True BASIC

The Original BASIC

Bronze Edition

John G. Kemeny
Thomas E. Kurtz
Edited by John Arscott & Anne Taggart

Bronze Edition Guide

for the True BASIC Language System

ISBN: 0-939553-39-2

Published by:
True BASIC
PO Box 204
Gaysville, VT 05746-0204 USA

MANUAL NUMBER: 7222/M

1-888 282-9873	Sales Department
1-888 282-9873	Fax (24-hour availability)
support@truebasic.com	Customer Support
http://www.truebasic.com	Website

Printed in the United States of America. 01/2002

Contents

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**.

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

CHAPTER

An Introduction to Programming 1

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:

Avery Catlin, *Let's Program It... in True BASIC*, True BASIC Press, 416 pp. Third Edition 1996. (ISBN 0-939553-34-1)

Stewart M. Venit & Sandra Schleiffers, *Programming in True BASIC: Problem Solving with Structure and Style*. PWS Publishing Co., 2nd Edition: 544 pp 1998. (ISBN 0-534-95351-4)

Brian D. Hahn, *True BASIC by Problem Solving,* VCH Publishers, 337 pp. 1988 (ISBN 3-527-26863-4)

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** for more information.

CHAPTER

Why True BASIC? 2

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.

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 language 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 indenting 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 computers were limited and did not benefit from new developments. Since 1985, True BASIC has provided an easy-to-use yet powerful, fully structured language for users of personal computers. Dr. Kurtz remains active in True BASIC affairs and has taken a leading role in insuring that this latest BRONZE Edition combines the traditional simplicity of BASIC with a wealth of powerful new features.

CHAPTER

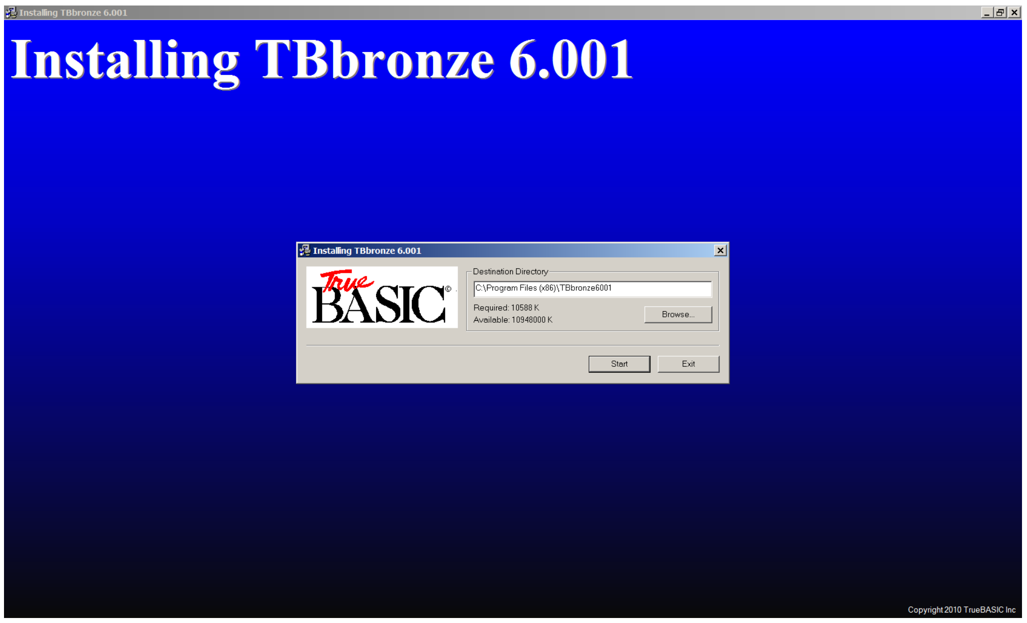
3

Installing True BASIC and Using the TB Editor

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.

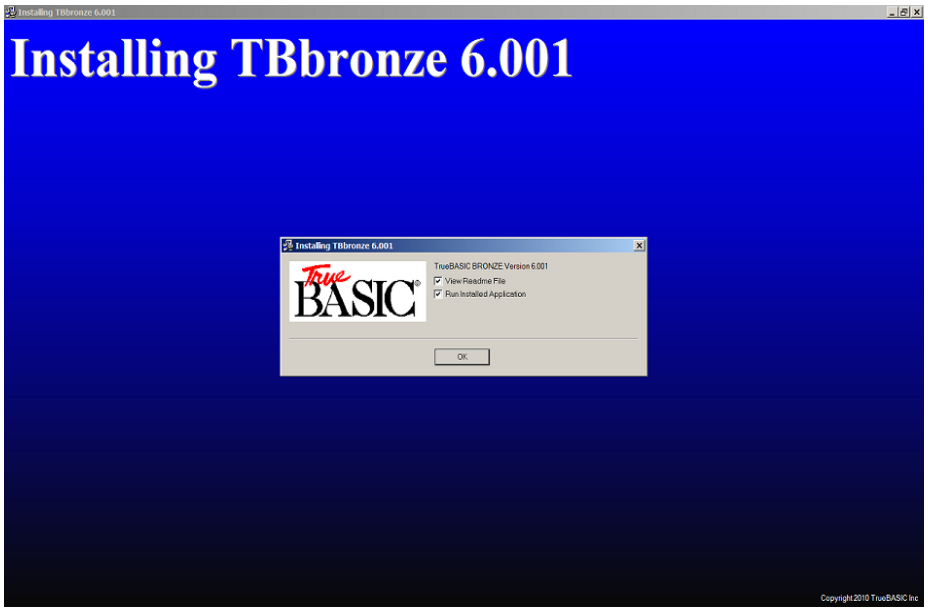
True BASIC runs on Windows 95, 98, 2000, ME, XP, Vista & Windows 7. The following instructions are based on a standard Windows Vista installation and should apply to most situations, but if further information is needed visit www.truebasic.com for more detail. Information on using True BASIC with other operating systems is also available at the website. Vista and Windows 7 users are advised to set the Properties to "Run as Administrator."

1. Locate the setup file (TBbronze6001setup.exe, for example) wherever it was saved during the download process or copied from the CD. By default, in Windows Vista downloaded files are saved in the Downloads folder in the main user directory.
2. Double-click the setup file to begin installation. The setup program will begin to run and you will see this initial screen. You can use the default destination directory or browse to select a different one; then click Start to begin setup.


Installing TBbronze 6.001
Installing TBbronze 6.001
Installing TBbronze 6.001
True BASIC
Browse...
Start
Exit
Copyright 2010 TrueBASIC Inc

Destination Directory
C:\Program Files (x86)\TBbronze6001
Required: 10588 K
Available: 10948000 K

3. The installer will run, unpack all the True BASIC files and save them onto your computer in the destination folder you selected. You will then see this screen:


Installing TBbronze 6.001
Installing TBbronze 6.001
Installing TBbronze 6.001
True BASIC
OK
Copyright 2010 TrueBASIC Inc

TrueBASIC BRONZE Version 6.001
[✓] View Readme File
[✓] Run Installed Application

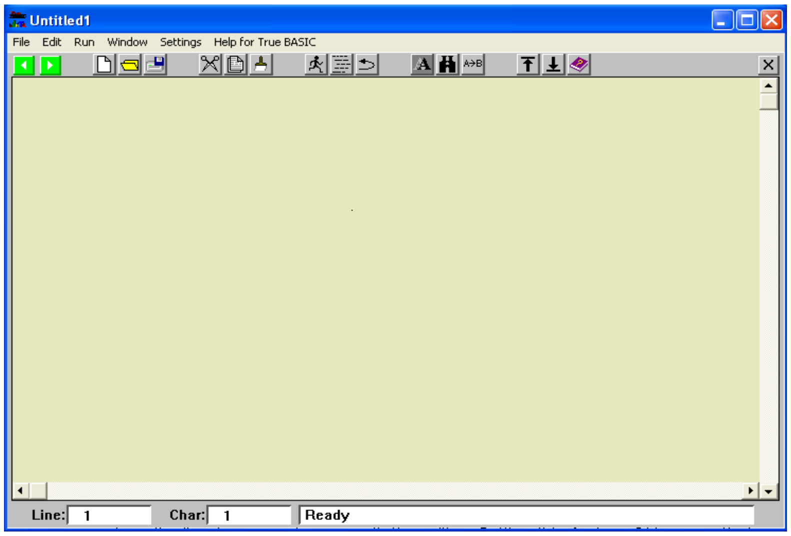
Installation is now complete! At this screen you can also choose to open the Read Me file to learn more about the software, launch the application, or neither. Click OK to get started.

Using the True BASIC 6.0 Editor

Even if you are familiar with word or text processors you are still advised to read these notes because the new True BASIC editor contains a number of unique features that are not available in previous TB editors or other text editors.

START UP

When you start True BASIC for the first time the screen will show an empty window labelled “Untitled 1” in the top left corner of the screen. In earlier versions at start up, True BASIC displayed a small file selector dialog box where you can click on the NEW button to start with an empty, untitled window.



There are several ways of starting the editor:

(1) Double click on the editor desk top icon.
(2) Set up a file association between TRU files and the editor.
(3) Drag file icons onto the editor icon
(4) Chain to the editor from another application

File associations can be set up from Windows control panel:
Select Folder Options.
Click the File types tab.
Scroll down the list of extensions and click TRU
Click the CHANGE button.
Click the BROWSE button and then navigate to TBeditor.exe in the folder where you installed the editor.

DEFAULT SETTINGS (settings menu)

The following settings have been pre-set, but you can customize them at any time. Any change in settings will remain in force until you next change them.

Save on close
The default setting is OFF, meaning that source code will NOT be saved automatically when you close or exit the editor. Setting this feature ON means that when you close the editor, then all currently open programs will be saved.

Back-up on save
The default setting is OFF, meaning that a back-up copy of your source code will NOT be automatically created whenever you save your program code. Setting this feature ON means that a back-up copy of your source program will be saved with the same program name, but with the extension BAK.

Confirm quit
The default setting is ON, meaning that whenever you attempt to close or exit the editor you will be asked if you are sure this is what you want to do. If this feature is switched OFF then you will not be asked if you are sure. The editor will shut down as requested.

Hotstart
The default setting is ON, meaning that when you start up the editor it will return to exactly the same conditions as it was when you last shut down. The programs you had open at the time will be re-opened and the last program you were using will be the focus. The position of the cursor will be the same as you left it. If this feature is switched OFF then the editor will start up with an empty 'Untitled' window.
This feature was incorporated in many earlier versions of the editor but never worked consistently.

Binder
The program that runs, compiles and binds source code is called TBsystem.exe. The default version is 5.5b19. This means that your executable programs do not need the 3 DLL files that older versions needed. However, there are certain features of the old TBsystem file that you may prefer, in which case 531TBsystem.exe can be used.

Aliases
Previous versions of the editor allowed programs to use short filenames instead of the full pathnames in LIBRARY statements. A list of alias pathnames was used by the editor in order to locate the short filename. This principle is incorporated in the new editor. The default list contains alias types {library}, {do} and {help}. A maximum of 9 alias types can be specified by the user. Aliases can only be used with literal filenames, e.g. "{library}TrueCTRL.trc"

Function keys
The default setting is OFF. When the switch is on it means that the function keys F2 to F9 work in a similar way to the DOS version function keys, e.g. F4 marks or highlights a block of text, F5 copies and pastes this block, and F6 cuts and pastes this block. If this feature is switched OFF then the editor will not respond to the function keys. This feature has been enabled in this version.

Short cuts
The default setting is ON, meaning that all menu items will be shown with their short cut keystrokes. If this option is turned OFF then the menu displays will only show the menu item and not the equivalent short cut keystroke.

SINGLE WINDOW
Unlike earlier versions of the editor, this version only has one window, whereas previous versions had one window for each open program. However, in the single window you can open up to 10 programs simultaneously. The two green arrow buttons allow you to switch between any of the open programs, just like the forwards and backwards buttons on a browser. The window title bar shows the name of the current program. You can also switch to a specific program by clicking your mouse on the program name in the list under the WINDOW menu. Note that the current program is ticked in this list. You can close any individual program by clicking the mouse on the close program button at the right hand end of the toolbar (black cross on a gray background). If you wish to compare two programs side by side, start two instances of the program. The number of open windows is limited only by your computer's memory.

RECENT PROGRAMS
When you close a program it gets deleted from the list of open programs but at the same time it gets added to the list of the 10 most recently used programs, which you can see under the WINDOW menu. If you click the mouse on any program in the recently used list, then this program will be opened and will become the focus.

COMMAND LINE
The command line in earlier versions appeared in its own window, called the command window, but in every other respect the new command line works the same way, i.e. you can type instructions on this line and the computer will execute them immediately without the need to select RUN. For example, you can type the word FORMAT and True BASIC will format (indent) your code. Similarly, if you type RUN then True BASIC will run the current program. If you type VER (version) then your will see the version number and date. Note that not all of the original commands will work in this version, in particular the PRINT *variablename* which allowed the user to stop the program and inspect the values of variables. This important feature has now been added to the BREAKPOINT feature instead. The command line also allows the user to specify alias names, e.g. ALIAS {myfolder} c:\Tbsilver. Aliases specified at the command line are only valid for the current session. When you shut down the editor, these aliases will not be remembered. The command line can also be used to specify a SCRIPT file, e.g.
SCRIPT myscript.txt.

SCRIPT FILES
When the editor first starts up it looks for a SCRIPT file called STARTUP.TRU. You can only use this script file to LOAD libraries and to specify ALIAS commands. All the other commands in a script file will be ignored. If STARTUP.TRU is not present in the same directory as the executable editor, the editor will carry on as normal. You can also use the SCRIPT command on the command line to specify a script file with another name, e.g. SCRIPT myscript.txt. Unlike the LIBRARY statement, there are no quote marks around the file names.

LOADING LIBRARIES

Loading libraries is an alternative to using the LIBRARY statement. You an use a script file to load one or more libraries into the editor, or you can specify the library on the command line, e.g.
LOAD mylib.tru, yourlib.tru.
Unlike the LIBRARY statement, there are no quote marks around the file names. Loaded libraries work as if the library routines have been incorporated into the main True BASIC language system. Any routine in the loaded libraries is therefore available to your program, in fact the loaded libraries are available to all your programs in the current session. When you shut down the editor the loaded libraries are cleared from memory and forgotten. You can also clear all loaded libraries by using the FORGET command on the command line.
CAUTION: if you are using line numbers in your program, remember to leave plenty of spaces between line numbers because the editor inserts extra code into your program to achieve the LOAD feature.

COMMENTS

An exclamation mark at the beginning of a line tells the computer to ignore the line because it is NOT a program instruction. These reminder notes are called "comments" and they can be used anywhere in your program. Indeed it is good practice to make comments after certain lines of code to remind yourself what that line of code is actually doing, because it is not always blindingly obvious. As an alternative to the exclamation mark (!) you may type the word REM, short for reminder.

A feature of the new editor is that if you highlight a block of text by dragging your mouse across the text, then you can comment all the lines in the text by typing the exclamation mark just once. This is a toggle action feature in that you also un-comment a block of lines by doing the same operation. The toggle action works on the basis that if any line does not have a comment mark then it will add one, whereas if the line already has a comment mark, it will be removed.

AUTO EXTENSIONS (TRU)

NEW or blank programs can be selected in the same way. By default the name of the blank program is shown on the window title bar as 'UNTITLED'. You can define an appropriate name for the program when you save it. It is a feature of the new editor that program names automatically have the extension TRU added to the name if no other extension exists. This was a feature of the old DOS editor.

SWITCHING FILES

Existing programs can be opened by selecting OPEN from the FILE menu or by selecting the OPEN button on the toolbar. In either case the new program will become the focus in the window and the blue title bar will show the program name. Any existing program previously displayed in the window will still remain open in a queue behind the focus program. You can move easily between the queue of programs just by clicking the green arrow buttons on the toolbar, or by selecting the program by name from the list under the WINDOW menu.
The editor keeps track of each program in the queue, so that when you switch from one to another, the cursor position is in the same place as it was when you last used the program. A maximum of ten programs can be open at the same time.

UNDO and REDO

UNDO can be applied to:
CUT
COPY
PASTE
FUNCTION F4 (select text block)
FUNCTION F5 (copy and paste)
FUNCTION F6 (cut and paste)
DELETE
TYPING
KEEP
INCLUDE
DO….
FORMAT
UPPER
LOWER
FORMS
TBILT
The UNDO menu item shows the current action, e.g. UNDO delete.

In the case of typing, UNDO will restore the original text before CONTINUOUS typing began. For example, suppose you type ABC anywhere on an existing line, then UNDO will remove ABC. You can still use the back-space key to remove individual letters. You might have typed say 20 lines of code, and when you press UNDO then all 20 lines will be removed. To limit the amount of text in a single "UNDO typing" group, you can break up the groups by highlighting a letter or word and then clear the highlight before carrying on with your typing. By selecting a highlight you are effectively creating a new activity, so the editor closes the current typing activity and opens a new activity called "Select" and waits for you to decide what you are going to do next. If you carry on typing, then the editor renames "Select" as "Typing".

Each time you begin an activity such as typing, cut, copy, paste, DO FORMAT etc., the editor takes a "photocopy" of the current program and keeps it in an internal array so that you can return to this copy if you want to UNDO the activity. There are ten elements in the internal array so you can use UNDO to backtrack ten times. It is unlikely that any program will exceed 2MB so all 10 "photocopies" will only take up 20MB of memory. Obviously you will perform more than just ten activities, so on the eleventh activity, the first internal array is re-used. This first-in-first-out process continues indefinitely.

For example; suppose you have already done 9 different activities and you are now typing, i.e. the tenth item in the undo list is typing. You now want to do a cut and paste operation so your undo list will now have two more items – "cut" and "paste" as the eleventh and twelfth activities. The internal array is limited to 10 elements so the whole list moves up two places to allow cut and paste to occupy elements 9 and 10. Your previous typing activity now occupies element 8. The two old elements at the top of the list are discarded. You now decide to back track 5 places up the list with the UNDO feature. Now you realize this is too far so you move down the list with the REDO feature by two places, i.e. element 7. Ahead of you there are still typing, cut and paste, but you decide that element 7 is where you want to be, so you begin typing your program again. This typing operation will now OVER-WRITE element 8 and subsequent activities will over-write elements 9 and 10. However, all the elements prior to element 7 will still be preserved.

When you use the UNDO feature the internal arrays are restored in reverse order. Likewise, when you use REDO, the internal arrays are used in forward order. The penalty for having this extensive feature is a slight delay when switching from one program to another as the internal arrays are downloaded to the hard drive. There are no significant delays when editing individual programs.

When you change the current program, these internal arrays are downloaded onto your hard drive, so that if you go back to this program, the undo features are still operational by uploading these internal arrays. This applies to all ten possible open programs. All the hard drive files associated with open programs are deleted when you EXIT the editor. Only the relevant hard drive arrays are deleted when you CLOSE a program.

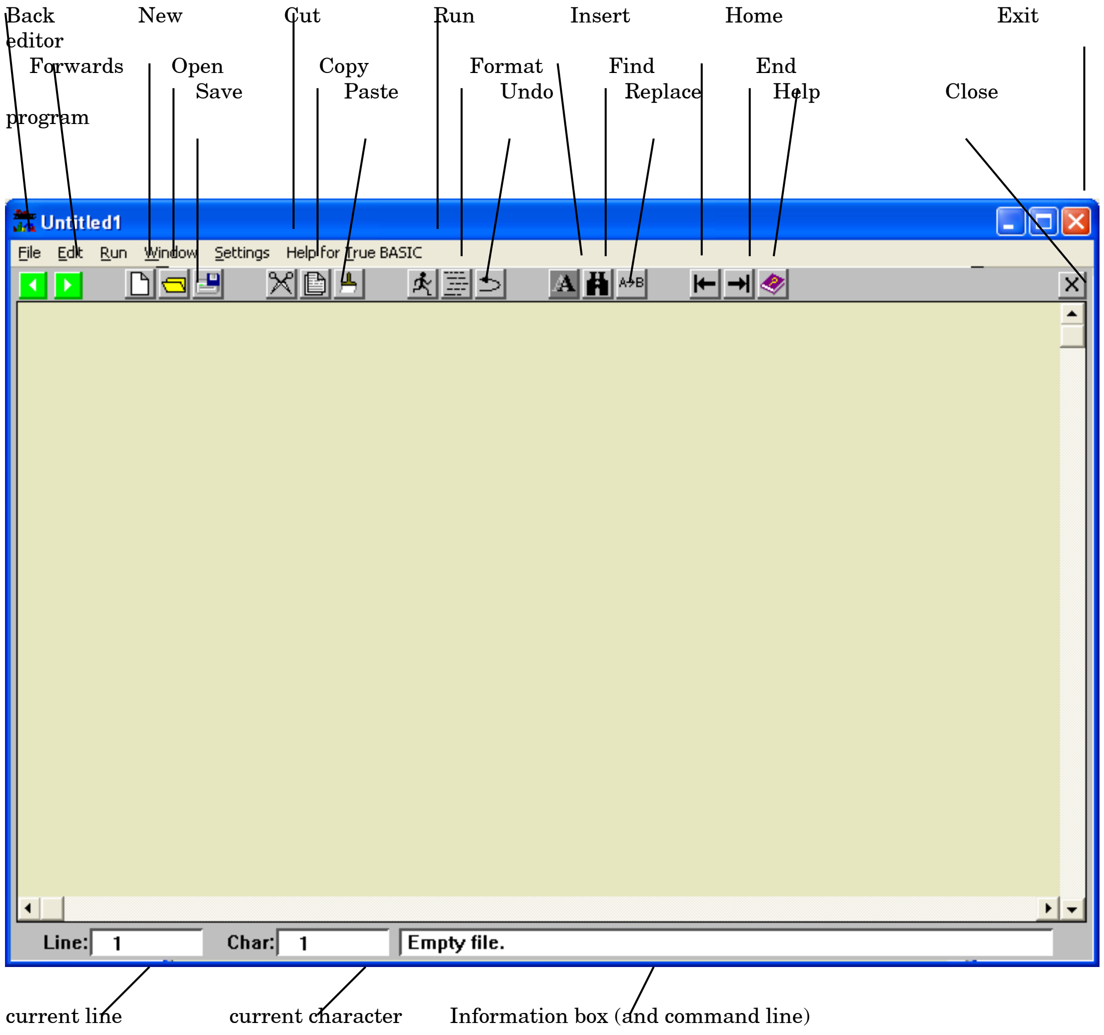
PRINTING HARDCOPY

The editor offers you two strategies for listing hardcopies of your programs. The first is a legacy method sometimes referred to LTPR or line printer. The second method treats the printer as a virtual window, which allows both PRINT and PLOT instructions. Some commercial printer (HP) drivers have difficulty responding to both these print methods. In version 6.007 an extra print method has been added, which uses a thirty party freeware program called “prfile32.exe” to execute hard copy printing. The editor automatically searches for this program. If it has been installed on your computer, then the editor will use it.

There is another third party freeware program called “hardcopy.exe” which installs an extra green printer icon in the top right corner of all windows. Clicking this button produces a hardcopy graphics picture of the current window. This can be used to hardcopy print the output window, for example, as well as visible portions of the current program in the editor window.

TOOLBAR

In this version of the editor there is also a toolbar at the top that gives you quick access to a number of frequently used features. When the cursor is in the toolbar zone it changes shape to a pointer; tooltips also appear identifying the function of each button on mouseover.



The central area (you can change this color later) is your working page. Below this page are two information boxes that tell you the current position of the cursor. The box on the left gives you the line number and the box on the right gives the character number counting from the left. On the right at the bottom of the window is an information box. This box is also the command line where you can type instructions directly to the computer. If you click the mouse inside the information box it will turn blue, and you can begin typing your instructions. For example, if you type the word “version” in the blue box, the computer will immediately respond with the current version number of this edition. Note that after v6.006 there is an additional red STOP button on the toolbar, located between the PASTE and RUN icons. Clicking this icon opens the dialog box that allows you to stop a running program.

UNWRAPPED TEXT

The most important difference between the True BASIC editor and other text editors is that when you type your instructions, the lines of text are NOT WRAPPED, i.e. when your typing reaches the right hand margin it just carries on and on. The text does not automatically drop to the next line down. The only way you can drop to the next line down is to press the RETURN key on your keyboard because this signifies the end of a line. The reason behind this method of operation is that True BASIC only allows ONE instruction per line, but that instruction may be too long to fit the width of the page, so the editor will always allow you enough space for your instruction regardless of how long it is. There is a scroll bar across the bottom of the editor page so that you can view anywhere along very long lines.

If long lines worry you, and you would prefer to see your all of your program without having to scroll across the page, then you can use the ampersand sign (&) to terminate a line as long as you begin the continuation line with an ampersand too.

WRAPPED TEXT

Under the EDIT menu there is an option that allows you to view wrapped text. For example you may wish to consult a text document during the course of writing a program. Please note that you cannot use COMPILE, RUN or BIND when you are in the WRAP mode. This feature has been enabled in this version.

OVER-TYPING

If you press the INSERT key the editor will change from inserting characters at the cursor point to over-typing at the cursor point. If you press the INSERT key again then insert typing will resume. Unlike all previous editors, this version indicates which of these two modes is operational, by illuminating the over-type icon on the toolbar.

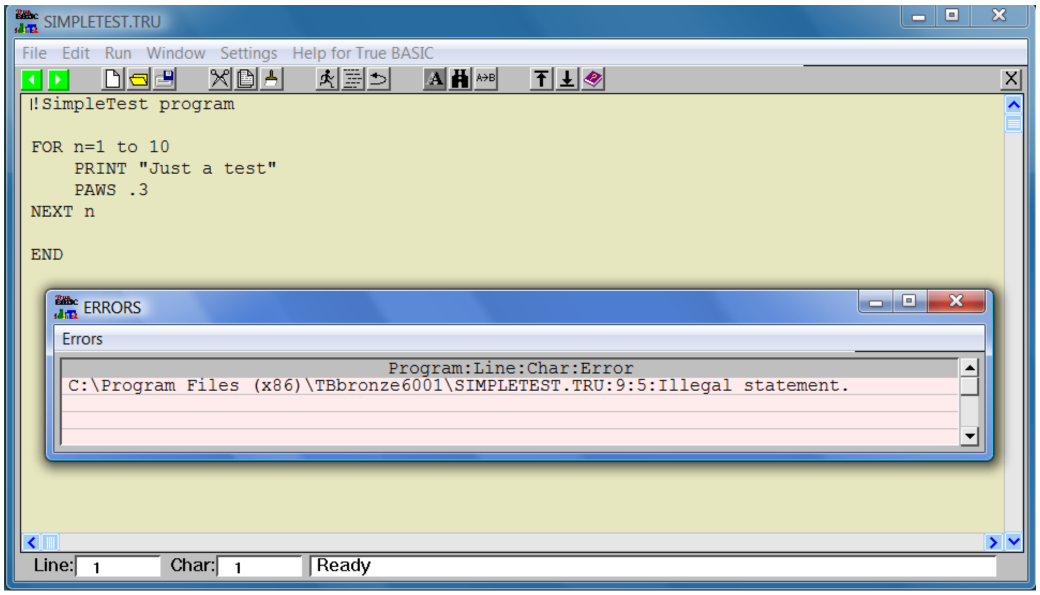
FORMATTING TEXT

The editor is very tolerant of the way you type the program instructions. You can use upper case or lower case or both. You can also add spaces as often as you like if they make things clearer to read. Indeed there is a utility feature built into True BASIC that will “format” your code, i.e. it will indent certain keywords to make the program easier to read and easier to understand the way it is structured. In a way it is a bit like using paragraphs and bullet points in ordinary text. You will find the format feature one of the most useful items on the True BASIC menu.

ERROR DETECTION

Whilst True BASIC is tolerant of the way you set your program out, like all other computer languages it is not so tolerant about the instruction code itself. When you type an instruction it has to be word perfect, and if there is any punctuation it has to be perfect too. It is not good enough to get it nearly right; it has to be perfect. Fortunately, True BASIC is wise enough to know that it is dealing with human beings that have a habit of making mistakes, so it has an extensive error detection system built in. When you attempt to RUN your program, if you have made any mistakes then True BASIC will almost certainly find them. In this version of the editor your source code is subjected to the error detection process whether you compile, run or bind your code.

Let us suppose that you have loaded the program SIMPLETEST.TRU and that we have incorrectly spelled the word PAUSE and we have used the word PAWS instead. If we attempt to RUN the program then we would expect the compiler to detect this error and report it. This will give you an opportunity to see how the error detection system works. Select RUN from the RUN menu.



```
!SimpleTest program

FOR n=1 to 10
    PRINT "Just a test"
    PAWS .3
NEXT n

END
```

```
                    Program:Line:Char:Error
C:\Program Files (x86)\TBbronze6001\SIMPLETEST.TRU:9:5:Illegal statement.
```

If the error detection process picks up an error or a series of errors, then these will be presented on screen in a separate error window in the form of a scrollable list. If you click your mouse on any line in the list then the corresponding error line in your code will be highlighted.

The information box shows that there was an error while running the program. The compiler detected the errors, and these are displayed in the error window in tabular form. If you click on any line in the table of errors, then the corresponding line in your program will be highlighted. The Preferences box allows the user to change the full line highlight to just the first character.

If you correct the error, the program will run successfully and prints the phrase ‘Just a test’ ten times in the output window. To exit from the program and return to the editor you must press any key or click the window with the mouse.

You are free to use upper or lower case but my advice would be to use lower case for all your variables and upper case for keywords. The built-in DO FORMAT option will automatically convert keywords to upper case for you, so you can use lower case for everything. There is a strong body of evidence that suggests lower case is much easier to read.

BREAKPOINTS

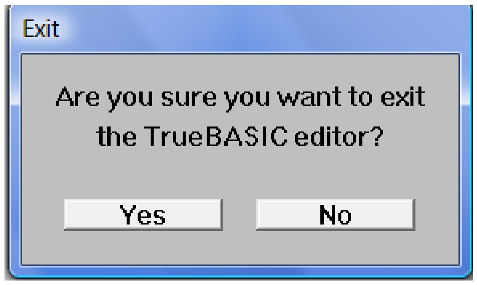
Breakpoints mark your program at the line where the cursor is positioned. The BREAKPOINT option under the RUN menu is normally disabled (grayed out) and only becomes active when you select DEBUG MODE from the SETTINGS menu. The breakpoint will appear as the word <<<BREAKPOINT>>> surrounded by angle brackets. If you RUN your program with breakpoints marked, the program will stop at the first breakpoint. A dialog box will give you the opportunity to continue running your program. All breakpoints are cleared when you toggle DEBUG MODE again.
If you add a series of variable names after a breakpoint, e.g.
<<<<BREAKPOINT>>>> a,b, string$
then when the program stops you will see a list of these variables and their current value. This is a very useful feature for locating bugs. For example, the breakpoint can be inserted inside a FOR…..NEXT loop to check how the value of variables change with each increment of the loop. The variables list dialog box will give you the opportunity to continue running your program.

FUNCTION KEYS

If the Function keys option under the SETTINGS menu is ticked then:
F2 will make the command line active (blue)
F3 will display the FIND window.
F4 will mark (highlight) the first line in a block of text. A second press of F4 will mark the end of a block of text.
F5 will copy and paste the text marked by F4 to the current cursor position.
F6 will cut and paste the text marked by F4 to the current cursor position.
F7 will undo the last operation.
F8 will toggle a breakpoint on the current line.
F9 will run the current program.
(NB. This feature has been enabled in this version.)

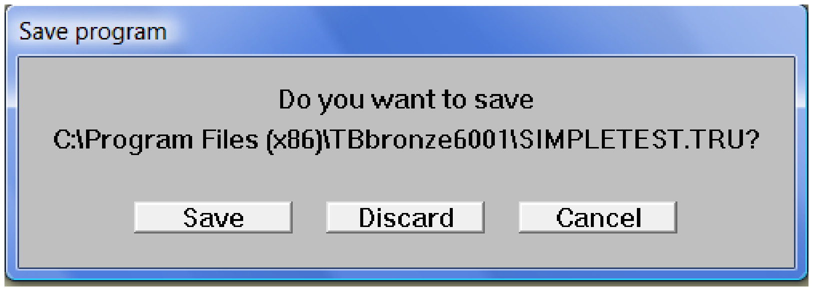
EXIT THE EDITOR

If you wish to exit the editor you must select EXIT under the FILE menu or you can click on the 'close window' button (white cross on a red background). You will be asked if you are sure you want to QUIT. You can eliminate this reminder by un-ticking the 'Confirm quit' option under the SETTINGS menu.


Exit

Are you sure you want to exit the TrueBASIC editor?

If you click on YES (or press the <RETURN> key then you will be asked if you wish to save your program if you have made any changes to the program since the last save operation. If you have not made any changes you will not be asked if you wish to save. Likewise if you have selected 'Save on exit' under the settings menu then your program will be automatically saved without displaying this dialog box.

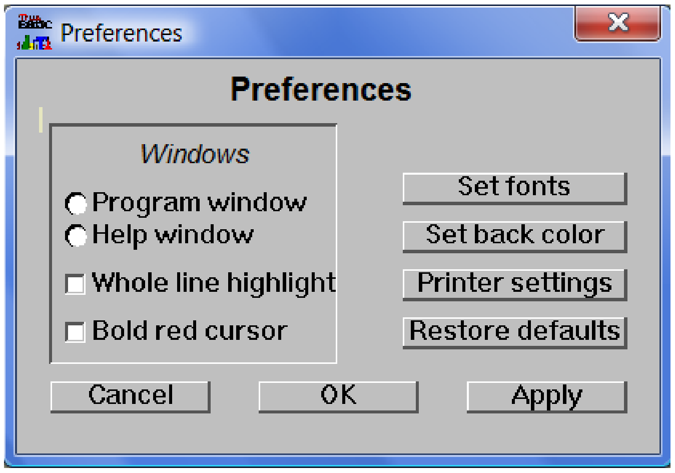

Save program

Do you want to save C:\Program Files (x86)\TBbronze6001\SIMPLETEST.TRU?

The same process will be applied to all currently opened programs before True BASIC finally terminates.

PREFERENCES

Under the SETTINGS menu, the user can select Preferences to set up the editor to suit the user.



To set the font or background color for the source program window, first click on the radio button labelled Program window, and then click on the button for font or background color.

If you want the whole line highlighted to indicate a compile error then click on the check box. Otherwise only the first character in the line where the error is located will be highlighted.

If you want the text cursor to change from a simple vertical black line to a bold red line that is easier to see, then click on the check box.

If you want to save a change that you have made then click the APPLY button. This will allow you to continue to make other changes. With each change you must click the APPLY button to save the change. When you have completed all your changes you must click the OK button to execute all your saved changes.

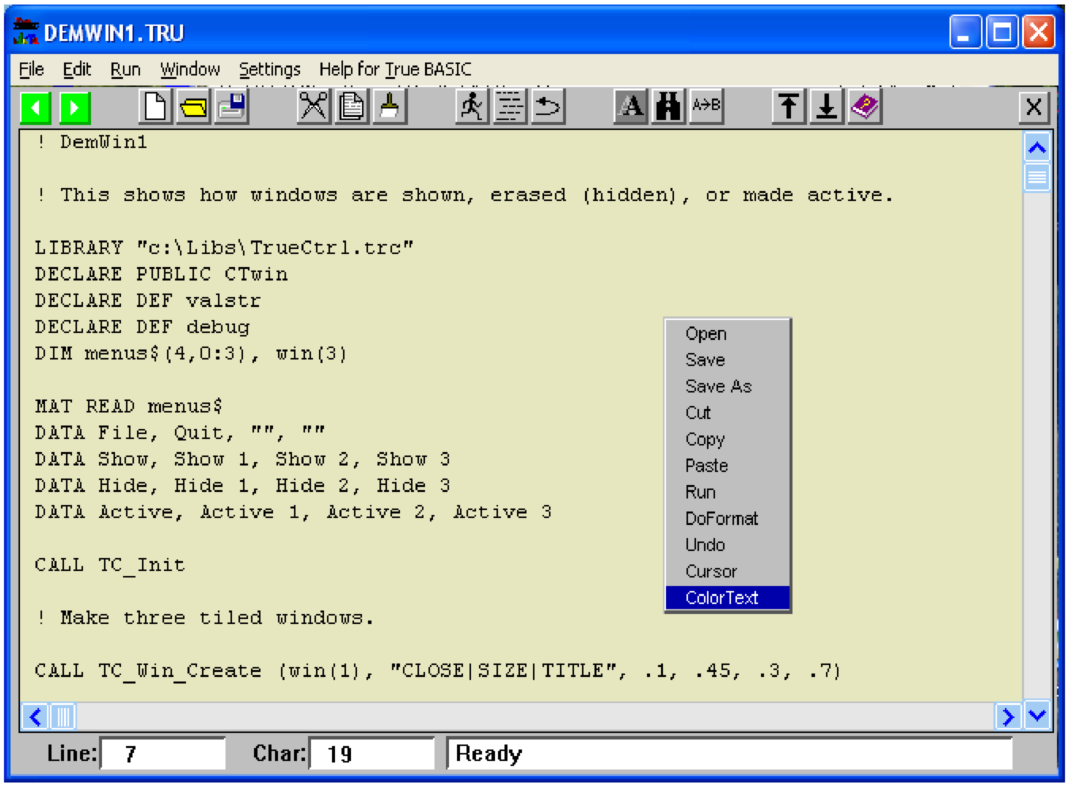
If you want the default settings to be restored then click on the appropriate button. If you click on either the APPLY button or the OK button, the defaults will be restored immediately.

If you click the CANCEL button at any time, then all saved changes will be ignored and the current settings will remain in force.

The printer settings allow you to change the number of print characters across the page and the number of lines down the page. The default is 80 characters and 60 lines.

RIGHT CLICK /SHORTCUT MENU

By clicking the right hand mouse button you can reveal the shortcut menu.



```
! DemWin1

! This shows how windows are shown, erased (hidden), or made active.

LIBRARY "c:\Libs\TrueCtrl.trc"
DECLARE PUBLIC CTwin
DECLARE DEF valstr
DECLARE DEF debug
DIM menus$(4,0:3), win(3)

MAT READ menus$
DATA File, Quit, "", ""
DATA Show, Show 1, Show 2, Show 3
DATA Hide, Hide 1, Hide 2, Hide 3
DATA Active, Active 1, Active 2, Active 3

CALL TC_Init

! Make three tiled windows.

CALL TC_Win_Create (win(1), "CLOSE|SIZE|TITLE", .1, .45, .3, .7)
```

Line: 7 Char: 19 Ready

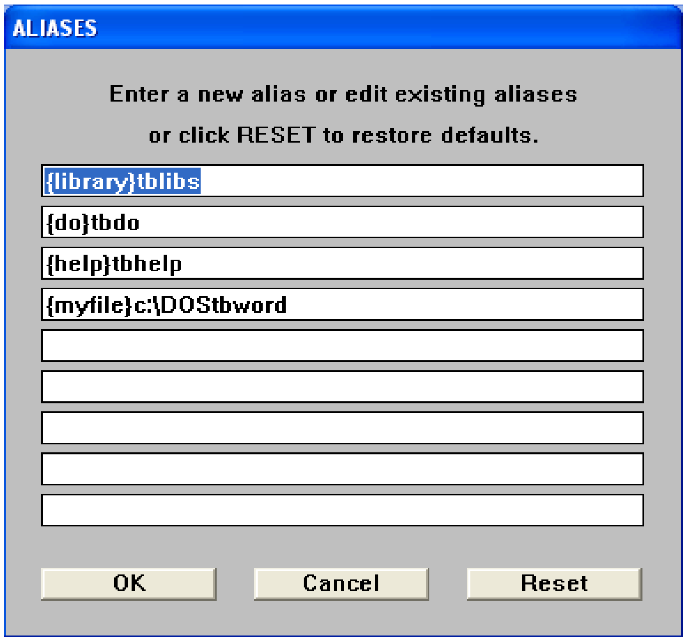
This menu works like the main menu. It will disappear as soon as you make a selection from the menu.

SETTING ALIASES

SET ALIAS under the PREFERENCES menu allows you to create new aliases. There are three default alias types, {library}, {do} and {help}.
You may add or edit more in the fields provided.

Note: it is important to used curly brackets around the alias type, followed immediately by the directory pathname. Aliases added under SET ALIAS are permanent, i.e. the editor will remember the aliases when you shut down so that when you restart the editor the aliases will still be in effect. Temporary aliases can be added at the command line or by means of a SCRIPT file. Temporary aliases are not remembered when the editor shuts down.

Some previous versions of the TBeditor allowed users to ignore the alias group name in curly brackets. The editor looked into the three default folders to see if the file was located in these folders. This feature has been preserved for the benefit of legacy code. In other words, as long as the file is in any of the three default alias folders then you do not need to specify the alias group name in curly brackets.



Enter a new alias or edit existing aliases
or click RESET to restore defaults.

{library}tblibs
{do}tbdo
{help}tbhelp
{myfile}c:\DOStbword

The reset button restores the three default alias types and clears the remaining fields.

COLORTEXT

This new feature will color certain words in your True BASIC source code. Currently these parts are:
Linenumbers (if any)
Comments
Keywords (i.e. statements)
Functions and definitions
CALLs and SUBs
Literal quotes and string variables
Aliases
Numeric variables and constants
Punctuation

Depending on the background color, a default set of 9 different colors is used to color these parts. The standard color numbers are:

BLACK (or dark backgrounds)

7 (gray) for Linenumbers
10 (green) for Comments
9 (blue) for Keywords (i.e. statements)
13 (magenta) for Functions and definitions
12 (red) for CALLs and SUBs
11 (cyan) for Literal quotes and string variables
14 (yellow) for Aliases

24 (orange) for Numeric variables and constants
-2 (white) for Punctuation

WHITE (or light backgrounds)
8 (dark gray) for Linenumbers
2 (green) for Comments
9 (blue) for Keywords (i.e. statements)
13 (magenta) for Functions and definitions
12 (red) for CALLs and SUBs
3 (dark cyan) for Literal quotes and string variables
6 (dark yellow) for Aliases
25 (brown) for Numeric variables and constants
-1 (black) for Punctuation

Users are also free to create their own custom list of 9 colors in a simple text file. However, these colors are applied regardless of the background color.

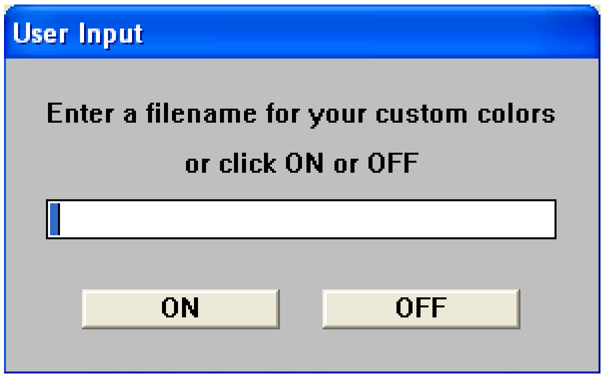
COLORTEXT can be activated from the SETTINGS menu or from the right click menu.

Defined functions will only be colored correctly if the function has previously been declared e.g. DECLARE DEF mydef.

Once COLORTEXT has been switched on, then all currently open source programs will be colored, except wrapped files.

As you type, the text color may change with each keystroke until you press the space bar or type a punctuation mark. At that point the text will take on a fixed color.

To switch COLORTEXT on click the ON button. To switch off COLORTEXT then click the OFF button. If you leave the filename field blank then the default colors are used. If you enter a filename (full pathname) then your list of custom colors will be used.


User Input

Enter a filename for your custom colors

or click ON or OFF

LINE NUMBERS

Legacy code often uses line numbers, and some users may prefer to continue working with line numbers, even though TrueBASIC does not require them. The TrueBASIC editor works with or without line numbers. There are several utility programs which allow users to number, renumber and un-number programs. It is important to note that programs are automatically re-numbered after CUT and PASTE operations or when lines are deleted. GOTO and GOSUB references are also updated during re-numbering.

AUTO LINE NUMBERING

The editor has a built-in feature that allows automatic line numbering. To invoke this feature the user must insert the following line as the first line of their program:

```
100 !AUTOLINENUM
```

Note that the line must begin with a line number followed by a space, followed by a comment mark (!). The keyword AUTOLINENUM is not case sensitive. The line number signifies the number you wish to start at. All subsequent lines are numbered in increments of 10. By embedding the automatic line numbering switch inside the program, means that the editor can detect which programs require numbering and those that don't. This leaves the user to move freely between programs without having to switch this feature on or off for each program.

If the keyword line is removed, then the program becomes just a regular manually numbered program. Likewise a manually numbered program can be made automatic by adding the keyword line at the beginning, regardless of whether the current program is already numbered or not.

DELETING TEXT

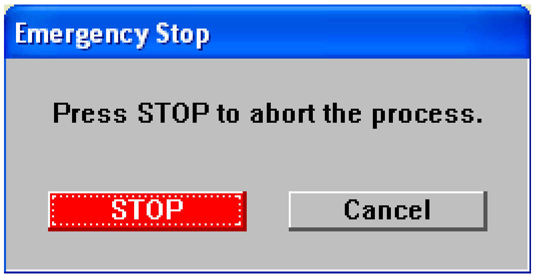
From the current cursor position, the DEL key will delete single characters ahead of the cursor. The BS (backspace) key will delete text behind the cursor. The DEL key will also delete any highlighted text. Similarly, the back space key will also delete highlighted text.

If a block of text is already highlighted when you PASTE any text from the clipboard then the highlighted text will be replaced by the pasted text.

Note that EDIT fields, i.e. input boxes such as those in the FIND box or the CHANGE box , will now allow pasted text as well as typed text.

STOPPING PROGRAMS

There may be times during the course of developing programs that you will attempt to run a program that has an error that hangs the computer, or in some other way doesn't terminate properly. For example you may have a DO…..LOOP statement with no EXIT DO to escape the loop. On the toolbar there is a red STOP icon which produces an Emergency Stop dialog box containing a STOP button. When you click on this button, the running process will abort immediately and you will be returned to the editor.



Press STOP to abort the process.

Note: if you are running a program, then it may produce a window that obscures the editor and the STOP toolbar icon. To reveal the editor window, click on the editor label on the taskbar or slide the program window out of the way to show the editor underneath.

HIGHLIGHTING TEXT

There are two ways to highlight text:

(1) By manually dragging the mouse across the text.

(2) By using the arrow keys in conjunction with SHIFT.

In the first method the highlighted text NEVER includes the end-of-line characters at the end of the last line highlighted. As a result, when this text is pasted into your text there are no “returns” or extra lines generated.

In the second case the end-of-line characters are ALWAYS included. As a result, when this text is pasted into your text then a new line is generated immediately after the end of the pasted text.

If you highlight any text prior to a paste operation, then the pasted text will replace the highlighted text.

If you highlight any text prior to typing at the keyboard, then the typed text will replace the highlighted text.

PEN COLOR (for lines of text)

A new text coloring feature has now been added. If you are modifying a program, you may wish to print the modifications in a different color so you can easily recognize what changes you have made. This cannot be done by changing the pen color because this will change the color of the whole text. Individual lines or blocks of text can now be colored by adding a color signature to each line. This done by highlighting the block of modified text and pressing the keys (#) for blue or (%) for red. This is a toggle action, so you can remove the color signatures by highlighting the same block of text and pressing either (#) or (%). The signatures (!#) or (!%) can also be added manually. The colored text can be run, compiled or bound in the normal way.

The True BASIC Editor menus

Normally you would use the mouse to click on menu headings, and then to click on items under the heading. Alternatively you can press the ALT key on the keyboard to activate the menu bar. The side arrow keys can be used to drive the heading highlight backwards and forwards across the menu headings. The up and down arrow keys can then be used to highlight individual menu items. Pressing the <RETURN> key will select the current menu item.

FILE MENU

- **NEW** - this option opens a new empty editing page in the main window with the default title "untitled" followed by a sequential number. A maximum of ten new and existing windows can be open at the same, and you can switch between them as often as you like.
- **OPEN** – will raise an open file dialog box where you can navigate through drives, folders and files to select the file of your choice. The list of files is limited to program files only, i.e. those files with the extension TRU or TRC. You can extend this by selecting ALL FILES in the file type box. When you select a file it will displayed with the file name as the window title. The information box will display the total number of lines in the program.
- **CLOSE** – will close the current program in the main window. This action is identical to clicking the mouse on the close button (black cross on a gray background). You will be asked if you wish to save the contents of the window. When a program is closed, the code is erased from the computer's memory.
- **SAVE** – will save the contents of the current window using the window title as the file name. The file will be saved in the same folder as the original version when the file was opened. In other words the new version will over-write the existing version.
  If the program is being saved for the first time, i.e. it is untitled, then you should make sure you give your program a meaningful name because there is every chance that in a matter of weeks you will forget what it is called, so you will have to hunt through your programs folder to see if you recognize the name. For example, if your program calculates the time of sunrise and sunset at any geographical location, then SUNSET.TRU would be an appropriate name. Calling your program MYPROG.TRU or ANYPROG.TRU is not very helpful and will certainly not jog your memory as you glance through your program folder. Clearly this advice becomes more important the greater the number of program files you have saved. It is not unusual for True BASIC programmers to have hundreds, if not thousands, of saved program files on their hard drives, purely because it is so easy to write programs in this language.
- **SAVE AS** – will raise a save file dialog box that will let you specify any name for the file and will allow you to save the file in a folder of your choice. The default file name is the same as the window title, and the default destination folder is the same as the original file when it was first opened.

- **UNSAVE** – is a drastic measure because it will completely delete any file that you specify. You will be asked if you are sure you want to delete the named file. Once you delete a file there is no way to recover it. This is NOT the same as dragging a file to the recycle bin.
- **RENAME** – changes the name of the current window. It does not send a copy of the current source text to a file. If the current source has already been saved to a file, then this existing file will remain unchanged. This corresponds to the RENAME command that executes exactly the same action.
- **PAGE SETUP** – this option presents you with a special dialog box that allows you to specify certain features of any printed output.
- **PRINT** – allows you to select all or a part of your program to be hard copy printed. You select the text by dragging the mouse to highlight the required text. You can also select text using the SHIFT KEY in combination with the DOWN-ARROW key. This procedure uses a high definition print method more suited to proportional fonts. At present this option only prints in COURIER 10 point font.
- **LISTING** - allows you to select all or a part of your program to be hard copy printed. You select the text by dragging the mouse to highlight the required text. You can also select text using the SHIFT KEY in combination with the DOWN-ARROW key. This procedure uses a standard print quality with 80 characters per line and 60 lines per page as default values. These default values can be changed under the SETTINGS menu. It is more suited to fixed pitch fonts such as COURIER and LUCIDA CONSOLE.
- **CHAIN TO….-** allows the user to select an executable file, i.e. with the extension .EXE, and to run this application directly from the editor. When the application is shut down, the editor is re-activated and continues where it left off.
- **CHAIN WINDOWS APP** – allows the user to select an application file, such as a WORD document file (with the extension .DOC) or an Excel spreadsheet file (with the extension .CSV). The editor will run the main application and will automatically load the selected file. Both the Windows application and the editor continue to be active.
- **EXIT** – will close all windows and shut down the True BASIC editor. You will be asked if you wish to save the program as each window is closed if the SAVE ON CLOSE menu option has NOT been selected.

EDIT MENU

- **UNDO** – this option will re-instate the original program text prior to a CUT or PASTE operation. In other words, if you perform a CUT or PASTE action and you decide that you have made a mistake and want to return to the original text before you made the mistake, then using UNDO will achieve this. There are now ten levels of UNDO available to the user - in other words, you can backtrack ten times. The menu is labeled with the operation that can be undone, e.g. UNDO paste. The menu item is labeled "can't UNDO" when you can no longer backtrack any further.

- **REDO** – this option allows you to effectively undo a previous undo operation. In other words you can reinstate the former text after you have done an UNDO operation. As with UNDO, you can REDO repeatedly.

- **CUT** – will copy any highlighted text to the clipboard, and will then erase that portion of text. Portions of text held on the clipboard can be inserted back into your program with the PASTE option. CTRL-X can be used as an alternative way to execute CUT.
- **COPY** – will copy any highlighted text to the clipboard, but will not erase that portion of text from your program. CTRL-C can be used as an alternative. Portions of text held on the clipboard can be inserted back into your program with the PASTE option.
- **PASTE** – will transfer text from the clipboard to the point immediately after the current cursor position in your program. CTRL-V can be used as an alternative. You can position the cursor anywhere in your program by clicking the mouse at that point. The cursor location boxes at the bottom of the editor window indicate the current line and character position. If any text is highlighted, PASTE will replace the highlighted text with textfrom the clipboard.
- **FIND** – will raise the find dialog box that allows you to specify and locate any word, part word or phrase in your program text. The search can be an exact match including upper and lower case, or the search can be independent of case. Normally the search begins at the current cursor position and proceeds to the end of your program. Alternatively you can "wrap" the search to include the whole of your program. The first instance of any text that matches your specification will be highlighted. After a FIND operation the FIND window stays on top, ready to be used again.
- **FIND AGAIN** – is a quick alternative to the FIND dialog box. Once the FIND search has located the first instance of a match, then you may use FIND AGAIN to progressively locate all the other instances.
- **CHANGE** – is similar to FIND except that when a match is found you have the option to replace the match with a specified alternative. You can replace the first instance of a match or you can replace all instances.
- **KEEP** – will retain the highlighted portion of your program and discard the rest. It is a quick way to delete large parts of your program.
- **INCLUDE** – will allow you to specify the name of a program file. The contents of this file will then be inserted in your program at the current cursor position.
- **SELECT ALL** – is a quick way to highlight the whole of your program text rather than dragging the mouse across all the text, which may occupy many pages.
- **MOVE TO** – this is a quick and useful way to place the cursor at a specific line or a specific word in your program. Alternatively you can select the name of a sub-routine from a list of all the sub-routines in your program, and the cursor will move to the start of that routine.
- **WRAP** – this option converts the current window to wrapped text, i.e. long lines are truncated at the edge of the window and are continued on the next line. In the WRAP mode the editor can be used as a general-purpose text reader. CAUTION: None of the options under the RUN menu will work while the window is in WRAP mode. Click the WRAP option again to restore normal programming mode.

RUN MENU

- **RUN** – this option will run the current program, i.e. the program in the front page of the editor window. When the program has finished running the title bar will tell you to click the mouse or press any key. Either action will return you to the editor. If True BASIC encounters any errors, these will be shown as a list. You may select any of the listed errors and the cursor will immediately go to the line and character position where the error occurred. Prior to running your program, the editor adds a few extra lines of code to a copy of your source program (this preserves the original) and it is this copy that is run. These extra lines include adding any loaded libraries and aliases. In the case of MODULES and EXTERNAL program units, no extra code is added and no loaded libraries are added. Remember that the default directory is where the bound program is launched from. CAUTION: if you are using line numbers, make sure to leave plenty of space between your line numbers to allow the editor room for the extra code between your lines. Intervals of 10 are normally sufficient.

- **BREAKPOINT** – will mark your program at the line where the cursor is positioned. BREAKPOINT is normally disabled (grayed out) and only becomes active when you select DEBUG MODE from the SETTINGS menu. The breakpoint will appear as the word <<<BREAKPOINT>>> surrounded angle brackets. This is a toggle action feature, i.e. if the line already has a breakpoint then it will be switched off, but if there is no breakpoint then one will be added. If you RUN your program with breakpoints marked, the program will stop at the first breakpoint. A dialog box will give you the opportunity to continue running your program. All breakpoints are cleared when you toggle DEBUG MODE again. If you insert a series of variable names immediately after the breakpoint, e.g. <<<BREAKPOINT>>>a,n,string$,xyz,b
  then when your program halts at the breakpoint a list of all these variables and their current values will be displayed. In this way you can track the changing values of any variable while the program is running. This is a valuable aid to debugging. CAUTION: if you are using line numbers, make sure to leave plenty of space between your line numbers (10 lines is usually sufficient) to allow the editor to insert extra code between your lines to achieve this breakpoint feature.
- **COMPILE** – will cause your program to be converted into a coded format that the computer understands. Unlike earlier editors, your program will be preserved. The compiled version will be automatically saved with the same file name and in the same folder as your source program, but the extension will be changed to TRC instead of TRU. Prior to the compiling process, the editor adds a few extra lines of code to a copy of your source program (preserving the original) and it is this copy which is compiled. These extra lines include adding any loaded libraries and aliases. In other words, a compiled program will run exactly like a source program. The exceptions to this rule are MODULES and EXTERNAL program units. In these two cases, no extra code or loaded libraries are added. Remember that the default directory is where the TRC program is launched from. CAUTION: if you are using line numbers, make sure to leave plenty (10 is usually sufficient) of space between your line numbers to allow the editor room for the extra code between your lines.
- **BIND** – is a special linking process that combines your program with any library modules and other resources to produce a stand-alone executable application. The default name of this application is the same as your original source code except the extension is changed to EXE instead of TRU. A dialog box allows you to change this name and to specify the folder where the executable file will be saved. NOTE: this feature is NOT available in the Bronze edition so the menu item is grayed out and disabled. Prior to the binding process, the editor adds a few extra lines of code to a copy of your source program (preserving the original) and it is this copy which is bound. These extra lines include adding any loaded libraries and aliases, but DO NOT include the code that retains the output window. In other words, when your program reaches the END statement, the proram will stop and the screen will clear. If you need to retain the output window, you must add the code yourself. For example, immediately before the END statement add the following line:
  CALL TBexitroutine
  This will preserve your last screen until the user presses any key or clicks the mouse.

Remember that the default directory is where the bound program is launched from. CAUTION: if you are using line numbers, make sure to leave plenty (10 is usually sufficient) of space between your line numbers to allow the editor room for the extra code between your lines.

- **TRACE** – is another feature that helps you locate errors in your program by stepping through your program line by line. Essentially TRACE puts a breakpoint on every line. TRACE is normally disabled (grayed out) and only becomes active when you select DEBUG MODE from the SETTINGS menu. All breakpoints are cleared when you toggle DEBUG MODE again.
- **DO** – is a general-purpose command in which you specify and run an EXTERNAL program unit. A file selector dialog box will assist you in locating the DO program of your choice. Note: the program RUNDO.TRU is NOT a DO program and will generate errors if you attempt to run it. Do not move or delete this file because you will no longer be able to run any DO programs.
- **DO FORMAT** – is a built-in routine that indents your program text depending on certain keywords in order to make the text more readable. It also helps you to locate errors because it aligns corresponding statements such as FOR… NEXT and DO….LOOP. If these statements are not perfectly aligned then there must be an error in the code between these statements. You will find that this menu option is one of the most frequently used features in the editor.
- **DO UPPER** – is a built-in routine that will convert the text of any True BASIC program to all upper case (capital letters).
- **DO LOWER** – is a built-in routine that will convert the text of any True BASIC program to all lower case (small letters).

WINDOW MENU

- **RECENT FILES** – this option displays a rolling list of the ten most recently closed files. As you close more files, older files will drop off the bottom of the list.

NOTE: At the bottom of the WINDOW menu there will be a list of all the program files that are currently OPEN. The list shows the full path name of each file. The current program file will be ticked. You may click the mouse on any of these file names to force the file to become the focus of the editor. When you close a program file it is removed from this list.

SETTINGS MENU

- **SAVE ON CLOSE** – this option sets an internal toggle action switch that automatically saves your program when you close the window. When the internal switch is active a tick will appear against this item. Click on this item again to cancel the internal switch and the tick will be erased. The default condition is OFF. The True BASIC editor will try to help you avoid catastrophic mistakes by presenting you with a dialog box that asks if you wish to save your program every time you click on the close window button.
- **BACKUP ON SAVE** – this option allows you to set an internal switch that will automatically produce a back-up copy of any program at the time you save the program. The back-up copy has the same name as the original file except the extension is BAK instead of TRU. When the internal switch is active a tick will appear against this item. Click on this item again to cancel the internal switch and the tick will be erased. This is known as a toggle action switch; click once for ON and click again for OFF. The default condition is ON.
- **DEBUG MODE** – is a toggle action switch that enables the BREAKPOINTS and TRACE options under the RUN menu. When DEBUG MODE is switched ON the item is ticked. When the switch is OFF the tick is erased and your program will run as normal. All breakpoints are removed when DEBUG MODE is switched OFF. The default condition is OFF.

- **CONFIRM QUIT** – is a toggle action switch that causes a dialog box to appear whenever you attempt to shut down True BASIC. The dialog box requires that you confirm your intention to shut down. This option will avoid shutting down when you did not mean to do this. When the confirm switch is active a tick will appear against this item. Click on this item again to cancel the internal switch and the tick will be erased. The default condition is ON.
- **HOTSTART** – is a toggle action switch that causes all the open files that you were using in the previous session with True BASIC to be loaded automatically when you start up True BASIC in the current session. When the hotstart switch is active a tick will appear against this item. Click on this item again to cancel the internal switch and the tick will be erased. The default condition is ON.
- **PREFERENCES** – raises a special dialog box where you can set the color of the editor window, and the name and color of the font used to print the text in the window. The default page color for the editor window is SAND. The default font for all editor windows is ASI MONO, 10 point PLAIN (regular) or COURIER, 10 point PLAIN, and the standard default font color is BLACK. As an alternative LUCIDA CONSOLE 10 point PLAIN can be used as a fixed pitch font. The preferences dialog box also allows you to set the number of characters that will be printed across the hard copy page and the number of lines that will be printed per page. The default settings are 80 and 60 respectively. Code lines longer than 80 characters will be wrapped in the hard copy print.
- **BINDER** – allows you to select which binder you wish to use, i.e. the older version binder that requires DLL files in order for executable programs to run, or the new binder (5.5b19) which does not require DLL files to run executable programs. Note that the new binder has a number of residual bugs that prevent some TrueCtrl objects from working correctly.
- **ALIASES** – this menu option allows you to add or edit the list of alias pathnames used by the editor to locate filenames used in LIBRARY statements. Note that when the file is located in a sub-folder of the directory where the new editor is located, e.g. Tblibs, then only the sub-folder name is used in the alias list. If the file is located in a different directory altogether, then the full path to that directory must be given, e.g. c:\my documents\my pictures. Do not use a trailing backslash.
  Aliases can also be used with the OPEN file statement provided the file already exists, i.e. CREATE OLD is specified. The filename must also be a string literal within quote marks and not a string variable.
  Legacy programs that used curly brackets and an alias group name, e.g. {mygroup} will be handled by the new alias system, even though the group name is ignored. Aliases that are added under the ALIAS menu are permanent, i.e. the editor will remember them so that when you shut down and restart, the aliases will still be in effect. Note that temporary aliases can be added on the command line or by means of a SCRIPT file. Temporary aliases are not remembered when the editor shuts down.
- **FUNCTION KEYS** – is a toggle action switch that enables or disables the function keys. This feature is now enabled in this version.
- **SHORT CUTS** – is a toggle action switch that shows or hides short cut keystrokes against each menu item. The default condition is ON, i.e. short cuts are shown.

- **COLORTEXT** – this feature uses different colors for line numbers (if any), key words, calls and sub-routines, definitions and functions, punctuation, aliases, strings and numeric variables. Two default color schemes are available depending on the background page color (light or dark). The user can also define a custom color scheme. This option allows the user to switch colortext on or off. Note that when colortext is ON then all current open source files will use color text except files that are wrapped. Colortext can be RUN, COMPILED and BOUND in the normal way.

HELP FOR True BASIC

- **HELP** – this option shows a small text window with a drop down index and an edit box that allows you to search the help file. You can resize this window to suit your purpose and it will remain at this size for the remainder of the session while you are working with True BASIC. The help files contain details of all the functions and statements in True BASIC and how to use these features. There are a number of other useful items of information in the help files including extracts from this book. You can select which help file you want to use from the CONTENTS menu. This help file will remain current until you change to another help file.
  The full alphabetical index will be shown when you click on the down arrow button to the right of the topics title. When you select a topic from the index, the text related to the topic will be shown in the main text box.
  If you are uncertain what you are looking for, you can type an associated word or concept in the search box then click on the green GO button. The program will then search the whole text in the current HELP file for a match and will display the results in the text box.
  A unique feature of the HELP option is that you can edit, change or add items to the help file using the EDIT or INSERT options under the HELP WINDOW menu. COPY and PASTE options also allow you to copy code fragments contained in the help text box and transfer these fragments direct to your program
- **FORMS** – this option is grayed out (disabled) in all versions. It is a new option to True BASIC but must be purchased separately. The application automatically enables this menu option to make it fully integrated with the editor. FORMS is not available in any earlier versions. This program allows the user to design window layouts using a simple graphical drag-and-drop interface. Most importantly, FORMS generates the program code to reproduce your design, and includes this code in your own program. You can use FORMS repeatedly to create or modify as many windows as you like. Each window may contain as many controls and objects as you need. The code generated by FORMS is a complete skeleton application that can be run immediately without further intervention by the user. Included in the code are comments to guide you to the point where you need to insert your own program code to respond to user input.
- **TBILT** – this option is grayed out (disabled) in all versions except Gold. It is a regular option to True BASIC but must be purchased separately. The application automatically enables this menu option. This is a free standing program that allows the user to design window layouts using a simple graphical drag-and-drop interface. The editor automatically chains to TBILT. Most importantly, TBILT generates the program code to reproduce your design, and leaves this code on the clipboard for you to paste into your own program.
- **ABOUT True BASIC** – will show you the version number, edition and release date of the version of True BASIC currently running.

- **MANUALS** – will display a selection list containing details of all the manuals available in the DOCS folder. When you select a manual from this list the program will automatically start up an Adobe PDF file containing the selected manual. When you close the Adobe Reader window, control is passed back to the True BASIC editor. You can also add your own manuals to the DOCS folder, provided the manual file itself is in PDF document format, and this manual will be automatically added to the list in the editor.

HELP WINDOW MENU

FILE

- **PRINT** – the current help file topic that appears in the text box will be copied to the hard copy printer.
- **RUN DEMOS** – first select a demo file by highlighting the name (drag the mouse across the name), then select RUN DEMOS. The file will be automatically loaded into editor window ready for you to run.
- **CLOSE** – this option closes the HELP window and returns the user to the main True BASIC editor.

EDIT

- **CUT** – this option is normally disabled (grayed out). It only becomes active when the MODIFY or INSERT options are selected. When active you can copy any highlighted text to the clipboard, and that portion of text will then be erased. Portions of text held on the clipboard can be inserted back into the help file with the PASTE option.
- **COPY** – will copy any highlighted text to the clipboard, but will not erase that portion of text from the help file. Portions of text held on the clipboard can be inserted into your program with the PASTE option on the editor menu.
- **PASTE** – this option is normally disabled (grayed out). It only becomes active when the MODIFY or INSET options are selected. When active you can transfer text from the clipboard to the point immediately after the current cursor position in the help file. You can position the cursor anywhere in the text box by clicking the mouse at that point.
- **MODIFY** – this is a toggle action option that allows you to edit the existing help file text. For example, you may include additional explanatory notes or more examples to the existing topic, or you may correct mistakes if you find any. The options CUT and PASTE also become active. First select the topic you wish to modify from the drop-down topics list, then click the MODIFY option. When you have completed your changes select the MODIFY option again. This will erase the active tick mark and will disable CUT and PASTE. At this point you will be given the option to SAVE your modified topic or DISCARD it. You must exit the HELP window for your modified topic to appear in the drop down list.
- **ADD NEW** – this is a toggle action option that allows you to add extra topics to the help file. First select the ADDNEW option to clear the text window ready for you to type in your topic. You must begin your topic with a title inside

angle brackets, e.g. <TITLE> and this will ensure that your topic will then appear in the alphabetical drop down list. Your topic can be of any length. When you have finished, click the ADD NEW option again. At this point you will be given the option to SAVE your new topic or DISCARD it. The ADD NEW toggle option will then be turned OFF and CUT and PASTE will be disabled. You must exit the HELP window for your new topic to appear in the drop down list.

- **IMPORT** – is an alternative to ADD NEW. It allows you to import additional help information that has been saved in an external file. In this instance multiple help topics can be inserted in one operation. Each imported topic must begin with a title in angle brackets. The import file can contain any number of topics. A dialog box will request the name of the file and its entire contents will be appended to the current help file. This is a very simple way to update your help file using files generated by others, e.g. True BASIC Forum Members or by True BASIC Inc. You must exit the HELP window for your imported topics to appear in the drop down list.

CONTENTS

Selecting any one of the following items determines which help file the editor will use. There are currently eight different help files that cover various aspects and library modules included with True BASIC. In turn this determines the list of topics that you can select from the drop-down list.

- **USING THE EDITOR** – is a series of topics related to using the editor and the help feature. The topics are arranged alphabetically. This item is common to all editions of True BASIC.
- **FUNCTIONS** – this section lists and explains all the built-in functions within True BASIC and again it is common to all editions. Most topics contain code that can be copied to your programs.
- **STATEMENTS** – this section lists and explains all the statements in True BASIC. Most topics contain code that can be copied to your programs. This section is common to all editions of True BASIC.
- **TRUECTRL** – this section details all the sub-routines in the library module and explains the syntax and how to use each routine with code examples that can be copied directly to your programs. This option is not available to users of the Bronze edition. Instead, BronzeTC is included.
- **TRUEDIAL** – this section details all the sub-routines in the dialog box library module and explains the syntax and how to use each routine with code examples that can be copied directly to your programs. This option is not available to users of the Bronze edition.
- **TRUECTX** – this section details all the sub-routines in the extended color and text library module and explains the syntax and how to use each routine with code examples that can be copied directly to your programs. This option is not available to users of the Bronze edition.
- **TRUETDX** – this section details all the sub-routines in the extended dialog box library module and explains the syntax and how to use each routine with code

examples that can be copied directly to your programs. This option is not available to users of the Bronze edition.

- **FORMS** – this section describes how to use the FORMS program to create windows and objects and automatically generate code. This option is not available to users of the Bronze edition.

Note: The editor automatically reads all TXT files that reside in the TBhelp folder and creates the CONTENTS list from these files. To add another help file to this list, all you have to do is drop the file into the TBhelp folder and the editor will do the rest.

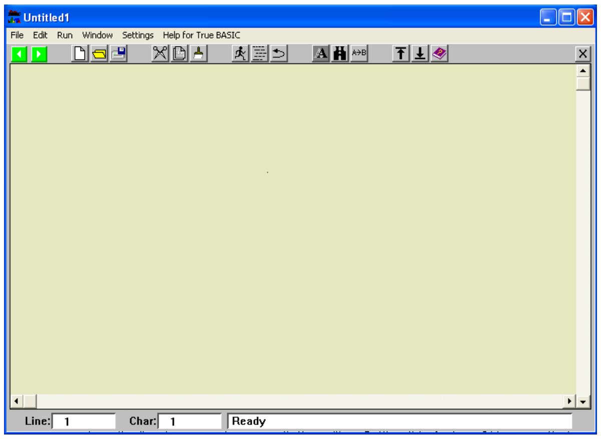
If you wish to add more help files you may use Notepad or the TB Editor to create additional menu items.

CHAPTER

4

Writing and Running Your First Program

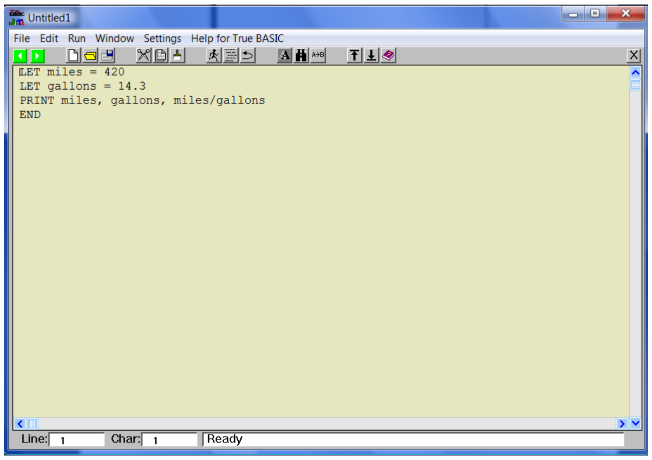
Start True BASIC, if you haven't already, as described in the preceding chapter. This time, instead of using an existing program, you'll create your own in the editing window. If you've just started True BASIC BRONZE Edition and chosen "New", you'll have a blank editing window called "Untitled 1" because 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.

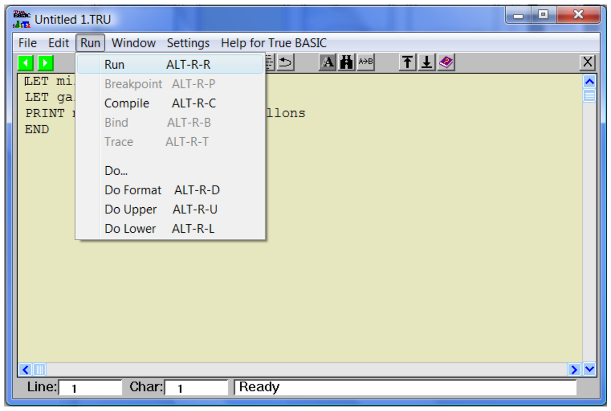
```
LET miles = 420
LET gallons = 14.3
PRINT miles, gallons, miles/gallons
END
```



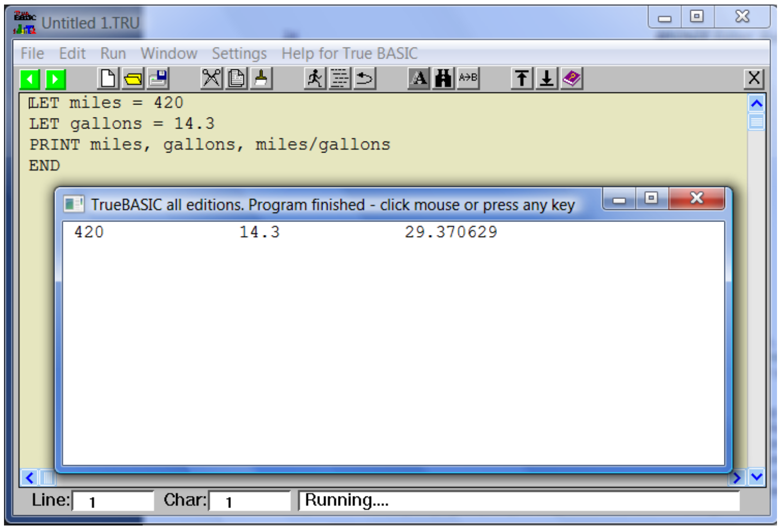
```
LET miles = 420
LET gallons = 14.3
PRINT miles, gallons, miles/gallons
END
```

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”:


Untitled 1.TRU
File Edit Run Window Settings Help for True BASIC
TrueBASIC all editions. Program finished - click mouse or press any key
Line: 1
Char: 1
Running....

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

```
420              14.3             29.370629
```

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:

```
    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.

CHAPTER

Modifying and Saving Programs 5

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 Detete (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

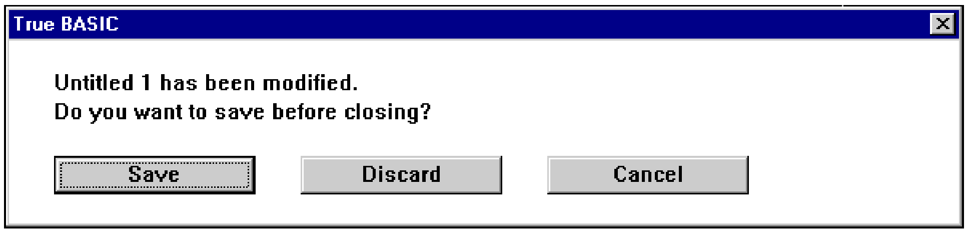
You've now made additional changes to the MPG program since you last saved it. What if you want to save these additions but you also want to keep the version as it was when you last saved it? In other words, you want two versions of the program — one with the data supplied by LET statements and one that 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



Untitled 1 has been modified.
Do you want to save before closing?

CHAPTER

Constants, Variables and Expressions 6

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:

Number Constants

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

Variables

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

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

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

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

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

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

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

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

Variable Names

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

Expressions and Formulas

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

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

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

For example:

```
miles/gallons
```

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

True BASIC does not notice spaces in expressions. For example, "a+b" means the same thing as "a + b", and "miles/gallons" is equivalent to "miles / gallons". Remember, however, that variable names cannot contain spaces.

Notice the symbols for multiplication and division. Computer keyboards don't usually contain the ÷ 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:

```
miles/gallons
```

not

```
miles ÷ gallons
```

or

```
 miles
-------
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+10/2
```

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

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

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

```
12/6*2
```

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

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

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

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:

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

✓ **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 to 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:

```
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:

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

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

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

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

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

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

Add the new PRINT statement 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.)

CHAPTER

7

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:

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

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

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

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

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

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

Your results should look something like this:

```
    How many miles? 312
    How many gallons? 8
    MilesGallonsMiles per gallon
    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.

```
    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:

```
    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 tl 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:

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

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

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

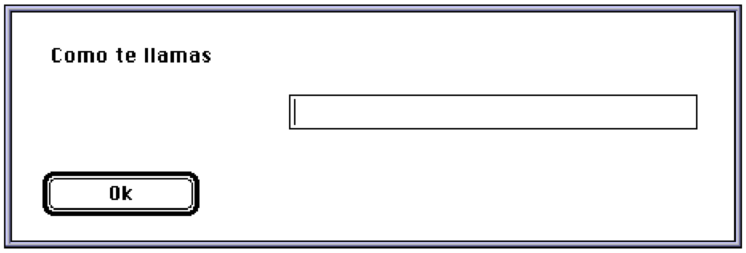
You can even enter no reply to a LINE INPUT by just pressing the Return key. (If you just press Return with an INPUT statement, True BASIC complains that you did not give enough input.)

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.

```
!  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:


Como te llamas

You can then type your name into the small box.

CHAPTER

Loop Structures 8

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

How a FOR-NEXT Loop Works

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

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

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

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

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

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

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

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

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

```
! 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:

```
Number          Square Root

 0               0
 .1              .31622777
 .2              .4472136
 .3              .547722256
 .4              .63245553
 .5              .70710678
 .6              .77459667
 .7              .83666003
 .8              .89442719
 .9              .9486833
 1.              1
```

This program uses the built-in function SQR to obtain the square root of *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:

```
! 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
```

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

```
Number          Square Root

 10              3.1622777
 9               3
 8               2.8284271
 7               2.6457513
 6               2.4494897
 5               2.236068
```

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

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

and run it again:

```
Number          Square Root

 10              3.1622777
 9               3
 8               2.8284271
 7               2.6457513
 6               2.4494897
 5               2.236068
 4
```

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

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

Nested Loops

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

As an illustration, open the demo program EXES:

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

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

    PRINT
NEXT row
END
```

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

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

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

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

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

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

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

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

    FOR triangle = 1 to 3          ! new loop starts here

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

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

    PRINT
NEXT row
END
```

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

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

An Introduction to Conditions

In the FOR-NEXT loop, you must specify how many times you want the loop to repeat. Computers, however, are quite capable of making decisions based on 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**:

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

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

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

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

```
! 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
```

Run the program:

```
    0             10000
    1             10550
    2             11130.25
    3             11742.414
    4             12388.247
    5             13069.6
    6             13788.428
    7             14546.792
    8             15346.865
    9             16190.943
    10            17081.445
    11            18020.924
    12            19012.075
    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.

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* > = 2 * *original*) is still met.

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

```
In 9 years, you'll have $ 20838.557
```

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
```

CHAPTER

Decision Structures 9

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

Simple IF-THEN Decisions

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

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

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

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

END
```

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

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

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

Single-line IF-THEN-ELSE Decisions

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

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

END
```

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

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

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

Multiple-Line Decisions

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

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

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

END
```

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

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

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

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

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

More About Counters

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

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

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

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

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

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

END
```

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

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

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

The RANDOMIZE Statement

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

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

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

FOR toss = 1 to 5
   IF Rnd<.5 then
      PRINT "Heads, you win"
      LET heads = heads + 1            ! Count heads
   ELSE
      PRINT "Tails, you lose"
      LET tails = tails + 1            ! Count tails
   END IF
NEXT toss

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

END
```

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

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

The STOP Statement

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

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

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

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

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

FOR chance = 1 TO 3
   PRINT "Enter your guess";    ! Ask for number
   INPUT guess! Get a guess

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

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

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

```
! Craps game.
!
RANDOMIZE

FOR game = 1 to 10              ! Play 10 games

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

   PRINT dice;                  ! Print this roll

   SELECT CASE dice             ! Branch on roll

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

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

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

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

NEXT game

END
```

CHAPTER

Formatting and Printing Your Program 10

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.

```
! 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.)

```
! 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
```

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 than 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 chossing **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.

CHAPTER
11

Editing Hints and Shortcuts

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.

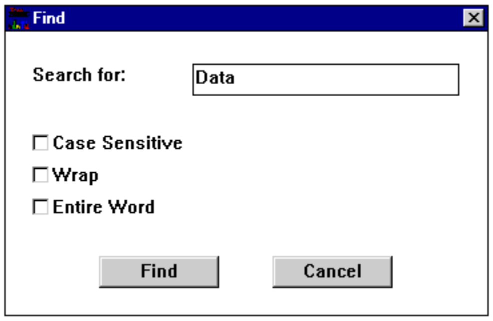
Find and Change

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
```



Search for: Data

To find the next occurrence of the word *data*, choose **Find Again** in the **Edit** menu. True BASIC will select the next occurrence of the word *data*, which is the first DATA statement.

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

```
dat
```

and press the RETURN key or click Find. True BASIC will select the next occurrence of *dat*, which is the first part of the next occurrence of the word *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 **Find** one more time. This time look for the word:

```
read
```

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

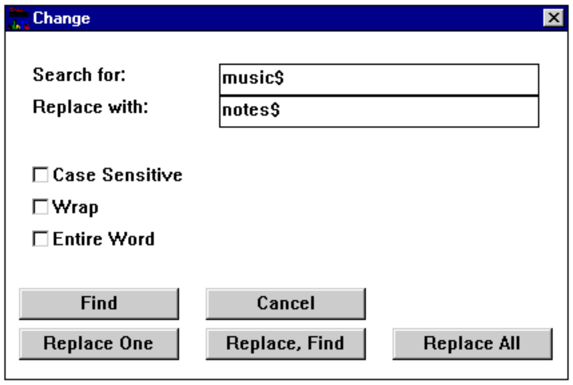
in the message line at the bottom of the Editing Window.

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.

√ **True BASIC always searches from the insertion point to the end of the program, and then stops, unless Wrap has been selected.**

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.


Change

Search for: music$
Replace with: notes$

☐ Case Sensitive
☐ Wrap
☐ Entire Word

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.

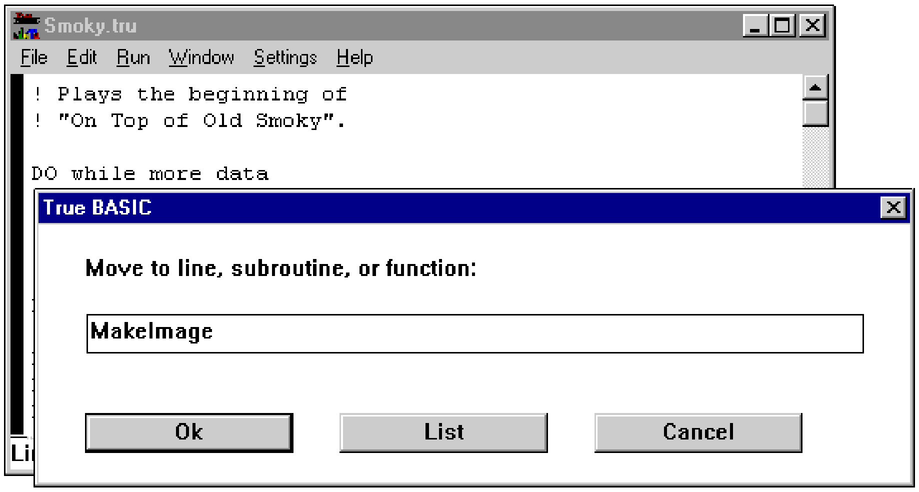
The **Include** command lets you add the contents of another file to your program. Put the insertion point at the place where you wish to add the new file and select **Include** from the **Edit** menu. You'll get a dialog box where you can specify any existing file in any directory on any disk. True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



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

DO while more data
```

Move to line, subroutine, or function:

MakeImage

CHAPTER
12

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.

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

FOR i = 1 to num_quest          ! Read all questions

    READ question$, answer$

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

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

NEXT i
```

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

    DATA 5

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

    END
```

The first executable statement after the initial comment lines is a READ statement. This "reads" the first item in the first DATA statement and assigns that value to the variable *num_quest*. The value of *num_quest* determines how many times the program goes through the FOR-NEXT loop.

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

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

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

√ **DATA statements may be placed anywhere in your program.**

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

✓ **READ and DATA statements can use either numbers or strings.**

You may freely mix strings and numbers in your DATA statements. Just be sure that the variable name type (numeric or string) is reading an appropriate type of data item. You can't read a string data item into a numeric variable, but you can read a number into a string variable. The TRIVIA program reads some numbers for the string variable *answer$*. This is perfectly legal in True BASIC, as long as you don't try to use that variable to do arithmetic calculations.

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

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

Checking for More Data

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

It is not always convenient to count the number of DATA statement 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.

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

DO while more data
```

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

   LOOP

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

   END
```

The DO WHILE MORE DATA statement means "keep looping while there are more data items to read". This is why the program still worked even when you copied and pasted an extra set of the DATA statements.

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

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

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

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

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

Reusing Data Values

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

✓ **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.

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

END
```

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

```
! 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."            ! And say bravo
   ELSE
      PRINT "No, the correct answer is "; answer$; "."
   END IF
```

```
LOOP until end #1

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

CLOSE #1

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.

```
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:

```
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 $"; lastdep_amt
   PRINT
   PRINT "Input all checks and deposits since "; bal_date$
ELSE
   PRINT "Input all checks and deposits."
END IF
!
```

```
! Get new transactions
!
PRINT "Enter one per line: use - for checks, + for deposits"
PRINT "Enter 0 (zero) when done"
!
DO
! Get new transactions
   INPUT amount
   LET curbal = curbal + amount   ! Update balance
   IF amount < 0 then
      LET lastcheck_amt = amount*(-1)
   ELSE IF amount > 0 then
      LET lastdep_amt = amount
   END IF
LOOP until amount = 0
!
LINE INPUT PROMPT "Date of last transaction: ": bal_date$
PRINT "Your current balance is $"; curbal
!
! Clear data file and enter new amounts
ERASE #1
! Remove any existing data
PRINT #1: bal_date$
PRINT #1: curbal; ","; lastcheck_amt; ","; lastdep_amt
CLOSE #1

END
```

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

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

```
July 4, 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: 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.

CHAPTER 13

Arrays and Matrices

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

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

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

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

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

One-Dimensional Arrays

This problem calls for array variables. An **array** is a variable that can hold several different values at once. You could think of 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 **INVNTORY** program to use two arrays, *item$* and *number* as shown below:

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

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

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

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

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

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

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.

item$

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

number

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

Figure 13.1 – Items in Arrays

Array Subscripts

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

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

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

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

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

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

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

The program will print the items in reverse order:

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

Array Bounds

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

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

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

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

INPUT PROMPT "What year are you interested in? ": year
IF springfield(year) > woodsville(year) then
   LET town$ = "Springfield"
ELSE
   LET town$ = "Woodsville"
END IF
PRINT "In"; year; town$; " had the largest population."

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

A sample run produces output such as:

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

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

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

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

Arrays of Two or More Dimensions

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

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

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

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

FOR i = 1 TO 10               ! Ask 10 questions
    LET n = Int(50*Rnd) + 1   ! Pick a number between 1 and 50
    PRINT "The capital of "; state$(n,1); " is";
    LINE INPUT capital$       ! Get the reply
    IF Lcase$(capital$) = Lcase$(state$(n,2)) THEN
       PRINT "RIGHT!"
    ELSE
       PRINT "Nope, it's "; state$(n,2); "."
    END IF
NEXT i
```

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

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

state$

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

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!

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

MAT READ item$, number

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

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

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

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

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

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

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

MAT READ item$, number

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

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

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

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

 4            2            1             4              2
```

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

CHAPTER 14

Functions and Subroutines

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

Subroutines

Call up the demo program CRAPS, which introduced the SELECT CASE structure 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."
      CASE ELSE                 ! Anything else
         LET POINT = dice       ! Remember that roll
         DO
            LET die1 = Int(6*Rnd + 1) ! Roll again
            LET die2 = Int(6*Rnd + 1) ! Both dice
            LET dice = die1 + die2
            PRINT dice;         ! Print this roll
         LOOP until dice = 7 or dice = point

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

NEXT game

END
```

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

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

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

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

   CALL Rolldice                ! Subroutine rolls dice

   SELECT CASE dice             ! Branch on roll

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

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

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

NEXT game

END
```

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

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

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

Subroutines with Parameters

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

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

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

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

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

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

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

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

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

This subroutine illustrates two kinds of parameters:

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

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

• 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, *answer* acquires the value 3.1622777, which is returned by the function SQR.

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

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

- functions can return only one value and
- functions cannot change the values of any parameters sent to them.

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

One-line Functions

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

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

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

FOR game = 1 to 10              ! Play 10 games

   LET dice = Rolldie + Rolldie ! Rolldie function twice

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

NEXT game

END
```

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

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

If you don’t define it first, True BASIC won’t know that you are referring to a function and not a variable 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:

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

INPUT n                     ! Input a number
PRINT Sgn(n)                ! Print its sign
PRINT Sgn(3-5*2)            ! And the sign of this formula
END
```

If you run this program and give 35 as input, you will see the following results:

```
? 35
1
-1
```

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

Global Variables

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

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

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

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

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

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

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

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

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

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

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

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

Bugs of this sort are very typical when you use **global** 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.

```
! Using an external function

DECLARE DEF XXX$

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

! XXX$ -- returns n x's

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

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

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

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

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

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

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

The LOCAL Statement

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

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

Now, XXX$ can be an internal function and you could safely use the variable names *i* and *s$* in the main program. Those variables 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.

CHAPTER
15

Libraries

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

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

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:

```
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
```

The above program uses the subroutine RollDice but doesn't use the function to flip a coin; you don't have to use everything in the library. But, you can expand CRAPS so that it flips a coin to decide which of two players goes first. Notice that you must use a

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

```
! Craps game.
!
LIBRARY "gameslib.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

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

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

NEXT game

END
```

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

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

Appendix C in this manual lists most of True BASIC's built-in functions. 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

They can also be compiled files. It doesn't make any difference. Source files usually have the extension ".tru" in their name, 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.

CHAPTER 16

Graphics

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

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.

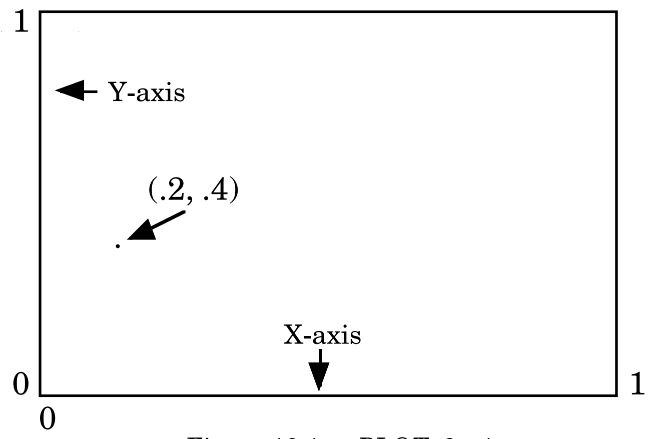


Figure 16.1 – PLOT .2, .4

A simple True BASIC program to draw this point on your screen has just two lines:

```
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.

```
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):

```
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:

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

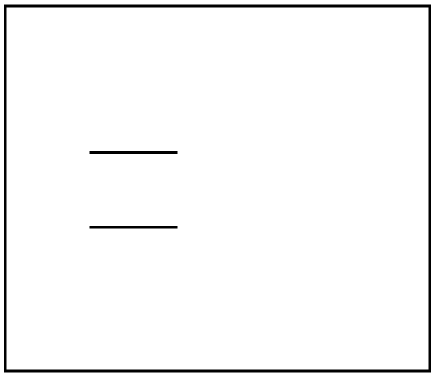


Figure 16.2 - Horizontal Lines

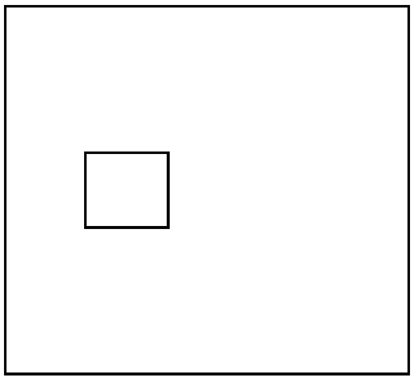


Figure 16.3 - A Box

Changing the Coordinates

As you saw above, True BASIC assumes the output coordinates go from 0 to 1 in both the horizontal and vertical directions. However, you can use a SET WINDOW statement to set any boundaries you want.

For example, if you want the coordinates to go from 0 to 10 in both directions, you could include the following statement before you give any PLOT statements:

```
SET WINDOW 0, 10, 0, 10
```

The first two numbers give the start and end values for the horizontal axis, the second numbers give the start and end for the vertical axis.

Your coordinate system need not begin at zero, and the horizontal and vertical axes need not match. For example if you were plotting a graph to show production of cars over this century, you might set your coordinates as follows:

```
SET WINDOW 1900, 1990, 0, 10000000
```

The horizontal axis would show the range of years, and the vertical axis would let you plot production amounts from 0 to 10,000,000.

You can change the coordinates within a program. All PLOT statements use the coordinate system specified in the most recent SET WINDOW statement.

Drawing Shapes

True BASIC gives you two ways to draw empty or solid shapes. The BOX statements are the easiest and fastest method.

The BOX LINES statement draws the outline of a square or rectangle. You give the coordinates of the left, right, bottom, and top edges in the same way as in the SET WINDOW statement. The following program outlines a square as shown in Figure 16.4.

```
! Draw a square
!
SET WINDOW 0, 30, 0, 20     ! 15,10 is center of window
BOX LINES 10, 20, 5, 15     ! Draw box with sides = 10
END
```

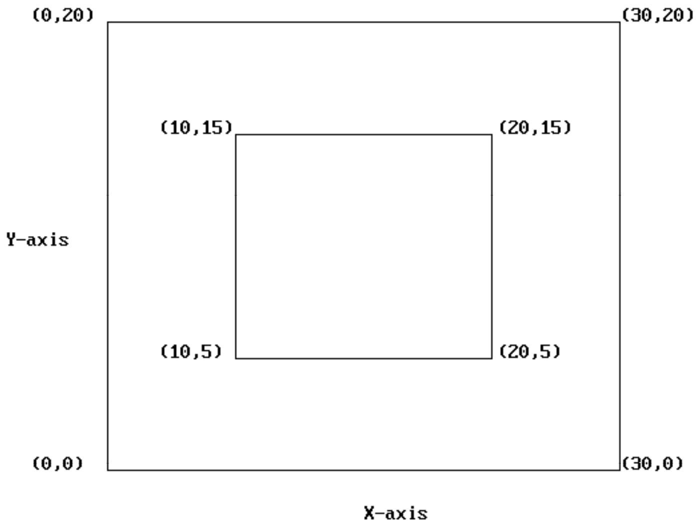


Figure 16.4 - BOX LINES 10, 20, 5, 15

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

```
! Draw a solid square
!
SET WINDOW -15, 15, -10, 10     ! 0,0 is center of window
BOX AREA -5, 5, -5, 5           ! 5*2 is length of each side
END
```

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
```

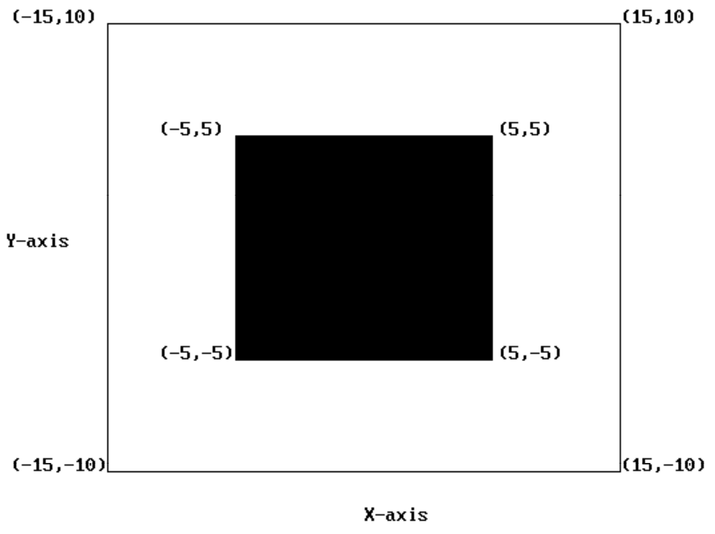


Figure 16.5 - BOX AREA -5, 5, -5, 5

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.

```
    ! Draw a knight
    !
    PLOT .2,.1;.8,.1;.8,.2;              ! Draw the outline
    PLOT .7,.25;.7,.3;.8,.4;.65,.7;.6,.9;.55,.9;
    PLOT .5,.82;.2,.75;.2,.6;.3,.6;.4,.55;
    PLOT .25,.45;.2,.37;.3,.3;.3,.25;.2,.2;.2,.1
    FLOOD .5,.5                          ! Fill it in
    END
```

The **PLOT AREA statement** connects a series of points **and** fills in the object. It works much as a series of PLOT statements except that PLOT AREA always connects the last point to the first. So you need not repeat the first point. The following statements draw and fill a triangle (Figure 16.8). Note that the PLOT AREA statement has a colon after the AREA keyword.

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

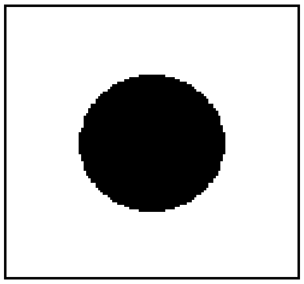


Figure 16.6
BOX CIRCLE and FLOOD

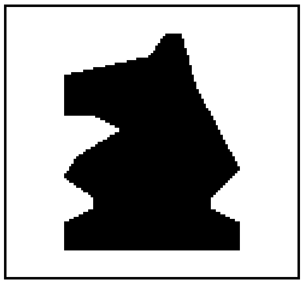


Figure 16.7
PLOT and FLOOD

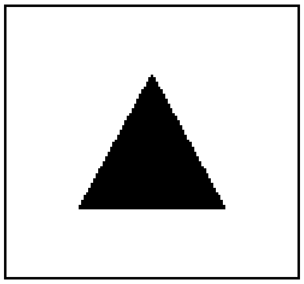


Figure 16.8
PLOT AREA

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:

```
    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.

Name	**Number**	**Meaning**
black	0	black
blue	1	blue
green	2	green
cyan	3	cyan
red	4	red
magenta	5	magenta
brown	6	brown (yellow on some monitors)
white	7	white
	8	gray
	9	bright blue
	10	bright green
	11	bright cyan
	12	bright red
	13	bright magenta
yellow	14	yellow (brown on some monitors)
	15	bright white

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

```
! Draw six squares
!
SET WINDOW -10, 10, -10, 10
BOX AREA -6, 6, -6, 6          ! Draw outer square in black
FOR i = 5 to 1 step -1         ! From large to small
    SET COLOR i! Change color
    BOX AREA -i, i, -i, i      ! Draw next square
NEXT i
END
```

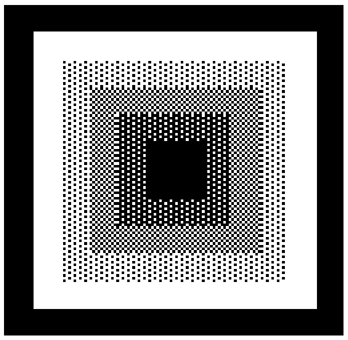


Figure 16.9 – Six squares.

If you have a color monitor, you can use the nine True BASIC color names (listed in the table above). If your computer can produce more colors, you can use color numbers and the SET COLOR MIX statement for greater variety. The color numbers you can use depend on the 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.

DRAW Knight

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;
```

```
       PLOT .25,.45;.2,.37;.3,.3;.3,.25;.2,.2;.2,.1
       FLOOD .5,.5                  ! Fill it in
    END PICTURE

    DRAW Knight

    END
```

Like subroutines and functions, pictures may be internal or external. External pictures may be stored in Library files.

Transformations

So far there doesn't seem to be any great benefit to defining a picture. The true power of pictures comes when you use them with transformations and parameters. **Transformations** let you move pictures or rotate, re-scale, or tilt them when you draw them. For example, you could replace the DRAW statement above with the following lines to draw lots of knights all over the screen.

```
SET WINDOW 0, 10, 0, 10
FOR x = 0 to 9
    FOR y = 0 to 9
        DRAW Knight with shift(x,y)
    NEXT y
NEXT x
```

The SHIFT transformation moves horizontal and vertical coordinates by the amounts you specify. The above statements use a larger coordinate system (SET WINDOW 0, 10, 0, 10) and then draw the knight 100 times within that window. Try it!

Similarly, you can double the size of the knight:

```
DRAW Knight with scale (2,2)
```

or make it twice as tall as wide:

```
DRAW Knight with scale (2,4)
```

SCALE multiplies the horizontal and vertical coordinates of your picture by the amounts you specify. Be aware that your scaled picture may become bigger than the window coordinates! Use a SET WINDOW statement to give enlarged coordinates if necessary.

Other transformations let you "shear" (or tilt) the picture or rotate the picture. You must give the amount of tilt or rotation in radians unless you include an OPTION ANGLE DEGREES statement first. You may then use degrees.

The SHEAR transformation leans vertical lines forward (clockwise) by the angle you specify. For example,

```
    OPTION ANGLE DEGREES                ! Use degrees
    DRAW Knight with shear (45)
```

makes the knight lean to the right by 45 degrees. Use a negative angle to lean a picture to the left. As with SCALE, you may have to use a SET WINDOW statement so that the picture doesn't lean out of the window.

ROTATE moves pictures counterclockwise (clockwise if you use a negative angle) around the (0,0) point in the window. Note that this is not the same as rotating a picture in place! You can easily rotate a picture out of coordinate window, unless you adjust coordinates with SET WINDOW or also shift the picture.

For example, if you rotate the knight 90 degrees, it would "fall on its face to the left" and be out of the standard coordinate system (0, 1, 0, 1). The upper right box of Figure 16.10 shows the knight drawn in the standard coordinate system with no transformations. The gray knight was rotated with the statements:

```
    OPTION ANGLE DEGREES                ! Use degrees
    DRAW Knight with rotate (90)
```

True BASIC rotates the knight about the point (0,0) and out of the standard coordinate window.

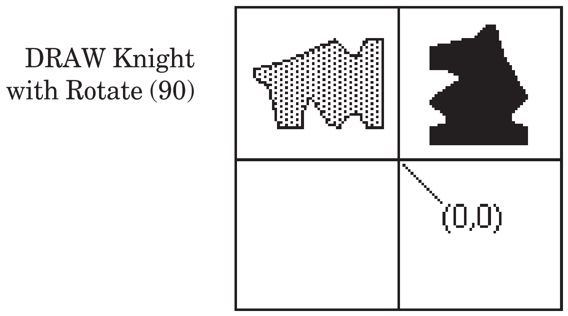


Figure 16.10 – RotateTransformation

You can **combine transformations** on one DRAW statement by placing an asterisk (*) between transformations. For example, you could rotate the knight and then move it back into the (0, 1, 0, 1) window:

```
OPTION ANGLE DEGREES                ! Use degrees
DRAW Knight with rotate (90) * shift (1,0)
```

When you use more than one transformation, True BASIC performs them in order from left to right. Because of this, the order of transformations can make a difference. You're most likely to get the results you expect if you use SHIFT as the last transformation.

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.

```
    LIBRARY "GraphLib.tru"
    SET WINDOW 0, 10, 0, 10
    CALL Ticks(1,1)
    END
```

Other Graphics Features

As you become more proficient, you might want to use some of True BASIC's other graphics statements. Several of these are 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 then 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.

Programming for Windows

In other parts of this manual you have been shown how to write *'top-down'* programs and routines. In other words the programs start at the beginning and finish at the end, and how the user proceeds through this sequence of events is controlled entirely by the person who wrote the program.

Programming for Windows requires a completely different approach because a window can contain many objects; such as menus, push buttons, edit fields, lists etc., and the user has complete freedom to activate any of these objects at any time. You, as the programmer must accept that you no longer control the process. Your job is to write a program that takes account of whatever the user wants to do in whatever order they elect to do it.

This is a profound difference in the way you think about and approach writing programs so make sure you understand this fundamental concept before you try to work with windows.

In TrueBASIC this is not as difficult as it sounds. Firstly because a special library module called BronzeTC has been constructed that makes it very easy to create windows and objects. In general, just one single line of code will produce a whole window or an object such as a list or push button.

At the start of all your windows programs you need to access this library with:

```
LIBRARY “BronzeTC.trc”
```

You also need to initialise this library with:

```
CALL TC_init
```

It is also good practice to restore everything to normal when you exit your program with:

```
CALL TC_cleanup
END
```

Secondly, this library contains a sub-routine called TC_event that solves the problem of finding out what actions the user has made with the mouse or the keyboard. Indeed, you will find that calling this sub-routine from inside a DO....LOOP structure in your programs makes programs much easier and quicker to write, because you can use the same *'skeleton'* program every time. The folder TBdemos contains such a *'skeleton'* program called TBstandard.TRU.

The skeleton program has been set out to allow for a menu, a push button, an edit field, a list button and both window scroll bars. You will see that the program consists of a simple event loop in which the type of event is analysed and each type calls an appropriate sub-routine to deal with the event. As an illustration the ID of a push button is call *mybutton*, but you can use any meaningful name. For example if the button quits the program then it would have a label "QUIT" so it would be convenient to use the ID name *quit*. Similarly the edit field ID has been called *myedit*, but if your edit field asks for a surname then it would be more sensible to call the edit ID *surname*. This rule applies to all object ID names.

Every event must have an appropriate response from the program, even if the response is to ignore the event. Usually the way to deal with an event is to call a sub-routine. This keeps everything nice and tidy because each routine only does one simple task – it deals with just one event. Collectively your program may be quite complex, but individual routines will be extremely simple. Think of the main program as being an index of events. You look up the index for an event and it points to a sub-routine where you will find the detail. This strategy will make trouble shooting and de-bugging so much easier. Even the task of creating objects to fill your window is best done inside a sub-routine.

In top-down programming you always control what the user does next, but in programming for windows you have to accept that the user may do absolutely anything, however bizarre or stupid, so you need to prepare for this. For example, suppose you have a program that essentially collects data for an address/phone book. In the window you will probably have several edit fields for name, address, phone number etc and a SUBMIT button, and maybe a CANCEL button. You are expecting the user to complete all the edit fields and then press the SUBMIT button, so your response to the SUBMIT button would be to file the data entered by the user.

In practice, the user may well press the submit button without completing all the edit fields, so you need to check whether this has been done before you write some incomplete data to your file. Similarly, you should also check whether the data that the user has entered is reasonable. If you want the user to enter a phone number then you need to check that what they have typed conforms to the pattern of numbers you normally expect for phone numbers. The library module Bronze_TD provides you with a range of dialog boxes that you can display to inform the user that they have done something wrong.

In many ways working with windows is a lot easier that writing ordinary programs simply because the code that generates objects and controls how they work has already been done for you. In normal programming a lot of your time is spent writing code that gets input information and even more code to display information on the screen. Windows objects do most of this for you, so you can spend more time and effort on what your program really does.

The next concept that you need to understand before you go any further is the difference between physical and logical windows. Physical windows are those that usually have a border, a title bar and buttons for closing and re-sizing the window. Logical windows are zones or areas within a physical window. Logical windows have no borders or title bars nor any control buttons. Every time you create a physical window, a logical window is also created to fill that window. Why have two windows? The reason is very simple; physical windows are where you put windows objects such as push buttons, edit fields and list buttons, and logical windows are where you use all the other TrueBASIC features such as PRINT, PLOT and image handing statements such as BOX SHOW.

It will help you to visualize how things will appear on the monitor screen if you think of the physical window as being on top. For example, if you color an area of the logical window with a BOX AREA statement, then you place a push button in the physical window, the button will appear on top of the colored area. If you change the colored area with another BOX AREA statement, then the button will still be on top.

The size and location of all windows are defined by four co-ordinates in this order;
left, right, bottom, top

The co-ordinates for windows are always relative to the **whole screen.**
The co-ordinates for objects inside windows are always relative to the **window**.

The final concept that you need to get your head around is that all windows and all objects inside windows each have a unique identity number. The reason why we need identities is again very simple. Suppose you have several push buttons labelled QUIT, CANCEL and SUBMIT, then you will want to know which of them the user has clicked on. The TC_event routine will tell you that a button has been pressed and it also gives the ID number of the button. If you have created multiple windows, then TC_event will also tell you the ID of the window the user is currently working with. The most difficult concept to grasp is that you don't need to know the value of the ID number, because you can use a variable name instead. You are free to use any variable name you like. Suppose you have a button that is labelled QUIT, then you might use the variable name *quit* as the ID for this button.

Remember, that there is nothing magical or mystical about windows and the objects inside them. They are just graphical images painted on the screen. However, behind the scenes, there is a great deal of code that makes the object appear to react in a particular way when you click the mouse inside the perimeter of the object. Take for example a simple push button, which you create with just one very simple line of code:

```
        CALL TC_PushBtn_create(id,"LABEL",xl,xr,yb,yt)
```

You don't need to worry about all the code that paints a rectangular area of the screen gray, and works out where the center is so that the word LABEL is printed in the middle in black. Nor do you have to concern yourself with working out how to color the edges of the rectangle to give the button a 3D effect of standing proud of the background using shadows on the bottom and right hand edges. When you click the button these shadows are reversed momentarily to give the impression the button has been pressed. All this code has been done for you in the library module BronzeTC.

Until now the output from your programs has been displayed on the screen in the default font, over which you had no control. BronzeTC now gives you control over the font, not just in the default window, but all windows, e.g.

```
CALL TC_win_SetFont(wid,”Times”,12,”Bold italic”)
```

Where `wid` is the ID of the window you wish the font to apply to.

You may change the font details any number of times, so it is possible to fill the screen with a variety of fonts in different sizes and styles, and in different colors, e.g.

```
CALL TC_win_SetFont(wid,”Times”,12,”Bold italic”)
Set Color 9 ! blue
PLOT TEXT, AT 20,50:”This is an example”
CALL TC_win_SetFont(wid,”Arial”,16,”Bold”)
Set Color 12 ! red
PLOT TEXT, AT 20,100:”of different fonts”
CALL TC_win_SetFont(wid,”Helvetica”,10,”Plain”)
Set Color 13 ! magenta
PLOT TEXT, AT 20,150:”in various colors”
```

Much of your programming effort will have been spent devising ways of getting user INPUT, and then even more time working out how to display the output data. The tools available in BronzeTC will make these tasks much easier and will give greater impact to your output.

Remember that all the code you use with BronzeTC will run without any changes if you later upgrade to the TrueCtrl and TrueDial library modules. This is because BronzeTC is a cut down version of the TrueCtrl library. Similarly BronzeTD is a reduced set of dialog boxes from the TrueDial library.

Demonstration programs featuring the objects in BronzeTC can be found in the Tbdemos folder.

To use these features correctly you will need to refer to the BronzeTC manual which is located in the documents folder. This manual also contains details of BronzeTD dialog boxes.

CHAPTER 17

Sound and Music

You've already seen the Demo Program SMOKY that plays the first few lines of "On Top of Old Smoky". True BASIC's PLAY and SOUND statements let you produce melodies and general sound effects on your computer.

The PLAY Statement

The PLAY statement lets you play simple melodies on your computer. When you use a PLAY statement, you give it a string consisting of codes for notes, tempo, and how the notes should be played.

Open the SMOKY program, run it again, and then take a look at the music codes in the DATA statements.

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

DO while more data

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

LOOP


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

END
```

Look at the first DATA statement, which represents the first six notes of "On Top of Old Smoky." The letters A through G represent the notes A through G. The other codes give True BASIC information about how to play the sequence of notes.

The letter **O followed by a digit** sets the current **octave**. The octaves start at C and go up to B, as on a piano keyboard. (Middle C is the first note in octave 5.) This song begins in the fourth octave, so the first string item is "O4".

Next, the letter **L followed by a digit** tells True BASIC the **length of the note or notes** to play. The larger the number with the code L, the shorter the length of the note. Therefore, "L4" means a quarter note, "L2" a half note, and "L1" a whole note. True BASIC plays all notes following an L code at that length until another L appears in the string expression.

After the first DATA statement sets the octave and the length of notes, "C C E G" tells True BASIC to play two C's, an E, and a G as quarter notes. The next note, however, is in the next octave, so you need another O code to set the octave to O5.

After O5, the next note is a 3/4 note C. This is done by changing the length to L2 (half note) and adding a dot after the letter C. The **dot multiplies the length of the note by 3/2**, just as it does in written music. The line ends by going back down to octave 4 and playing another 3/4 note, A.

The remaining string data use these codes to play the next three lines of the song. You may type the letters in the codes in upper or lowercase. Also, the spaces between the codes don't matter to True BASIC, but they do make the program easier to read!

True BASIC has other music codes that give you more control over the notes and the way they're played. The letter **T sets the tempo**, or speed, for the rest of the melody. The number given with T represents the number of quarter notes played in one minute. If you don't specify the tempo, True BASIC plays 120 quarter notes per minute. Add the code T180 to the first DATA statement, and run Smoky again.

The **ML code plays music legato**, and **MS plays staccato**. (Legato means play the music smoothly with a connection between successive notes. Staccato means play the music briskly with no connection between notes.) Add some of these codes to Smoky, and run it again. You can use the **MN code to set the music style back to normal**.

You can include **sharps** and **flats** in your music by adding a "+" or "#" after the note to indicate a sharp, or "-" after the note to indicate a flat. You can also write lengths of single notes by putting the appropriate digit after the letter for that note. For example, the first two lines of "America" in the key of F would look like this:

```
F4 F4 G4 E4. F8 G4
A4 A4 B-4 A4. G8 F4
```

The letter **R stands for rest**. The number given with R has the same meaning as the numbers associated with the code L. That is, R4 means rest for the length of a quarter note, R2 means rest for the length of a half note, etc.

The following table summarizes the PLAY codes.

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

The SOUND Statement

The SOUND statement makes your computer emit sounds that are not necessarily musical notes. You specify the frequency of the sound in Hertz (cycles per second) and the duration of the sound in seconds. For example, the statement:

```
SOUND 440, 10
```

plays concert A, which has a frequency of 440 Hertz, for 10 seconds.

CHAPTER
18

Correcting Errors and Debugging

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.

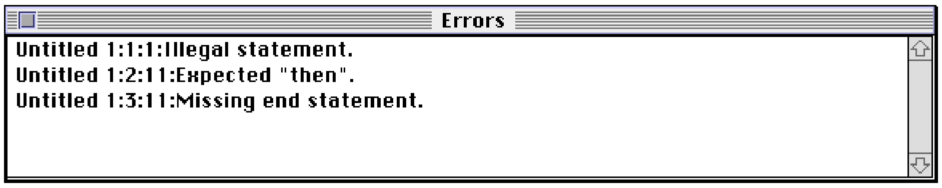
Illegal Statements

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 statemen**t, 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":

```
    PIRNT  "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:



```
Untitled 1:1:1:Illegal statement.
Untitled 1:2:11:Expected "then".
Untitled 1:3:11: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 PIRNT. 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.

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.

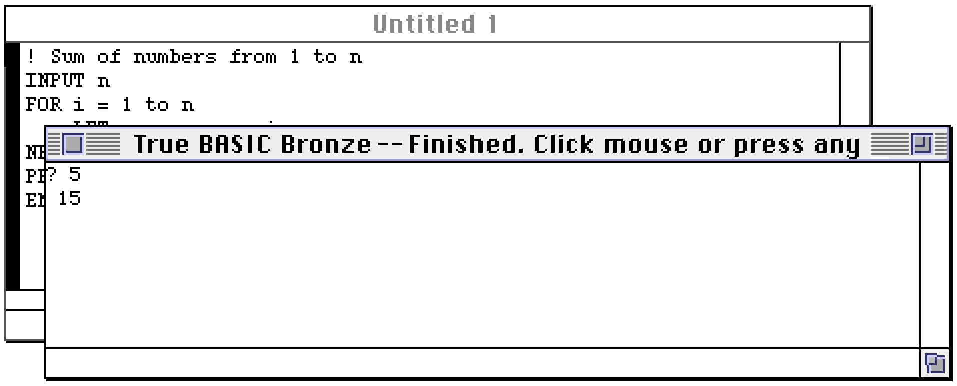
If you accidentally close the Variable window, you can reopen it by selecting it from the Windows menu of Editing window.

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:

```
! 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.)



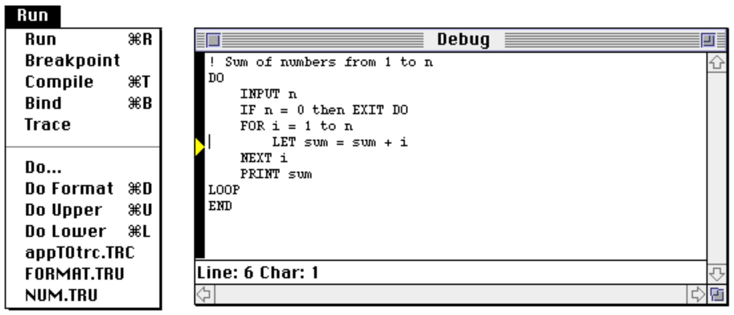
```
! Sum of numbers from 1 to n
INPUT n
FOR i = 1 to n
```

```
? 5
 15
```

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!)

```
! Sum of numbers from 1 to n
DO
   INPUT n
   IF n = 0 then EXIT DO
   FOR i = 1 to n
       LET sum = sum + i
   NEXT i
   PRINT sum
LOOP
END
```

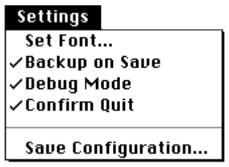
When you run this program and enter 5, it prints 15 as it should. But when you now enter 3, it prints not 6, but 21, which is a wrong answer.



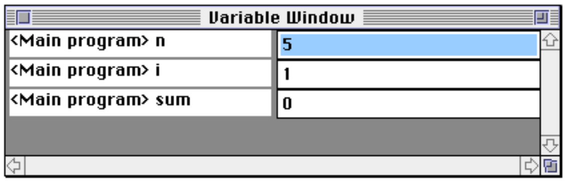
```
! Sum of numbers from 1 to n
DO
    INPUT n
    IF n = 0 then EXIT DO
    FOR i = 1 to n
        LET sum = sum + i
    NEXT i
    PRINT sum
LOOP
END
```

Line: 6 Char: 1

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.



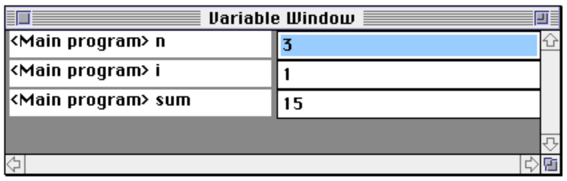
Make sure Debug Mode is checked in the Settings menu. Now place the cursor in front of the line 'LET sum = sum + 1', which is the workhorse line in the program. Now choose Run from the Run menu. The program will stop almost immediately at the breakpoint. The Variable Window will look like this:



<Main program> n	5
<Main program> i	1
<Main program> sum	0

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.



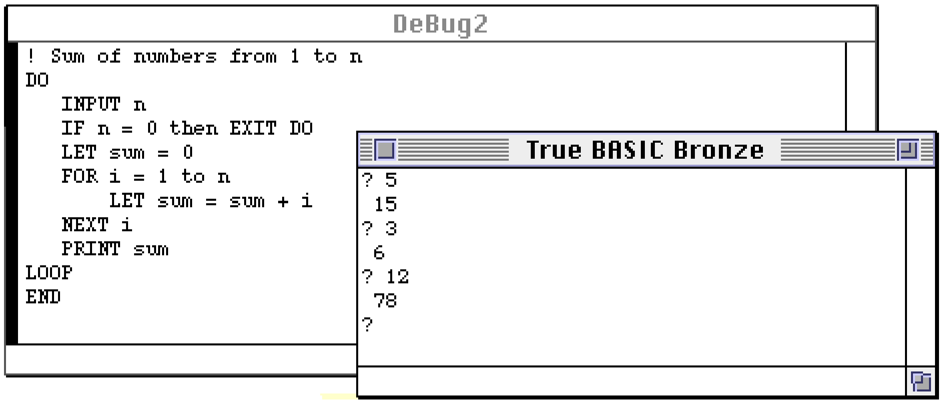
<Main program> n	3
<Main program> i	1
<Main program> sum	15

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

```
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!



```
! Sum of numbers from 1 to n
DO
   INPUT n
   IF n = 0 then EXIT DO
   LET sum = 0
   FOR i = 1 to n
       LET sum = sum + i
   NEXT i
   PRINT sum
LOOP
END
```

```
? 5
 15
? 3
 6
? 12
 78
?
```

APPENDIX A

ASCII Character Set

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.

Decimal	Name	Hex	Decimal	Name	Hex	Decimal	Name	Hex
000	nul	00	029	gs	1D	058	:	3A
001	soh	01	030	rs	1E	059	;	3B
002	stx	02	031	us	1F	060	<	3C
003	etx	03	032	space	20	061	=	3D
004	eot	04	033	!	21	062	>	3E
005	enq	05	034	“	22	063	?	3F
006	ack	06	035	#	23	064	@	40
007	bel	07	036	$	24	065	A	41
008	bs	08	037	%	25	066	B	42
009	ht	09	038	&	26	067	C	43
010	lf	0A	039	’	27	068	D	44
011	vt	0B	040	(	28	069	E	45
012	ff	0C	041	)	29	070	F	46
013	cr	0D	042	*	2A	071	G	47
014	so	0E	043	+	2B	072	H	48
015	si	0F	044	,	2C	073	I	49
016	dle	10	045	-	2D	074	J	4A
017	dc1	11	046	.	2E	075	K	4B
018	dc2	12	047	/	2F	076	L	4C
019	dc3	13	048	0	30	077	M	4D
020	dc4	14	049	1	31	078	N	4E
021	nak	15	050	2	32	079	O	4F
022	syn	16	051	3	33	080	P	50
023	etb	17	052	4	34	081	Q	51
024	can	18	053	5	35	082	R	52
025	em	19	054	6	36	083	S	53
026	sub	1A	055	7	37	084	T	54
027	esc	1B	056	8	38	085	U	55
028	fs	1C	057	9	39	086	V	56

Decimal	Name	Hex	Decimal	Name	Hex	Decimal	Name	Hex
087	W	57	101	e	65	115	s	73
088	X	58	102	f	66	116	t	74
089	Y	59	103	g	67	117	u	75
090	Z	5A	104	h	68	118	v	76
091	[	5B	105	i	69	119	w	77
092	\	5C	106	j	6A	120	x	78
093	]	5D	107	k	6B	121	y	79
094	^	5E	108	l	6C	122	z	7A
095	_	5F	109	m	6D	123	{	7B
096	`	60	110	n	6E	124	|	7C
097	a	61	111	o	6F	125	}	7D
098	b	62	112	p	70	126	~	7E
099	c	63	113	q	71	127	del	7F
100	d	64	114	r	72			

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
END
```

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
END
```

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
END
```

APPENDIX **B**

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	RANDOMIZE
DIM	REM
PAUSE	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	ON GOTO
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 TEXT
DRAW	SOUND
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	
ASK MAX COLOR	MAT PLOT POINTS
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
	SET DIRECTORY
ASK WINDOW	SET MODE
BOX DISK	SET NAME
GET KEY	WINDOW
GET MOUSE	

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 PLOT AREA
MAT WRITE	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	MAT INPUT #n:
ASK #n: DATUM	MAT LINE INPUT #n:
ASK #n: ERASABLE	MAT PRINT #n:
ASK #n: FILESIZE	
ASK #n: FILETYPE	READ #n:
ASK #n: MARGIN	
ASK #n: NAME	SET #n: MARGIN
ASK #n: ORGANIZATION	SET #n: POINTER
ASK #n: POINTER	SET #n: RECORD
ASK #n: RECORD	SET #n: RECSIZE
ASK #n: RECSIZE	SET #n: ZONEWIDTH
ASK #n: RECTYPE	
ASK #n: SETTER	UNSAVE
ASK #n: ZONEWIDTH	WRITE #n:

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 statments 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 CALLstatement.

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 occurs lexically before the first reference to that array.

DO Loop

```
DO { | WHILE condition | UNTIL condition | }
     . . .
     EXIT DO   [optional]
     . . .
LOOP {  WHILE condition | UNTIL condition | }
```

The DO statement can contain either a WHILE or UNTIL part, or nothing, and the same for the LOOP statement. There can be any number of EXIT DO statements.

DRAW Statement

```
DRAW picture name (arg 1, ..., arg n)
DRAW picture name (arg 1, ..., arg n) WITH trans *... *  trans
trans::     SCALE (size)
            SCALE (xsize, ysize)
            ROTATE (angle)
            SHIFT (xshift, yshift)
            SHEAR (angle)
```

The (argument-list) is optional. The DRAW statement causes the picture named to be drawn on the screen, just as if the DRAW statement were replaced by the code of the picture definition. The angles in ROTATE and SHEAR are measured in radians unless OPTION ANGLE DEGREES is in effect.

If the WITH clause is present, then the transformation applies applies to PLOT, FLOOD, and MAT PLOT statements (but not BOX statements) in the picture before drawing it. If a picture also contains DRAW statements with WITH clauses, then the final transformation is the "product" of the transformations along the way. The transformation consists of shifts, rotations, shears, or changes of scale, or any sequence thereof.

SCALE with one argument is the same as SCALE with two arguments with the same scale factor applied to both the x- and y-directions. That is, SCALE(a)= SCALE(a,a).

ROTATE causes the picture to be rotated counter-clockwise through the given angle.

SHIFT causes the picture to be shifted in the x-direction by an amount given by the first argument, and in the y-direction by an amount given by the second argument.

SHEAR causes the picture to be tilted clockwise through the specified angle. That is, it leaves horizontal lines horizontal, but tilts vertical lines through the given angle.

END Statement

The END statement must be the last statement of a program and is required. Only one END statement is allowed. The file that contains the program can also contain external procedures and modules following the END statement. Executing the END statement stops the program.

END DEF Statement

The END DEF statement must appear as the last statement of a DEF structure.

END IF Statement

The END IF statement must appear as the last statement of an IF structure.

END PICTURE Statement

The END PICTURE statement must appear as the last statement of a PICTURE structure.

END SELECT Statement

The END SELECT statement must appear as the last statement of a SELECT structure.

END SUB Statement

The END SUB statement must appear as the last statement of a SUB structure.

EXIT DEF Statement

```
EXIT DEF
```

The EXIT DEF statement jumps to just beyond the END DEF statement of the innermost function that contains it, and is optional.

EXIT DO Statement

```
EXIT DO
```

The EXIT DO statements jumps to just beyond the LOOP statement of the inner-most DO loop containing the EXIT DO, and is optional.

EXIT FOR Statement

```
EXIT FOR
```

The EXIT FOR statement jumps to just beyond the NEXT statement of the inner-most FOR loop containing the EXIT FOR, and is optional.

EXIT PICTURE Statement

```
EXIT PICTURE
```

The EXIT PICTURE statement jumps to just beyond the END PICTURE statement of the innermost picture that contains it, and is optional.

EXIT SUB Statement

```
EXIT SUB
```

The EXIT SUB statement jumps to just beyond the END SUB statement of the innermost subroutine that contains it, and is optional.

EXTERNAL Statement

```
EXTERNAL
```

The EXTERNAL statement must appear at the start of a LIBRARY file of external procedures.

FLOOD Statement

```
FLOOD xcoord, ycoord
```

FLOOD will fill, with the current foreground color, the closed graphical region containing the point whose x-coordinate is xcoord and whose y-coordinate is ycoord.

FOR Loop

```
FOR forvar = numeric-expression TO numeric-expression STEP numeric-expression
   ...
   EXIT FOR   [optional]
   ...
NEXT forvar
```

The simple numeric variable (not a numeric array element) in the NEXT statement must be the same as the numeric variable appearing in the FOR statement. The STEP part is optional. If missing, the increment is 1.

IF Statement

IF *condition* THEN *simple-statement* ELSE *simple-statement*

If the condition is “true,” then the *simple-statement* following the keyword THEN will be executed, following which control will pass to the next line.

If the condition is “false,” and the ELSE clause is present, its simple-statement will be executed, following which control will pass to the next line. If the ELSE clause is not present, then control will pass directly to the next line.

IF Structure

```
IF condition1 THEN
   ...
ELSEIF condition2 THEN
   ...
ELSEIF condition3 THEN
   ...
ELSE
   ...
END IF
```

The IF structure can have 0 or more ELSEIF parts and 0 or 1 ELSE. If ELSE is present, it must follow any ELSEIF part. The keyword ELSEIF can be spelled ELSE IF.

If *condition 1* is “true,” the statements immediately following are executed, up to the first ELSEIF, ELSE, or END IF, following which control jumps to the statement following the END IF.

If *condition 1* is “false,” control passes to the first ELSEIF part following the IF line. If *condition 2* is “true,” the statements immediately following it are executed, up to the next ELSEIF, ELSE, or END IF, following which control passes to the statement following the END IF line. If *condition 2* is “false,” this process is repeated.

If there are no more ELSEIF parts, then control is passed to the ELSE part, and the statements following the ELSE line are executed, up to the END IF line. If there is no ELSE part, control is passed to the statement following the END IF line.

INPUT Statement

INPUT variable, ..., variable

INPUT PROMPT *string-constant*: variable, ..., variable

When the INPUT statement is executed, the program awaits an input-response from the user. The input-response consists of quoted-strings and unquoted-strings, separated by commas.

The items in the input-response are assigned to the variables in the INPUT statement. String variables can receive any input-item, but numeric variables can receive only input-items whose characters form a numeric-constant. The rules are similar to those for READ and DATA statements.

LET Statement

LET variable = *formula*

The LET statement computes the formula on the right of the equal sign and then assigns the value to the variable on the left of the equal sign.

LIBRARY Statement

LIBRARY *quoted-string* …, *quoted-string*

The LIBRARY statement names the file or files containing external routines needed by the entire program.

LINE INPUT Statement

LINE INPUT stringvar$, ..., stringvar$

LINE INPUT PROMPT *string-constant*: stringvar$, ..., stringvar$

A LINE INPUT statement requests one or more lines of input from the user. The first line is supplied to the the first stringvar$, the second to the second, and so on. All characters in the response-line are supplied, including leading and trailing spaces, embedded commas, and quote marks.

LOCAL Statement

LOCAL variable, ..., variable

A LOCAL statement specifies that the variables named in it are local to the routine containing the statement. If an array is named in a LOCAL statement, it must also include its subscript bounds. The LOCAL statement is normally irrelevant in external routines, since all variables except parameters are automatically local, but it can be important in internal routines. The LOCAL statement can be used in conjunction with the OPTION TYPO statement to avoid typographical errors in variable names.

LOOP Statement

The LOOP statement may occur only as the last statement of a DO loop, and is required. (See the DO Loop.)

MAT INPUT Statement

    MAT INPUT array, ..., array

MAT INPUT assigns values from the input-response to the elements of the arrays, in order. There must be a separate input-response for each array 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; ...; point
    PLOT LINES: point; ...; point;
    PLOT point; ...; point;

PLOT LINES plots the line-segments that connect the points. A line is drawn from the previous point to the first point if and only if the beam was left on.

The following two statements are equivalent:

    PLOT   x1, y1; x2, y2; x3, y3
    PLOT LINES: x1, y1; x2, y2; x3, y3

If the PLOT LINES and PLOT statements end with a semicolon, the beam stays on so that subsequent PLOT LINES or PLOT statements will continue plotting the line without a break; otherwise, the beam is turned off.

PLOT AREA Statement

    PLOT AREA: point; ...; point

PLOT AREA plots the polygon defined by connecting the points and fills it with the current foreground color. The last point need not repeat the first point, as the line segment needed to close the polygon is automatically supplied.

PLOT TEXT Statement

PLOT TEXT, AT point: textstring$

PLOT TEXT plots the text string in the current color at the point specified in the AT clause.

Vacuous PLOT Statement

PLOT
PLOT LINES
PLOT LINES:

These statements turn off the beam in case a previous PLOT or PLOT LINES statement ended with a semicolon. They have no effect if the beam is already off.

PRINT Statement

PRINT
PRINT print-list
PRINT USING string: *using-list* *(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

SELECT CASE *select-expression*
CASE *case-specifier*
. . .
CASE *case-specifier*
. . .
CASE ELSE
. . .
END SELECT

case-specifier::	*case-part,* …, *case-part*
case-part::	constant
	constant TO constant
	IS *relational-operator* constant

The SELECT CASE structure may have zero or more CASE parts, and zero or one CASE ELSE parts, but must have at least one of either a CASE or CASE ELSE part. The constants in a *case-specifier* must be of the same type (numeric or string) as the *select-expression* in the SELECT CASE statement.

The *select-expression* in the SELECT CASE statement is first evaluated. The *case-specifier* in the first CASE part is then examined. If it satisfies any of the *case-parts*, then the statements following that CASE statement are executed and control passes to the first statement following END SELECT.

If no *case-part* in the first CASE statement is satisfied, then the second CASE statement is examined in a like manner, and so on.

If no CASE statement is satisfied, then the statements following the CASE ELSE statement are executed. If no CASE statement is satisfied and there is no CASE ELSE part, then an exception occurs.

SET BACK Statement

SET BACK colornumber

SET BACK colorname$

SET BACK is an abbreviation for SET BACKGROUND COLOR. SET BACK with colornumber 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:

MAGENTA	CYAN	WHITE
RED	BLUE	GREEN
YELLOW	BROWN	BLACK
BACKGROUND		

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 CALLstatement 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.

APPENDIX
C

Built-in Functions

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

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

Mathematical Functions *(continued)*

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

Date and Time Functions

Function	Result
DATE	Year and day of year as a number
DATE$	Year, month, and day of month as a string
TIME	Seconds since midnight
TIME$	24-hour clock time as a string

String to Number Functions

Function	Result
CHR$(x)	Character represented by ASCII number x
ORD(x$)	Ordinal position of x$ in ASCII character set
NUM(x$)	Numeric value of IEEE 8-byte string
NUM$(x)	IEEE 8-byte equivalent of numeric value
STR$(x)	Changes number to a string
VAL(x$)	Changes string containing digits to a number

String Transforming Functions

Function	Result
LCASE$(x$)	Change letters to lowercase
UCASE$(x$)	Change letters to uppercase
LTRIM$(x$)	Remove leading blanks
RTRIM$(x$)	Remove trailing blanks
TRIM$(x$)	Remove leading & trailing blanks
REPEAT$(x$,n)	x$ repeated n times

String Search Functions

Function	Result
LEN(x$)	Number of characters in x$
POS(x$,y$,n)	First occurrence of y$ in x$ after character n
POSR(x$,y$)	Ditto POS but starting from the end
CPOS(x$,y$)	First occurrence in x$ of any character in y$
CPOSR(x$,y$)	Ditto CPOS but starting from the end
NCPOS(x$,y$)	First occurrence in x$ of any character not in y$
NCPOSR(x$,y$)	Ditto NCPOS but starting from the end

Array Functions

Function	Result
DET(a)	Determinant of the square matrix a
DOT(a,b)	Dot product of vectors a and b
LBOUND(a,n)	Lower bound of dimension n for array a
UBOUND(a,n)	Upper bound of dimension n for array a
SIZE(a,n)	Number of element in dimension n of array a

MAT Functions *that can appear only in MAT assignment statements*

Function	Result
CON	Array of ones
IDN	Identity matrix
INV(a)	Inverse of array a
NUL$	Array of empty strings
TRN(a)	Transpose of array a
ZER	Array of zeroes

The descriptions in the alphabetical list use the following terms:

(numeric-expression)	numeric expression
(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

CEIL(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

CHR$(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

CON redim
CON

CON is an array constant that yields a numeric array consisting entirely of ones. CON can appear only in a MAT assignment statement.

COS Function

COS(numeric-expression)

Returns the value of the cosine function. The argument is assumed to be in radians or degrees depending on whether the current OPTION ANGLE is RADIANS (default) or DEGREES.

COSH Function

COSH(numeric-expression)

Returns the value of the hyperbolic cosine function.

COT Function

COT(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 geneology 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 *r*numeric-expression. If there is no second argument, *arrayarg* must be one-dimensional array, and LBOUND returns the lowest value (lower bound) for its subscript.

LCASE$ Function

```
LCASE$(string-expression)
```

Returns the value of string-expression with all ASCII uppercase letters converted into lowercase. Characters outside the range of the ASCII uppercase letters are unchanged.

LEN Function

```
LEN(string-expression)
```

Returns the length (that is, the number of characters) of the argument string-expression. All characters count, including control characters and other nonprinting characters.

LOG Function

```
LOG(numeric-expression)
```

Returns the natural logarithm of numeric-expression, which must be greater than 0. The natural logarithm of *x* may be defined as that value *v* for which e^v = x, where e = 2.718281828....

LOG10 Function

```
LOG10(numeric-expression)
```

Returns the common logarithm of numeric-expression, which must be greater than 0. The common logarithm of *x* is defined as that value *v* for which 10^v = x.

LOG2 Function

LOG2(numeric-expression)

Returns the logarithm to the base 2 of numeric-expression, which must be greater than 0. The logarithm to the base 2 of *x* is defined as that value *v* for which 2^v = x.

LTRIM$ Function

LTRIM$(string-expression)

Returns the value of string-expression but with leading blank spaces removed. Trailing spaces, if any, are retained.

MAX Function

MAX (numeric-expression, numeric-expression)

Returns the larger of the values of the two arguments.

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 numberical 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, rnumeric-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 rnumeric-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$(rnumeric-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(rnumeric-expression)

TAB(rnumeric-expression, rnumeric-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.

APPENDIX

D

Explanations of Error Messages

This appendix contains a partial list of True BASIC error messages, in alphabetic order. Error messages referring to statements or features not introduced in this book are omitted.

The number following some messages is the error number for 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 is 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:

```
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 dimensions. 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 subscript 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 contact 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.

APPENDIX

Making Your Own DO Programs E

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

   ```
   EXTERNAL
   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

   ```
   END SUB
   ```

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 upper case letters into lowercase, or all lower case letters into uppercase. Write the following subroutine:

```
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 sourse 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.

APPENDIX

PRINT USING Statement

F

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.

```
x          PRINT x            PRINT USING "###": x
-          -------            --------------------
1          | 1                            |  1
12         | 12                           | 12
123        | 123                          |123
1234       | 1234                         |***
```

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

--                    ------------------------------
0                                |$0,000,000.00
1                                |$0,000,001.00
1234                             |$0,001,234.00
1234567.89                       |$1,234,567.89
x                     PRINT USING "$*,***,***.##": 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 *carets* (^) that must immediately follow the *decimal-part*. The following example shows how.

```
x                     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:

   The Floating Header

First	Second	Positive	Negative
$	+	"$+"	"$–"
$	–	"$ "	"$–"
$	none	"$"	error
+	$	"+$"	"–$"
+	none	"+"	"–"
–	$	" $"	"–$"
–	none	" "	"–"
none	none	""	error

   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.

```
PRINT x                     PRINT USING "$$$$$$$-#,###.##": x
-------                     ---------------------------------
| 0                                      |        $     .00
| 1                                      |        $    1.00
|-1                                      |        $-   1.00
| 4321.5                                 |       $ 4,321.50
|-4321.5                                 |       $-4,321.50
| 1.23456789e+7                          |   $ 12345,678.90
|-1.23456789e7                           |   $-12345,678.90
| 1000000000                             | $ 1000000,000.00
|-1000000000                             | $-1000000,000.00
```

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.

```
USING          String to be Printed
string     "Ok"       "Hello"    "Goodbye"
------     ------      -------    ---------
"<####"     |Ok        |Hello     |*******
"#####"     | Ok       |Hello     |*******
">####"     |   Ok     |Hello     |*******
```

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.

```
s$                     PRINT USING "$#,###,###.##": s$
-------------          -------------------------------
"Cash"                            |   Cash
"Liabilities"                     | Liabilities
"Accounts Receivable"             |*************
```

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 calculated results or data in many easy-to-understand formats. Investing time in learning the many capabilities of these statements will pay rich dividends.

APPENDIX

G

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:

```
PRINT #2: name$, "Age:"; 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:

```
SET #3: MARGIN 70
SET #3: ZONEWIDTH 10
```

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

```
ASK #2: MARGIN m
ASK #2: 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:

```
PRINT #2: name$; Tab(45); "Age:"; 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:

```
LET form$ = "###########################>   Age: ##"
PRINT #2, 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:

```
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:

```
...
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 Str-Lib library.

```
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 structures 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** statement 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
```

```
CASE "STRING"
     READ #1: s$
CASE else
     ! type$ = "NONE" if at the end of the file
     ! type$ = "UNKNOWN" if can't tell
END SELECT
```

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:

```
! 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:

```
! 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:

```
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:

```
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
```

```
      READ #9: next$              ! Examine each record
      IF next$ = word$ then EXIT FOR
   NEXT r
   IF r > last_rec then LET rec = 0 else LET rec = r
END SUM
```

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:

```
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:

```
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:

```
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:

```
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:`

```
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:

```
CALL Packb(x$,Maxnum,16,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 `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:

```
ASK #1: FILESIZE fs
READ #1, BYTES fs: x$
LET j = 1
FOR i = 1 to Len(x$)/2
    LET list(i) = Unpackb(x$,j,16)
    LET j = j+16
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.

APPENDIX

H

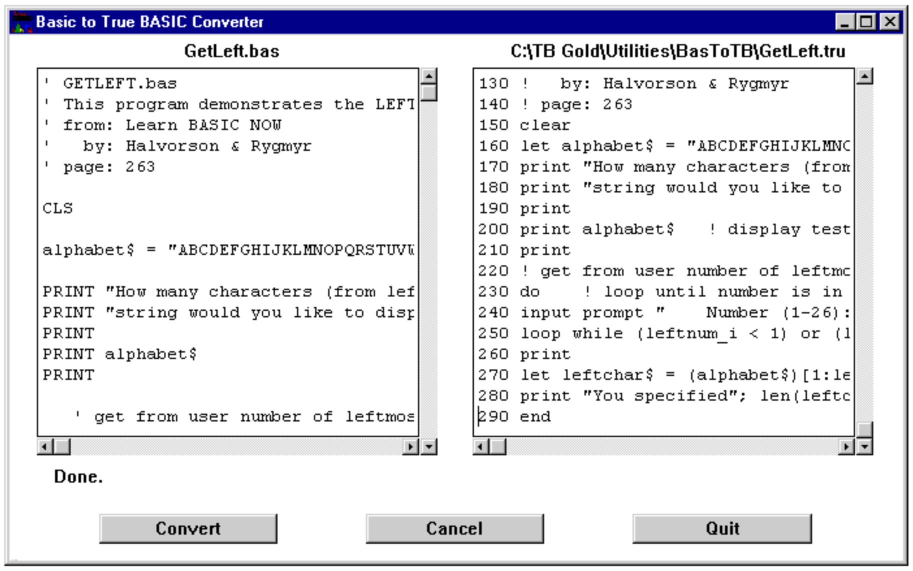
BASIC to True BASIC Converter

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.

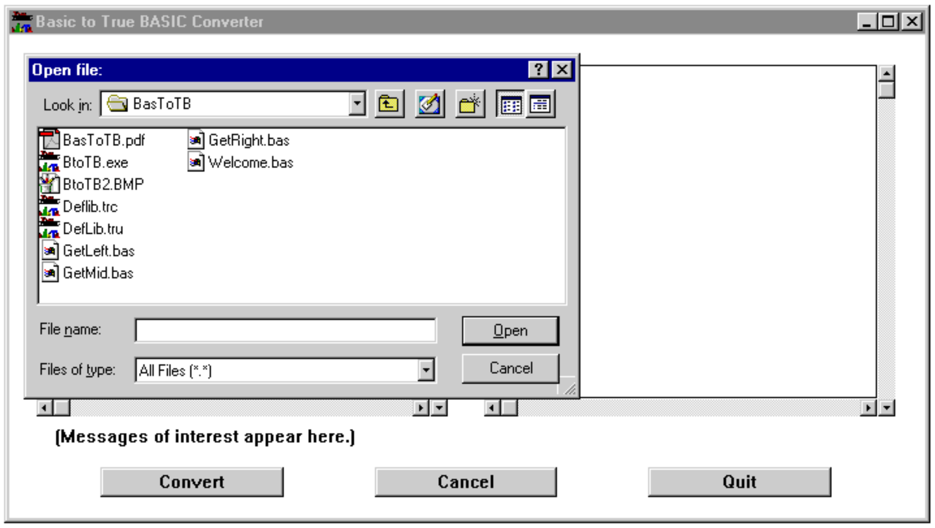

Basic to True BASIC Converter
GetLeft.bas
C:\TB Gold\Utilities\BasToTB\GetLeft.tru
Convert
Cancel
Quit

```
' GETLEFT.bas
' This program demonstrates the LEFT
' from: Learn BASIC NOW
'   by: Halvorson & Rygmyr
' page: 263

CLS

alphabet$ = "ABCDEFGHIJKLMNOPQRSTUVW

PRINT "How many characters (from lef
PRINT "string would you like to disp
PRINT
PRINT alphabet$
PRINT

   ' get from user number of leftmos
```

```
130 !   by: Halvorson & Rygmyr
140 ! page: 263
150 clear
160 let alphabet$ = "ABCDEFGHIJKLMNO
170 print "How many characters (from
180 print "string would you like to
190 print
200 print alphabet$    ! display test
210 print
220 ! get from user number of leftmo
230 do     ! loop until number is in
240 input prompt "    Number (1-26):
250 loop while (leftnum_i < 1) or (l
260 print
270 let leftchar$ = (alphabet$)[1:le
280 print "You specified"; len(leftc
290 end
```

Done.

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 whichyou 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 line feed 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,

```
LET a% = b!
```

is converted to the True BASIC

```
LET a_i = 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:

Basic	True BASIC
`x`	`x`
`x%`	`x_i`
`x!`	`x_s`
`x#`	`x_d`
`x&`	`x_li`

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 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,

```
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:

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

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 tobe 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.

INDEX