ports)
troubleshooting, 68, 199
PRIVATE statement, 150
Procedures (See Functions and Subroutines)
PROGRAM statement, 163

Programs
closing, 84
compiled, 130, 195, 210-211, 213
creating, 7, 9, 12, 23, 27, 118
deleting, 75
extensions, 20
formatting, 65-67, 69, 71, 140-141
listing names of (See FILES command)
naming, 7, 23
opening, 20, 27, 39, 48, 50, 53, 66, 69, 72, 83, 87, 90-91, 93, 97, 129, 135, 139
printing, 7, 30, 38, 42-43, 50-51, 58, 65, 67-69, 88, 90-91, 96
renaming, 72
saving, 26-27, 29, 31, 39, 128, 210
stopping, 21, 61, 74, 82, 84-85, 110, 140-143, 155, 160, 174, 186, 197
PUBLIC statement, 189
Quitting, 13, 31, 132, 188, 201-204
Quote mark, 30, 39, 44, 65, 79, 158, 186, 190, 196

RAD function, 179
Raising to a power (See Arithmetic operators)
Random files (See Files)
RANDOMIZE statement, 61-62, 163, 179
READ statement, 74-75, 77-79, 99, 159, 163, 183, 188, 197
Reclaiming memory (See Memory)
Record files (See Files)
RECSIZE clause (See Open-clause)
RECTANGLE, 120-121, 132, 152
Redimensioning, 170
Relational operators, 52, 189
REM statement, 163
REMAINDER function, 179
RENAME command, 196
Renaming files (See Files)
REPEATS function, 179
Repeating statements (See Loops)
Reserved words, 193
RESET statement, 87, 91
Resolution (See Graphics resolution and Pixels)
RESTORE statement, 81, 163
RETURN (See CHAIN statement)
Return key, 19, 21, 23-24, 28, 30, 37, 45, 59, 73-74, 205
RETURN statement, 45
RND function, 61-62, 179
ROTATE transformation (See Transformations)
ROUND function, 180
RTRIM function, 180
RUN command, 21, 69, 198
Run menu, 21, 24, 67, 141, 143, 204-206, 209-210
Running programs (See Programs)
RUNTIME function, 180
SAVE command, 29
Saving files (See Files)
images (See BOX KEEP statement)
programs (See Programs)
screens, 124
workspaces (See STORE command)
SCALE transformation (See Transformations)
Scientific notation (See Exponential notation)
Scope, 10
Screen
  clearing, 124-125, 153
coordinates, 187
default, 122, 165
determining, 149
graphics, 122-124
Scroll bar
Scrolling, 18
Screen modes (See also individual modes)
default, 122, 165
determining, 149
graphics, 122-124
Scripts
  files (See Files)
startup, 134
Selecting text (See Text)
Semicolon (See Print separators)
SENSITIVE, 74-75, 130, 132
SET statements
  BACK, 149, 164
  BACKGROUND COLOR (See SET BACK statement)
  COLOR MIX, 124, 149
  COLOR, 122-124, 149, 164-165, 194
  CURSOR, 149
  DIRECTORY, 149
  MARGIN, 43, 148, 150
  MODE, 149, 165
  NAME, 149
  POINTER, 150
  RECORD, 150
  RECSIZE, 150
  TEXT JUSTIFY, 129, 149
  WINDOW, 119-124, 126-130, 149, 165, 187, 200
ZONEWIDTH, 43, 148, 150
SGN function, 108, 180
SHARE statement, 200
Shared variables (See Variables)
SHEAR transformation (See Transformations)
SHIFT transformation (See Transformations)
Signum function (See SGN function)
Simple-statement, 157
SIN function, 180
Sine function (See SIN function)
SINH function, 180
SIZE function, 181
Slash operator (See Arithmetic operators)
SOUND statement, 137, 165
Source window, 27-29, 75
Splitting lines, 28
Square root function (See SQR function)
SQR function, 181
Stand-alone programs (See Applications)
Startup file (See Scripts)
Step size, 48-49
Stopping a program run (See Programs)
STOP statement, 61-62, 165
STR$ function, 168, 181
Strings
  as arguments, 172, 175, 177, 182
  in arrays, 94, 169-170, 176-177, 182, 192
  concatenating (See Concatenation)
  as constants (See Constants)
  converting to numbers (See VAL and NUM functions)
  expressions (See Expressions)
  formatting, 55, 178, 183
  joining (See Concatenation)
  maximum length, 55, 176
  quoted, 43, 79, 88, 153, 186
  substrings, 40
  unquoted, 153, 160
  as variables (See Variables)
  Structures (See individual structures)
  STRWIDTH function, 181
SUB structure, 148, 155, 166
Subdirectories (See Directories)
Subroutines
  built-in, 151
  external, 110-113, 126, 156
  internal, 111, 126
  invoking, 45, 104-106, 112, 128, 153, 166, 173, 192
  as structures, 103, 166, 192
Subscripts
  lower bound, 175
  range of, 173
  upper bound, 182
Substrings (See Strings)
Subtraction (See Arithmetic operators)
Switching directories (See Directories)
Switching disk drives (See Disk Drives)
Switching files (See Files)
Switching modes (See Screen modes)
TAB function, 43, 181
TAN function, 181
Tangent function (See TAN function)
TANH function, 168, 182
TD_LINEINPUT, 45
Text
  changing, 75
copying, 71, 73
cursor (See Cursor)
cutting, 71-72
deleting, 71, 205
files (See Files)
ingcluding, 75, 196
justification (See SET TEXT JUSTIFY statement)
keeping, 75
modes (See Screen modes)
output, 38, 129, 153
pasting, 71, 73
selecting, 28, 71-73, 75, 205
TEXTEDIT, 11, 121-122, 124-125, 130-134, 151
text editor
LINE, 18, 73, 85, 187
TIME function, 182
Time, setting, 34, 141
TIMES$ function, 182
Timing a program (See Programs)
TRACE utility
Transformations
and BOX statements, 127, 155
combining, 128
and DRAW statements, 126-128, 155, 160
on pictures, 126, 128, 160
ROTATE, 126-128, 154
SCALE, 154-155
SHEAR, 126-127, 154-155
SHIFT, 126, 128, 154
Transpose (See TRN array function)
Trigonometric functions, 160
TRIM$ function, 182
TRN array function, 169, 182
TRUNCATE function, 182
UNPACKB function, 151
Unquoted strings (See Strings)
UNTIL (See DO loop)
Upper bounds (See Subscripts)
USE statement, 166
USING (See PRINT USING statement)
USING$ function, 183
VAL function, 183, 195, 198, 200
Variables
global, 109-110, 112
index, 47-51, 189
local, 110-112, 116, 141, 158-159, 200
public, 200
shared, 105, 109, 111-112, 116, 200
Vectors (See Arrays)
WHEN ERROR IN, 188, 195
WHEN structure, 140, 151, 166, 185, 195
WHILE (See DO loop)
WINDOW statement, 119-120, 126-127, 165, 187, 200
Windows
activating, 19, 117
clearing, 149, 153
closing, 84, 131, 133, 142, 188, 204
coordinates, screen, 187
coordinates, window, 119, 126-127, 165
framing, 128
opening, 27, 123, 139, 141, 187, 205
switching, 19
WITH (See CHAIN statement and DRAW statement)
ZER array constant, 183
Zonewidth, 43, 148, 150