TRUE BASIC

The Original BASIC

Gold Edition

Gold Edition Reference Guide

For the True BASIC Language System

Edited by Thomas E. Kurtz, John Arscott, Anne Taggart

ISBN 9-939553-42-2

Published by:
True BASIC
Gaysville, VT 05746 USA

MANUAL No: 7232/M

http://www.truebasic.com	True BASIC website
support@truebasic.com	Technical questions and support
sales@truebasic.com	e-mail sales address
1-888-282-9873	Phone/Fax (24-hour service)

Printed in the United States of America. 01/00

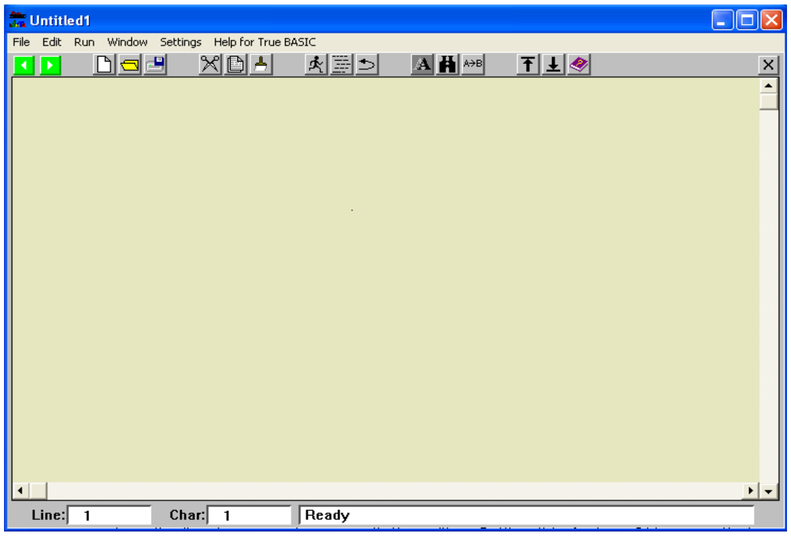
Contents

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.

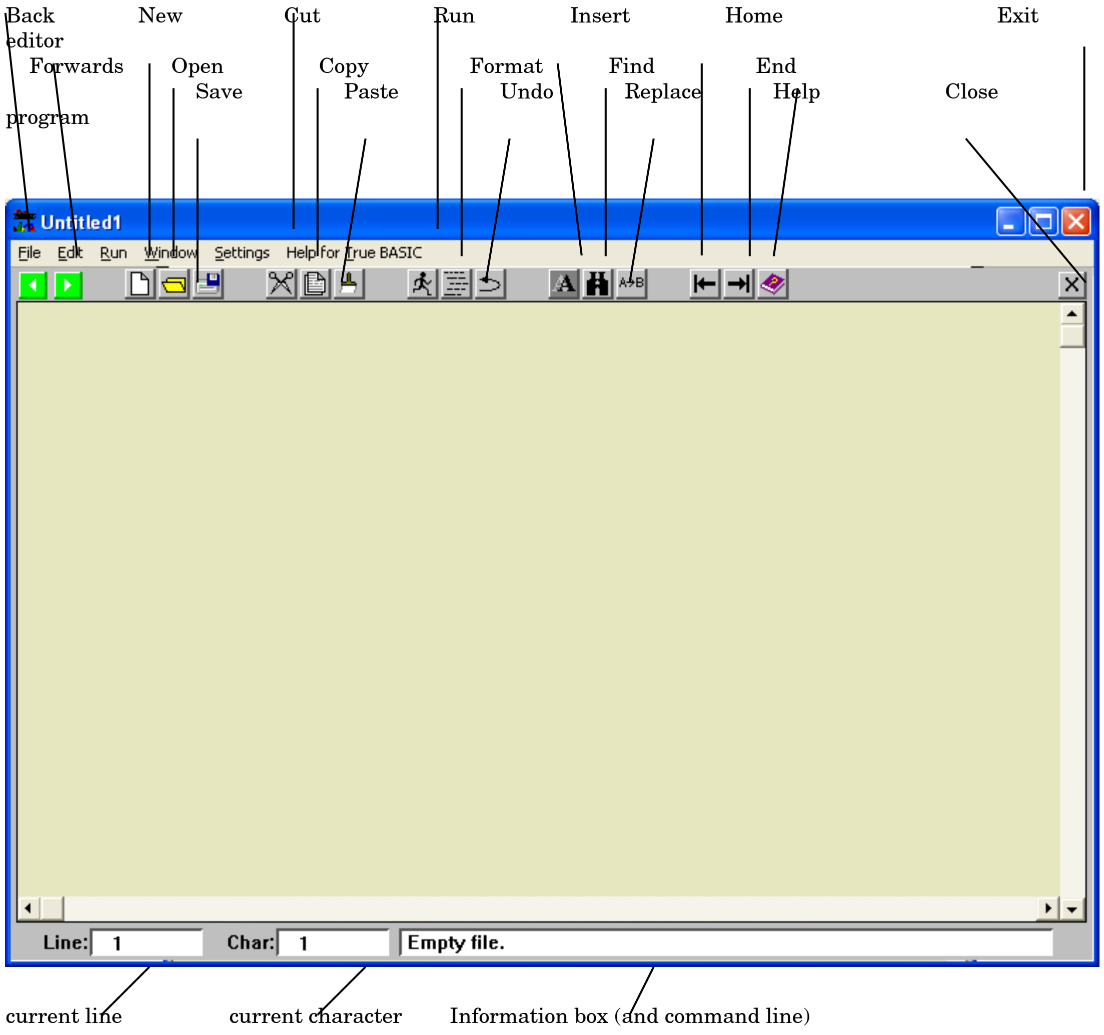
PRINTING HARDCOPY

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.

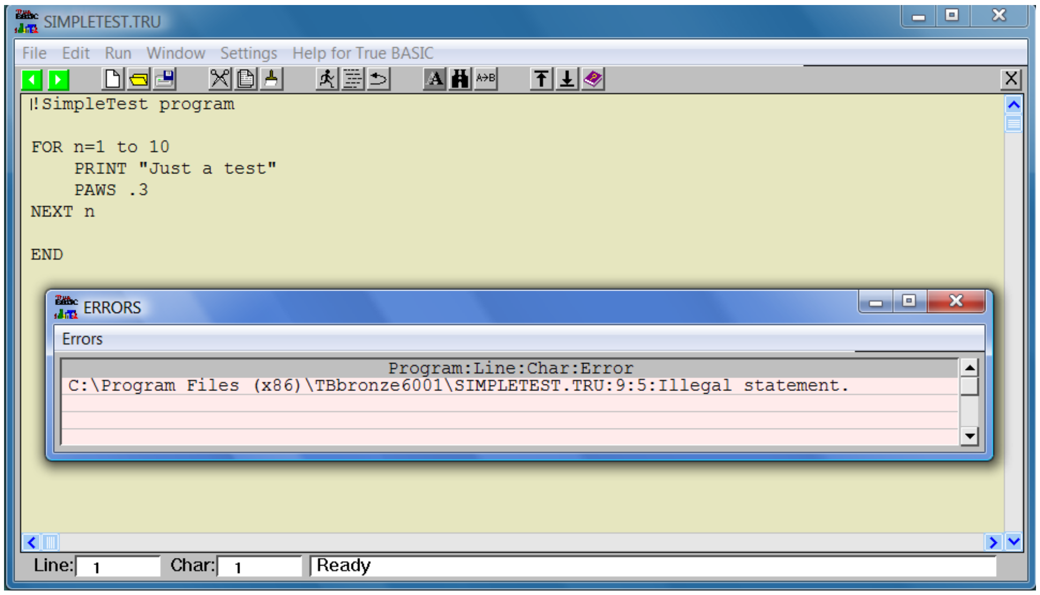
FORMATTING TEXT

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

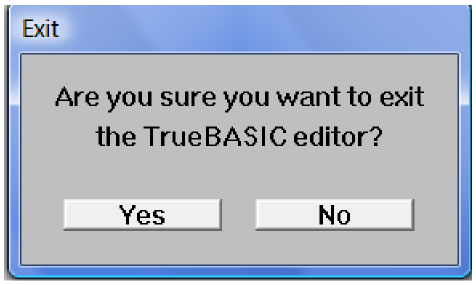
Breakpoints mark your program at the line where the cursor is positioned. The BREAKPOINT option under the RUN menu is normally disabled (grayed out) and only becomes active when you select DEBUG MODE from the SETTINGS menu. The breakpoint will appear as the word <<<BREAKPOINT>>> surrounded by angle brackets. 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

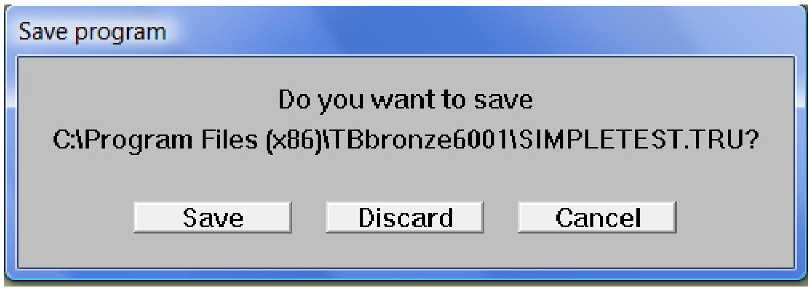
If you wish to exit the editor you must select EXIT under the FILE menu or you can click on the 'close window' button (white cross on a red background). You will be asked if you are sure you want to QUIT. You can eliminate this reminder by un-ticking the 'Confirm quit' option under the SETTINGS menu.



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.



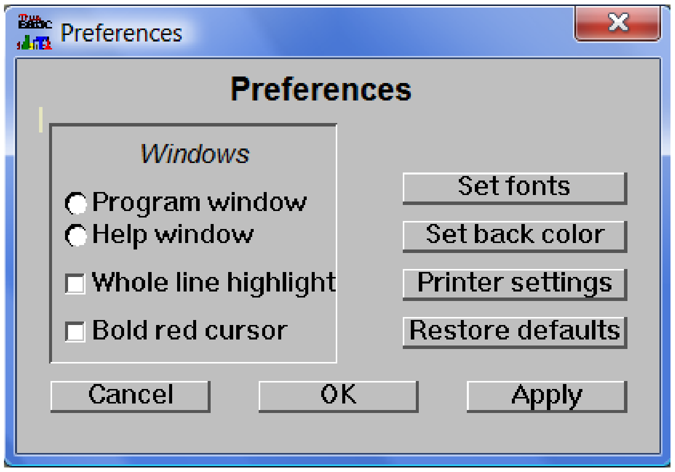
Save program

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.

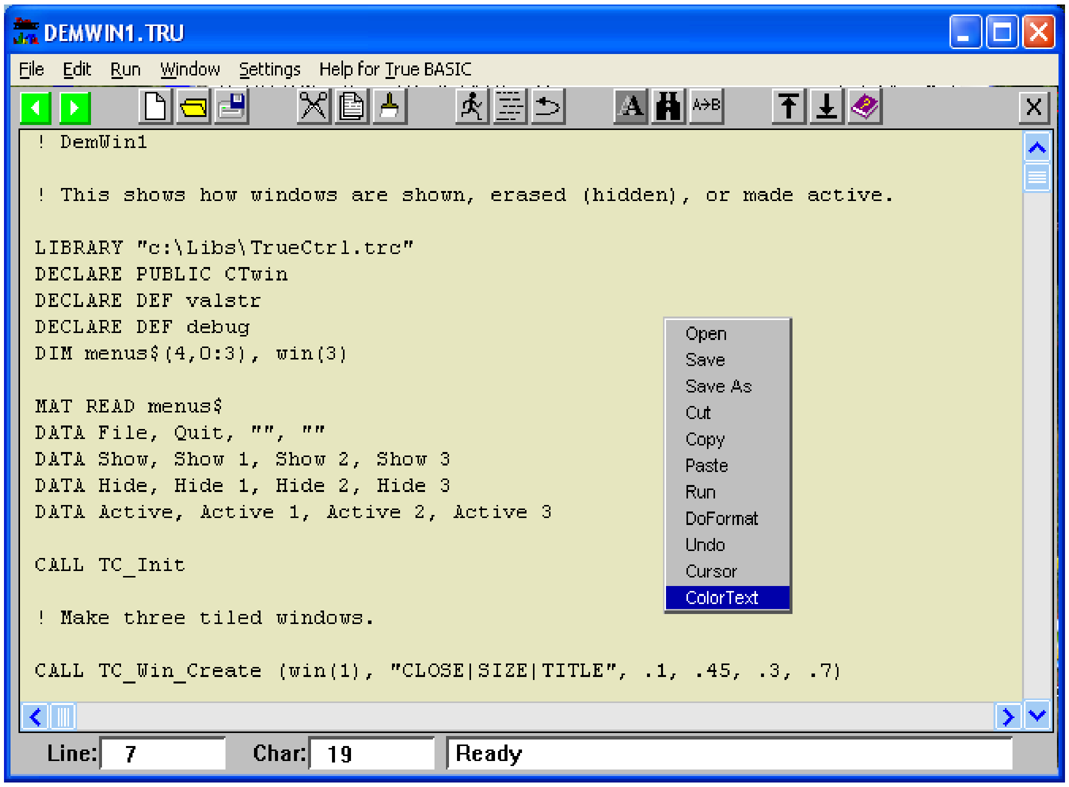
If you want the default settings to be restored then click on the appropriate button. If you click on either the APPLY button or the OK button, the defaults will be restored immediately.

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)
```

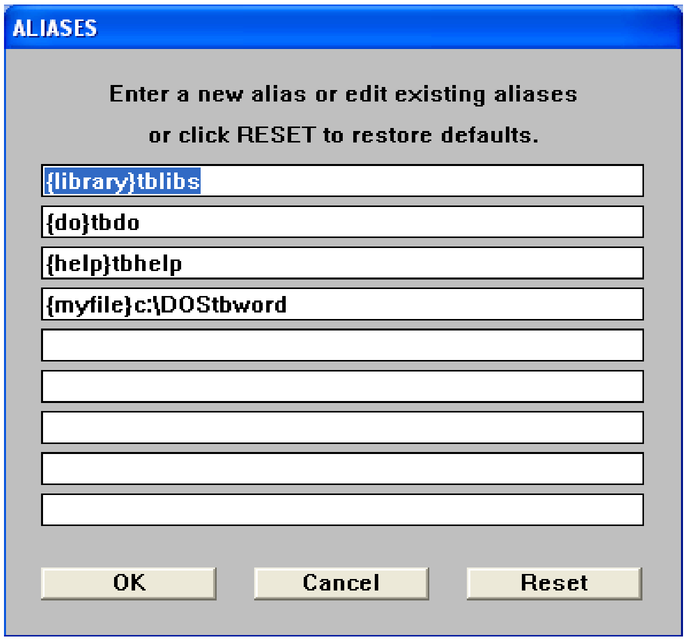
This menu works like the main menu. It will disappear as soon as you make a selection from the menu.

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.

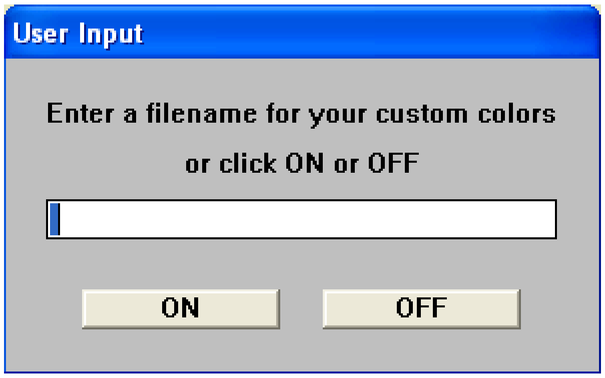
COLORTEXT can be activated from the SETTINGS menu or from the right click menu.

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.



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

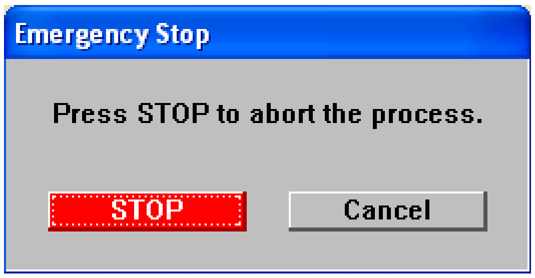
From the current cursor position, the DEL key will delete single characters ahead of the cursor. The BS (backspace) key will delete text behind the cursor. The DEL key will also delete any highlighted text. Similarly, the back space key will also delete highlighted text.

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

1

A Word on Style

Before you begin to program, you need to know something about the programming style of the language you plan to use. A language's style determines how you may arrange a program's source code.

The style of a programming language is largely determined by rules. True BASIC, like most programming languages, supports two types of style: ***required style*** — rules you must obey— and ***conventional style*** — optional standards that help make programs easier to read, understand, and adapt or expand. Take a look at the following program that implements a simple guessing-game. This program shows the "flavor" of True BASIC style and illustrates both required and conventional elements of the language:

```
! Set up the program and get initial number
CLEAR
RANDOMIZE
LET answer = Int(Rnd*10) + 1

! Display the title and instructions
PRINT "A guessing game."
PRINT "Enter your guess as number between 1 and 10."
PRINT "Enter 0 to quit."

! Allow the user to play an unlimited number of games
DO
   INPUT PROMPT "Your guess: ": guess
   LET guess = Int(guess)                        ! Use next lowest integer
   IF guess < 1 then                             ! User has quit
      EXIT DO
   ELSEIF guess > 10 then                        ! Guess out of range
      PRINT "Your guess must be between 1 and 10!"
   ELSEIF guess = answer then                    ! Correct guess
      PRINT "Correct! What a guess!"
      PRINT "I'm thinking of another number."
      PAUSE 3                                    ! Act like we're thinking
      PRINT                                      ! Blank line to start series
      LET answer = Int(Rnd*10) + 1               ! Get new answer
   ELSE                                          ! Incorrect guess
      PRINT "Wrong! Try again!"
   END IF
LOOP
! All done
PRINT "Thanks for playing."
END
```

Required Style

There are certain rules you must obey if you want your True BASIC programs to run. Fortunately, True BASIC is designed so that you do not need to know all the required style rules before you begin to program. You should, however, know the required style rules that apply to a particular structure before you attempt to use it in a program.

Most of this manual describes the required style rules for each of True BASIC's statements and structures. This section introduces a few fundamental requirements that apply to all programs and all structures.

Look again at the simple guessing-game program at the beginning of this chapter. It illustrates several requirements of the True BASIC language.

[!] Note: Every True BASIC program must have one, and only one, END statement.

While it doesn't really "do" anything, the **END** statement is vital to the operation of a True BASIC program. The **END** statement indicates the end of the main program; it tells True BASIC where to stop executing code.

As you'll see later (Chapter 10, "User-Defined Functions and Subroutines") the **END** statement is not necessarily the last statement in the document containing your program. You may have external structures stored in the same document after the **END** statement, but the **END** statement must end the main program unit.

If you attempt to run a program with no **END** statement, the program halts with the message "Missing end statement." If you run a program containing more than one **END** statement, the program halts with the message "Statement outside of program."

[!] Note: Each True BASIC statement must begin with a keyword. If additional information follows, there must be a space after the keyword.

Look at the sample program again. Some lines are blank or contain only comments (beginning with a !), but each "executable statement" begins with a keyword. Some keywords stand by themselves, such as the **CLEAR**, **RANDOMIZE**, and **END** statements. Others usually or always include additional information, such as the **LET** and **PRINT** statements. A space must always follow a keyword used with additional information. The use of spaces in the rest of the statement is generally optional. Throughout this manual we represent keywords in all uppercase letters so that they are clearly distinguishable, but you are not required to do so.

Let's look a bit more closely at the **LET** and **PRINT** statements — perhaps the two most commonly used statements in True BASIC. Each statement has certain required style rules that it must follow.

The **LET** statement assigns a value to a variable. Each begins with the keyword LET followed by the name of the variable to which the value is to be assigned. The variable name is followed by an equal sign (=) and the value to be assigned to the specified variable. The value being assigned may be an expression containing mathematical or string operators:

```
LET guess = Int(guess)
LET answer = Int(Rnd*10) + 1
```

Complete details about the rules for the **LET** statement and for constants, variables, and expressions are discussed in Chapter 2, "Constants, Variables, and Expressions." (See also the **OPTION NOLET** statement later in this section.)

The **PRINT** statement obeys required style rules of its own. The PRINT keyword is usually followed by one or more items to be printed, and multiple items can be separated by commas or semicolons. The **PRINT** statements in the sample program each print one string constant, as in:

```
PRINT "Thanks for playing."
```

Notice that the sample program also uses a blank **PRINT** statement, which produces a blank line. The complete required style rules for **PRINT** statements are discussed in Chapter 3, "Output Statements."

[!] Note: Names representing entities such as variables, arrays, and routines may be of any length and contain letters, digits, and underscore characters. All names must start with a letter and contain no spaces; all names representing string items must end with a dollar sign ($).

True BASIC makes no distinction between uppercase and lowercase. Here are some legal variable names:

```
guess               name$
answer              first_name$
V25                 CityStateCode$
```

For complete information on variable names see Chapter 2, "Constants, Variables, and Expressions."

[!] Note: True BASIC allows only one statement per line. Thus, the end of a line indicates the end of a True BASIC statement.

The rules for some statements do allow multiple parts or serve dual purposes, but you can still put only one such statement on a line. For example, the **INPUT PROMPT** statement combines the functionality of the **PRINT** and **INPUT** statements by letting you specify a message to be printed when the program asks for input, but it is itself a single statement:

```
INPUT PROMPT "Your guess: ": guess
```

The **IF** structure often occupies several lines as it does in the sample program where it uses two **ELSEIF** statements and one **ELSE** statement. However, there is also a one-line **IF** statement for simpler decisions. For example in the sample program, the single line:

```
PRINT "Wrong! Try Again!"
```

could be replaced by the single-line IF structure:

```
IF guess < answer then PRINT "Too low!" else PRINT "Too high!"
```

These are the fundamental elements of True BASIC's required style: All programs must have one, and only one, **END** statement. Each executable statement must begin with a keyword. Each keyword must be followed with a space or end-of-line character, and there may be only one statement per line.

The one exception to the above is the **OPTION NOLET** statement that lets you omit the LET keyword from assignment statements. We do not recommend you use the **OPTION NOLET** statement, as it "violates" the keyword rule and can lead to confusing error messages.

There is one more element of required style that applies only when you elect to use line numbers. As you can see, True BASIC does not require line numbers. Though we don't recommend them (see the end of this chapter), you may use them. But if you do so, be aware of this additional rule: If you use line numbers, every line in your program — including blank and comment lines — must have a line number.

Conventional Style

As long as your programs obey both the general required style rules mentioned above and the specific rules associated with the statements used, they will run. True BASIC does not pay any attention to the conventional style of your programs. Conventional style, however, helps you create and maintain your programs with a minimum of frustration.

There are many elements of conventional style, and there are also many philosophies of how best to apply them. This section introduces you to some of the most common conventional style elements and philosophies. Be aware, however, that this is primarily intended to introduce you to the options and give you some ideas. As you read it, think about what makes the most sense to you. You should develop your own conventional style philosophy to enhance the readability of your programs. If you implement your style philosophy from the outset, you can enhance your enjoyment and the productivity of your programming.

Let’s look again at the program from the beginning of the chapter, repeated here for your convenience:

```
! Set up the program and get initial number
CLEAR
RANDOMIZE
LET answer = Int(Rnd*10) + 1
! Display the title and instructions
PRINT "A guessing game."
PRINT "Enter your guess as number between 1 and 10."
PRINT "Enter 0 to quit."
! Allow the user to play an unlimited number of games
DO
   INPUT PROMPT "Your guess: ": guess
   LET guess = Int(guess)                          ! Use next lowest integer
   IF guess < 1 then                               ! User has quit
      EXIT DO
   ELSEIF guess > 10 then                          ! Guess out of range
      PRINT "Your guess must be between 1 and 10!"
   ELSEIF guess = answer then                      ! Correct guess
      PRINT "Correct! What a guess!"
      PRINT "I'm thinking of another number."
      PAUSE 3                                      ! Act like we're thinking
      PRINT                                        ! Blank line to start series
      LET answer = Int(Rnd*10) + 1                 ! Get new answer
   ELSE                                            ! Incorrect guess
      PRINT "Wrong! Try again!"
   END IF
LOOP
! All done
PRINT "Thanks for playing."
END
```

Some things you might notice are the use of space, indenting, and comments to make the structure and function of the program easier to follow.

True BASIC lets you add or omit spaces as you see fit, within certain required style rules. You may add spaces anywhere but in the middle of a keyword, name, or two-character symbol (such as >=). You may also omit spaces where it causes no confusion, but you must be sure to have a space after each keyword.

You may also insert blank lines in the program. Often, blank lines are used to separate logical blocks in a program. True BASIC simple ignores these additional lines when it runs the program.

True BASIC also allows any number of spaces to appear at the beginning of a line. (In line numbered programs, the number must come first followed by any number of spaces.) True BASIC was designed this way to allow for a variety of indentation styles. Typically, you will find that your programs are much easier to understand if you indent the bodies of structures such as loops, decisions, and procedures. The “True BASIC Environment” chapter in the Introduction section shows how the True BASIC editor can indent programs for you.

Comments are another powerful tool for making your programs easy to read. Comments are notes to yourself (and anyone else who needs to understand your code) about how a program works. True BASIC ignores comments when it runs your program.

True BASIC comments begin with an exclamation mark (!). You can place them on a separate line or at the end of any line. True BASIC ignores anything to the right of an exclamation point (that is not part of a string constant). Thus, inserting an explanation point at the beginning of a line is also an easy way to temporarily disable a statement. You may also use the **REM** statement to insert comments into your code, but the **REM** statement must occupy a line of its own:

```
REM  This is a guessing game program.
```

As mentioned in the above section, True BASIC lets you use names, or identifiers, of any length. Therefore, you may use names that describe the purpose of your variables, subroutines, and functions. You will find that this makes your programs much easier to understand and debug.

True BASIC ignores the case of letters, except as they appear in string constants. Thus, the keywords **LET**, **Let**, and **let** are identical to True BASIC. Likewise, the variable names answer, **Answer**, and **ANSWER** are seen as the same name.

In the sample code throughout this documentation, we use capital letters for keywords, begin the names of functions and subroutines with a capital letter (and capitalize the beginning of each subsequent "word"), and start variable names with a lowercase letter. We feel that this creates clear, readable programs, but you should feel free to develop your own stylistic philosophy.

[!] Note: Unlike some other forms of the BASIC language, True BASIC does not penalize you with a slower execution speed when you add spaces, comments, or blank lines or when you use meaningful variable names. There is no advantage in squeezing as much as possible on a single line. Feel free to write readable programs. Making a program understandable will not make it slower.

Many of True BASIC's structures themselves contribute to the readability of programs. The **IF** and **DO** structures used in the sample program are two examples. Other structures such as **FOR** structures, **SELECT CASE** structures, subroutines, functions, and modules also help you organize your program into clearly understandable compartments. These structured programming features also make it easier to write programs, letting you divide the task into smaller blocks that you can often test independently. Generous use of spaces and indenting make these structures even easier to understand.

These structures also free True BASIC from the need for line numbers. Line numbers are required by older versions of the BASIC language, but they are optional in True BASIC. None of True BASIC's structures or statements (described in the main parts of this manual) require line numbers, but you may number your programs and use statements such as the **GOTO** or **GOSUB** statements from older forms of BASIC. While this makes it relatively easy to run older programs, we don't recommend line numbers as a desirable conventional style. Programs written without line numbers are generally much easier to understand and maintain. Remember, however, that if you do use line numbers, you must number all the lines in the source file. Appendix E describes how the older line-number control statements work.

As a final illustration of conventional style, here is a program that is functionally identical to the sample program earlier in this chapter, but it violates most of the conventional rules we've been discussing. It does obey all the required rules, however, and True BASIC will run it without complaint:

```
1000 CLEAR
1010 RANDOMIZE
1020 LET ANSWER=INT(RND*10)+1
1030 PRINT "A GUESSING GAME."
1040 PRINT "ENTER YOUR GUESS AS A NUMBER BETWEEN 1 AND 10."
1050 PRINT "ENTER 0 TO QUIT."
1060 INPUT PROMPT "YOUR GUESS: ":GUESS
1070 LET GUESS=INT(GUESS)
1080 IF GUESS<1 THEN GOTO 1210
1090 IF GUESS>10 THEN GOTO 1130
1100 IF GUESS=ANSWER THEN GOTO 1150
```

```
1110 PRINT "WRONG! TRY AGAIN!"
1120 GOTO 1060
1130 PRINT "YOUR GUESS MUST BE BETWEEN 1 AND 10!"
1140 GOTO 1060
1150 PRINT "CORRECT! WHAT A GUESS!"
1160 PRINT "I'M THINKING OF ANOTHER NUMBER."
1170 PAUSE 3
1180 PRINT
1190 LET ANSWER=INT(RND*10)+1
1200 GOTO 1060
1210 PRINT "THANKS FOR PLAYING."
1220 END
```

We hope you'll agree that the earlier program is much easier to understand. This is because the earlier program uses a well defined conventional style philosophy. It uses comments to clarify the code; it uses case, blank lines, and extra spaces to make the code easier to read; and it uses structured programming and consistent indentation to make its logic easier to analyze.

The line-numbered program was not made deliberately complex. It merely uses structures that rely on line-numbers and ignores most of the conventions introduced in this chapter (though it does use multi-character variable names).

In summary, the look and feel of a programming language is determined by both its required style and the conventional style it allows. Exactly what you adopt as your own conventional style philosophy is not so important as the fact that you develop a style that makes your programs easier to write, understand, and expand, and that you use it consistently.

CHAPTER

2

Constants, Variables, and Expressions

The fundamental purpose of virtually all programs is to manipulate data — both numbers and letters. This data is most often represented within the program in the form of constants and variables — and expressions that calculate new values from constants and variables. Here's a simple program that illustrates these three representations of data:

```
INPUT PROMPT "Number of items? ": n
LET cost, price = 0
FOR i = 1 to n
    INPUT PROMPT "Price of item? ": price
    LET cost = cost + (price * 1.04)  ! Include 4% sales tax
NEXT i
PRINT "Your cost with the sales tax is ";
PRINT USING "$$$$#.##": cost
PRINT "Press any key to end program."
GET KEY k                             ! Hold output until a key is pressed
END
```

Constants are data values you put directly into the source code. Since their values cannot change unless you change the source code, constants remain unchanged during the running of the program. In the program above, the sales tax is the constant 1.04 representing a 4% sales tax. To change this, you must change the program itself.

Variables are named representations of data. They associate a name with some data value. True BASIC provides many ways for you to assign and change the values of a variable during a program run. This is why the names used to represent the data are called variables. In the program above, the variable n retains the same value throughout the run, but the values of price and cost change each time through the **FOR** loop (explained in Chapter 6 "Loop Structures").

Constants and variables may be combined with operators and functions to create expressions. ***Expressions*** are similar to formulas; they represent data values that are calculated during the program run based upon the values of their elements at the time of calculation. In the program above, an expression computes a new value for cost each time through the **FOR** loop. The program uses the previous value of cost on the right side of the equal sign; the new value of cost equals the old value plus the new price plus the sales tax.

Many programming languages have several different data types. True BASIC, however, simplifies the programming process by distinguishing between only two types of data: numbers and strings.

Thus, True BASIC uses numeric constants, numeric variables, and numeric expressions as well as string constants, string variables, and string expressions. This chapter introduces you to each of these data representations. You will find more formal definitions of these concepts in Chapter 17.

Numeric Values

As far as the programmer is concerned, True BASIC treats all numeric values equally. It does not force you to distinguish between integer and real values or limit the size of particular values.

If you are familiar with one of the many programming languages that forces these distinctions and limits, then you can treat each numeric value in True BASIC as the equivalent of a double-precision floating value. If that value may be more appropriately interpreted as an integer value, True BASIC will convert it internally. For calculating memory requirements, however, you should assume that each numeric value will occupy eight bytes (which is a standard IEEE format for representing numbers).

Constants

Numeric constants must contain at least one digit. They may contain a decimal point, and they may start with a plus or minus sign. Examples are:

```
12                   3456543
3.1416               -123                -.0003
```

You may not use commas or spaces within numeric constants. Thus, 1,234,567 or 12 345 are not legal numeric constants.

You may also use exponential notation (sometimes called scientific notation) to represent numeric constants. In exponential notation, an ordinary numeric constant is followed by the letter e and an integer. The integer designates the power of 10 that multiplies the number. Here are some examples of this, along with the same number in ordinary notation:

```
1e3                         1000
12.3e10             123000000000
1e-3                             .001
1.234e-5                         .00001234
```

Variables

A numeric variable represents a numeric value, as do the letters “x” or “y” in algebra. The name of the numeric variable may be as long as necessary. Numeric variable names must start with a letter and may contain digits and the underscore character (_). Since spaces and hyphens are not allowed in variable names, the underscore character is often used to create hyphenated or multi-word names. Some examples of valid numeric variable names are:

```
i
last
x3
FirstNumber
next_in_line
```

The same rules apply for names of numeric arrays, numeric functions, and subroutines, but you may not use the same name to represent two different things. In other words, if you have a numeric variable called score, you may not also have a numeric array or function with the same name. Remember that True BASIC treats capital and lowercase letters identically, so that last and Last are the same variable.

When you run a program, True BASIC sets the initial value of all numeric variables to 0. You may of course use the **LET** statement or other assignment statements to “initialize” variables to any value you wish. In fact, it is a good habit to initialize all variables near the beginning of the program, even if you set them equal to 0.

Expressions

You may use constants and variables to build numerical expressions. The following arithmetic operators are available:

Arithmetic Operators

Operator	Meaning
+	addition
-	subtraction
*	multiplication
/	division
^	exponentiation
( )	parentheses

True BASIC follows the usual algebraic conventions. And, as in algebra, you may use the plus and minus signs as in +5 or –(x+y).

The following are examples of numeric expressions:

```
last - first + 1
numerator / denominator
-(z+2.35) + abc*def^2
(x^2-3) / (a+5)
```

The ***order of evaluation*** is the order in which operations within an expression are carried out. In True BASIC, the order of evaluation follows a common mathematical convention. Exponentiations are carried out first, from left to right. Multiplications and divisions are computed next, from left to right. Additions and subtractions, from left to right, are carried out last. You may use parentheses to achieve a different order (or simply to clarify the conventional computation). The following examples illustrate these rules:

Order of Evaluation

Expression	**Computed As**	**Result**
5–4*3	5–(4*3)	–7
2^3^2	(2^3)^2	64
–3^2	–(3^2)	–9
6/2*3	(6/2)*3	9
6/(2*3)	6/(2*3)	1
6+4/2+3	(6+(4/2))+3	11
(6+4)/(2+3)	(6+4)/(2+3)	2

Compare the last two examples in particular. Omission of parentheses in the denominator is a common error.

The precision with which True BASIC evaluates numeric expressions may vary from machine to machine, but generally adheres to the IEEE eigh-byte standard, which yields about 15 significant digits. The range of numeric values that can be handled also may vary, but at least is in the range 1e-300 to 1e300.

When a computed value is too large to be represented by True BASIC, an ***overflow error*** occurs. If the value computed is too close to 0, True BASIC substitutes 0, and no error occurs.

The program at the beginning of this chapter used numeric constants, variable, and expressions. Here is another simple program illustrating various representations of numeric data:

```
LET cord = 4 * 4 * 8                    ! Cubic feet
LET width = 7.25                        ! Feet
LET height = 5.5                        ! Feet
LET log_length = 16                     ! Inches
LET depth = (log_length/12) * 3
PRINT "Your woodpile contains";
PRINT (depth * height * width) / cord ; "cords of wood."
GET KEY k                               ! Press any key to end program
END
```

When run, this program produces the output:

```
Your woodpile contains 1.24609 cords of wood.
```

String Values

String values are composed of characters. A ***character*** is a single letter, digit, punctuation mark, or other mark as allowed by your computer. You may use any character available on your computer as part of a string, although you may not be able to type all the available characters from your keyboard. For a listing of the set of characters available on your computer, see Appendix A.

Each character occupies one byte of memory, and the size of a string in bytes is equal to the number of characters it contains. True BASIC allows strings to be as long as necessary, limited only by the available memory. On most operating systems the upper limit is more than four gigabytes; on Windows 3.1 it is about sixteen million characters.

Constants

A string constant is any sequence of characters enclosed in quotation marks. The **PRINT** and **INPUT PROMPT** statements in the two sample programs earlier in this chapter contain string constants. Other examples are "x", "first", "2001", "This is a sentence." and "X#*q30m". Within string constants, True BASIC does distinguish between capital and lowercase letters. Thus, "last" and "Last" are different string values.

A ***null string*** (or empty string) is a string that contains no characters. You represent a null string as a pair of quotation marks with no space between them: "". A space between the quotation marks represents a string containing a single space character, not the null string.

To include a quotation mark within a string constant, you must enter the quotation mark twice. Thus, "x""y" is a three-character string, with a quotation mark as the middle character. A more interesting example is:

```
"He said ""I don't believe it!"" and smiled."
```

Variables

The names of string variables differ from numeric variable names simply in that they end with a dollar sign ($). Otherwise, the rules for naming them are the same. Examples are:

```
name$
first_name$
week31$
Who_knows_what$
```

You use the same rules for names of string arrays and string-valued functions, but you may use a particular name for only one kind of object. In other words, if you have a string variable called *name$*, you may not also have an array with that name.

When you run a program, True BASIC sets the initial value of all string variables to the null string (""). As with numeric variables, it is a good habit to initialize all your string variables at the beginning of a program, even if you set them to the null string.

Expressions

Two operations let you build more complex string expressions out of constants and variables: concatenation and substrings.

Concatenation is the process of adding one string to the end of another, or gluing strings together. You use the ampersand sign (&) for concatenating two string values. Thus the string expression:

```
"Mo" & "men" & "tum"
```

produces the string "Momentum".

You may also use a substring expression to specify a portion of a string. A ***substring expression*** consists of a string constant, variable, or expression followed by [a:b], where a indicates the starting character of the substring and b the ending character. Thus,

```
"Momentum"[3:5]
```

represents the string "men". You may use parentheses instead of square brackets, as in:

```
"Momentum"(3:5)
```

In the expression [a:b], if b is larger than the number of characters in the string, then True BASIC uses the length of the string in place of b. If a is less than 1, then True BASIC substitutes a value of 1. If a is larger than the length of the string, or a is greater than b, the result is an empty string. Thus, any values of a and b are legal. For example, the program:

```
PRINT "House"[3:100]
PRINT "House"[-5:20]
PRINT "House"[4:4]
PRINT "House"[5:3]
PRINT "Done"
END
```

produces the following output:

```
use
House
s

Done
```

Note that the substring expression in the fourth line returns the null string because a is larger than b, and thus a blank line is printed.

You may use parentheses to control the order of evaluation when combining substring expressions with other string expressions. If you don't use parentheses, substring extraction occurs before concatenation. Thus,

```
"Abcde" & "fghijklm"[3:7]
```

equals "`Abcdehijkl`", while

```
("Abcde" & "fghijklm")[3:7]
```

equals "`cdefg`".

Here's a simple program that uses string constants, variables, and expressions:

```
LET first_name$ = "Abraham"
LET last_name$ = "Lincoln"
PRINT first_name$ & " " & last_name$ & ", 16th President of the US"
PRINT "Subscription Code:  " & last_name$[1:4] & first_name$[1:1]
END
```

It produces the following output:

```
Abraham Lincoln, 16th President of the US
Subscription Code:  LincA
```

Assignment Statements

The **LET** statement is the primary means of assigning values to variables. In assigning values you must assign numeric values to numeric variables and string values to string variables. (True BASIC does not perform automatic type conversions.) Examples of numeric assignments are:

```
LET e = 2.718282828
LET answer = Sin(pi/2) * Exp(-x^2)
LET length = last - first + 1
LET k = k + 1
LET i, j = 2
```

In an assignment statement, the expression to the right of the equal sign (=) is evaluated first, and the resulting value is assigned to the variable to the left of the equal sign. This means that you can use the variable appearing on the left in the expression on the right, and its old value will be used in the expression. In the next to the last example above, the present value of k increases by one and becomes the new value of k.

Note that a simple variable may only contain a single value. Thus, assigning a value to a variable will completely overwrite that variable's previous value. (See Chapter 9 on "Arrays and Matrices" for information on variable structures that contain multiple values.)

You may specify more than one variable name to the left of the equal sign, as long as you separate names by commas. In this case, the resulting value of the expression on the right will be assigned to each variable listed on the left. In the last example above, both i and j are set equal to 2.

Similarly, values of string expressions may be assigned to string variables:

```
LET name$ = "George Washington"
LET b$ = a$[3:7]
LET answer$ = answer$ & def$
LET a$, b$ = ""
```

You may also use the **LET** statement to change a portion of a string. To do so, you simply put a substring expression to the left of the equal sign. For example, the code fragment:

```
LET a$ = "bookkeeper"
LET a$[2:4] = "ee"
```

creates the string "beekeeper". The string "ee" replaces the second through fourth characters "ook".

If, in an assignment to x$[a:b], the value of b is less than the value of a, then True BASIC makes an insertion *before* character number a. For example, the code fragment:

```
LET x$ = "hose"
LET x$[3:2] = "u"
LET x$[1:0] = "The "
```

will result in the string x$ with the value "The house". Note that when you make assignments to substrings, the length of the string may change.

If you start your program with an **OPTION NOLET** statement, you can omit the LET keyword in assignment statements.

[!] Note: We urge caution in the use of the OPTION NOLET statement, since error messages may be less clear when it is in effect. The OPTION NOLET statement also destroys the simple structure of True BASIC, in which each statement starts with a keyword.

Although the **LET** statement is the most straightforward and commonly used method of assigning values to variables, True BASIC provides several additional ways to assign values to variables. There are several forms of the **INPUT** statement, which allow the person running the program to specify values for variables. The **READ** statement lets the program read data that exists elsewhere in the source code. And the **MAT** statement assigns values to entire arrays. In addition, the **INPUT** and **READ** statements may be used to assign values to variables based upon the contents of data files. These statements are described in greater detail in later chapters.

CHAPTER

3

Output Statements

Generally, a program produces some form of output to let the user know what the "answer" is. In True BASIC the primary mechanism for generating output is the **PRINT** statement.

The **PRINT** statement displays textual output on the computer screen. This textual (as opposed to graphical) output may be string constants; the values of variables, arrays, functions, or expressions; or any combination you specify. You may also use the **PRINT** statement to send output to a printer or a file.

The **PRINT USING** statement lets you format your output in a more careful, sophisticated way. For instance, it lets you specify the number of digits to use for printing numeric values or the exact alignment for printing string values.

This chapter introduces some simple options for sending textual output to the computer screen, a printer, or a file using the **PRINT** and **PRINT USING** statements. Other methods of producing output are described in Chapter 12 "Files for Data Input and Output," Chapter 13 "Graphics," and Chapter 14 "Interface Elements."

Basic Printing

To understand how the **PRINT** statement works, you must first understand the concept of the text cursor. The ***text cursor*** is the point on the screen at which the next text to be printed will appear. When True BASIC first opens a window, the text cursor is in the upper left-hand corner of that window. Thus, if your program prints to that window, the text starts in the upper left-hand corner.

[!] Note: True BASIC automatically opens an output window for simple programs; Chapters 13 "Graphics" and 14 "Interface Elements" describe how you can create and open additional windows. The text cursor itself is normally not visible although you can control its location. The text cursor appears on the screen only when an input statement expects a response from the user; the next chapter describes input statements.

The **PRINT** statement displays the value of expressions at the current text cursor. For example, the code fragment:

```
LET name$ = "Rumplestiltskin"
LET abcd = 1999
LET x = 48
LET y = 24
PRINT name$
PRINT abcd
PRINT (x+y)/2
```

will print a string and two numbers:

```
Rumplestiltskin
 1999
 36
```

Each of these items, the string and the two numbers, appears on a separate line. This is because each **PRINT** statement normally prints an end-of-line character after printing the specified value. This end-of-line character moves the text cursor to the beginning of the next line.

Controlling Line Breaks

You may also use semicolons or commas to combine several items in one **PRINT** statement and to prevent the automatic end-of-line character after a **PRINT** statement. The use of the semicolon to print items consecutively is described below; the use of the comma to print in columns is described in the next section.

At the end of the **PRINT** statement, the semicolon suppresses the end-of-line character. For example, the code segment:

```
PRINT "The numbers are ";
PRINT (x-y)/2;
PRINT "and ";
PRINT (x+y)/2
```

would print the following output if x equals 3 and y equals 7:

```
The numbers are -2 and  5
```

You may also use semicolons to combine expressions to be printed consecutively by one **PRINT** statement. When several numeric and/or string expressions are listed in a single **PRINT** statement, they are often referred to as that statement's ***print items***. The punctuation mark used to separate the print items (in this case the semicolon) is called a ***print separator***. For example, the following statement, which contains four print items and three print separators, is equivalent to the four **PRINT** statements in the preceding example:

```
PRINT "The numbers are "; (x-y)/2; "and "; (x+y)/2
```

Since there is no punctuation at the end of the last **PRINT** statement in the first example or the at the end of the **PRINT** statement in the second example, True BASIC will move the text cursor to the beginning of the next line after it carries out those statements. Therefore, the output of the next **PRINT** statement would start on the next line.

PRINT statements are often used inside a loop to print a series of values. (Loops are discussed in detail in Chapter 6 "Loop Structures.") Here is a simple example:

```
FOR n = 1 to 100 step 2
    PRINT n;
NEXT n
PRINT
PRINT "Done"
END
```

The semicolon used at the end of the first **PRINT** statement tells True BASIC to leave the text cursor at the end of each number printed, causing the odd numbers to appear, one after the other, on the same line. (True BASIC puts spaces around numbers as described below.) If the next number would go beyond the current margin, True BASIC prints it on a new line.

The second **PRINT** statement, which has nothing after it, simply prints the end-of-line character, moving the text cursor to a new line. Without it, future output would continue at the end of the list of odd numbers, which may be in the middle of a line. Because of this blank, or vacuous, **PRINT** statement, the word "Done" appears on a new line.

The output of this program would look something like this:

```
 1  3  5  7  9  11  13  15  17  19  21  23  25  27  29  31  33  35  37  39
 41  43  45  47  49  51  53  55  57  59  61  63  65  67  69  71  73  75  77
 79  81  83  85  87  89  91  93  95  97  99
Done
```

Printing Blank Lines

You can also use vacuous **PRINT** statements to insert blank lines into your output. Because each **PRINT** statement with no ending punctuation moves the text cursor to the beginning of the next line, such statements are equivalent to adding extra "line-feed" characters. In the following version of the loop that prints odd numbers, there would be a blank line before the string "Done":

```
FOR n = 1 to 100 step 2
    PRINT n;
NEXT n
PRINT             ! "Turns off" semicolon
PRINT             ! Prints blank line
PRINT "Done"
END
```

Conventions for Printing Numbers and Strings

True BASIC follows certain conventions to print numbers in a convenient format. Positive numbers and zero start with a space, while negative numbers start with a minus sign. All numbers end with a space, so that they do not run together when you use semicolons to separate output.

If a number can be represented as an integer of no more than twelve digits, the number is printed as an integer. If a number that is not an integer can be represented by eight digits and a decimal point, that form is used, but trailing zeroes after the decimal point are not printed. If none of the above apply, then the number is printed in exponential (scientific) notation, as explained in Chapter 2 "Constants, Variable, and Expressions." Values represented in decimal format that contain many decimal places are rounded to eight significant digits before printing. The following examples show how all these cases would be printed:

Printing of Numeric Values

Value	PRINT output
123456789012	123456789012
1234567890123	1.2345679e+12
0.12345678	.12345678
0.123456789	.12345679
12345.6789123	12345.679

Occasionally, you may see a numeric value printed as an integer followed solely by a decimal point; this indicates that the value is not a "true" integer, but is rather a real value that is extremely close to the displayed integer value. You should interpret such values as approximations.

If you need numeric values printed with greater accuracy or in a different format, use the **PRINT USING** statement, discussed later in this chapter.

[!] Note: These formatting rules simply govern the printed accuracy of numeric values; they do not affect the accuracy of the values' internal representations. True BASIC always maintains the highest possible degree of accuracy for its internal representations of numeric values.

In contrast to numbers, strings are always printed "as is." For example, the following program fragment:

```
DIM day$(7)
MAT READ day$
DATA Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
FOR i = 1 to 7
    PRINT day$(i);
NEXT i
PRINT
```

prints the names of the days of the week all run together:

```
SundayMondayTuesdayWednesdayThursdayFridaySaturday
```

If you want spaces, you should include them in the string, or you may print a string made up of one or more spaces, as in the example below. This version of the program fragment inserts spaces to make sure that the days do not run together:

```
DIM day$(7)
MAT READ day$
DATA Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday
FOR i = 1 to 7
    PRINT day$(i); "   ";
NEXT i
PRINT
```

It would produce the following output:

```
Sunday   Monday   Tuesday   Wednesday   Thursday   Friday   Saturday
```

Erasing Output

If printed output reaches the bottom of the output window, lines are automatically scrolled off the top of the window to make room for more text at the bottom. You can erase the output window at any time with a **CLEAR** statement:

```
CLEAR
```

The **CLEAR** statement erases all existing output and moves the text cursor back to the upper-left corner of the output window. If a new background color has been set (see Chapter 13, “Graphics”), **CLEAR** also fills the window with that new color.

Printing in Columns

Along with the semicolon, True BASIC allows the comma as a print separator. The comma instructs True BASIC to move the text cursor to the beginning of the next print zone before printing the next print item. Thus, you can use commas to generate columns of textual output.

Print zones are logical divisions of the current window that divide the area between the left edge of the current window and the margin into equal sized “columns.” The ***margin*** of a window is the maximum length of a line of text in that window. Both margins and print zones are specified as a number of fixed-width characters. (True BASIC normally prints output using fixed-width fonts rather than the variable-spaced fonts used with many word processors.)

Unless a program specifies otherwise, True BASIC examines the output window, establishes a default margin at its right edge, and sets up print zones with a width of sixteen characters each. If the margin is not evenly divisible by sixteen, then the right-most print zone will be less than sixteen characters wide.

Print zones are very convenient for tabular output. For instance, if you replace the semicolon with a comma in the earlier example that prints odd numbers, the numbers are printed in columns. Another example program is:

```
PRINT "Number", "Square", "Cube"
FOR n = 1 to 15
    PRINT n, n^2, n^3
NEXT n
END
```

The output of this program would be displayed in three columns, with the header labels lined up with the corresponding numbers (remember that True BASIC prints a space before each positive number):

```
Number          Square          Cube
 1               1               1
 2               4               8
```

```
  3               9               27
  4               16              64
 ...
```

If you attempt to print past the margin on a given line, True BASIC prints as much as possible on the line, then moves to the next line.

The **SET MARGIN** and **SET ZONEWIDTH** statements let you customize the default settings. For example,

```
SET MARGIN 70
SET ZONEWIDTH 10
```

will limit a line to 70 characters and provide seven zones of 10 characters each. You may specify the margin by as large a positive integer as you wish. The zonewidth cannot be greater than the margin.

The margin may exceed the limits of the current window. In such a case, True BASIC will continue printing on a line as required until it reaches the margin, but you will not be able to see any text that lies beyond the right edge of the window.

You may use the **ASK MARGIN** and **ASK ZONEWIDTH** statements to find out the current settings, as in:

```
ASK MARGIN m
ASK ZONEWIDTH z
```

If you had issued the two **SET** statements above, these two **ASK** statements would set m equal to 70 and z equal to 10.

Printing at Specific Screen Locations

As you have seen, True BASIC prints text at the text cursor location, and semicolons and commas in the **PRINT** statement control the placement of the text cursor after an item is printed. These **PRINT** conventions are quite useful for controlling the cursor when you want the next text to follow the current text.

Often, however, you may want to print text at particular positions on the screen that may not relate sequentially to other text items. The **SET CURSOR** statement and the **TAB** function let you move the text cursor location to control where the next **PRINT** statement output will appear; these are described in this section. Other methods of printing text at specific screen locations are provided by the **PLOT AT** statement described in Chapter 13 "Graphics" and the True Controls routines described in Chapter 14 "Interface Elements."

The **SET CURSOR** statement lets you put the text cursor anywhere you wish. With the **SET CURSOR** statement, you specify the row and column (or character) position of the text cursor within the current window. For example, the statement:

```
SET CURSOR 10, 43
```

will move the text cursor to row 10 and column 43 (or the 43rd character position in the 10th row) of the current window. The statement:

```
SET CURSOR 1, 1
```

moves the cursor to the upper left-hand corner of the current window. Rows are counted from the top and columns from the left. Each row is as high as necessary to accommodate the highest character in the character set, and each column is wide enough to accommodate the widest character, if you are using a variable-spaced font.

You can find the current cursor location with the following form of the **ASK CURSOR** statement:

```
ASK CURSOR row, col
```

Here row and col are assigned the current row and column position of the text cursor.

Remember that the text cursor itself — normally a flashing line or box — is visible only when the program expects the user to provide some input. Chapter 4 on "Input Statements" explains how you can use the **SET CURSOR** statement to turn the cursor on or off; Chapter 14 on "Interface Elements" describes library routines that let you change the shape of the cursor.

If you specify a row or column number that is outside the current window, an error results. If you need to know how many rows and columns exist in the current window, the **ASK MAX CURSOR** statement will tell you:

```
   ASK MAX CURSOR maxrow, maxcol
```

This statement will set maxrow to the largest row number and maxcol to the largest column number possible for the current window. Because window size can vary depending on the computer running the program, and can sometimes be altered by the user, this statement lets you avoid "Cursor out of bound" errors.

Here is an example program that specifically positions the text cursor. It fills the window with birthday greetings, with each line indented more than the line above.

```
   ASK MAX CURSOR maxrow, maxcol          ! Number rows, columns
   LET text$ = "Happy Birthday!"
   LET slack = maxcol - Len(text$)        ! Extra spaces on line
   LET ind = Int(slack / maxrow)          ! Indentation

   FOR row = 1 to maxrow                  ! Use every row
       SET CURSOR row, row*ind            ! Each line indented
       PRINT text$;
   NEXT row
   END
```

Notice the semicolon that ends the **PRINT** statement. This is necessary to allow the final birthday greeting to appear in the last row of the window. Without it, True BASIC moves the text cursor to the beginning of the following line after printing the last greeting, forcing the window's contents to scroll. Try running the program with and without this semicolon to see the difference.

Sometimes you wish to position the text cursor in the midst of a **PRINT** statement. You can use the **TAB** function to do this, as in the following example:

```
   PRINT name$; Tab(4, 20); x
```

This statement will print the value of name$, then set the cursor to row 4 and column 20 and print the value of x there. The semicolon after the **TAB** function indicates that the cursor should stay at the desired position.

The **TAB** function with a single value moves the cursor to the specified column on the present line. If it is already past this column, the cursor moves to that column on the next line. This version of **TAB** is useful for creating variable-width zones. (Remember that print zones are of equal width.) For example, the following code segment:

```
   PRINT "Name"; tab(25); "Age"; tab(30); "Phone number"
   PRINT
   FOR i = 1 to n
      PRINT name$(i); tab(25); age(i); tab(30); phone$(i)
   NEXT i
```

creates output in three columns. The first column may contain up to 24 characters, the second up to 5 characters, and the third column may extend to the current margin. Some possible output follows:

```
   Name                    Age  Phone number

   Sallie Smythe            12  907-333-4352
   Olaf Larsen              56  703-256-2626
   Juan Martinez            43  802-778-9991 extension 445
   Pierre La Fontaine       27  602-664-1221
```

Formatted Printing

True BASIC also provides the **PRINT USING** statement as a sophisticated and flexible way of formatting output. The **PRINT USING** statement works differently from the **PRINT** statement in that it ignores print zones and follows only the format string you specify with it. A ***format string*** is a string expression that determines the format of the output exactly, using fields (composed of place holders) and, possibly, constants (composed of characters).

This section illustrates some of the more common uses of the **PRINT USING** statement. Appendix D contains a more formal discussion of the details.

Numeric Format Fields

First consider a simple example of formatting numeric values:

```
PRINT USING "-###.###": -Pi, 9^3
```

The format string "`-###.###`" causes the **PRINT USING** statement to print a numeric value with exactly three decimal places, adding zeroes or rounding if necessary. It allows up to three places before the decimal point and reserves a space for the sign if the value is negative. Leading zeroes will not be printed. Thus, the output for the above statement would appear as:

```
-  3.142 729.000
```

You may define the format string within the **PRINT USING** statement, or you may define a string variable for use with **PRINT USING**. The following two statements produce the same output as the example above:

```
LET format$ = "-###.###"
PRINT USING format$: -Pi, 9^3
```

In these examples, the format string contains only one field. A ***field*** is the format specification for a single print item. Since there is only one field but two print items, the field is used twice — once for each print item. Notice that each item uses eight spaces with no extra spaces separating the two print items. True BASIC follows the format string precisely and adds no additional spaces regardless of data type or punctuation between print items. If you want spaces to precede or follow each number you must include those spaces within the format string:

```
LET format$ = "  -###.###  "
PRINT USING format$: -Pi, 9^3
```

The ***field length*** is the number of character positions that the value will occupy when printed. In the format string "-###.###", the field length is eight spaces, which includes the spaces reserved for the sign and the decimal point. Numeric values are aligned with the decimal within a field (that is, they are printed decimal justified). This means that if several numbers are printed with the same format string, one under the other, any decimal points (or commas) will line up. If no decimal point is present in the field, numeric values are aligned with the right edge of the field (right justified).

In the format string "-###.###", the first position in the field is reserved for a negative sign; it may not be occupied by a digit. Here are some examples; notice the last one in particular.

Examples of Output Formatted with the String: "-###.###"

Number	Output
17	17.000
17.1234	17.123
-123.4687	-123.469
-17.2	- 17.200
12345	********

If a number cannot be printed in the specified format, asterisks are printed instead, as in the last example. The field's length determines the number of asterisks printed.

Format fields are composed of place holders. ***Place holders*** are characters that reserve space within the field for a specific character or range of characters that may appear in the printed value. The format string "-###.###" contains three different place holders: the minus sign (–), the pound sign (#), and the decimal point (.).

There are other place holders that you may use to compose fields in format strings. The following table summarizes all the place-holder characters valid for numeric values:

PRINT USING Place Holders

Place Holder	Reserves Space For
Leading Characters	
-	minus sign if required, blank for positive numbers; repeat for “floating” minus sign
+	plus or minus sign always printed; repeat for “floating” sign
$	dollar sign; repeat for “floating” dollar sign
Digit Characters	
#	digit or leading space (or negative sign if no leading sign in format string)
%	digit or leading zero
*	digit or leading asterisk (*)
Other Characters	
.	decimal point; digits to the right of the decimal point are always printed, rounded if necessary
,	comma or blank if there are no digits to the left
^	exponent part of scientific notation; must use from three to five carets (^)

Here are some examples showing how place holders would print the variable answer with the value 1234.5:

Examples of PRINT USING Place Holders

PRINT USING Statement	Output
`PRINT USING "#,###,###" :      answer`	`   1,235`
`PRINT USING "-#####.##" :      answer`	`  1234.50`
`PRINT USING "+#####.##" :      answer`	`+ 1234.50`
`PRINT USING "+#####.##" :     -answer`	`- 1234.50`
`PRINT USING "$##,###.##" :     answer`	`$ 1,234.50`
`PRINT USING "$-#,###.##" :    -answer`	`$-1,234.50`
`PRINT USING "%%%,%%%.%%" :     answer`	`001,234.50`
`PRINT USING "******" :         answer`	`**1235`
`PRINT USING "#.#^^^" :         answer`	`1.2e+3`

Like the minus (-) sign, the plus (+) and dollar ($) signs reserve spaces for leading characters. When you use the plus sign, the appropriate sign — positive or negative — is always printed. You may use the dollar sign along with the plus or minus sign.

Like the number (or pound) sign (#), the percent (%) and asterisk (*) characters reserve spaces for digits and a leading minus sign if required. But they also tell True BASIC to always print something in that space: the percent character prints leading zeroes and the asterisk prints leading asterisks if necessary. You may not mix different digit characters within one format item. Thus, “####” and “%%%%” describe four-character fields, the first with leading spaces and the second with leading zeroes, but “%%##” is not allowed.

The comma reserves a place for a comma if appropriate; it is printed only when there are digits to the left of the comma. The caret (^) reserves spaces for scientific notation. You must reserve from three to five spaces — for the “e”, the plus or minus sign, and one to three digits. Notice that True BASIC rounds the numeric value when necessary to fit the format

On occasion, you may want a leading sign to appear immediately to the left of the left-most digit, rather than in a fixed position as in the examples above. You can easily do this using a ***floating place holder***. To indicate a floating place holder, you use a place holder that normally reserves a particular space for a single character, such as the plus or dollar sign, but you repeat it over a range of spaces. The position of the specified character “floats” within this range as needed so that it always appears as far right in the range as possible.

Consider these examples where the value of answer is 2.34:

Examples of Floating Place Holders

PRINT USING Statement	Output
`PRINT USING "-###.##": -answer`	`-  2.34`
`PRINT USING "---#.##": -answer`	`-2.34`
`PRINT USING "$###.##": answer`	`$  2.34`
`PRINT USING "$$$$.##": answer`	`$2.34`

Constants and Multiple Fields in Format Strings

Any character that is not a valid place holder (including a space) that appears in a format string is interpreted literally (with the exception of the string place holders < and > discussed below). Thus, you can include text as a ***constant*** in a format string, as in:

```
PRINT USING "The answer is #,###.###": answer
```

Here the constant text is printed as it appears, and the value of the numeric variable answer is printed according to the format field. For example:

```
The answer is 5,439.780
```

You can also include more than one field in a format string. The print items will be inserted into the fields in relative order from left to right. For example, the following formats print a table of sines and cosines:

```
LET format$ = "-#.###   -#.######   -#.######"
FOR x = 0 to 9 step 2
    PRINT USING format$: x, sin(x), cos(x)
NEXT x
```

The spaces between the format fields provide equal spacing between the columns of output:

```
      .000     .000000    1.000000
     2.000     .909297   - .416147
     4.000   - .756802   - .653644
     6.000   - .279415     .960170
     8.000     .989358   - .145500
```

If there are more print items than there are fields, then the fields will be reused from left to right until all of the print items have been printed. If there are more fields than there are print items, then the format string will end at the beginning of the unused field. For example:

```
LET tot4 = 796
LET tot5 = 1113
PRINT USING "Fiscal Yr ####: $$#,###K    " : 1994, tot4, 1995, tot5, 1996
```

produces the following output:

```
Fiscal Yr 1994:  $  796K    Fiscal Yr 1995:  $1,113K    Fiscal Yr 1996:
```

Here are some more examples of **PRINT USING** formats that use multiple fields:

Examples of Multiple PRINT USING Fields

Statement	Output
`PRINT USING "+#.# on $#,###.##"; i, d`	`+9.5 on $3,527.30`
`PRINT USING "## and ": 12.3, 12.7, 14`	`12 and 13 and 14 and`
`PRINT USING %% plus %%": 3`	`03 plus`

Do take care in designing your format strings and the **PRINT USING** statements that use them. Consider the following nonsense program:

```
PRINT USING "Answer #1 is ##.#. Good job!": 2.6
END
```

When run, this program produces the unexpected output:

```
   Answer 31 is
```

The reason for this output is quite simple: the pound sign before the digit 1 (intended to represent the number sign) is interpreted as the first numeric field. Therefore, True BASIC rounds the value of 2.6 to 3 and prints it in this field. Other characters, including the 1, are printed as constants. Since no second print item is available for the second field (which we intended to be the only field), the format string is printed only to the first character in that field.

Now consider the following potential solution:

```
   PRINT "Answer #1 is ";
   PRINT USING "##.#. Good job!": 2.6
   END
```

Since the problematic pound sign is no longer part of the **PRINT USING** statement, this form of the program solves that part of the problem, but it also brings up another problem. When you try to run this program, you get an error message claiming that you have a “Badly formed USING string.” This means there is now something illegal in the format string.

This problem is a little trickier to find, but it makes sense. The second period (intended to end the first sentence) is directly adjacent to the format field. Since it is a valid place holder, True BASIC considers it part of the field. However, this results in a field with two decimal points which is not possible, so True BASIC generates the error. (This error didn’t occur in the first example because True BASIC stopped using the string before it reached that point.)

To fix this problem, modify the program as follows:

```
   PRINT "Answer #1 is ";
   PRINT USING "##.#": 2.6;
   PRINT ". Good job!"
   END
```

When you run this version of the program it produces the output originally intended:

```
   Answer #1 is  2.6. Good job!
```

Notice that a semicolon at the end of a **PRINT USING** statement has the same effect as at the end of a **PRINT** statement.

String Format Fields

True BASIC also lets you format string values with the **PRINT USING** statement, but the options are more limited. The **PRINT USING** statement with string values is most useful if you wish to define fixed length fields and control the justification (alignment) of the string values within those fields.

You may print strings with any numeric format item. Unless you specify otherwise, True BASIC centers the string within the field, adding spaces if needed. (If necessary, there will be one more space to the right than to the left.) If the string is too long to fit the format field, asterisks are printed instead, just as with numbers.

You may also tell True BASIC to align a string to the left or right within the format field using two special string place holders, “<” and “>”:

String PRINT USING Place Holders

Place Holder	**Reserves Space For**
#	character or space
<	character or space; left justifies string value in field
>	character or space; right justifies string value in field

In case of more than one “<” and/or “>”, the left most one decides.

Here are some examples:

```
LET name$ = "zebra"
PRINT USING "##########": "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
PRINT USING "##########": name$         ! Center string
PRINT USING ">#########": name$         ! Align to right
PRINT USING "<#########": name$         ! Align to left
```

These statements produce the following output

```
**********
  zebra
     zebra
zebra
```

The ability to format strings with any valid numeric format field gives you added flexibility. The following code fragment shows how you can print titles over columns:

```
LET form$ = "<##################   ###   $###,###"
PRINT USING form$: "Name", "Age", "Salary"
FOR i = 1 to n
    PRINT USING form$: name$(i), age(i), salary(i)
NEXT i
```

This would produce output like the following:

```
     Name                   Age        Salary
     Frank Williams          56      $ 57,999
     Vicki Mantle            49      $113,400
     Rudy Garland            32      $ 85,000
```

Whenever possible, True BASIC will try to fit the print items to the format fields. However, you must remember that print items are associated with fields from left to right. True BASIC will not match string values with string fields and numeric values with numeric fields. It is up to you to ensure that print items appear in the appropriate order for the format fields.

The **PRINT USING** statement can be tremendously useful. Although this section introduces most common uses, you may wish to review the formal specifications in Appendix D at some point.

Printing to a Printer

At times, it is necessary or convenient for your programs to print to a printer rather than a window on the screen. Just as you can use the **PRINT** or **PRINT USING** statement to display the values of numeric and string expressions on the screen, you can instruct True BASIC to send that information to the printer instead. Before you do so, however, you must first open a channel to the printer.

A ***channel*** is a connection between your program and an input or output device, such as a printer or a file. Within your program, you can open channels to several such devices. The program distinguishes between the channels by channel numbers. A ***channel number*** is any integer value between 0 and 999, preceded by a pound sign (#); it indicates a specific channel to a device. Note that channel #0 is reserved for the default logical window (where True BASIC automatically sends output unless you specify otherwise) as discussed in Chapters 13 “Graphics” and 14 “Interface Elements.” Each open channel must have a unique channel number.

To use a channel, you must first open it with the **OPEN** statement. The following program opens a channel to the printer and prints to it:

```
OPEN #1: PRINTER
PRINT #1: "Beginning of list:"
FOR i = 1 to 20
    PRINT #1: i
```

```
NEXT i
PRINT #1: "End of list"
CLOSE #1
END
```

When you run this program, it sends all of its output to the printer; nothing appears on the screen. If you wanted the same information to appear on the screen as well, you would need to modify the program as follows:

```
OPEN #1: PRINTER
PRINT #1: "Beginning of list:"           ! Print to printer
PRINT "Beginning of list:"               ! Print to screen
FOR i = 1 to 20
    PRINT #1: i                          ! .. to printer
    PRINT i                              ! .. to screen
NEXT i
PRINT #1: "End of list"
CLOSE #1
PRINT "End of list"
END
```

In this program, the **PRINT** statements without a channel number send their output to the screen, while those that specify channel #1 send their output to the printer (because the **OPEN** statement associated channel #1 with the printer).

Some operating systems limit the number of channels that may be open simultaneously. Therefore it is good practice to use the **CLOSE** statement as shown above to close channels when you no longer need them. Once you close a channel, you may reuse that channel number for a different channel.

You may also use the **PRINT USING** statement to send formatted output to a printer. Consider the following program that prints a table of natural logarithms for the numbers 1 to 100 on a printer:

```
OPEN #1: PRINTER
LET form$ = "   ###    ##.###"
PRINT #1, USING form$: "Num", "Log"
FOR i = 1 to 100
    PRINT #1, USING form$: i, Log(i)
NEXT i
PRINT #1: "End of table"
END
```

You cannot use the **SET CURSOR** statement to position the printer's print head, nor does the **CLEAR** statement have any effect on printer output. You may, however, adjust the margin and zone width, and you may use print separators and the single-argument **TAB** function to position your output when sending it to the printer. The following lines demonstrate the proper formats:

```
PRINT #1: name$; Tab(25); age; Tab(30); phone$

PRINT #1: name$, age, phone$

SET #1: MARGIN 60

SET #1: ZONEWIDTH 6
```

Printing to a File

You may also use the **PRINT** statement to send textual output to a file. This is very similar to using the **PRINT** statement with a printer (as described in the previous section); the only major difference is the form of the **OPEN** statement used.

Consider the following variation on the example from the previous section:

```
OPEN #1: NAME "MyFile", CREATE NEWOLD
ERASE #1
PRINT #1: "Beginning of list:"
FOR i = 1 to 20
    PRINT #1: i
NEXT i
PRINT #1: "End of list"
CLOSE #1
END
```

This program opens a channel to the file named MyFile in the current directory (normally the directory in which the program is saved). If a file with this name already exists in the current directory, then it will be used; otherwise, a file with the name MyFile will be created. The CREATE NEWOLD clause in the **OPEN** statement specifies this behavior.

Since the program may have opened an existing file, you can be sure you are working with a “blank slate” by using the **ERASE** statement to erase the file’s current contents before continuing.

As with printing to a printer, you cannot use **SET CURSOR** to arbitrarily position the cursor within a file — and you must use an **ERASE** statement rather than **CLEAR** to remove the contents of a file. You may use the single-argument **TAB** function and the **SET MARGIN** and **SET ZONEWIDTH** statements to control spacing across each line of printing to a file. You may also use the **PRINT USING** statement to further control printing to files.

You will find much more information on using files in Chapter 12, “Files for Data Input and Output.”

CHAPTER

4

Input Statements

Most useful programs have some flexibility built into them; that is, they can produce different results when provided with different data. Commonly, the user of the program provides this data and thus has some control over the program's behavior.

In True BASIC, the simplest way to obtain data from the user is the **INPUT** statement. The data provided by the user may consist of numeric or string values, and it may come from the keyboard or a file.

This chapter introduces the fundamentals of user input. It discusses the **INPUT** statement, the **LINE INPUT** statement, and the **GET KEY** statement for getting single keystrokes. For information on graphical input from the **GET MOUSE** and **GET POINT** statements, see Chapter 13 "Graphics." For input from menus, buttons, check boxes, special edit fields, and dialog boxes, see Chapter 14 "Interface Elements."

Basic Input

Chapter 2 on "Constants, Variables, and Expressions" illustrates the use of the **LET** statement to assign data values to variables. Although it is extremely useful, the **LET** statement is limited in that you must know the value you wish to assign when you are writing the program. To change the value, you must change the program.

Since it is impractical and undesirable for the user to change the source code before each run, True BASIC provides the **INPUT** statement to let the user assign values to variables during the run. The **INPUT** statement complements the **PRINT** statement. While the **PRINT** statement lets your program give information (the "answer") to the user, the **INPUT** statement lets the user give information (the "question") to your program.

The **INPUT** statement pauses the program, prints a question mark (?) at the current text cursor position, and then displays the text cursor (if it is turned on). A visible text cursor indicates that the user must enter data before the program will continue.

The number and type of variables specified as input items determine the number and type of values the user must supply. The program may include as many input items as necessary in an **INPUT** statement, separating them with commas. Consider the following example lines:

```
INPUT a
INPUT x, y
INPUT name$, age, phone$
```

The first line expects the user to enter a single number, the second line expects two numbers, and the third expects three pieces of information — a string, a number, and another string (in that order).

The user must enter the proper number of items, of the proper types, in the proper order, and then press the Enter (or Return) key to tell the program to continue. Before continuing, the program matches the values entered with the list of input items from left to right. If the number and types of items do not match exactly, then True BASIC prints a message and asks the user to re-enter the data. If they do match, True BASIC assigns each entered value to the corresponding input item. Thus, if a user responds to the third example line above by typing:

```
? Chris Jones, 32, 555-4321
```

then the program will assign the string value "Chris Jones" to name$, the numeric value 32 to age, and the string value "555-4321" to phone$. Notice that the user must separate multiple items by commas.

Prompting the User

Although the user must enter the data correctly, the **INPUT** statement does not tell the user how many items to enter or what type they must be. You can help the user of your program by printing a ***prompt*** or description of the data expected before the **INPUT** statement. Here's an example:

```
FOR i = 1 to 10
  PRINT "Numerator, denominator";
  INPUT n, d
  PRINT n/d
NEXT i
```

The first **PRINT** statement prints the prompt, indicating that two numeric values are expected and that they should be separated by a comma. The prompt also tells the user that the first value will be used as the numerator, and the second as the denominator. The semicolon at the end of this **PRINT** statement ensures that the question mark printed by **INPUT** immediately follows the prompt. Thus, the program prints:

```
Numerator, denominator?
```

The pairing of **PRINT** and **INPUT** statements is so common that True BASIC provides a single statement that accomplishes both:

```
INPUT PROMPT "Numerator, denominator? ": n, d
```

The **INPUT PROMPT** statement prints the specified prompt in place of a question mark. The space after the question mark at the end of the prompt string helps to make the input easier to read on the screen.

Supplying Input

As noted above, the user must enter the correct number and type of responses, separating multiple items with a comma and ending with the Return or Enter key. In response to the following statement:

```
INPUT PROMPT "Item, number purchased, cost? ": item$, num, price
```

the user must enter three items — one string and two numbers:

```
Item, number purchased, cost? apples, 6, .49
```

True BASIC can handle common incorrect responses to **INPUT** statements. If the user enters too few or too many items or enters something other than a number for a numeric item, True BASIC prints a message stating the problem and asking the user to retype the entire input. For example, here are some incorrect replies to the above example:

Input Error Messages for: INPUT item$, num, price

User Response	Error Message
`apples, 6`	`Too few input items. Please reenter input line.`
`apples, 6, $.49`	`String given instead of number. Please reenter input line.`
`apples, 6, 4, .20`	`Too many input items. Please reenter input line.`

Notice that the error messages are fairly general. You as programmer should use prompts to make input requests as clear as possible.

If so many input items are requested that the input does not fit on a single line, the user may end the line with a comma. True BASIC will display a question mark at the beginning of the next line so the user can continue typing input.

You can also write the program so that the user can anticipate future input requests. If an **INPUT** statement ends with a comma, True BASIC does not complain about excess input; it saves any extra input, and uses it to fulfill future input requests. In the following example, the user may respond to the first prompt by typing ten numbers (separated by commas) and the program will work five division problems without any further prompt:

```
FOR example = 1 to 5
```

```
      INPUT PROMPT "Numerator, denominator? ": n, d,
      PRINT n; "/"; d; "="; n/d
   NEXT example
```

True BASIC strips off leading and trailing spaces from input items. If you wish to furnish a string that has either leading or trailing spaces, you must enclose the string value in quotes:

```
   ? "   February   "
```

A similar situation exists with a string that contains a comma. Since commas separate input items, you must use quotes around a string that contains commas. For example:

```
   ? "Washington, George"
```

If you omit the quotes, True BASIC would treat this as two separate input items.

The **INPUT** statement also requires repeated quotes if you wish to use quotes within a string enclosed in quotes. For example:

```
   ? "Eisenhower, Dwight David ""Ike"""
```

The **LINE INPUT** statement described below can alleviate some of the problems of entering complex strings. Whatever form of input statement you use, you must keep in mind the rules that govern the input of data and ensure that the user of your program is aware of these rules. Printed documentation is helpful, but **PRINT** statements and clearly worded input prompts are usually the best methods for assisting users unfamiliar with programming or True BASIC.

Programming for Errors

We've mentioned True BASIC's ability to recognize errors that violate the rules for supplying input and the programmer's responsibility for supplying adequate input prompts to prevent such errors. Allowing the user to assign values to variables during a run increases your responsibilities as a programmer in other ways as well.

When your program is "closed" to user input (such as a program that relies exclusively on **LET** statements), you have complete control. If errors are going to occur, it is quite likely that you will find them during the testing phase. When your program is open to user input, however, you have no control over the user's actions. Therefore, your program must be prepared for any eventuality; it should make every reasonable attempt to protect its users from their own folly.

As an illustration, consider the division program discussed above. This program works fine, until the user enters a value of 0 as the denominator. Since division by zero is mathematically undefined, True BASIC generates an error if your program attempts such an operation. Thus, if the user enters 0 for the denominator in this example, the program will stop and display the message "Division by zero."

This program could, and probably should, be rewritten to check for a denominator value of zero. If the program encounters such a value, it can simply print a message indicating that division by zero is undefined and skip that particular division problem:

```
   FOR i = 1 to 10
     PRINT "Numerator, denominator";
     INPUT n, d
     IF d = 0 then
        PRINT "Denominator cannot be zero; please re-enter."
     ELSE
        PRINT n/d
     END IF
   NEXT i
```

This process is known as "handling" the error, and it is discussed in detail in Chapter 16 "Error Handling."

Inputting Complete Lines

The previous sections show how you must use quotes to enter complex string values for the **INPUT** statement. However, those rules can be unnecessarily burdensome to the user of a program. When properly used, the **LINE INPUT** statement can simplify the process of entering strings with commas, quotes, or leading and trailing spaces.

The **LINE INPUT** statement can get input for string variables only. It expects the user to enter one line for each input item, signaling the end of a line with the Return (or Enter) key. The **LINE INPUT** statement assigns the entire contents of each line, including spaces and punctuation marks, to the corresponding string variable in the input list. For example, the statement:

```
LINE INPUT name$, street$, city_state_zip$
```

prints three question marks for the three lines of input it expects. The user need not be concerned with spaces, commas, or quotes within the lines:

```
? Joshua "Skip" Silverstein
? 1154 Wise Avenue, No. 16B
? Fairview, MA  01077
```

The **LINE INPUT PROMPT** statement is a variation that lets you specify a prompt string to be printed in place of the default question mark, as in the following example:

```
LINE INPUT PROMPT "Type a sentence: ": sent$
```

LINE INPUT statements also let the user input nothing by simply pressing the Return (or Enter) key. In the above request for name and address for example, the user could press just Return (or Enter) to enter a null (empty) string for street$ if they have no street address:

```
? Joshua "Skip" Silverstein
?
? Fairview, MA  01077
```

Because it can accept empty input, you can use the **LINE INPUT** statement to make the program pause until the user is ready:

```
LINE INPUT PROMPT "Press Return to continue.": s$
```

Here, it doesn't matter what the user enters; once they press the Return (or Enter) key, the program will continue.

Inputting Keys

With both the **INPUT** and the **LINE INPUT** statements, the user must press the Return (or Enter) key at the end of the input. Often, however, you want the user to press a single key as input, perhaps to determine what the program does next. The **GET KEY** statement lets your program obtain a single keystroke from the user.

The statement:

```
GET KEY z
```

waits until the user presses a key, then translates that key into a corresponding number and assigns that number to the variable z. The **GET KEY** statement does not display anything on the screen – it does not display a prompt, nor does it echo the value of the key pressed. However, if the cursor is currently on, it will be displayed while the **GET KEY** statement waits for a keystroke.

If the key pressed directly corresponds to an element of the current character set, the numeric code for that character is assigned to the specified variable. Special keys that do not correspond to elements of the standard character set, such as function and cursor keys, are assigned a number above 255.

Since all the operating systems under which True BASIC runs support the ASCII character set, you can assume that the standard character and punctuation keys on the "main" keyboard will return the same values regardless of the operating systems. Appendix A lists the ASCII character set with numeric code values. For example, if the user presses a lowercase "q", the number 113 is assigned to the **GET KEY** variable. (Note that lowercase and uppercase letters have different codes.)

Special keys such as function and cursor keys, however, may vary among types of computers. Also some operating environments use key combinations for accented or other special characters. For example, for some accented characters you would hold down the appropriate modifier key and press an associated key. You may then need to press a third key without holding the modifier key. The accented letter or special character will then appear. See the documentation supplied with your operating environment for a complete list of the characters you can type in this way. All such characters are represented by a unique character code and count as a single keystroke for the **GET KEY** statement.

You can use the following program to identify code numbers for special keys or key combinations under any operating system:

```
DO
   GET KEY k
   PRINT k
LOOP UNTIL k = 27        ! Escape key stops program
END
```

Run this program and press any key or modifier and key combination. The number corresponding to that key will appear on your screen. Remember that the Shift, Option, Alt, Control, and Command (C) keys can be used as modifier keys. Hold down the modifier key while you press another key to get the modified values. If you press a key before the **GET KEY** statement is executed, True BASIC will save it for later use — provided, of course, there is room in the key buffer as explained below.

You cannot read keystrokes that are shortcuts for active menu items; True BASIC executes the menu item instead. (Menus are described in Chapter 14 on “Interface Elements.”)

Useful Built-in Functions

The **CHR$** and **ORD** functions are often used with the **GET KEY** statement. The **CHR$** function returns the string character that corresponds to the numeric code provided as its argument. Thus, you can “translate” **GET KEY** input back to the character typed:

```
DO
   GET KEY k
   IF 32 < k and k < 127 then
      PRINT Chr$(k); " is"; k
   ELSE
      PRINT "Key is not a printable character"
   END IF
LOOP UNTIL k = 32    ! Space bar stops program
END
```

The **ORD** function returns the numeric character code of the single-character string provided as its argument. The resulting value matches the value returned by the **GET KEY** statement for printable characters. For example, if you want to stop the program when the user presses a “Q” or “q” you could use the **ORD** function to test the user input:

```
PRINT "Press Q if you wish to quit; any other key to continue.";
GET KEY quit
IF quit = Ord("Q") or quit = Ord("q") then STOP
```

For more about built-in functions, see Chapter 8 “Built-in Functions.”

An Example

Because the **GET KEY** statement pauses the program until the user responds, you can use it to force a program to pause until the user is ready to continue. Although you have already seen how to use the **LINE INPUT** statement for the same purpose, the **GET KEY** statement is particularly well suited to this task because the user may press any single key without a following Return (or Enter) key, and the input is not displayed on the screen.

If you want the program's future action to be based on which key the user presses, use a **GET KEY** statement in conjunction with a **SELECT CASE** structure as described in Chapter 5 on "Decision Structures." For example:

```
PRINT "Self-study menu. Press the appropriate letter"
PRINT "   (A)ddition        (S)ubtraction"
PRINT "   (M)ultiplication   (D)ivision"
SET CURSOR "off"                        ! Don't show text cursor for input
GET KEY choice
LET test$ = Lcase$ ( Chr$(choice) )      ! Translate to lowercase character
SELECT CASE test$
CASE "a"
     CALL Add                           ! User-defined subroutine
CASE "s"
     CALL Subtract                      ! User-defined subroutine
CASE "m"
     CALL Multiply                      ! User-defined subroutine
CASE "d"
     CALL Div                           ! User-defined subroutine
CASE ELSE
     STOP
END SELECT
```

Testing for Key Input

If the user response is optional, you want to test whether the user has pressed a key without stopping the program. You can do this by testing the special logical expression, KEY INPUT. KEY INPUT is true if there is a character waiting in the key buffer, and false otherwise. Characters go into the key buffer when the user presses a key before being prompted to do so (see below).

The following example lets the user end a long tabular output by pressing the escape key. Note that the escape key corresponds to ASCII number 27.

```
FOR x = 0 to 10 step .1
    PRINT x, Sin(x)
    IF KEY INPUT then
       GET KEY z
       IF z = 27 then STOP        ! Escape key pressed
    END IF
NEXT x
```

Once the user presses a key, KEY INPUT will be true until a **GET KEY** or some form of the **INPUT** statement is executed (since those are the only statements that remove keys from the key buffer).

The Key Buffer

The **GET KEY** statement actually gets the value of the key pressed from the key buffer, not directly from the keyboard. The ***key buffer*** is a reserved portion of memory that stores keystrokes as the user enters them. Because keystrokes go into the key buffer the instant they occur, you generally need not worry about a fast typist getting ahead of your program. Thus, the **GET KEY** statement waits for a keystroke only when the buffer is empty; if there is a keystroke in the buffer, **GET KEY** uses that keystroke. Each time it is executed, the **GET KEY** statement removes the "oldest" keystroke from the buffer.

Occasionally, you may want your program to clear out, or "flush," the keyboard buffer. This is useful if you are worried that the user may press more than one key, or if you want to eliminate the possibility of a stray keystroke being accepted as valid input. Whenever you need to flush the key buffer, you may use code similar to the following:

```
DO WHILE KEY INPUT
   GET KEY k
LOOP
```

Turning the Text Cursor On and Off

Along with a question mark and any prompt the program may print, True BASIC normally displays the text cursor as a flashing line or box when it expects input. The text cursor is displayed at its current position, which you can control with the **SET CURSOR** statement (see Chapter 3 “Output Statements”). If you use an **INPUT PROMPT** or **LINE INPUT PROMPT** statement, the prompt is printed at the text cursor, and the text cursor is moved to the right of the prompt string.

You can use a form of the **SET CURSOR** statement to control whether the cursor is displayed when input is required. The statement:

```
SET CURSOR "OFF"
```

makes the cursor invisible during input, while the statement:

```
SET CURSOR "ON"
```

makes it visible again when user input is expected.

The **ASK CURSOR** statement:

```
ASK CURSOR c$
```

lets you find the current state of the cursor. True BASIC assigns to the variable c$ the value “ON” or “OFF”. See Chapter 14 “Interface Elements” for a routine that lets you change the shape of the text cursor.

Inputting from a File

As with the **PRINT** statement, you can use the **INPUT** and **LINE INPUT** statements to obtain input from files. To do so, you need simply open a channel to a text file and specify that channel number with the **INPUT** statements that should take their input from the file rather than the keyboard. (See Chapter 3 for an introduction to opening a channel to a file; Chapter 13 “Files for Data Input and Output” describes the use of files in greater detail.)

As an illustration, consider the following program that displays the contents of a text file on the screen:

```
INPUT PROMPT "Name of file to display: ": fname$
OPEN #1: NAME fname$
DO WHILE MORE #1
   LINE INPUT #1: line$
   PRINT line$
LOOP
END
```

This program obtains a file name from the user (at the keyboard) and uses that value in the **OPEN** statement to open a channel to that file. Once the channel is open, the **DO** loop will repeat as long as there is more information on that channel. Each pass through the loop reads a line from the file with the **LINE INPUT** statement (note the channel number), and prints it to the current window with the **PRINT** statement (note the lack of a channel number).

You will find much more information on using files in Chapter 13 “Files for Data Input and Output.”

Other Forms of Input

You may find that typed text input is inappropriate for certain applications. For instance, you may want to allow the user to specify a particular spot on the screen, or you may want to track the current position of the mouse. Chapter 13 “Graphics” describes the **GET MOUSE** statement and the **GET POINT** statement that provide graphical coordinate input.

You can also create elements such as menus, radio buttons, check boxes, push button, edit fields, and dialog boxes to enhance the appearance and ease-of-use of your programs. Chapter 14 “Interface Elements” explains how to create such elements and how to process input from them.

CHAPTER

5

Decision Structures

True BASIC carries out most statements in the order in which they appear in the source code. This means that most simple programs proceed from "top" to "bottom" when run. With this sort of "linear execution," each statement is used once and only once.

However, many problems require more flexibility than strict linear execution allows. The best solutions may require that certain statements execute more than once or perhaps not at all under some circumstances.

True BASIC uses structures to achieve this flexibility. A ***structure*** is a specialized construct that allows the program to control which statements get executed and when. This chapter introduces decision structures — a structure that lets your program decide which statements to execute and which statements to ignore. Later chapters introduce other structures, including loops, defined functions, subroutines, pictures, and error handlers.

True BASIC has two different decision structures: the **IF** structure and the **SELECT CASE** structure. Both structures let you "branch" to a specific set of statements and ignore others. Thus, you can write programs that proceed in different ways depending upon the value of logical expressions.

Logical Expressions

True BASIC allows you to compare the values of numeric and string expressions using logical expressions. A ***logical expression*** (sometimes called a Boolean expression) is an expression that can be evaluated as having either a true or false value. You form a logical expression by using a relational operator to compare two numerical expressions or two string expressions. The relational operators are:

Relational Operators

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

When you compare two expressions with a relational operator, the resulting logical expression is either true or false — either the relation holds or it doesn't. For instance, two numeric values are either equal or not; there is no "maybe."

When comparing numeric values, True BASIC uses standard mathematical conventions. When comparing string values, True BASIC uses the order of characters specified by the character set. (Since most operating systems use the ASCII character set, this manual assumes that set for all examples. For an ordered listing of the ASCII character set, see Appendix A.)

The ASCII character set ranks all letters alphabetically, but all uppercase letters come before lowercase letters, so that "Z" is earlier in the alphabet than "a". Most other characters, such as punctuation marks and digits, come before letters in the ASCII character set. (Exceptions include { | } ~ and the delete character.) A few examples should clarify these concepts:

Examples of Comparisons

Expression	Value
`4 - 2 = 1 + 1`	`true`
`3.5 > 2^2`	`false`
`2.35 <= 2.35`	`true`
`"Apple" < "Pear"`	`true`
`"apple" < "Pear"`	`false`
`"tree"[3:4] < "grass"`	`true`
`"123" > "abc"`	`false`

More complex logical expressions, called ***compound conditions***, may be built using the logical operators NOT, AND, and OR.

Logical Operators

`NOT` reverses the value of the logical expression given as its operand. For instance, if the NOT operator is applied to an expression with a true value, the value of the resulting compound condition will be false. Therefore,

`(3+2 = 5)`	is true and
`not (3+2 = 5 )`	is false.

`AND` evaluates the logical expression on its left and the one on its right and returns a value of true only if both logical expressions are true.

`OR` evaluates the logical expression on its left and the one on its right and returns a value of false only if both logical expressions are false.

[!] Note: True BASIC evaluates a compound condition from left to right but only as far as is necessary to determine whether it is true. This process is known as ***short circuiting*** and is very useful in avoiding errors. For example, the complex condition

```
      b <> 0 and a/b > 10
```

is safe. That is, no error will result if b is 0 because a/b will not be evaluated: if the first expression in an AND condition is false then the entire expression must be false, so True BASIC stops evaluating the compound condition. Similarly, if the first expression in an OR condition is true then the entire expression must be true so True BASIC evaluates no further.

To illustrate the behavior of the logical operators consider how True BASIC would evaluate the following complex condition:

```
  x < y or not(x-2 = z) and (a$ = "Yes" or b$ = "No")
```

First, the complex condition would be broken down into the operands of the first `OR` operator as follows:

```
  x < y
  not(x-2 = z) and (a$ = "Yes" or b$ = "No")
```

The expression `x < y`, as the left operand, would be evaluated first. If its value is true, then the value of the entire expression is true, and True BASIC stops evaluating. In such a case, the right operand is not even considered.

However, if the value of `x < y` is false, then the value of the complex condition

```
  not(x-2 = z) and (a$ = "Yes" or b$ = "No")
```

must be evaluated. To do so, this complex condition is broken down into the operands of the `AND` operator as follows:

```
not(x-2 = z)
(a$ = "Yes" or b$ = "No")
```

Again the left operand is evaluated first. The left operand is the complex condition not`(x-2 = z)`. This complex condition has only one operand, `(x-2 = z)`, and the NOT operator serves to reverse the value of this expression. Thus, if the value of the expression `(x-2 = z)` is true, then the value of the complex condition not`(x-2 = z)` is false. If the first operand is false, then the AND condition must be false and the second operand is ignored.

However, if the first operand of the AND operator is true, then the second operand must be evaluated to determine the value of the AND condition. This operand consists of another complex condition which may be broken down as operands of the OR operator:

```
a$ = "Yes"
b$ = "No"
```

If the expression a$ = "Yes" is true, then the entire OR condition is true and the right operand is ignored. However, if the left operand is false, then the right operand must be evaluated to determine the value of the OR condition.

The value of the OR condition within the parentheses is then used to resolve the value of the AND condition. In turn, the value of the AND condition is used to resolve the value of the initial OR condition. This then provides the value of the original complex condition.

The IF Structure

Logical expressions are commonly used in the **IF** structure, which lets your program make decisions. Simple **IF** structures choose between only two options, while more complex **IF** structures can select from several choices.

Single-branch IF Structures

The simplest form of the **IF** structure simply determines whether a particular block of statements will be executed. For example:

```
IF age >= 100 then
   PRINT "Congratulations!"
END IF
```

In this code segment, the "Congratulations!" message will be printed only when the value of age is greater than or equal to 100. The **IF** structure begins with an **IF** statement containing a logical expression and the keyword THEN. An **END IF** statement marks the end of the structure.

Because of its simplicity, this **IF** structure could also be written as the following single-line **IF** statement:

```
IF age >= 100 then PRINT "Congratulations!"
```

Single-line **IF** statements may have only one statement following THEN, and they have no **END IF** statement.

The **IF** structure is usually more flexible, however, since you can specify any number of statements to be executed if the logical expression is true. For example:

```
IF age >= 100 then
   PRINT "Congratulations!"
   LET bonus = 10
END IF
```

Everything between the **IF** and **END IF** statements will be executed if the condition is true.

Two-branch IF Structures

A slightly more complicated form of the **IF** structure allows you to specify two different blocks of statements; one to be executed if the logical expression is true, and another to be executed if it is false.

For example, suppose you are playing a guessing game. The computer has picked a number and stored it in the variable n. Your guess is stored in the variable guess. You could then use the following **IF** structure to determine whether the guess is correct:

```
IF guess = n then
   PRINT "Right"
ELSE
   PRINT "Wrong"
   PRINT "It was "; n
END IF
```

This structure, like the earlier example, starts with an **IF** statement and ends with an **END IF** statement. Notice that the position of the **ELSE** statement defines two distinct blocks of statements; one for a correct answer, and another for an incorrect answer. If you guessed correctly, the logical expression is true so the program uses the first block to print the "Right" message and ignores the second block. If you were wrong, the logical expression is false and the program ignores the first block and uses the second to print "Wrong" and give you the correct answer. Thus, the program follows one of two different courses of action, depending on the values of the variables guess and n. The indentation used here, which helps to display the structure, is based on the style conventions discussed in Chapter 1 "A Word on Style."

You may use the single-line form of the **IF** statement with a two-way decision as long as the THEN and ELSE keywords each have just one statement:

```
IF guess = n then PRINT "Right" else PRINT "Wrong"
```

Notice that again there is no **END IF** statement with a single-line **IF** statement. Single-line **IF** statements may have only one statement after the THEN keyword and one statement after the optional ELSE keyword. True BASIC interprets any **IF** statement with a statement following the THEN keyword on the same line as a single-line **IF** statement.

Multiple-branch IF Structures

You may use the **ELSEIF** statement to create more intricate branches. To illustrate a three-way branch, here is a short program for quadratic equations:

```
PRINT "Enter the three coefficients: "
INPUT a, b, c                      ! The coefficients
LET discr = b^2 - 4*a*c            ! The discriminant
IF discr = 0 then                  ! One root
   PRINT "The root is:"
   PRINT -b/(2*a)
ELSEIF discr > 0 then              ! Two roots
   LET s = Sqr(discr)              ! Take square root
   PRINT "The roots are:"
   PRINT (-b+s)/(2*a)
   PRINT (-b-s)/(2*a)
ELSE                               ! Complex roots
   PRINT "No real roots"
END IF
END
```

In this example, the **IF** structure defines three distinct blocks of statements. It is important to remember that, regardless of the number of blocks it contains, a single **IF** structure will execute one, and only one, of these blocks. Once a condition is satisfied, its associated block of statements is executed and the program continues with the line following the **END IF** statement; all other blocks in the **IF** structure are ignored.

You may use as many **ELSEIF** statements as necessary within an **IF** structure, but you may include only one **ELSE** statement (which should appear as the last option). Each **ELSEIF** statement must specify its own logical expression followed by the keyword THEN. As you have seen, each **IF** structure must begin with an **IF** statement (ending with the keyword THEN) and end with an **END IF** statement.

There is no form of the single-line **IF** statement that lets you include more than two possible actions. You must use a multi-line **IF** structure for three or more possible branches.

Nested IF Structures

The flexibility of the **IF** structure may be further enhanced by ***nesting*** — the process of defining one structure within another. The inner structure is said to be "nested" within the outer structure. The nested structure must be completed before the containing structure can be completed.

Here is an example of nesting. First, note that the above three-way branch for quadratic equations did not check whether a is 0. You could solve that problem by expanding the original **IF** structure to a four-way branch beginning with a test for a = 0 and nesting another **IF** structure in that new first branch as shown below:

```
PRINT "Enter the three coefficients: "
INPUT a, b, c                         ! The coefficients
LET discr = b^2 - 4*a*c               ! The discriminant

IF a = 0 then                         ! New test with nested structure
   IF b = 0 and c = 0 then            ! Begin nested structure
      PRINT "Any number is a solution."
   ELSEIF b = 0 then
      PRINT "There is no solution."
   ELSE
      PRINT "The root is:"
      PRINT -c/b
   END IF                             ! End nested structure

ELSEIF discr = 0 then                 ! (continue as above)
   PRINT "The root is:"               ! One root
   PRINT -b/(2*a)

ELSEIF discr > 0 then                 ! Two roots
   LET s = Sqr(discr)                 ! Take square root
   PRINT "The roots are:"
   PRINT (-b+s)/(2*a)
   PRINT (-b-s)/(2*a)

ELSE                                  ! Complex roots
   PRINT "No real roots"
END IF

END
```

The SELECT CASE Structure

If all the choices in the decision structure are based on the value of a single numeric or string expression, it is often more convenient to use a **SELECT CASE** structure.

The formation of a **SELECT CASE** structure is similar to that of an **IF** structure. The **SELECT CASE** statement, which indicates the beginning of the structure, contains the expression to be evaluated. An **END SELECT** statement indicates the end of the structure. Within the structure, you may use as many **CASE** statements as necessary to define blocks of statements that will be executed for specific values of the specified expression. True BASIC evaluates the expression and then executes the block of statements indicated by the first appropriate **CASE** statement; any remaining blocks are ignored. For example:

```
SELECT CASE n
CASE 2                                  ! If the number is 2
     PRINT "Even prime"
CASE 3, 5, 7                            ! If the number is 3, 5, or 7
     PRINT "Odd prime"
CASE 1, 4, 9                            ! If the number is 1, 4, or 9
     PRINT "Perfect square"
CASE else                               ! If anything else
```

```
     PRINT "Composite, not a square:"
END SELECT
```

When executing this segment of code, True BASIC determines the value of n and looks for the first **CASE** statement that specifies a matching value. When it encounters an appropriate **CASE** statement, it executes the block of statements immediately following that statement (up to the next **CASE** statement) and continues with the line following the **END SELECT** statement. If there is no **CASE** statement that specifically matches the value of n, then the block of statements following the **CASE ELSE** statement is executed. If there is no **CASE ELSE** statement, then an error occurs and the program stops.

The **CASE ELSE** statement is optional, but if used it must be the last case in the structure. Since a **CASE** statement is not required to have a block of statements associated with it, a program can "ignore" a particular case simply by having no statements between that **CASE** and the next. Thus, you may wish to include an empty **CASE ELSE** block to avoid errors.

The **CASE** statements may specify only constant values; variables and expressions may be used only in the **SELECT CASE** statement. While the above example demonstrates the use of discrete constants in the **CASE** statements, you can build much more powerful **SELECT CASE** structures using ranges and relations on the **CASE** statements.

A ***range*** specifies a range of values for which the **CASE** statement holds true. To specify a range, use the keyword TO with constants specifying the low and high ends of the range, as in:

```
CASE 10 TO 20
```

Ranges are inclusive. For instance, the above example would hold true for values of 10, 20, or any value greater than 10 and less than 20.

A ***relation*** specifies a relationship between the value of the expression specified in the **SELECT CASE** statement and a constant value for which the **CASE** statement holds true. To specify a relation, use the keyword IS followed by one of the relational operators and a constant value, as in:

```
CASE IS < 0
```

This **CASE** statement holds true for any **SELECT CASE** expression whose value is less than 0.

Consider the following example that demonstrates the use of ranges and relations with a string expression. This program counts characters of various types. It uses the facts that space is the first regular character and that control characters (carriage returns, line feeds, etc.) come before regular characters in the ASCII character order.

```
LINE INPUT PROMPT "Enter a line of text: ": line$
FOR c = 1 to Len(line$)
    SELECT CASE line$[c:c]          ! Current character
    CASE "0" to "9"                 ! If it is a digit
         LET number = number + 1
    CASE "A" to "Z"                 ! If it is uppercase
         LET uc = uc + 1
    CASE "a" to "z"                 ! If it is lowercase
         LET lc = lc + 1
    CASE is < " "                   ! If a control character
         LET control = control + 1
    CASE else
         LET other = other + 1
    END SELECT
NEXT c
PRINT "The line contained:"; number; "numbers and"; uc + lc;
PRINT "characters."
PRINT "There were"; other + control; "other characters in the line."
END
```

CHAPTER

6

Loop Structures

Often, you will find that you want to repeat a block of statements many times. True BASIC provides two loop structures that let your programs execute the same statements several times. **FOR** structures, often called **FOR** loops, repeat a block of statements a specified number of times. **DO** structures, or **DO** loops, repeat a block of statements until a certain condition is satisfied.

This chapter introduces **FOR** and **DO** loops, as well as the **EXIT** statements that allow you to escape from the body of a loop.

FOR Loops

A **FOR** structure, or **FOR** loop, executes a block of statements a predetermined number of times. You form a **FOR** structure using a **FOR** statement and a **NEXT** statement.

The **FOR** statement controls the number of times the loop will be repeated by defining the ***index variable***, its ***initial value***, its ending or ***limit value***, and its ***increment***. The structure uses the index variable to monitor the number of passes through the loop. After each time through the loop, True BASIC increases the index variable by the value of the increment. As soon as the index variable becomes greater than its limit value, the program goes to the statement following the **NEXT** statement.

Here is a simple example using the **PLOT** statement, which is explained in more detail in Chapter 13 “Graphics.”

```
! Plot the square root function
SET WINDOW 0, 10, 0, 4
FOR x = 0 to 10 step .1
    PLOT x, Sqr(x);
NEXT x
END
```

Here the index variable x starts with value 0 and increases in steps of 0.1 until its value reaches 10. The **NEXT** statement that indicates the end of the loop’s body must specify the same index variable as the **FOR** statement that begins the loop. For each value of x, the ***body*** of the loop (the statements between the **FOR** and **NEXT** statements) executes exactly once. The STEP clause may be omitted from the **FOR** statement, in which case an increment of 1 is used:

```
! Table of square roots
PRINT "Number", "Square Root"
FOR number = 1 to 10
    PRINT number, Sqr(number)
NEXT number
END
```

Note that the index variable is increased at the **NEXT** statement. Thus, upon completion of the loop, the index variable equals the first value that exceeds the limit value. Hence in the first example, x has a value of 10.1 upon completion of the loop, while in the second example, number is equal to 11 after the loop.

If the initial value of the index is greater than the limit value, the loop is not executed. For example:

```
! Sum of odd numbers to n
INPUT n
LET sum = 0
FOR i = 1 to n step 2           ! Odd numbers only
    LET sum = sum + i           ! Add them up
NEXT i
PRINT sum                       ! Answer
END
```

If a value of 0 is supplied for n, then the body of the loop is not executed at all, and an answer of 0 is printed — which is correct!

Negative increments are also allowed, in which case the loop continues until the value of the index variable is less than the limit value. For example,

```
FOR x = 3.2 to 1.3 step -0.5
```

executes the loop with x = 3.2, 2.7, 2.2, 1.7 and exits with x = 1.2. In the case of a loop with a negative increment, the loop body will not be executed at all if the initial value is less than the limit value.

Beware of unintentionally changing the value of the index variable inside the body of the loop. Although you may do so, it can lead to unexpected results.

Occasionally, you may want to exit from a **FOR** loop before the index variable completes its defined sequence. True BASIC provides the **EXIT FOR** statement for exactly this purpose. When an **EXIT FOR** statement is executed, True BASIC immediately skips to the line following the **NEXT** statement. Upon such an exit, the index variable retains the value it had when the **EXIT FOR** statement was executed.

The **EXIT FOR** statement is typically used as part of a decision structure. For example, you may want to examine a series of values until some condition is met:

```
! Find smallest integer whose 5th power
!    is greater than a billion
FOR n = 1 to 100                 ! Examine each integer
    IF n^5 > 1e9 then EXIT FOR
NEXT n
PRINT n                          ! First integer to satisfy condition
END
```

If you find yourself using an **EXIT FOR** statement that is not part of a decision structure, then you most likely don't need the loop that contains it.

DO Loops

Often, you do not know how many times you will need to execute the body of a loop. Instead, you want to repeat the loop until a condition is met. The **DO** loop fulfills this need.

A **DO** structure, or **DO** loop, starts with a **DO** statement and ends with a **LOOP** statement. The following program illustrates the simplest form of the **DO** structure:

```
DO
   PRINT "Happy Birthday!"
   PRINT "And many happy returns."
LOOP
END
```

The **DO** loop in this program will repeat forever – that is, until the user stops the program. (To stop a running program, see the "True BASIC Environment" chapter in the Introduction section.) Loops that run forever are called ***infinite loops***.

Although infinite loops are useful sometimes, you will usually want a loop that ends once a condition is met. True BASIC provides three ways of ending a **DO** loop. You may attach a condition to the **DO** statement or to the **LOOP** statement, or you may use an **EXIT DO** statement within the loop.

You have two options for attaching a condition to the **DO** statement: the WHILE clause and the UNTIL clause. The WHILE clause follows the DO keyword and specifies a condition as a logical expression:

```
INPUT PROMPT "Initial sum, annual interest rate? ": sum, interest
LET mo_rate = interest/12

DO WHILE sum < 1000
   LET sum = sum * (1 + mo_rate)
   LET months = months + 1
LOOP

PRINT "It will take"; months; "months at"; interest; "to earn $1,000."
END
```

As long as the value of the logical expression is true, the body of the loop will be executed repeatedly. Before each pass through the body of the loop, True BASIC checks the value of the condition; as soon as it becomes false, the program continues with the line immediately following the **LOOP** statement.

The UNTIL clause is used the same way, except that the loop continues until the condition becomes true. Thus, the following two **DO** statements have the same effect:

```
DO WHILE sum < 1000
DO UNTIL sum >= 1000
```

You may also attach a WHILE or UNTIL clause to the **LOOP** statement. The behavior of the loop will differ only in when the condition is checked. With the WHILE or UNTIL on the **DO** statement, the condition will be checked before each pass through the loop. Thus, there is a possibility that the body may never be executed. In the above example, if the user enters an initial value greater than 1000, the program will skip the body of the loop.

On the other hand, when the WHILE or UNTIL clause is on the **LOOP** statement, the condition will be checked after each pass through the loop. This guarantees that the body of the loop is always executed at least once. For instance:

```
! Ask whether we should continue
DO
   CALL Game_Sub                  ! User-defined subroutine
   PRINT "Shall I continue";      ! Ask question
   INPUT answer$
LOOP WHILE answer$ = "yes"
```

Here the **LOOP** statement contains the WHILE clause since the loop's body must be executed at least once before there is an answer to check.

While it is possible to specify a WHILE or UNTIL clause for both the **DO** and **LOOP** statements, this is seldom necessary. Adding clauses to the top and bottom of a loop makes the loop's behavior difficult to understand, and you should avoid this technique in all but exceptional circumstances.

Occasionally, you may want to exit from the body of a DO loop without waiting until the next pass. To do so, use the **EXIT DO** statement. When an **EXIT DO** statement is executed, True BASIC immediately skips to the line following the next **LOOP** statement.

As with the **EXIT FOR** statement, you will typically use an **EXIT DO** statement as part of a decision structure, as in the following code segment:

```
! Ask whether we should continue
DO
   CALL Step_One                              ! User-defined subroutine
   INPUT PROMPT "Shall I continue": answer$   ! Ask question
   IF answer$ = "no" then EXIT DO
   CALL Step_Two                              ! User-defined subroutine
LOOP
```

You may find it convenient to use an **EXIT DO** statement as well as a WHILE or UNTIL clause attached to the **LOOP** or **DO** statement. One might represent the normal termination, while the other may be an exit under special conditions.

Nested Loops

As with decision structures, you may nest a loop within another loop structure, and you may nest loops within decision structures or vice versa. You may nest loops and decision structures several layers deep; in other words, a nested structure may in turn contain another nested structure. The important rule is that the nested structure must be completed before the containing structure continues.

Here's an example of nested loops and decision structures:

```
! Print triangular patterns of letters
FOR row = 1 to 6
    FOR triangle = 1 to 3
        FOR xcount = 1 to row
            IF triangle = 1 then
               PRINT "a";
            ELSEIF triangle = 2 then
               PRINT "b";
            ELSE
               PRINT "c";
            END IF
        NEXT xcount
        PRINT,           ! Move to next column for next triangle
    NEXT triangle
    PRINT                ! Move to next row
NEXT row
END
```

This program uses three nested loops and a decision structure to produce the following output:

```
a              b              c
aa             bb             cc
aaa            bbb            ccc
aaaa           bbbb           cccc
aaaaa          bbbbb          ccccc
aaaaaa         bbbbbb         cccccc
```

To better understand the operation of these structures, it is worthwhile to study this code in detail.

The program contains three nested **FOR** loops and one **IF** structure. Notice how the indentation helps identify the nested structures. One loop governs the number of rows of text contained in each triangle. One loop controls the number of triangles. And the third loop forms the triangular shapes by varying the number of characters printed on each line. The decision structure determines which letter is used for each triangle.

The outer-most loop uses row as its index variable. As you can infer from the name of its index variable, this is the loop that controls the number of rows printed. To change the number of rows used in the printed figure, simply change the limit value of this loop. Each pass through the body of this loop is responsible for printing one row of the final image.

The body of this outer-most loop contains the middle loop and a vacuous **PRINT** statement that ensures that each row starts on a new line. The middle loop uses triangle as its index variable and controls how many triangles are printed — changing the limit value of this loop will change the number of triangles. Each pass through the body of this middle loop prints the current row of a single triangle.

To accomplish this, the body of the middle loop contains the third and final loop plus a **PRINT** statement containing nothing but a comma to force the text cursor to the next print zone. The inner-most loop uses count as its index variable and the current value of row as its limit value. Since the value of row increases by one with each pass through the outer-most loop, using it as the limit value of the inner-most loop results in the inner-most loop being executed once on the first pass through the outer-most loop, twice on the second pass, and so forth. Each pass through this loop prints one character in the current row of the current triangle.

The body of the inner-most loop contains an **IF** structure that determines which letter to print based upon the value of triangle. Since the value of triangle changes with each pass through the middle loop, its value is used to print a different letter for each triangle.

Nested structures give you lots of power and flexibility; however, they can also create extremely complex programs that are difficult to debug and maintain. While nested structures provide the best solution to many programming needs, it is important that you understand how such nesting works before you use it.

CHAPTER

7

Data as Part of the Program

Earlier chapters illustrate two methods for assigning values to variables. You can use the **LET** statement for direct assignments or use the various input statements to let the user make assignments while the program is running. A third method allows your program to assign values to variables from blocks of data stored in the program itself. This assignment method consists of two statements: the **DATA** statement, which defines the values, and the **READ** statement, which assigns those values to variables.

As with the **INPUT** statement, the **READ** statement can also assign values read from a data file. This chapter discusses only built-in data (data stored in the program itself). You will find a discussion of data files in Chapter 12 "Files for Data Input and Output."

Data Blocks

A **DATA** statement consists of the keyword DATA followed by a data list. A ***data list*** is a series of numeric and/or string constants separated by commas. The items in the data list should follow the same rules as data entered in response to an **INPUT** statement. String constants containing leading or trailing spaces, commas, quotation marks, or an exclamation point must be enclosed in quotes. Quotation marks that are not part of the data must be doubled. The null (or empty) string ("") is a valid string constant and may be used as a data item.

DATA statements are simply storage areas for information used by the program; they are not executed. Thus, you may put them anywhere in the program; common placements are just before the **END** statement to keep them out of the way, or near their associated **READ** statements to make the program easier to understand. Before it runs the program, True BASIC collects all the data items into a single ***data pool***. The data pool will contain each of the data lists in the order in which they appear in the program.

When you start writing programs with several program units, the rules for **DATA** statements get a bit more complicated. For now, it will suffice to note that each program unit has its own separate data pool, which contains only the contents of the **DATA** statements located within that program unit. For more information on data pools and program units, refer to Chapter 11 "Libraries and Modules."

Reading Data

The **READ** statement assigns values from the data pool to specific variables. It is similar to the **INPUT** statement, except that it obtains its information from the data pool rather than from the user. **READ** statements, like most True BASIC statements, are executed as they are encountered, and they always take the next data item from the data pool.

To understand the concept of the next data item, you must first understand the concept of the data pointer. The ***data pointer*** indicates which data item in the data pool should be used next. True BASIC handles the data pointer automatically, so you don't need to worry about it. When the data pool is created, the data pointer "points to" the first data item in the pool. Each time a **READ** statement uses a data item, True BASIC automatically moves the data pointer to the next item. When the last item in the pool has been read, the pointer is set beyond the end of the pool. If your program tries to execute a **READ** statement when the pointer points beyond the end of the pool, True BASIC generates an error.

Consider this example:

```
READ x, y                    ! 3, 4
PRINT x + y                  ! 7
READ x, y                    ! 2, 3
READ a$, z                   ! "Answer is," 4
PRINT a$; (x + y) * z        ! "Answer is, " (2 + 3) x 4
GET KEY k                    ! Hold output until a key is pressed
DATA 3, 4, 2
DATA 3, Answer is, 4
END
```

This program has two **DATA** statements, which are combined to form the data pool:

```
3
4
2
3
Answer is
4
```

The first **READ** statement reads the first two items from the data pool and leaves the data pointer at the third item. Thus, the variable x will be assigned the value 3, and y will be assigned the value 4. The program prints the sum of these values (7) before it executes the second **READ** statement.

The second **READ** statement reads the value 2 into x and the value 3 into y, leaving the data pointer pointing at the fifth data item. Notice that the fifth data item is not a valid numeric constant; it must be read into a string variable. The third **READ** statement does just that, reading that value into the string variable a$ and reading the sixth item into the variable z. These variables are then used to produce the second line of output.

Thus, the entire output of the program will be:

```
 7
Answer is 20
```

Note that the **DATA** statements must list items in the precise order in which they will be read by the **READ** statements. You may not read a non-numeric value into a numeric variable, but you may read any value into a string variable. And while you may read fewer items than are contained in the data pool, you may not read more items than are in the data pool.

Checking for More Data

Since **READ** statements are often used inside loop structures to repeat the same block of statements with several different values, True BASIC provides two special logical expressions that your program may use to avoid reading past the end of the data pool. These expressions are MORE DATA and END DATA.

The logical expression MORE DATA is true as long as the data pointer is not pointing beyond the last data item in the pool. The logical expression END DATA is true only when the data pointer has passed the last item in the pool. You may use these anywhere you can use a logical expression, such as the condition in an **IF** statement or as the condition in a WHILE or UNTIL clause. For example, consider the following program:

```
PRINT "Number", "Square Root"
DO WHILE MORE DATA
   READ x
   PRINT x, Sqr(x)          ! Show square root
LOOP

DATA 1,2,3,5,6,7,8,10,11,12
END
```

Because of the flexible way in which this program is written, extending the printed table is a simple matter of adding additional data items.

Another mechanism for detecting the end of the data pool is the IF MISSING clause in the **READ** statement, as in:

```
DO
   READ IF MISSING then EXIT DO: x
   PRINT x
LOOP
```

In this example, the **EXIT DO** statement will be executed only if the data pointer has passed the last data item in the pool.

Reusing Data

Sometimes, your program will need to use the same data pool more than once. The **RESTORE** statement sets the data pointer back to the first item in the pool, allowing its reuse. For example, the program:

```
FOR n = 1 to 1000
   READ x
   PRINT x;
   IF END DATA then RESTORE
NEXT n

DATA 1, 2, 3, 4, 5, 6, 7, 8, 9, 10
DATA 99, 98, 97, 96, 95
END
```

will print 1,000 numbers, using the 15 numbers in the data pool repeatedly.

If your program uses line numbers, then you may specify a line number with the **RESTORE** statement. In that case, True BASIC moves the data pointer to the item in the data pool that corresponds to the first item in the **DATA** statement at that line number.

CHAPTER

8

Built-in Functions

True BASIC provides built-in, or pre-defined, functions that perform a wide range of operations. This chapter introduces many commonly used built-in functions; additional functions are introduced in other chapters as appropriate. If you cannot find a built-in function that performs the operation you require, you can define your own functions as described in Chapter 10 “User-defined Functions and Subroutines.”

Function Basics

A ***function*** is a structure that simplifies a complex operation into a single step. Functions act as “black boxes.” They accept some input value or values and process that input in a defined manner to produce or “return” an output value. As long as you know how and when to use a particular function, you need not be bothered by how it actually works.

Consider, for example, the process of taking the square root of a numeric value. If you had to define this process in your program every time you needed a square root, your programs would require extra code and you would probably tire of entering the same code over and over.

Fortunately, True BASIC has a built-in function that compresses the entire square root operation into a single step. Using the **SQR** function, your program can easily find the square root of any number greater than or equal to zero:

```
DO
   INPUT PROMPT "Enter a number: ": n
   IF n < 0 then EXIT DO
   PRINT "The square root of"; n; "is"; Sqr(n)
LOOP
END
```

The **SQR** function displays the square root of a numeric value provided by the user.

If a function returns a numeric value, as the **SQR** function does, it is a ***numeric function***. Functions that return a string value are ***string functions***. Other functions may return logical values or entire arrays (as discussed in the following chapter). A function’s name reflects the type of value it returns; a function returning string values has a name that ends with a dollar sign ($).

The input values that you provide when you use a function are called its ***arguments***, and the value the function returns is called its ***return value***. Often a function’s return value is referred to simply as that function’s value. Note that even though functions may require several arguments as input, a function returns one and only one value (except for the special array functions).

In the program example above, the variable n is the argument to the **SQR** function. The **SQR** function takes the value of n and returns its square root. True BASIC uses the function’s return value at the place where the function is invoked. In the above example that value is displayed by the **PRINT** statement; it could also be part of an expression used in an assignment statement:

```
LET x = ( Sqr(z) + y ) / z
```

Not all functions require arguments, but those that do follow very strict rules. The function’s definition determines the number, type, and order of its arguments. A function definition works with specific ***parameters***. When you invoke the function, you usually must supply matching arguments as input for those parameters. You specify the ***argument list*** for a function in parentheses after the function's name, separating multiple arguments by commas.

For example, the **REPEAT$** function uses a string parameter and a numeric parameter, in that order. It returns a string formed by the string parameter repeated the number of times specified by the numeric parameter. When you use the function you must provide a string argument followed by a numeric argument:

```
PRINT Repeat$("Tra la! ", 4)
```

The function will then substitute the values of the arguments for the parameters to compute its return value:

```
Tra la! Tra la! Tra la! Tra la!
```

Some of True BASIC's built-in functions are defined with optional parameters. They assume some default value for these optional parameters if you do not supply matching arguments when you invoke the function. For example, the **ROUND** function rounds numeric values. It uses two numeric parameters: the value to be rounded and the number of places it should be rounded to the right or left of the decimal point. If you omit the second argument, **ROUND** assumes 0 for that parameter. For example:

```
PRINT Round (123.4567, 3), Round (123.4567), Round (123.4566, -2)
```

prints the following:

```
 123.457        123            100
```

True BASIC matches the arguments you specify when you invoke a function with the parameters in that function's definition based solely on their position in the argument list. In other words, the value of the first argument in the invocation will be used as the value of the first parameter in the definition, and so on. If the type of an argument does not match the type of the corresponding parameter, then True BASIC generates an error.

You may use a function anywhere you would use an expression. A function invocation, along with its associated argument list, is itself an expression and may be used to build more complex expressions. Here are some typical uses of numeric and string functions (the functions themselves are introduced in the following sections):

```
LET answer = 3 * Log(2*z - 7.2)
LET z = Exp(-x) * Cos(2*t - 1)
DO WHILE Sin(x) < .5
LET reply$ = Lcase$(input$)
IF Ucase$(continue$[1:1]) = "Q" then STOP
```

Arguments to functions may be constants, variables, or expressions — as long as they are the correct type. Because functions are themselves expressions, you may use them as arguments to other functions. Here are some examples:

```
LET n = Int(Rnd * 10) + 1              ! Random integer between 1 and 10
LET answer = Round(Sqr(x), 2)          ! Round square root to 2 decimal
places
PRINT Repeat$ (echo$, Int(Rnd*5)+1)    ! Repeat a random no. of times
```

Numeric Functions

Many of True BASIC's built-in numeric functions, like the **SQR** function, perform mathematical operations that would be difficult or impossible to implement with the mathematical operators discussed in Chapter 2 "Constants, Variables, and Expressions." This section introduces most of the built-in mathematical and trigonometric functions along with two other functions that examine the numeric capabilities of the computer running your program. (Other functions that return numeric values are introduced in the "String-handling Functions" and "Time and Date Functions" sections of this chapter.)

Most numeric functions take one or two arguments which may be numeric constants, numeric variables, or any other numeric expression. As noted above, numeric functions are themselves numeric expressions so they may be used as arguments to other functions:

```
LET answer = Max(Sin(x), Cos(2*x))  ! Larger of sine or cosine
```

Mathematical Functions

The following table summarizes the built-in mathematical functions. The numeric arguments in the table are represented by `x`, `y`, or `n`.

Mathematical Functions

Function	**Result**
ABS(x)	Absolute value of x
SGN(x)	Sign of x; returns 1 if positive, -1 if negative, 0 if x = 0
SQR(x)	Square root of x
EXP(x)	The natural exponent of x, or ex where e = 2.718281828...
MIN(x,y)	Smaller of two numbers
MAX(x,y)	Larger of two numbers
MOD(x,y)	Remainder when x is divided by y
REMAINDER(x,y)	Remainder when x is divided by y
ROUND(x,n)	Value of x rounded to n decimal places; n assumed to be 0 if not specified
TRUNCATE(x,n)	Value of x truncated to n decimal places; n assumed to be 0 if not specified
RND	A pseudo-random number greater than or equal to 0 and less than 1
LOG(x)	Natural logarithm of x
LOG10(x)	Common logarithm of x (base 10)
LOG2(x)	Logarithm to the base 2 of x
INT(x)	Greatest integer <= x
IP(x)	Integer part of x
FP(x)	Fractional part of x
CEIL(x)	Ceiling of x or least integer >= x

While most of these functions are direct parallels of their mathematical counterparts, a few warrant special attention.

The **SGN**, or signum, function is often used in mathematics and is very useful for programming. The value of `Sgn(x)` is +1, 0, or –1, depending on whether `x` is positive, zero, or negative.

The **MOD** function has many uses. One use is to test whether one number is a multiple of another, in which case the **MOD** function returns a value of zero. Consider the following program that finds out if a number is odd or even:

```
DO
   INPUT PROMPT "Enter a number (0 to quit): ": n
   IF n = 0 then
      EXIT DO
   ELSEIF Mod(n,2) = 0 then
      PRINT "The number "; n; "is even."
   ELSE
      PRINT "The number "; n; "is odd."
   END IF
LOOP
END
```

The **REMAINDER** function is a variant of the **MOD** function that uses a different convention for negative numbers. For more details on the subtle differences between these two functions see their formal descriptions in Chapter 18 “True BASIC Statements and Built-in Functions and Subroutines.”

When you want to round off a numeric value, use the **ROUND** function which rounds `x` to `n` decimal places. If the value of `n` is 0, or if the second argument is omitted, then `x` is rounded to an integer. To round to the left of the decimal point, use a negative value for n. For example, -3 will round `x` to the nearest thousand. The **TRUNCATE** function is similar, but simply drops any extra digits. The number 1.7 rounded to an integer becomes 2, but when truncated, the .7 is dropped and it becomes 1.

The **RND** function requires no arguments. It generates a pseudo-random number greater than or equal to 0 and less than 1. Each time the **RND** function is invoked, a new number is returned. While the numbers are not truly random, True BASIC's random numbers stand up well under statistical tests and hence allow the simulation of chance events.

To simulate a game with a 37% probability of winning, you could use the following code segment:

```
IF Rnd <= .37 then
   PRINT "You win"
ELSE
   PRINT "You lose"
END IF
```

The **RND** function can also simulate the rolling of a die. Since the result of rolling a die is always an integer ranging from one to six, you could use the statement:

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

If executed repeatedly, this statement will produce integers from one to six with equal probabilities.

To facilitate debugging, True BASIC produces the same sequence of pseudo-random numbers each time you run a program. Once the program is debugged, however, you will almost certainly want it to behave differently each time it is run.

To force the program to produce a different sequence of pseudo-random numbers each time it is run, simply insert a **RANDOMIZE** statement near the beginning of your program. This statement produces a new "seed" for the random-number generator, resulting in a new series of pseudo-random numbers. Please note that the **RANDOMIZE** statement need only be executed once; it is neither necessary nor desirable for a program to execute it repeatedly.

The **INT** function returns the greatest integer that is less than or equal to its argument, and the **CEIL** function returns the least integer that is greater than or equal to its argument. For instance, `Int(2.34)` returns `2` while `Ceil(2.34)` returns `3`. Similarly, `Int(-2.34)` equals `-3` and `Ceil(-2.34)` equals `-2`.

A variant of the **INT** function is the **IP** function, which returns the integer part of its argument. The two functions work differently for negative numbers, however: **IP** essentially strips the decimal part of the number away returning the value to the left of the decimal point, while **INT** returns the greatest integer less than or equal to its argument. Thus, both `Int(3.267)` and `Ip(3.267)` equal `3`, but `Int(-3.267)` equals `-4` and `Ip(-3.267)` equals `-3`. The **FP** function returns the fractional part of a number. It is always true that `Ip(x) + Fp(x) = x`.

[!] **Note:** True BASIC also provides some built-in subroutines. Subroutines differ from functions in that they do not return a value in the same way and must be invoked with a **CALL** statement. Subroutines are discussed in detail in Chapter 10 "User-defined Functions and Subroutines."
However, here we introduce the **DIVIDE** subroutine, which may be invoked as follows:

```
CALL DIVIDE(x, y, q, r)
```

The **DIVIDE** subroutine performs integer division of `x` by `y` and assigns the value of the quotient to `q` and the remainder to `r`.

Trigonometric Functions

Trigonometric functions are summarized in the following table. Most take one or two numeric arguments, indicated by x and y in the table.

Examples of Trigomonetric Functions

Function	Result
PI	The constant 3.1415...
SIN(x)	Sine
COS(x)	Cosine
TAN(x)	Tangent

SEC(x)	Secant
CSC(x)	Cosecant
COT(x)	Cotangent
ATN(x)	Arctangent
ACOS(x)	Arccosine
ASIN(x)	Arcsine
COSH(x)	Hyperbolic cosine
SINH(x)	Hyperbolic sine
TANH(x)	Hyperbolic tangent
DEG(x)	Converts x from radians to degrees
RAD(x)	Converts x from degrees to radians
ANGLE(x,y)	Counter-clockwise angle between positive x-axis and point (x, y)

The **PI** function requires no argument and always returns the value of that famous constant (3.1415...). It is a function and not a variable; it would not make sense to assign a value to it. The **PI** function is useful in trigonometric formulas such as:

```
LET z = Sin(x + Pi/4)
```

Unless your program specifies otherwise, True BASIC assumes that values representing angles in trigonometric functions are measured in radians. If you want to work with angles measured in degrees, you can change True BASIC's behavior with the **OPTION ANGLE** statement. This statement takes two forms. The form

```
OPTION ANGLE degrees
```

instructs True BASIC to assume degrees for the arguments to all subsequent trigonometric functions. The angle measure set by the **OPTION ANGLE** statement remains in effect until the end of the program (or the current program unit) or until another **OPTION ANGLE** statement changes the setting. You may use

```
OPTION ANGLE radians
```

to return to radian measures if necessary.

If you need to convert a single value from radians to degrees, use the **DEG** function, which takes the radian value to be converted as its only argument. Use the **RAD** function to convert a single value from degrees to radians.

The **ANGLE** function returns the angle measured counterclockwise between the positive x-axis and the point specified by its arguments. The first argument represents the x-coordinate of the point, and the second the y-coordinate. The return value is in radians or degrees depending upon the current angle measure option.

While any numeric expression may serve as a numeric argument, certain mathematical functions specify values that are illegal when used as arguments. For example, `Log(-2), Tan(Pi/2)` and `Angle(0,0)` will produce errors.

The accuracy of the trigonometric and transcendental functions on most operating systems is the full accuracy spedified by the IEEE standard for eigh-byte arithmetic, that is, about 15 significant digits.

Range-of-numbers Functions

True BASIC provides two numeric functions — the **MAXNUM** and **EPS** functions — that help you discover the range of numbers available on your computer. The values of these functions depend on what computer you use.

The **MAXNUM** function, which does not accept an argument, returns the largest positive number expressible by the computer currently running the program.

The **EPS** function returns the smallest positive number that "makes a difference" when added to or subtracted from the value of the argument. Thus, `Eps(0)` is the smallest positive number expressible by the computer currently running the program. Similarly, if `Eps(1e15)` equals 4 on your computer, then adding or subtracting 3 to 1e15 will not change its value, and you know that the fractional part of your number is meaningless — and even the last digit of the integer part is suspect.

String-handling Functions

True BASIC also provides several functions to help you work with strings. Some string-handling functions transform a string argument into another string. Other string-handling functions, however, search for values within strings and are actually considered numeric functions because they return numeric values. This section introduces these two groups of functions plus the **USING$** function. (See the section on "Time and Date Functions" for two more functions that return string values.)

String Search Functions

The following table lists those string-handling functions that return a numeric variable. These are generally classified as string search functions:

Examples of String Search Functions

Function	Result
LEN(x$)	Number of characters in x$
POS(x$,a$,c)	First occurrence of a$ in x$ at or after character number c
CPOS(x$,ch$,c)	First occurrence of a character in x$ from string ch$ at or after character number c
NCPOS(x$,ch$,c)	First occurrence of a character in x$ not from string ch$ at or after character number c
POSR(x$,a$,c)	Last occurrence of a$ in x$ at or before character number c
CPOSR(x$,ch$,c)	Last occurrence of a character in x$ from string ch$ at or before character number c
NCPOSR(x$,ch$,c)	Last occurrence of a character in x$ not from string ch$ at or before character number c

The **LEN** function finds the number of characters in the string value supplied as its argument. The **LEN** function used with substring expressions is extremely useful when you need to process a string one character at a time. For example, the following program:

```
LINE INPUT PROMPT "Enter a line of text: ": line$
LET count = 0
FOR i = 1 to Len(line$)
     IF line$[i:i] = "," then LET count = count + 1
NEXT i
PRINT "Number of commas in line:"; count
END
```

counts the number of commas in a line input by the user. Notice how the **LEN** function limits the **FOR** loop so that the body is executed once for each character in line$. Within the body of the loop, a substring expression uses the index variable i to examine the i-th character of the string, counting it if it is a comma.

The other functions in the above list allow your program to search for the occurrence of one string inside another.

Pos(x$,a$,c) searches in x$ for the next occurrence of the substring a$. The search can start at a specific character position in x$, specified by c. However, if c is omitted, the search starts at the beginning of the string. The **POS** function can be very useful for parsing a string. ***Parsing*** is the process of breaking a string into its separate components. For instance, the following program parses a line into individual words:

```
LINE INPUT PROMPT "Enter a line: ": line$
LET line$ = line$ & " "
LET start = 0
DO
   LET end = Pos(line$," ",start) - 1   ! End of 1st word from 'start'
   PRINT line$[start:end]               ! Print from 'start' to end of word
   LET start = end + 2                  ! Reset value of 'start'
   IF start > Len(line$) then EXIT DO
```

```
LOOP
END
```

As it finds each word, it prints that word on a line by itself, as follows:

```
Enter a line: Hello, who are you?
Hello,
who
are
you?
```

This simplified implementation of the parsing process assumes that words are separated by exactly one space. A slightly more comprehensive version can be written using the **CPOS** and **NCPOS** functions.

`Cpos(x$,ch$,c)` and `Ncpos(x$,ch$,c)` treat the string `ch$` as a list of characters. The **CPOS** function searches `x$` for the first occurrence of any character from this list, while the **NCPOS** function searches `x$` for the first occurrence of any character not from the list. As with the **POS** function, the final numeric argument, `c`, is optional; if `c` is specified, the search begins at that character position.

Thus, the parsing program could be written as follows:

```
LET delim$ = " ,.?!;-()"
LINE INPUT PROMPT "Enter a line: ": line$
LET line$ = line$ & " "
LET start = Ncpos(line$,delim$)
DO
   LET end = Cpos(line$,delim$,start) - 1
   PRINT line$[start:end]
   LET start = Ncpos(line$,delim$,end+1)
   IF start = 0 then EXIT DO
LOOP
END
```

This version of the parsing program is significantly more flexible. It allows several standard punctuation characters to act as the separators between words (such characters are typically called ***delimiters***), and it will ignore multiple separators between words.

The **POSR**, **CPOSR**, and **NCPOSR** functions search in reverse; that is, they work backwards from the end of the string to be searched.

The following table illustrates the behavior of the seven string search functions. For each function, the return value is the number (or position) of a character where a match is found. If no match is found, the return value is 0:

Examples of String Search Functions

Example	Returns	Interpretation
`LEN("Hello there!")`	12	Number of characters in "Hello there!"
`POS("Harvard","ar")`	2	First occurrence of "ar"
`POS("Harvard","ar",3)`	5	First occurrence of "ar" at or after character 3
`POS("Harvard","ra")`	0	First occurrence of "ra"
`POSR("Harvard","ar")`	5	First occurrence starting from the end
`CPOS("Harvard","dv")`	4	First occurrence of a character from the string"dv"
`CPOSR("Harvard","dv")`	7	First occurrence starting from the end
`NCPOS("Harvard","dv")`	1	First occurrence of a character not in the string "dv"
`NCPOSR("Harvard","dv")`	6	Ditto, but starting from the end
`NCPOS("Harvard","adrv",2)`	0	First occurrence of a character not in "adrv" at or after the second character

String Transform Functions

Along with the numeric string-search functions, True BASIC has several built-in string functions that transform their string argument in some way:

String Transform Functions

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

[!] **Note:** Functions do not actually change the values of their arguments. Thus, when a function is referred to as transforming or changing its argument, the statement should be interpreted as meaning that the function returns a value that represents a transformation of the value of its argument. After the function invocation, however, the value of the actual argument remains unchanged.

The **LCASE$** and **UCASE$** functions can be useful for testing user input. Consider, for example, a situation where you expect the user to answer a prompt with the input “Yes”. If you use the test:

```
IF Lcase$(answer$) = "yes" then
```

then it does not matter whether the user types YES, Yes, yEs, or yes in response to your prompt. Allowing the user to ignore case when not important makes your programs more user friendly.

The **LTRIM$**, **RTRIM$**, and **TRIM$** functions are handy when you are using substring expressions. For example, if you are testing the user’s input based on its first character, you might strip off leading spaces to ensure that you are testing the first real character. The following variation on the previous example would accept `Y, y, Yes, yep!, "  Yes, sir  "`, and any other phrase beginning with the letter “`y`” as an affirmative response:

```
IF Lcase$(Trim$(answer$)[1:1]) = "y" then
```

As shown earlier, the **REPEAT$** function generates strings with repeating patterns. The string specified as the first argument is repeated the number of times specified by the second (numeric) argument.

The USING$ Function

Another useful string function is the **USING$** function, which is related to the **PRINT USING** statement. The **USING$** function returns a string formatted by a format string according to the same rules as the **PRINT USING** statement. The **USING$** function does not print the resulting string, however. For example,

```
LET result$ = Using$("##.###", 13.756812)
```

assigns the string value `"13.757"` to the variable `result$`. The value returned by the **USING$** function may be displayed anywhere on the screen, stored in a file, or further manipulated.

Converting Between Strings and Numbers

True BASIC maintains a strict distinction between numeric and string values. Most operations and functions are specific to one or the other. For instance, you cannot subtract two strings, nor can you concatenate two numbers. Sometimes, however, you need to work around these distinctions by converting between strings and numbers.

True BASIC provides several built-in functions that allow you to convert between data types. The simplest of these are the **STR$** and **VAL** functions.

The **STR$** function returns the string representation of the numeric value given as its argument. The form of the string representation is the same as would be produced by a **PRINT** statement, except that leading and trailing spaces are not included. Thus, the **STR$** function lets you print a numeric value without its leading or trailing spaces. For instance:

```
LET n = 2
LET m = 3
PRINT Str$(n); "+"; Str$(m); "="; Str$(n+m)
```

would produce the output:

```
2+3=5
```

If the value of a string expression follows the rules for a numeric constant, then the **VAL** function can convert that string value into its equivalent numeric value. This can be useful when you are processing numeric input entered as strings. Consider the following variation on the earlier parsing example that takes a delimited list of numbers and prints their sum:

```
LET delim$ = " ,.?!;-()"
LINE INPUT PROMPT "Enter a series of numbers: ": line$
LET line$ = line$ & " "
LET sum = 0
LET start = Ncpos(line$,delim$)
DO
   LET end = Cpos(line$,delim$,start) - 1
   LET sum = sum + Val(line$[start:end])
   LET start = Ncpos(line$,delim$,end+1)
   IF start = 0 then EXIT DO
LOOP
PRINT "Their sum is"; sum
END
```

The **CHR$** and **ORD** functions play a similar role for single characters. The **ORD** function translates a single character string into its corresponding code number in the current character set — usually the ASCII character set. If `c$` is a one-character string, then `Ord(c$)` returns the code number corresponding to that character in the current character set. For example,

```
GET KEY z
IF z = Ord(".") then EXIT DO
```

will jump out of a **DO** loop when the user types a period. **ORD** also accepts longer names for some ASCII characters. Thus, `ORD("BS")` returns 8, which is the character code for the "back space" character; see Appendix A for a list of ASCII codes and names. When its argument is the null string, the **ORD** function returns a value of –1.

The **CHR$** function goes the other way, converting a code number into its corresponding character. If `n` is a number in the range 0 to 255, `Chr$(n)` returns the character corresponding to that number in the current character set. You can use **CHR$** to translate **GET KEY** input into a string value:

```
DO
   GET KEY k
   IF 32 < k and k < 127 then
      PRINT Chr$(k); " is"; k
   ELSE
      PRINT "Key is not a printable character"
   END IF
LOOP UNTIL k = Ord(" ")              ! Space bar stops program
END
```

You can also use **CHR$** to introduce non-printing characters into your program. For instance, `Chr$(9)` returns the tab character, and `Chr$(27)` returns the escape character.

Advanced programmers may wish to "pack" numeric values into string variables. This technique can save memory and is often required when working with the machine at a lower level. The **PACKB** subroutine allows you to pack a numeric value into a specific set of bits within a string variable. A call to this built-in subroutine takes the following form:

```
CALL Packb(s$,b,nb,number)
```

This call instructs True BASIC to represent the numeric value of `number` (rounded) using `nb` bits and to store the result into `s$` beginning at bit position `b`. (The value of `number` is rounded to an integer if necessary, and special conventions apply to negative integers and to integers that do not fit into the specified number of bits.) You can save large amounts of memory space (at the expense of computation speed) by packing integers into a string.

If the bit position `b` is larger than the length of the string, the packed number is added to the end of the string. This can be useful because if you keep appending to a given string, you don't have to keep track of the bit position — instead, just use a very large number for b, such as the **MAXNUM** function.

To recover the numeric value from the string, use the **UNPACKB** function:

```
LET number = Unpackb(s$,b,nb)
```

This statement reverses the operation of the previous call to the **PACKB** subroutine. It converts nb bits from `s$` beginning at bit position b and returns the resulting integer value.

An example of packing and unpacking is included in the section on byte files in Chapter 12 "Files for Data Input and Output." For a more detailed description of the **PACKB** subroutine and the **UNPACKB** function see Chapter 18 "True BASIC Statements and Built-in Functions and Subroutines."

Time and Date Functions

True BASIC provides two string and two numeric functions that let you get time and date values from your computer's internal clock.

The **TIME** function returns the time as the number of seconds since midnight (so use caution when using it in programs that may be run overnight). This makes it easy to time a program:

```
LET t1 = Time                            ! Starting time in seconds
     DO
         CALL RunQuiz (score)            ! User-defined subroutine
     LOOP until score > 90.
LET t2 = Time                            ! Time at completion
PRINT "Time elapsed:"; t2-t1; "seconds"
```

The **TIME$** function returns the time measured by the 24-hour clock as a string in the format `"HH:MM:SS"`, where `HH` represents the hour, `MM` the minute, and `SS` the second. For example: `"08:05:45"` or `"15:16:10"`.

The **DATE** function returns a numerical representation of the current date consisting of the last two digits of the year followed by the ordinal number of the current day within the year. For example, **DATE** would return the value 95041 on February 10, 1995. Because earlier dates always have lower numerical values, this format is useful for sorting date values.

The **DATE$** function returns the date as a string in the format `"YYYYMMDD"`, where `YYYY` represents the year, `MM` the month, and `DD` the day. Thus, `DATE$` would return `"19960714"` on July 14, 1996. Although not the standard date format, dates in this format can be easily sorted. You can also print substrings of these functions to produce desired output. For example:

```
LET today$ = DATE$[5:6] & "/" & DATE$[7:8] & "/" & DATE$[3:4]
PRINT "Today is "; today$
END
```

would produce output such as:

```
Today is 07/14/94
```

[!] Note: For any calculations involving dates cross the year 2000 boundary, we recomment using DATE$. The DATE function will not work properly, as it supplies only the last two digits of the year number.

CHAPTER

9

Arrays and Matrices

An ***array*** is a data structure that allows you to group several numeric or string variables under a single name. It may be one-dimensional ***list*** or ***vector*** or a two-dimensional ***table*** or ***matrix***, or it may have several dimensions. An array may contain either string or numeric values, but a given array may not contain both types of values.

Arrays are an extremely powerful tool for organizing the data used by your program. This chapter introduces the basics of arrays and matrices, as well as several advanced topics, to help you use these powerful data structures efficiently and effectively.

Array Basics

Often, your programs will use a large number of logically related values. Consider, for example, a program to work with a teacher's grade book for a class of 15 students over the course of a semester.

Such a program would need to manipulate several values, including the names of the students and their grades. Using simple variables, sometimes called ***scalars***, the program would need 15 variables to store the names. If there are ten grades per semester, then the program would need another 150 simple variables to store the grades. As you can imagine, building such a program using simple variables would be a real headache.

Fortunately, True BASIC offers arrays for the temporary storage and manipulation of related values. An ***array*** is a named collection of numeric or string values. You can think of an array as a group of variables with the same name. For instance, the grade-book program might use two arrays — one named `names$` containing the student names and another named `grades` containing their grades.

[!] Note: Like any variable, values stored in an array remain there only during the program run. You must supply the values to your program as some form of input — from the keyboard, from **DATA** statements, or from a file. If you wish to preserve array data for future use, you should have the program write the array contents to a file. Using arrays along with data files gives you a powerful tool for manipulating large amounts of data. This chapter gets you started with arrays; see Chapter 13 "Files for Data Input and Output" for more information on using data files.

The individual variables contained in an array are called its ***elements***. All of the elements of a given array must be of the same type — either numbers or strings. Thus, arrays are often referred to as numeric arrays or string arrays, depending upon the values they can contain. You may use any valid numeric variable name for a numeric array, as in `grades`, or any valid string variable name for a string array, as in `names$`. However, you may not use the same name for both a simple variable and an array.

Before you can use an array, you must define it in a **DIM** (dimension) statement. A **DIM** statement tells True BASIC that you will be using the specified name as an array variable. The form of the name tells True BASIC whether you will be using the array to store string or numeric values, and what follows specifies the number of dimensions and the size of each dimension. For example:

```
  DIM names$(15), grades(15, 10)
```

defines the string array `names$` as a list of 15 elements. Each element in `names$` may be treated as a separate string value. It also defines the numeric matrix `grades` as a table with 15 rows (in the first dimension) and 10 columns (in the second dimension). The array dimensions specified in a **DIM** statement must be numeric constants, not variables or expressions.

Each element in an array is assigned a number called a ***subscript***. Unless you specify otherwise, these numbers begin at 1. For instance, the fifteen items in the `names$` array would be numbered 1 through 15. Likewise, the fifteen rows in the `grades` matrix would be numbered 1 through 15, and the 10 columns would be numbered 1 through 10.

You use subscripts to refer to specific elements within an array. You specify the subscript in parentheses after the array name. For instance, `names$(5)` represents the fifth item in the `names$` array. With multi-dimensional arrays, you must specify a subscript for each dimension. For instance, `grades(5,3)` represents the item in the third column of the fifth row of the `grades` matrix. You may use any valid numeric expression as a subscript; if the value of the subscript is not an integer, True BASIC will round it to the nearest integer.

When you use a subscript to refer to a specific element of an array, you may use that array element as you would a simple variable of the same type:

```
LET score(i) = i * x
LET cost(n) = october(n,3) * d(3)
PRINT name$(7), age(k)
```

To better understand the use of arrays, let's consider an example. The following program sets up a multiplication table and uses it:

```
DIM product(10,10)

FOR i = 1 to 10                    ! For each row
    FOR j = 1 to 10                ! For each column in current row
        LET product(i,j) = i*j
    NEXT j
NEXT i
DO
   INPUT PROMPT "Enter two integers to multiply; use 0,0 to end: ": a, b
   IF a = 0 and b = 0 then EXIT DO
   PRINT a; "*"; b; "="; product(a,b)
LOOP
END
```

The two **FOR** loops assign values to each element of the two-dimensional array `product`. The value for each element equals its row subscript multiplied by its column subscript. The **INPUT PROMPT** statement gets two numeric values from the user. The program then uses those values as subscripts and prints the value of the corresponding element in `product` — which is the product of the two subscripts.

If one of the input numbers (rounded if necessary) is less than 1 or greater than 10, then a "Subscript out of bounds" error results and the program stops. You can usually adapt your programs to avoid such errors. For example, you could modify this program by placing the **PRINT** statement in an **IF** structure:

```
DIM product(10,10)
FOR i = 1 to 10
    FOR j = 1 to 10
        LET product (i,j) = i*j
    NEXT j
NEXT i
DO
   INPUT PROMPT "Enter two integers to multiply; use 0,0 to end: ": a, b
   IF a = 0 and b = 0 then
      EXIT DO
   ELSEIF a < 1 or a > 10 or b < 1 or b > 10 then
      PRINT "Please re-enter, using two numbers from 1 to 10"
   ELSE
      PRINT a; "*"; b; "="; product(a,b)
   END IF
```

```
LOOP
END
```

There are other ways of preventing errors from stopping your program. For more information see Chapter 16 "Error Handling."

The lowest valued subscript in each dimension is that dimension's ***lower bound***. The highest valued subscript in each dimension is that dimension's ***upper bound***. If a **DIM** statement specifies only one numeric constant per dimension, True BASIC uses that number as the dimension's upper bound and assumes 1 as the lower bound. However, you may also specify a dimension's lower bound in a **DIM** statement. For example:

```
DIM profit(1980 to 1995), count(-10 to 10, 3)
```

defines the numeric array `profit` as having a lower bound of 1980 and an upper bound of 1995. Thus, `profit` has 16 elements, which you may refer to using the year as the subscript. In a similar fashion, this statement defines the numeric matrix `count` as having a lower bound of -10 and an upper bound of 10 in the first dimension and a lower bound of 1 and an upper bound of 3 in the second dimension. Thus, the matrix `count` has 63 elements divided into 21 rows of 3 columns each. Note that you may use a colon (:) in place of the word TO in array dimensions.

Once an array has been defined to have a certain number of dimensions, the number of dimensions cannot be changed. However, you can change the size of each dimension. Several ways of altering the size of a dimension are discussed in the sections that follow.

It is important to realize that the **DIM** statement reserves memory (RAM) for each array that it defines. Thus, you should avoid dimensioning arrays you don't need or making arrays much larger than required. Even if you never use them, every element in every array contains a value (0 or the null string if you haven't assigned a specific value) and thus takes up space in memory. The size of your arrays is limited only by the amount of memory available; however, if you are not careful, you may find yourself running out of memory.

Memory considerations are especially important with multi-dimensional arrays. True BASIC lets you create arrays of many, many dimensions, but you should avoid them unless you have good reason to use them. Multi-dimensional arrays can use up memory very quickly. The number of elements in a multi-dimensional array is the product of the sizes of its dimensions. For example:

```
DIM big(20,20,20,20)
```

defines a four-dimensional numeric array `big`. Doing the math, we get 20 x 20 x 20 x 20 = 160,000 elements! Since each numeric value in True BASIC takes up eight bytes, the number of bytes of memory required for `big` equals (roughly) the number of elements times 8, or well over a megabyte of memory. Although it might look relatively innocent, the array `big` could easily cause your program to run out of memory.

As a more extreme example, consider this: a 20-dimensional numeric array, even if each subscript could take on only two values, would require eight megabytes. The point is, quite simply, don't add extra dimensions to an array unless they are absolutely necessary.

Array Input and Output

As the above examples demonstrate, you can treat an individual element of an array as you would treat a simple variable. If you wanted to do something to all the elements in an array, you could use a loop to repeat the same operation for each element:

```
DIM names$(15)
FOR i = 1 to 15
   INPUT name$(i)
NEXT i
```

However, you will often find it easier to perform operations on an entire array at once. True BASIC provides several specialized **MAT** statements that allow you to do exactly that. The simplest examples of **MAT** statements are the **MAT READ**, **MAT INPUT**, and **MAT PRINT** statements. The first of these reads all the elements of the array from **DATA** statements, the second gets the values for the elements from the user, and the last displays all of the values in the array.

The **MAT READ** statement allows your program to read the values for one or more arrays from the current data pool of items in **DATA** statements. It works very much like the **READ** statement except that it reads values into an entire array. For multi-dimensional arrays it uses ***odometer order***, that is, the last subscript runs through its range, then the next subscript (to the left of the last) is increased by one, etc. For a two-dimensional array this means that it reads the first row, then the second row, etc. For example:

```
DIM day$(7), pay(3,2)
MAT READ day$, pay
DATA Monday, Tuesday, Wednesday, Thursday
DATA Friday, Saturday, Sunday
DATA 5.25, 7, 3.75, 12.10, 4.15, 5.35
```

The array `day$` will contain the days of the week. The matrix `pay` is 3 rows by 2 columns, hence:

```
pay(1,1)=5.25      pay(1,2)=7
pay(2,1)=3.75      pay(2,2)=12.10
pay(3,1)=4.15      pay(3,2)=5.35
```

The **MAT INPUT** statement works as does an **INPUT** statement with multiple input items. The **MAT INPUT** statement expects to receive all the elements of the array in the same order as they would be provided for the **MAT READ** statement. The user must enter the correct number of items, in the correct order, separated by commas. If the user enters too few or too many items, they will be asked to re-enter the data. The user can end a line with a comma if more space is needed to enter all the input.

Like the **INPUT** statement, the **MAT INPUT** statement displays a question mark as its default prompt. You can overrule the question mark by using the **MAT INPUT PROMPT** statement. For example,

```
DIM list(7)
MAT INPUT PROMPT "Type 7 numbers: ": list
```

The **LINE INPUT** statement also has an equivalent **MAT** statement:

```
DIM text$(15)
MAT LINE INPUT text$
```

This code segment will prompt the user with question marks for 15 lines of input. It will read each line as one element of the string array `text$`.

The **MAT PRINT** statement displays the contents of an entire array. It prints elements in odometer order, beginning a new line after each row. It leaves a blank line after each array, or after two-dimensional sections of higher dimensional arrays. This makes it much easier to recognize the shape of the array. The normal convention is to print items in print zones, as if a comma had been used as the print separator between elements. As with the **PRINT** statement, the zone width is normally 16 characters, but you may reset it with a **SET ZONEWIDTH** statement. For example, consider two **MAT PRINT** statements added to the earlier sequence:

```
DIM day$(7), pay(3,2)
MAT READ day$, pay
DATA Monday, Tuesday, Wednesday, Thursday
DATA Friday, Saturday, Sunday
DATA 5.25, 7, 3.75, 12.10, 4.15, 5.35
MAT PRINT pay
MAT PRINT day$
END
```

This program produces the following output:

```
 5.25            7
 3.75            12.1
 4.15            5.35

Monday          Tuesday         Wednesday       Thursday        Friday
Saturday        Sunday
```

If you use a semicolon after the array name in a **MAT PRINT** statement, True BASIC will print the elements in a row close together, as if you had used a semicolon as the print separator between elements. For example:

```
MAT PRINT pay; day$
```

will print the elements of the numeric array `pay` close together, and the elements of the string array `day$` in zones. Having no punctuation after the last array name has the same effect as having a comma after it. (Remember that True BASIC prints a space before and after each positive numeric value.)

```
 5.25  7
 3.75  12.1
 4.15  5.35

Monday          Tuesday         Wednesday       Thursday        Friday
Saturday        Sunday
```

To dictate the format of the output, use a **MAT PRINT USING** statement. The **MAT PRINT USING** statement prints the individual elements of the array, formatted by the format string, just as if they were printed one by one with **PRINT USING** statements. If there are more elements than fields in the format string, the string is reused, starting from the beginning. For example:

```
LET format$ = "##.###   "
MAT PRINT USING format$: pay
```

will print numbers rounded to three decimal places and leave three spaces after each number. In this example, two values are printed per line representing the two columns of each row in the table:

```
 5.250    7.000
 3.750   12.100
 4.150    5.350
```

Keep in mind that you may sometimes want to use a loop rather than a **MAT** statement to input or print all the elements of an array. Consider the following simple version of a grade-book program that finds the average grade for each student:

```
DIM names$(6), grades(6,3), averages(6)
MAT READ names$, grades
FOR s = 1 to 6                            ! For each student
    LET grade_total = 0
    FOR g = 1 to 3                        ! Add grades
        LET grade_total = grade_total + grades(s,g)
    NEXT g
    LET averages(s) = grade_total/3       ! Compute average grade
NEXT s
MAT PRINT names$, averages                ! Print results
DATA J. Andersen, S. Bree, D. Cordoza, G. Davison, A. Ellis, M. Feinstein
DATA 90, 92, 96, 90, 88, 85, 78, 84, 83
DATA 77, 79, 81, 85, 89, 84, 85, 94, 86

GET KEY k                              ! Wait for keystroke
END
```

The **MAT PRINT** statement first prints all values of `names$` and then all values of `averages`, giving the following output:

```
J. Andersen     S. Bree         D. Cordoza      G. Davison      A. Ellis
M. Feinstein

 92.666667       87.666667       81.666667       79              86
 88.333333
```

However, you might prefer the output you would get if you replace the statement:

```
MAT PRINT names$, averages
```

with the loop:

```
FOR n = 1 to 6
    PRINT names$(n), averages(n)
NEXT n
```

In this case, the corresponding elements from each array are printed together, as follows:

```
J. Andersen     92.666667
S. Bree         87.666667
D. Cordoza      81.666667
G. Davison      79
A. Ellis        86
M. Feinstein    88.333333
```

What if you find it easier to enter the data keeping student names and grades together, as follows?

```
DATA J. Andersen, 90, 92, 96
DATA S. Bree,     90, 88, 85
DATA D. Cordoza,  78, 84, 83
...
```

To do that you will need to replace

```
MAT READ names$, grades
```

with a more complex, nested loop so that you read an entire row of `grades` for each element in `names$`:

```
FOR n = 1 to 6                  ! For each student
   READ names$(n)               ! Read one name
   FOR g = 1 to 3               ! Read three grades
      READ grades(n,g)
   NEXT g
NEXT n
```

When you are deciding whether to use a **MAT** statement or an equivalent loop with multiple arrays, be sure you consider how the data will be organized as well as how you wish to write the program code.

Any of the **MAT** input and output statements may be used with a channel number to get input from a file or to print results to a printer or file. For example:

```
DIM names$(15)
OPEN #1: name "StuNams"
OPEN #10: printer
MAT LINE INPUT #1: names$           ! Get names
MAT PRINT #10: names$               ! Print to printer
....
```

Arrays used with data files are very helpful in managing and manipulating large sets of related data. Keep in mind, however, that data in a file must be in the correct order and format for the input statement you'll use. If you print to a file and intend to input that data again later, you must print the data in an acceptable format. This issue is discussed in Chapter 12 "Files for Data Input and Output."

Redimensioning Arrays

Unlike many other programming languages, True BASIC allows your program to change the number of elements in each dimension of an array while it is running. This process is known as ***redimensioning***, although the word is somewhat misleading since the number of dimensions cannot be changed. The lower and upper bounds of each dimension, however, can be changed, and so the array may become larger or smaller. You can redimension arrays with the **MAT REDIM** statement or by specifying the new dimensions in certain **MAT** statements.

Here's an example of the **MAT REDIM** statement:

```
DIM table(0 to 2, 5)
MAT REDIM table(4, -1 to 6)
```

The first statement creates the numeric array `table` with three rows numbered 0 to 2 and five columns numbered 1 to 5. The **MAT REDIM** statement changes `table` so that it has four rows numbered 1 to 4 and eight columns from -1 to 6.

Redimensioning has many uses. A typical use is to have the program ask the user for a list and its size, as in the example below. This technique lets you use the same program for arrays with different number of elements, and it helps conserve memory by keeping arrays as small as possible.

```
DIM name$(1)
INPUT PROMPT "How many names: ": n
MAT REDIM name$(n)
MAT INPUT PROMPT "Enter them: ": name$
...
```

You must still use the **DIM** statement to declare that `name$` is a one-dimensional string array, but the number of elements you specify is irrelevant, since you will redimension it when you supply `n` in the **MAT REDIM** statement.

This combination of statements is so common that True BASIC lets you redimension the array directly in a **MAT INPUT**, **MAT LINE INPUT**, or **MAT READ** statement. You can combine the **MAT REDIM** and **MAT INPUT** statements above to form a single **MAT INPUT** statement that includes a variable for the dimension of `name$`:

```
DIM name$(1)
INPUT PROMPT "How many names: ": n
MAT INPUT PROMPT "Enter them: ": name$(n)
...
```

Here is a similar example that reads an array from **DATA** statements:

```
DIM table(10,10)
READ m, n                              ! Actual size
MAT READ table(m,n)                     ! Read correct size
DATA 3, 4
DATA 1,2,3,4,5,6,7,8,9,10,11,12
...
```

This example sets up the array, then reads the first two data items which are used to define the actual size of the table. With this trick, the program can work for tables of any size without your having to rewrite the program. You need to change only the data lists in the **DATA** statements.

The **MAT** assignment statement, described in the "Array Assignment" section below, can also redimension an array when you assign values to its elements.

Finally, there is a version of the **MAT INPUT** statement that allows you to input a one-dimensional array of an unspecified size:

```
MAT INPUT list(?)
```

The question mark in the statement instructs the program to accept a list of any length. True BASIC adjusts the upper bound of the subscript to make the size exactly right for the number of elements the user enters. The user must separate items with commas and end a line with no comma to indicate the end of the input items. Note that you can use the question mark (?) only with the **MAT INPUT** statement and only for one-dimensional arrays.

Redimensioning with the **MAT READ** or **MAT INPUT** statement changes both the shape and the contents of an array. **MAT REDIM** will preserve all or part of the contents of an array. While this may be useful, you should be careful in using it, as shown below.

Suppose that the 2-by-2 array `sample` contains the following values:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

and you then use the statement:

```
MAT REDIM sample(3,2)
```

The array is now:

$$\begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 0 & 0 \end{pmatrix}$$

as you would expect. But if you use the statement:

```
MAT REDIM sample(2,3)
```

the result would be:

$$\begin{pmatrix} 1 & 2 & 3 \\ 4 & 0 & 0 \end{pmatrix}$$

which may not be what you want.

Two precautions will prevent this problem. If you wish to use the **MAT REDIM** statement and retain the previous contents of the array, then

- do not change the lower bound(s), and
- redimension only one-dimensional arrays or only the first dimension of two-dimensional arrays.

The **MAT REDIM** statement can be extremely useful when you need to make the most of the memory available to your program. A dimension of 0 effectively removes the array from memory, thus freeing the space it occupied. Therefore, when you need as much memory as you can muster, redimension any unnecessary arrays to zero elements in each dimension. Beware, however, that in doing so you will lose the contents of that array; use this technique only when you no longer need the contents of an array.

Functions That Find Array Sizes

Along with letting you redimension arrays, True BASIC provides functions that let you find out the current sizes of arrays. Three functions may be used to discover the ranges of subscripts:

Subscript Range Functions

Function	**Result**
LBOUND(array,d)	Lower bound of subscript in dimension `d` of `array`
UBOUND(array,d)	Upper bound of subscript in dimension `d` of `array`
SIZE(array,d)	Total number of elements in dimension `d` of `array`

If the array has only one dimension, you may omit the second argument `d` in the argument list of the **LBOUND** and **UBOUND** functions. If you omit the second argument `d` in the argument list of the **SIZE** function, the function returns the total number of elements in the entire array. For example, the following program inputs a list and prints it in reverse order:

```
DIM list(10)
PRINT "Enter a list of numbers: "
MAT INPUT list(?)              ! Input any number of items & redimension
LET n = Size(list)             ! How many numbers
FOR i = n to 1 step -1         ! Reverse order
    PRINT list(i);
NEXT i
END
```

These array subscript functions are also useful for writing array-handling subroutines. The following code segment searches the array `name$` for a particular name `n$`:

```
LET u = Ubound(name$)
FOR i = 1 to u                          ! i will be position in list
    IF name$(i) = n$ then EXIT FOR
NEXT i
IF i > u then LET i = 0                 ! Not found
```

Array Assignment

As you have seen, loop structures provide a concise and convenient mechanism for processing the individual elements of an array in series. The following program segment that copies the contents of one array into another:

```
DIM source(0), target(0)
PRINT "Enter a list of numbers: "
```

```
MAT INPUT source(?)            ! Input any number of items & redimension
LET n = Size(source)           ! How many numbers entered
MAT REDIM target(n)            ! Redimension target array
FOR i = 1 to n                 ! Copy source to target
    LET target(i) = source(i)
NEXT i
```

Just as the **MAT READ**, **MAT INPUT**, and **MAT PRINT** statements let you input or print an entire array instead of using a loop to operate on each element, the **MAT** statement lets you copy the entire contents of one array to another. The following equivalent of the previous example shows how it simplifies array operations:

```
DIM source(0), target(0)
PRINT "Enter a list of numbers: "
MAT INPUT source(?)            ! Input any number of items & redimension
MAT target = source            ! Copy source to target
```

The **MAT** statement assigns values to each element in an array. You can think of it as a specialized version of the **LET** statement that operates exclusively on arrays.

When you assign the values in one array to another by using an array variable on the right side of the **MAT** statement, the two arrays must be of the same type (numeric or string) and must have the same number of dimensions. If necessary, True BASIC automatically adjusts the upper bounds of the array being assigned to. For example:

```
DIM growth(2,1991 to 1993), temp(1,1)
MAT READ growth
DATA 25.6, 13.92, 15.2, 29.89, 12.64, 28.01
MAT temp = growth
```

Here the array `temp` takes on the values of the array `growth`, and its upper bounds are adjusted to the proper size. Note that the lower bounds are not changed. Thus, the new dimensions for `temp` are (2,3) after the assignment. To avoid unexpected results, you may wish to specifically dimension arrays used in assignments to have the same lower bounds.

While the **MAT** statement is commonly used to assign the contents of one array to another, you can also use it to assign the same value to each element of an array. If the value to the right of the equal sign is a constant, expression, or variable representing a single value, that single value is assigned to every element of the array specified to the left of the equal sign. For example:

```
DIM name$(10), grades(10,6), factor(10,6), init(6)
INPUT f
MAT name$ = "unregistered"
MAT grades = 100
MAT factor = f
MAT init = ((f*100) / 5)
```

These statements assign the string `"unregistered"` to all 10 elements in the array `name$`, the value 100 to all elements in `grades`, and the value input for the variable `f` to all elements in `factor`. Each element in `init` takes on the value of the expression `(f*100)/5`. Note that you must always use parentheses around an expression representing a single value in a **MAT** statement, as in the last line of the example above.

Built-in Array Constants

True BASIC provides several built-in array constants that you may use with the **MAT** statement. ***Array constants*** are special functions that return a particular array; they may be used only in a **MAT** statement on the right side of the equal sign. For example:

```
MAT word$ = Nul$                     ! all elements null strings
MAT table = Zer(3,4)                 ! 3-by-4 table, all elements 0
MAT table = Con                      ! all elements 1, or "constant"
MAT table = 7*Con(2,6)               ! 2-by-6 table, all elements 7
```

If you give no dimensions with the array constant, the array being assigned to keeps its current dimensions. If you do provide dimensions, the array being assigned to will be redimensioned to the size specified. Note, however, that the lower bound will not be changed, and the upper bound will be adjusted so that the resulting array has the proper number of elements. Consider, for example:

```
  DIM test(10)                      ! A 10-element array
  MAT test = Zer(3:9)               ! A 7-element array
  PRINT Lbound(test), Ubound(test)
  END
```

The **DIM** statement in this example defines the array `test` as having a lower bound of 1 and an upper bound of 10. Given the dimensions supplied with the **ZER** array constant used in the **MAT** statement, you might expect the new lower bound to be 3 and the new upper bound to be 9. However, if you run this program you will see that the lower bound of `test` remains 1 and the upper bound becomes 7. True BASIC retains the original lower bound of 1 and adjusts the upper bound so that `test` contains the same number of elements as in the constant `Zer(3:9)`.

True BASIC's array constants are as follows:

Examples of Array Constants

Constant	Result
`CON`	An array with every element set to 1

`IDN`	An identity matrix. It must be a square two-dimensional array, and it will have elements set to 1 running along the diagonal from top left to the bottom right. All other elements are set to 0. Since the array must be square (both dimensions the same size), you may redimension the array being assigned to by specifying the bounds of one or both dimensions.
	For example `Idn(4)` is equivalent to `Idn(4,4)` and contains the elements:
	`1 0 0 0`
	`0 1 0 0`
	`0 0 1 0`
	`0 0 0 1`

`NUL$`	An array with each element set to the empty or null string `""`
`ZER`	An array with each element set to 0

Array Arithmetic

True BASIC supports all of the fundamental mathematical operations for arrays and matrices. Many of these operations use the standard arithmetical operators with arrays as their operands. Other operations require special built-in functions. This section introduces operator-based array arithmetic and the following section discusses the function-based options. Both sections assume that you are familiar with the general concepts of matrix arithmetic, and make no attempt to teach these concepts.

True BASIC's array arithmetic expressions follow the normal mathematical rules for matrix arithmetic. The array to which the result is assigned must always have the correct number of dimensions, but True BASIC automatically redimensions it to the correct shape by changing the upper bounds. Note that as with array assignment described above, True BASIC redimensions an array by changing the upper bounds only; it does not change the lower bounds. We recommend that all arrays involved in matrix arithmetic have the same lower bounds. Additional restrictions for each operation are described below.

For array expressions, True BASIC allows only a single operator per **MAT** statement. The array operations are addition, subtraction, multiplication, and scalar multiplication.

Array Addition and Subtraction

In array addition and subtraction the specified operation is applied to all pairs of corresponding elements. Thus, the arrays must be of exactly the same size and shape (same number of dimenstions and same number of elements in each dimension). Although the lower and upper bounds for the dimensions are not important, the number of elements is. In the statement:

```
MAT c = a + b     ! Sum of corresponding elements of two arrays
```

the first element in `b` will be added to the first element in `a`, then the second elements in each will be added, then the third, and so forth until all pairs of corresponding elements have been added. The resulting array of sums will be assigned to `c`, and `c` will be redimensioned (if necessary) so that it has the same size and shape as `a` and `b`.

For a more complete example of array addition and subtraction, consider the following program:

```
DIM a(5), b(5), sum(1), diff(1)
MAT READ a, b
DATA 1, 2, 3, 4, 5
DATA 5, 4, 3, 2, 1
MAT sum = a + b
MAT diff = a - b
MAT PRINT sum
MAT PRINT diff
END
```

which produces these results:

```
 6             6             6             6             6

-4            -2             0             2             4
```

Notice that the target arrays `sum` and `diff` must be defined in the **DIM** statement. Although True BASIC will resize them as necessary, it will not create them automatically.

As long as they are of the same size and shape, you can add and subtract two arrays of any number of dimensions.

Array Multiplication

Array multiplication may be applied only to one- and two-dimensional arrays. If both arrays are one-dimensional, they must be the same size and the product will be a one-dimensional array containing one element. The array specified as the first (or left hand) operand of the multiplication operator will be treated as a row vector, and the array specified as the second (or right hand) operand of the multiplication operator will be treated as a column vector. The result will be the "dot product" of the two operand arrays. For example:

```
DIM a(3), b(3), product(1)
MAT a = 2
MAT b = 4
MAT product = a * b
MAT PRINT a
MAT PRINT b
MAT PRINT product
END
```

gives the results:

```
 2             2             2

 4             4             4

 24
```

If the arrays to be multiplied are two-dimensional, the number of columns in the first array must equal the number of rows in the second. The result will have the number of rows of the first array and the number of columns of the second. In other words, if `a` is an `l` by `m` array, then `b` must be `m` by `n`, and the result will be `l` by `n`.

Array Multiplication

$$\underset{a(2,3)}{\begin{array}{|c|c|c|}\hline 1 & 2 & 3 \\ \hline 4 & 5 & 6 \\ \hline \end{array}} \quad * \quad \underset{b(3,4)}{\begin{array}{|c|c|c|c|}\hline 1 & 2 & 3 & 4 \\ \hline 5 & 6 & 7 & 8 \\ \hline 9 & 10 & 11 & 12 \\ \hline \end{array}} \quad = \quad \underset{c(2,4)}{\begin{array}{|c|c|c|c|}\hline 38 & 44 & 50 & 56 \\ \hline 83 & 98 & 113 & 128 \\ \hline \end{array}}$$

As an example of matrix multiplication, consider the following program:

```
DIM a(2,3), b(3,4), product(1,1)
MAT a = 2
MAT b = 4
MAT product = a * b
MAT PRINT a
MAT PRINT b
MAT PRINT product
END
```

which gives the following results:

```
2              2              2
2              2              2

4              4              4              4
4              4              4              4
4              4              4              4

24             24             24             24
24             24             24             24
```

If one array operand of the multiplication operator is one-dimensional and the other is two-dimensional, the product will be one-dimensional. If the first array operand is one-dimensional, it is treated as a row vector (single row with multiple columns) and must match the first dimension of the second array. If the second array operand is one-dimensional, it is treated as a column vector (many rows in one column) and must match the second dimension of the first array.

In other words an array with `m` elements may be multiplied by an `m` by `n` array, or an `l` by `m` array may be multiplied by an array with `m` elements. In the first case the product will be a one-dimensional array with `n` elements, while in the second case the product will be a one-dimensional array with `l` element. For example:

```
DIM a(2), b(2,4), product(1)
MAT a = 2
MAT b = 4
MAT product = a * b
MAT PRINT a
MAT PRINT b
MAT PRINT product
END
```

produces:

```
2              2

4              4              4              4
4              4              4              4

16             16             16             16
```

Scalar Multiplication

You can also multiply an array by a single, non-array value. Such non-array values are often called ***scalar*** values. The scalar value used in the multiplication may be a positive numeric constant, a numeric variable, or any numeric expression enclosed in parentheses. The scalar value must appear as the first operand and may not be preceded by a sign; thus, to use a negative scalar value you must use a variable or enter the value as an expression in parentheses, such as (-4). The array operand, which must appear as the second operand, may be of any number of dimensions. Each element of the array operand is multiplied by the scalar value producing an array of the same size and shape as the array operand. For example:

```
DIM apples(2,3), income(1,1)
LET cost = .59
MAT READ apples
DATA 27, 14, 52, 22, 29, 7
MAT income = cost * apples              ! Scalar multiplication
MAT PRINT USING "$##.##   ": income
GET KEY k                               ! Wait for keystroke
END
```

produces:

```
$15.93   $ 8.26   $30.68
$12.98   $17.11   $ 4.13
```

Multiplication is the only array operation allowed for a scalar. If you wish to add or subtract the same value from all elements of an array, create an array with all elements containing that value and use array addition or subtraction as described previously. Note as well that the only operators defined for array arithmetic are addition, subtraction, and multiplication.

Built-in Functions for Array Operations

True BASIC also supplies some built-in functions that take one- or two-dimensional arrays as their arguments. Some return a single numeric value and others return array values. These functions allow you to perform calculations on arrays beyond those supported by the array operators described in the previous section.

Functions for Array Operations

Function	Result
`DET`	The determinant of the last square two-dimensional array inverted by the **INV** function; returns 0 if no array has been inverted
`DET(x)`	The determinant of the square two-dimensional array `x`; returns 0 if `x` is singular
`INV(x)`	The inverse of the square two-dimensional array `x`; if `x` is singular or nearly singular True BASIC gives an error message
`DOT(x,y)`	The dot product of the two one-dimensional arrays `x` and `y`, which must have the same number of elements
`TRN(x)`	The transposition of the two-dimensional array `x`

The **DET** function returns a numeric scalar value representing the determinant of the square matrix specified as its argument. If the square matrix is singular, the **DET** function returns a value of 0. If no array argument is specified, the **DET** function returns the determinant of the last matrix inverted with the **INV** function. If no matrix has been inverted, a value of 0 is returned.

The **INV** function returns the inverse of the square matrix specified as its argument. The result will be a square matrix of the same dimensions as the argument. If the argument is singular, or nearly singular, then True BASIC generates an error. Since matrix inversions are notoriously susceptible to round-off errors, it is wise to check the value returned by the **DET** function after each inversion to determine if the matrix just inverted was almost singular. If the **DET** function returns a value that is nearly zero, the inverted results are probably meaningless.

The following program demonstrates the use of the **INV** and **DET** functions:

```
DIM scores(2,2), inverse(1,1)
MAT READ scores
DATA 1,2,3,4
MAT inverse = Inv(scores)
MAT PRINT scores
MAT PRINT inverse
PRINT "Determinant: "; Det
END
```

This program produces the following output:

```
 1              2
 3              4

-2.             1.
 1.5           -.5

Determinant: -2
```

The **DOT** function returns a numeric scalar value representing the dot product, or inner product, of the two one-dimensional arrays specified as its arguments. The two arrays must be one-dimensional and must have the same number of elements. Here is a typical use of the **DOT** function:

```
DIM price(12), amount(12)
MAT INPUT PROMPT "Prices: ": price
MAT INPUT PROMPT "Amounts: ": amount
LET total = Dot(amount, price)   ! Total purchase price
...
```

Each element of the `amount` array is multiplied by the corresponding element of the `price` array and the value of `total` is set equal to the sum of these products.

The **TRN** function returns the transposition of the two-dimensional array specified as its argument. The result will be a two-dimensional array with the rows and columns of the argument array interchanged. In other words, the transposition of an `m` by `n` matrix named `x` gives an `n` by `m` matrix which we will call `r` such that `r(i,j)` equals `x(j,i)`. For example, the program:

```
DIM Scores (2,4), Transpose(1,1)
MAT READ Scores
DATA 27, 14, 34, 52, 22,12, 29, 7
MAT Transpose = Trn(Scores)
MAT PRINT Scores
MAT PRINT Transpose
END
```

produces the results:

```
 27             14             34             52
 22             12             29             7

 27             22
 14             12
 34             29
 52             7
```

Remember that you may use only one array or matrix operation per **MAT** statement. You may build more complex matrix expressions, however, using a series of **MAT** statements. The following program illustrates the computation of such a complex expression:_

```
! Compute the inverse of i - (1/2)*a*Trn(a)

DIM a(3,2), i(3,3), x(3,3), c(3,3)
```

```
MAT READ a
DATA .1, .2, .3, .4, .5, .6
MAT x = Trn(a)
MAT x = a * x
MAT x = (1/2) * x    ! (1/2)*a*Trn(a)
MAT i = Idn(3)
MAT x = i - x
MAT c = Inv(x)       ! Inverse
MAT PRINT c
PRINT Det(x)         ! Check determinant
END
```

In this example, only the shape of the array `a` is relevant; the other arrays are redimensioned by **MAT** statements. Note that the "working matrix" `x` occurs on both the left and right side of the equal sign in several statements. True BASIC always evaluates the right side first, then redimensions the matrix on the left (if necessary) and assigns the answer to it. This is why the program above works correctly.

Sorting and Searching the Contents of Arrays

As you work with arrays, you may want to sort the elements within them or search for a particular element. True BASIC provides several subroutines that can sort or search your arrays. A subroutine is a block of statements that carry out a specific task. As with functions, you may need to specify one or more arguments when you invoke a subroutine. Subroutines differ from functions in that you invoke them with a **CALL** statement and they return values via the arguments you specify — thus a subroutine may return more than one value or array.

The sorting and searching subroutines are not built into True BASIC, they are contained in the SortLib.TRC "library file" that is placed in the TBLIBS directory when you install True BASIC. This section introduces these subroutines and the SortLib.TRC library. For complete information on subroutines and library files, see Chapters 10 "User-defined Functions and Subroutines" and Chapter 11 "Libraries and Modules." Briefly, a **LIBRARY** statement must name the file or files containing any non-built-in subroutines your program will use. You then invoke the subroutine with a **CALL** statement that includes arguments for the subroutine, as follows:

```
! Print an alphabetized list of names

LIBRARY "c:\TBSilver\TBLIBS\SortLib.TRC"    ! Use appropriate pathname

DIM names$(1)
PRINT "Input names (last name first), typing a comma"
PRINT "after each name except the last."
MAT INPUT names$(?)
CALL SortS (names$())             ! Sort string array names$

FOR i = 1 to Ubound(names$())
    PRINT names$(i)               ! Print sorted array, one name per line
NEXT i

END
```

At the **CALL** statement, True BASIC carries out the task defined by the specified subroutine. That task may assign new values to the argument in the **CALL** statement as it does for the `names$` array above. (The **SortS** subroutine is described more fully below.)

Most of the sorting and searching subroutines are in the library file SortLib.TRC (saved in TBLIBS). The **LIBRARY** statement must use the appropriate "pathname" format to indicate the location of the library file for the computer you will use to run your program.

SortLib.TRC includes subroutines for simply and quickly sorting, reversing, or searching the elements of both numeric and string arrays. These simple sorts are very fast and require little memory as the sort is done "in place" — the original array is replaced by the values of the sorted array.

SortLib.TRC also includes subroutines for "pointer sorts" and customized "comparison sorts." A ***pointer sort*** does not change the original array, but instead creates an index array containing subscript pointers to the sorted values of the original array. Pointer sorts are helpful if it is important to keep the original order for ties or if you wish to use the sorted order of one array to print values from other "parallel" arrays.

A ***comparison sort*** lets you customize the way values will be sorted or sort values based on one or more specific "key fields." You may customize both simple and pointer sorts.

Simple Sorts and Searches

The simple sorting and searching subroutines in the SortLib.TRC library are as follows:

Simple Sorting and Searching Subroutines

Subroutine	Library	Result
`SortN(a())`	SortLib	Sorts numeric array a into ascending order
`SortS(a$())`	SortLib	Sorts string array `a$` into ascending order
`ReverseN(a())`	SortLib	Reverses order of elements in numeric array `a`
`ReverseS(a$())`	SortLib	Reverses order of elements in string array `a$`
`SearchN(a(),n,i,f)`	SortLib	Searches the sorted numeric array a for the value n; if found, reports the subscript value as n and f as a non-zero value
`SearchS(a$(),s$,i,f)`	SortLib	Searches the sorted string array `a$` for the value `s$`; if found, reports the subscript value as `n` and `f` as a non-zero value

The **SortN** and **SortS** subroutines sort numeric and string arrays into ascending order. They arrange values according to the standard meaning of True BASIC's <= operator. Numeric values are sorted in ascending order by value and strings are sorted according to code values for the standard character set; thus, uppercase characters come before lowercase characters (see Appendix A for code values of the standard character set).

To use these subroutines, you must name SortLib.TRC in a **LIBRARY** statement (using the pathname format appropriate for your computer) and invoke the subroutine with a **CALL** statement. Both of these subroutines require one argument — a numeric array for the **SortN** subroutine or a string array for the **SortS** subroutine. When the **CALL** statement is executed, the subroutine sorts the elements in the array passed as an argument. The subroutine changes the contents of the array, returning an array with all of its elements sorted into ascending order. For example, the following program:

```
! Sort a list of scores

LIBRARY "SortLib.TRC"                    ! Use appropriate pathname

DIM scores(10)
DO WHILE MORE DATA
   LET i = i + 1
   READ scores(i)
LOOP
DATA 99.5, 87.5, 89, 93.25, 89, 75, 80
CALL SortN (scores())                    ! Sort in ascending order
MAT PRINT scores                         ! Print the result
END
```

prints the following:

```
 0              0              0              75             80
 87.5           89             89             93.25          99.5
```

Notice that the **SortN** subroutine sorts the entire array. It keeps all tie values and includes zeroes for unassigned values.

The **SortS** subroutine works similarly for string arrays, merging in null strings for any unassigned values. Remember that characters are sorted by their character-code value. Thus the program:

```
! Sort a list of phrases

LIBRARY "SortLib.TRC"                         ! Use appropriate pathname

DIM words$(10)
DO WHILE MORE DATA
   LET i = i + 1
   READ words$(i)
LOOP
MAT REDIM words$(i)                           ! Eliminate unassigned elements
DATA zebra, orangutan, Tiger
DATA apples, apple pie, tiger
DATA "widget, small", "widget, large"
CALL SortS (words$())                         ! Sort in ascending order
MAT PRINT words$                              ! Print the result
END
```

produces the following output:

```
Tiger           apple tarts     apples          orangutan       tiger
widget, large   widget, small   zebra
```

Here, "`Tiger`" and "`tiger`" are not equivalent because uppercase and lowercase letters are different characters; and "`Tiger`" precedes all the lowercase words because the uppercase alphabet precedes lowercase letters in the standard character code. Similarly "`apple tarts`" comes before "`apples`" because the space character comes before letters in the standard character set. (If you wish to sort uppercase and lowercase letters as equivalents you could create a second array with all uppercase or lowercase elements and sort that or use a comparison sort as explained below.)

Notice that the above program eliminates the problem of null strings by redimensioning the array to eliminate unassigned elements before sorting.

The **SortN** and **SortS** subroutines are very fast and, because they work "in place" in the original array, they use very little memory beyond that already required by the array itself. They gain speed at the sake of complexity, however. The additional sorting subroutines described in the rest of this chapter give you greater control over how elements are sorted.

The **SortN** and **SortS** subroutines sort in ascending order only. You can easily sort in descending order, however, by reversing the elements after a simple ascending sort. The **ReverseN** and **ReverseS** subroutines (also contained in SortLib.TRC) reverse the order of the numeric or string array argument, respectively. For example, you could adapt the numeric sort example above to print scores in reverse order by adding a call to the **ReverseN** subroutine:

```
! Sort a list of scores

LIBRARY "SortLib.TRC"                         ! Use appropriate pathname

DIM scores(10)
DO while more data
   LET i = i + 1
   READ scores(i)
LOOP
DATA 99.5, 87.5, 89, 93.25, 89, 75, 80
CALL SortN (scores())                         ! Sort in ascending order
CALL ReverseN (scores())                      ! Reverse the order
MAT PRINT scores                              ! Print the result
END
```

This program prints the following:

```
 99.5           93.25          89             89             87.5
 80             75             0              0              0
```

Similarly, you could use the **ReverseS** subroutine with a sorted string array.

The **SearchN** and **SearchS** subroutines search sorted numeric or string arrays for a specified value. These subroutines take four arguments: the array to be searched, the value to search for, and two numeric variables that are used to return the results of the search. For example:

```
LIBRARY "SortLib.TRC"           ! Use appropriate pathname

DIM array(20)
FOR i = 1 to 20
   LET array(i) = Int(100*Rnd) + 1  ! Create array of 20 random numbers
NEXT i
CALL SortN(array())             ! Sort random numbers in ascending order
DO
   INPUT PROMPT "Enter a number from 1 to 100 (0 to quit): ": number
   IF number <= 0 then EXIT DO
   CALL SearchN (array(), number, index, found)
   IF found <> 0 then
      PRINT number; "found at element"; index
   ELSE
      PRINT number; "not found"
   END IF
LOOP

END
```

The last argument, `found` in the example above, returns 0 if the search value (`number` above) is not found or a non-zero value if it is found. The third argument, `index` above, returns the subscript of the element if the value is found. If the value is not found, `index` equals the subscript where the value would have been stored in the sorted array if it existed.

The **SearchS** subroutine works the same way except that the array and search value must be strings. Note that the array passed to the **SearchN** or **SearchS** subroutine must be sorted into ascending order.

[!] Note: The **SearchN** and **SearchS** subroutines must be passed arrays that are already sorted into ascending order because these subroutines use a binary search, which is generally much faster than an element-by-element, or "sequential," search. In a binary search, the subroutine looks first at the element at the mid-point of the array. If that element does not equal the search value, the routine finds out if it is greater than or less than the search value. The binary search then "ignores" the half of the array that is too small or too large and looks at the mid-point of the remaining range of elements. This process continues until the value is found or two consecutive elements are found to bracket the search value. The values of the array must therefore be arranged in order from smallest to largest.

Pointer Sorts

Though quick and efficient, the **SortN** and **SortS** subroutines lose the original order of elements in the array. This could be a problem if, for example, you have separate arrays for names, addresses, and phone numbers. If you use the **SortS** subroutine to change the order of elements in the array of names, you would lose the relationship between the arrays. For such cases, you should use a "pointer sort." The two basic pointer sorts are implemented by the **PSortN** and **PSortS** subroutines for numeric and string arrays, respectively.

Subroutine	**Library**	**Result**
`PSortN(a(),i())`	SortLib	Performs a "pointer sort" on values in numeric array `a` and stores the sorted pointers, or indices, in the array `i`
`PSortS(a$(),i())`	SortLib	Performs a "pointer sort" on values in string array `a$` and stores the sorted pointers, or indices, in the array `i`

In a ***pointer sort,*** the original array is not changed. Instead, the subroutine sorts the values and creates an index array whose elements are subscripts pointing to the sorted values of the first array. For example, here's an adaptation of the earlier phrase-sorting program:

```
! Sort a list of phrases

LIBRARY "SortLib.TRC"             ! Use appropriate pathname

DIM words$(10), index(10)
DO WHILE MORE DATA
   LET i = i + 1
   READ words$(i)
LOOP
MAT REDIM words$(i), index(i)     ! Eliminate unassigned elements
DATA zebra, bananas, orangutan
DATA apples, kiwis
CALL PSortS (words$(), index())   ! Sort in ascending order
PRINT "The words$ array contains the values:"
MAT PRINT words$                  ! Print the original array
PRINT "The index array contains the values:"
MAT PRINT index                   ! Print the index array
PRINT "Thus, the sorted values of words$ are:"
FOR n = 1 to i
   PRINT words$(index(n)),        ! Print words$ elements in sorted order
NEXT n
PRINT

END
```

This gives the results:

```
The words$ array contains the values:
zebra          bananas        orangutan      apples         kiwis

The index array contains the values:
 4              2              5              3              1

Thus, the sorted values of words$ are:
apples         bananas        kiwis          orangutan      zebra
```

Note that the order of elements in `words$` has not changed, but `index` gives the order in which the elements of `words$` should be read to produce a sorted list. The first word in the sorted list is element 4 or `"apples"`, the second word is element 2 or `"bananas"`, and so on. As with simple sorts, pointer sorts sort the entire array. Thus, if an array contains unassigned elements, the resulting index array will contain pointers for all the zeros or null strings in the array.

Pointer sorts are especially useful if your program uses "parallel" arrays. If you have one array containing student names and another containing their grades, you can do a pointer sort on grades and then print both arrays in sorted order using the index array. The following example does just that, and uses the **ReverseN** subroutine to sort the index array in descending order:

```
! Sort students by grades

LIBRARY "SortLib.TRC"              ! Use appropriate pathname

DIM names$(5), grades(5), index(5)
MAT READ names$
MAT READ grades
DATA Adams, Bell, Cosi, Du, Eisen
DATA 77,    94,   88,   80, 95
CALL PSortN (grades(), index())    ! Sort in ascending order
CALL ReverseN (index())            ! Reverse indices
FOR n = 1 to 5
   PRINT names$(index(n)), grades(index(n))
NEXT n
PRINT

GET KEY k                          ! Wait for keystroke
END
```

This produces the following output:

```
Eisen          95
Bell           94
Cosi           88
Du             80
Adams          77
```

Pointer sorts are also helpful if you need a ***stable sort***, which keeps the original order in case of ties. This is not important for simple sorts where the whole string is compared. But if you sort by just part of the data, as you can do with comparison sorts, this may be important.

Customized Sorts and Searches

The sorting routines you've seen so far all use the usual True BASIC <= operator to sort values into ascending order, and they consider the entire value in making the comparison. ***Customized*** sorting and searching routines let you sort or search on one or more key parts of the data or define exactly how to compare two values. Thus, you could choose a sort that ignores the difference between uppercase and lowercase letters, or devise one that sorts roman numerals in the correct order.

SortLib.TRC contains custom-sorting versions of all the sorting and searching subroutines introduced so far (except the reversing routines, for which custom comparisons are not needed):

Customized Comparison Sorting Subroutines

Subroutine	Result
`CSortN(a())`	Sorts numeric array `a` in ascending order using customized comparison routine called `CompareN`
`CSortS(a$())`	Sorts string array `a$` in ascending order using one or more special options
`CSearchN(a(),n,i,f)`	Searches the sorted numeric array `a` for the value `n` using a customized comparison routine called `CompareN`; if found, reports the subscript value as `i` and `f` as a non-zero value
`CSearchS(a$(),s$,i,f)`	Searches the sorted string array `a$` for the value `s$` using one or more special options; if found, reports the subscript value as `i` and `f` as a non-zero value

`CPSortN(a(),i())`	Performs a "pointer sort" on values in numeric array `a` using a customized comparison routine called `CompareN` and stores the sorted pointers, or indices, in the array `i`
`CPSortS(a$(),i())`	Performs a "pointer sort" on values in string array `a$` using one or more special options to customize the sort, and stores the sorted pointers, or indices, in the array `i`

First, let's see how we can sort a string array ignoring case. Before calling the subroutine **CSortS**, we call the subroutine **Sort_IgnoreCase**.

```
! Sort a list of phrases ignoring case

LIBRARY "SortLib.TRC", "CompCase.TRC"   ! Use appropriate path names
DIM words$(10)
DO while more data
   LET i = i + 1
   READ words$(i)
LOOP
MAT REDIM words$(i)
DATA zebra, ELEPHANTS, Tiger
DATA apples, tiger, Llama
DATA Widget, Oranges
CALL Sort_IgnoreCase                    ! Treat upper- and lowercase alike
CALL CSortS (words$())                  ! Sort in order defined in CompareS
MAT PRINT words$                        ! Print the result
END
```

This program prints the following:

```
apples         ELEPHANTS      Llama          Oranges        Tiger
tiger          Widget         zebra
```

Besides ignoring case, there are several other options.

Special Customized Sorting Options

Subroutine	Result
CALL Sort_Off	Remove all special string sorting options
CALL Sort_ObserveCase	Do not ignore case (default)
CALL Sort_IgnoreCase	Ignore distinction between upper- and lowercase
CALL Sort_NiceNumbers_on	Use "intuitive" ordering for numbers within strings
CALL Sort_NiceNumbers_off	Ignore numbers in strings (default)
CALL Sort_NoKey	No key fields (default)
CALL Sort_OneKey (f1, t1)	One key field
CALL Sort_TwoKeys (f1, t1, f2, t2)	Two key fields

As an example of a sort based on a key field, here's a program that sorts strings based on area code and then last names within area codes:

```
LIBRARY "SortLib.TRC"                   ! Use appropriate pathname

DIM phonelist$(4)
MAT READ phonelist$
DATA "Smith     Rosario   802-543-1234"
DATA "Li        Steven    617-123-1200"
DATA "Arndt     J. K.     802-331-3333"
DATA "de Forbe  Francis   205-256-2424"
CALL Sort_TwoKeys (20, 22, 1, 9)
CALL CSortS (phonelist$())
```

```
FOR i = 1 to 4
   PRINT phonelist$(i)
NEXT i

END
```

This produces the output:

```
de Forbe  Francis   205-256-2424
Li        Steven    617-123-1200
Arndt     J. K.     802-331-3333
Smith     Rosario   802-543-1234
```

As an example of sorting numbers "intuitively," imagine you have strings containing numbers, such as A1, A2, A3, A10, B1, B2, B12, and so on. The **SortS** or **PSortS** subroutine would arrange these as:

```
A1    A10   A2    A3    B1    B12   B2
```

Calling the subroutine **Sort_NiceNumbers_on** before calling **CSortS** subroutine would sort them as follows, putting the numeric text in proper numeric sequence:

```
A1    A2    A3    A10   B1    B2    B12
```

You may use customized comparisons with searches as well. Since the **CSearchN** and **CSearchS** subroutines both use the binary search method, the data must first be sorted. For example, if you wish to search the phone list in the example above by last name, you should first sort by the last-name field, as follows.

```
LIBRARY "SortLib.TRC"                      ! Use appropriate pathname

DIM phonelist$(4)
MAT READ phonelist$
DATA "Smith     Rosario   802-543-1234"
DATA "Li        Steven    617-123-1200"
DATA "Arndt     J. K.     802-331-3333"
DATA "de Forbe  Francis   205-256-2424"
CALL Sort_IgnoreCase
CALL Sort_OneKey (1, 9)
CALL CSortS (phonelist$())                 ! Sort using chars 1 through 9 only
INPUT PROMPT "Enter last name: ": find$

LET find$ = (find$ & "        ")[1:9]      ! Make 9 characters long


CALL CSearchS (phonelist$, find$, index, found)      ! Search with option
IF found <> 0 then
   PRINT phonelist$(index)
ELSE
   PRINT "Not found"
END IF

END
```

Observe that you must use the same customized options for the search phase as for the previous sort phase.

You can write your own customized comparison routines. Note that the sort and search subroutines whose names begin with the letter "C" invoke a special comparison subroutine CompareN or CompareS, for numeric arrays and string arrays, respectively. The `CompareN` and `CompareS` subroutines use the same three-parameter format:

```
CompareN (a,b,r)
```

or

```
CompareS (a$,b$,r)
```

The first two parameters pass the two values to be compared — numeric or string — from the sorting routine to the comparison routine. The third parameter must be a numeric variable that returns a value to indicate the sort order, as follows:

> `r` must return -1 if `a` should come before `b`, i.e., `a < b`
>
> `r` must return 0 if `a` and `b` tie, i.e., `a = b`
>
> `r` must return 1 if `a` should come after `b`, i.e., `a > b`

If you have special sorting requirements, you should examine the source code for the sorting library, found in `SortLib.tru.` You can precede each line in **CompareN** or **CompareS** with an exclamation point "!", which is the comment character for True BASIC. Then simply write your own routine without exclamation points! Use the existing routines as patterns. The routines **CSortS**, etc., are already set up to call a subroutine by the name **CompareS**, etc.

CHAPTER

10

User-defined Functions and Subroutines

True BASIC has three structures that let you break up a program into smaller units: user-defined ***functions***, ***subroutines***, and ***pictures***. These structures make it easier to write and debug programs because you can build large programs from small pieces. This chapter shows how to define and use your own functions and subroutines. Pictures are for graphics only and are discussed in Chapter 13 "Graphics."

You've already seen many of the functions built into True BASIC in Chapter 8 "Built-in Functions." However, you may not always find a built-in function that calculates the value you need; you may wish to define your own functions. Like a built-in function, a user-defined ***function*** always returns a single value of a specific type and is unable to change the value of any arguments passed to it. You may use (or "invoke") a function wherever you may use an expression of the same type. Once the function has been evaluated, its return value is substituted at the place it is invoked.

A ***subroutine*** is a block of statements that carries out one or more tasks. A subroutine does not return a value as a function does, but it can return values by changing the values of arguments passed to it. You invoke a subroutine with a **CALL** statement.

Within their definitions, functions and subroutines may invoke themselves. This is known as ***recursion***.

Functions and subroutines may be part of your main program, in which case they are ***internal procedures*** and they share all variables with the rest of the main program. Functions and subroutines may also be placed outside the main program as ***external procedures*** whose variables are local or "sealed off" from the main program and any other external procedures.

Defining Functions

[!] **Note:** The keywords **DEF** and **FUNCTION** are synonymous throughout the True BASIC language and may be used interchangeably. In the following discussion, the **DEF** keyword is used solely for the sake of simplicity. You may use **FUNCTION** if you feel that it makes your programs easier to read and maintain.

You use either a single-line **DEF** statement or a more complex **DEF** structure to define your own functions. The simple **DEF** statement:

```
DEF Sech(x) = 1 / Cosh(x)
```

defines the hyperbolic secant function. Once you have defined a function in your program, you may use it in any appropriate expression, such as:

```
LET factor1 = Sech(-.123)
```

Each function has a ***name*** and may have ***parameters*** whose values you supply when you invoke the function. In the example above, `Sech` is the name of the function and `x` is its one numeric parameter. When you invoke the `Sech` function, you must supply a single numeric value as an ***argument***.

Before evaluating the function, True BASIC will assign the value of the argument to the variable used as the parameter. In the example above the function is invoked with `Sech(-.123)`, where the value –.123 is the argument. True BASIC assigns this value to the parameter variable `x` in the function definition. Thus, the expression `1/Cosh(x)` is evaluated as `1/Cosh(-.123)`. The value of this expression becomes the ***return value*** of the function and is substituted in place of the function invocation in the **LET** statement. Note that the value of the argument is assigned to the parameter variable; functions cannot change the values of arguments passed to them.

[!] Note: When variable arguments are passed to a function, the values of those arguments are assigned to the corresponding parameters within the function definition. True BASIC considers the parameters in a function definition to be distinct from the arguments; that is, an argument and its matching parameter are different variables. Because they are different variables, changes to parameters within the function definition have no effect on the values of their corresponding arguments. Thus, functions cannot change the values of the arguments passed to them. This parameter passing mechanism is called ***passing by value.***

The **DEF** statement restricts the definition of the function to a single line, which is fine for simple functions. Often, however, more complex function definitions require looping or decision structures that cannot be expressed on a single line, or they may simply be easier to create as a series of steps. For functions longer than a single line, you use a **DEF** structure. For example:

```
DEF Gcd(a,b)                    ! Greatest common divisor
    DO
       LET r = Mod(a,b)         ! Remainder in division
       IF r = 0 then EXIT DO    ! We are done
       LET a = b                ! Else iterate
       LET b = r
    LOOP
    LET Gcd = b                 ! Value of the function
END DEF
```

Here `Gcd` is the name of the function, and `a` and `b` are the parameters. A **DEF** structure begins with a **DEF** statement that names the function and its parameters and ends with an **END DEF** statement. A **LET** statement must assign a value to the function name (`Gcd`) before the end of the definition. Once you've defined the function, you may use `Gcd` with two numeric arguments as a numeric expression. For example:

```
PRINT Gcd(121,55)
```

will print the value 11.

The rules for naming functions are the same as those for naming variables. However, you may not use the same name for both a variable and a function. The return value of a function may be either a number or a string; for string functions, the name must end with a dollar sign. Before you may use a function in a program, you must first define it with a **DEF** statement or structure or you must name the function in a **DECLARE DEF** statement (see the discussion of external functions later in this chapter).

Parameters may be numeric variables, string variables, or names of arrays. When you invoke a function, the arguments you provide must match the parameters named in the **DEF** statement. True BASIC matches arguments and parameters by the order in which they appear:

```
DEF   abcdef       (x,     z$,    u)
                    ↑       ↑      ↑
LET n = abcdef     (3.2, "dog",   y)
```

Here the value 3.2 is assigned to `x`, `"dog"` is assigned to `z$`, and the value of `y` is assigned to `u`. The value of `abcdef` is then computed according to its definition, and the answer is assigned to `n`. `"Cat"` would not be legal as the last argument, since you cannot assign a string to the numeric variable `u`.

You may also define functions without parameters. For example:

```
DEF Die = Int(6*Rnd) + 1                    ! Simulate one die
LET dice2 = Die + Die                       ! Sum of two dice
END
```

You may exit from the middle of a **DEF** structure with an **EXIT DEF** statement. Take care that you assign a value to the function before the exit, or the function will return the default value (0 or the null string).

Arrays as Parameters

Here's an example of a function that uses an array as a parameter. It finds the largest value in a one-dimensional numeric array:

```
DEF Max_element(a())
    LET largest = a(Lbound(a))
    FOR i = Lbound(a)+1 to Ubound(a)
        IF a(i) > largest then LET largest = a(i)
    NEXT i
    LET Max_element = largest
END DEF
```

An array parameter is indicated by empty parentheses, or "bowlegs," with commas to tell True BASIC how many dimensions that array contains. The number of commas is always one less than the number of dimensions. Thus, `()` represents a one-dimensional array, `(,)` a two-dimensional array, and `(,,)` a three-dimensional array.

When the function is invoked, the use of bowlegs with the array argument is optional. Thus, either of the following is legal:

```
PRINT "The highest score is"; Max_element(scores)

PRINT "The highest score is"; Max_element(scores())
```

as long as `scores` has been declared as a one-dimensional array in a **DIM** statement. It is the argument `scores` that requires a **DIM** statement, not the parameter `a()`. Remember that the parameter will be assigned the value of the argument. Therefore, the size of `a` will be adjusted to reflect the size of `scores`.

[!] **Note:** Since each parameter in a function definition is a separate copy of its associated argument, arrays as function parameters are not particularly efficient. Each time you invoke a function containing an array parameter, True BASIC must create a copy of the array passed as an argument. For large arrays, this can require a lot of time and memory. If you need to pass arrays to a procedure, try to use a **SUB** structure instead — **SUB** structures pass array parameters by a more efficient mechanism.

Recursive Functions

Within a function definition, the name of the function may normally occur only on the left side of the equal sign (=) in a **LET** statement. That is why we used the temporary variable `largest` for the computations within `Max_element`, assigning a value to the function name only at the end of the definition. An exception to this rule is ***recursion***, whereby a function may invoke itself. It is useful in many circumstances, but it is somewhat tricky to master. If you are unfamiliar with recursive programming, you may want to find a book on programming techniques or some other resource from which to learn the technique.

One of the most common examples of a recursive function is the calculation of a factorial:

```
DEF Fact(n)                                 ! Factorial function or n!
    IF n = 0 then
       LET Fact = 1
    ELSE
       LET Fact = n*Fact(n-1)               ! Recursion
    END IF
END DEF
```

If you ask for `Fact(7)`, the value of the function is `7*Fact(6)`, which invokes the same function to compute `Fact(6)` and so on. Notice, however, that when the value of the parameter is 0, the function no longer invokes itself, thus ending the "recursive chain" and allowing the previous invocations of the function to be completed in reverse order.

Variables Within Function Definitions

The simplest way to use a function is to define it at the beginning of the program that will use it; in other words, the function definition is "internal" to the main program. If you do this, however, you must be careful with the variables you use within the function definition. Look again at the `Max_element` function defined earlier in this chapter:

```
DEF Max_element(a())
    LET largest = a(Lbound(a))
    FOR i = Lbound(a)+1 to Ubound(a)
        IF a(i) > largest then LET largest = a(i)
    NEXT i
    LET Max_element = largest
END DEF
```

Notice that, besides finding the largest value in the array, `Max_element` changes the values of `largest` and `i`. If the invoking program uses variables by the same name (and the `Max_element` definition is contained in that program), the program's variables are also changed — the variables are ***global*** to the program and the function definition. Be careful how you use such a function. For instance, if you invoke the function inside a loop that starts with the statement:

```
FOR i = 1 to 10
```

chaos will result. Or, if you had previously defined a variable named `largest`, its value would be changed whenever this function is invoked.

You can avoid this sort of problem in two ways. First, you could list `largest` and `i` in a **LOCAL** statement within the **DEF** structure. This would force True BASIC to create separate variables named `largest` and `i` for each invocation of the function, preventing any changes to these variables from affecting those outside the **DEF** structure. Or, you could define `Max_element` as an "external function" where variables are effectively "sealed off" from the rest of the program. The **LOCAL** statement and external procedures are described later in this chapter.

Defining Subroutines

Functions are useful, but they're not ideal for all situations. For even greater flexibility in organizing your programs, True BASIC lets you define subroutines in addition to functions.

Subroutines, unlike functions, do not have a return value. This means that they cannot be invoked as part of an expression; instead, subroutines are invoked with a **CALL** statement. As you will see below, subroutines also differ from functions in the mechanism used to pass parameters — subroutines may change the values of their arguments. The differences between functions and subroutines make each type of structure better suited to certain uses than the other.

In general, functions are most useful when you need to calculate and return a single string or numeric value, and subroutines are most useful everywhere else. Subroutines help you organize your programs by packaging well-defined tasks into discrete structures. This packaging of tasks makes it easy to reuse that code both within the current program and in future projects. You need to define a subroutine only once but you may call it as many times as necessary.

For example, programs frequently ask questions that require a "yes" or "no" answer. A good program would check whether the answer really was legal and allow the user to type answers in upper or lower case. The following subroutine meets all of these needs, and can be quite useful:

```
SUB Yes_no(qu$, ans$)                  ! Get a "yes" or "no"
    DO
```

```
        PRINT qu$;                          ! Ask the question
        INPUT ans$                          ! Get the answer
        LET ans$ = Lcase$(ans$)             ! All lc, easy to check
        IF ans$ = "yes" or ans$ = "no" then EXIT DO    ! Ok
        PRINT "Answer 'yes' or 'no'"        ! Else try again
     LOOP
  END SUB
```

You may invoke this subroutine as follows:

```
  CALL Yes_no("Shall I go on", a$)
  IF a$ = "no" then STOP
```

The **SUB** structure begins with a **SUB** statement that names the subroutine and any parameters, and it ends with an **END SUB** statement. In the example above, `Yes_no` is the name of the subroutine and `qu$` and `ans$` are its two string parameters. When you invoke the `Yes_no` subroutine with a **CALL** statement, you must supply a string value as its first argument and a string variable as its second argument.

Let's examine this distinction between the two arguments a bit further. The first parameter `qu$` is an ***input parameter***, which means that the definition of `Yes_no` uses — but does not change — its value. Thus, if you want the **PRINT** statement within `Yes_no` to print anything, you must specify a string value as the first argument when you call the subroutine. Note that the input-parameter value may be supplied as a constant, expression, or a variable; it's the value that's important .

The second parameter `ans$` is not used until it has been given a value by the **INPUT** statement within the subroutine. Its value at invocation is inconsequential; however, its value upon completion of the subroutine contains the user's answer. Thus, `ans$` is an ***output parameter***. You must specify a variable for its argument, so the subroutine can change its value. When the value of the output parameter is changed the value of its argument variable also changes. The terms input parameter and output parameter clarify how the parameter is used — whether it requires a value or a variable as an argument. Note that a single parameter may act as both an input and an output parameter, in which case the input value must be supplied as a variable and the value of that variable will be changed by the subroutine.

[!] Note: When a variable argument is passed to a subroutine, the corresponding parameter is treated as an equivalent name for the same variable. Because they are the same variable, changes to a parameter within the subroutine definition will have an immediate and matching effect on the value of its corresponding argument — if that argument is a variable. Thus, subroutines, unlike functions, can easily change the values of the arguments passed to them. This mechanism of passing parameters is called ***passing by reference***.

In the example above, the subroutine is invoked with the statement:

```
  CALL Yes_no("Shall I go on", a$)
```

where the string constant `"Shall I go on"` is the first argument and the string variable `a$` is the second argument. Before it executes the code in the **SUB** structure, True BASIC assigns the value of the string constant to the parameter `qu$` and associates the parameter `ans$` with the variable `a$`. The subroutine prints a prompt using the parameter `qu$`, and assigns the lowercase equivalent of the user's response to the parameter `ans$`. Once it reaches the **END SUB** statement, the program continues with the statement immediately following the **CALL** statement. Since the parameter `ans$` and the argument `a$` are essentially different names for the same variable, the lowercase value stored into `ans$` within the subroutine is also the new value of `a$`.

While a subroutine can modify any variable that is used as an argument in the **CALL** statement, it cannot modify expressions used as arguments. Thus, arguments such as `sum, name$, score(7)`, and `titles$()` may be changed, but variables within expressions such as `x+3, a$ & b$`, and `c$[2:4]` cannot be changed. Remember that while `x` is a variable, `(x)` is an expression. Therefore, if you want to make sure that the value of an input variable is not changed, enclose it in parentheses. This trick allows you to pass parameters to a subroutine by value rather than by reference.

You may exit from the middle of a subroutine with an **EXIT SUB**. In the `Yes_no` subroutine, you could use **EXIT SUB** in place of **EXIT DO**.

Arrays as Parameters

As with functions, you may use arrays as parameters in subroutines. Here's a subroutine that uses a numeric array and numeric variable to accomplish the same thing as the earlier function example that finds the largest element in an array:

```
SUB Largest (a(), value)           ! Largest value in a list
    LET b1 = Lbound(a)             ! Find the bounds
    LET b2 = Ubound(a)
    LET value = a(b1)              ! Assume first is largest
    FOR i = b1+1 to b2             ! But compare to all others
        IF a(i) > value then LET value = a(i)
    NEXT i
END SUB
```

The same rules apply to specifying array parameters for subroutines as for functions (although the passing mechanism is distinctly different). Each array parameter must be followed by empty parentheses, or bowlegs. For instance, `a()` defines `a` as a one-dimensional array, `a(,)` defines it as a two-dimensional array, and `a(,,)` as a three-dimensional array. In the **CALL** statement, the parentheses are optional. Thus, either of the following is legal:

```
CALL Largest (prices(), v)

CALL Largest (prices, v)
```

as long as `prices` has been previously declared as a one-dimensional array in a **DIM** statement. It is the argument `prices` that requires a **DIM** statement, not the parameter `a`.

It is more efficient to pass arrays to subroutines than to functions. Arrays are passed to functions by value; the argument array must be copied to the parameter array, which takes time and storage space. With subroutines, however, arrays are passed by reference. Argument and parameter array names are associated to refer to the same array, hence the array is not duplicated. As with scalar variables, subroutines can change the values within argument arrays.

Channel Numbers as Parameters

Subroutines may also use channel numbers as parameters (functions may not). For example:

```
SUB OpenFile (qu$, #1)                ! Open a file
    PRINT qu$;                        ! Prompt user
    INPUT f$                          ! Name of file
    CLOSE #1                          ! In case #1 open
    OPEN #1: name f$
END SUB

CALL OpenFile("Data file", #3)        ! Invoke it
```

The file opened as channel #1 within the subroutine is associated with channel #3 in the calling program. Channel numbers and their uses are described fully in Chapter 12 "Files for Data Input and Output."

Variables within Subroutine Definitions

As with functions, you must be careful with any non-parameter variables you use within subroutines that are "internal" to (i.e., defined within) the main program. Look again at the `Largest` subroutine defined earlier in this section:

```
SUB Largest (a(), value)            ! Largest value in a list
    LET b1 = Lbound(a)              ! Find the bounds
    LET b2 = Ubound(a)
    LET value = a(b1)               ! Assume first is largest
    FOR i = b1+1 to b2              ! But compare to all others
        IF a(i) > value then LET value = a(i)
    NEXT i
END SUB
```

Notice that, in addition to finding the largest `value` in the array, the definition of `Largest` changes the values of `b1`, `b2`, and `i`. If the subroutine is part of the program that calls it, those variables are “global” — they are the same as any variables with the same name in the main program. For instance, if you invoke the subroutine inside a loop that starts with the statement:

```
FOR i = 1 to 10
```

chaos will result. Also, if you had previously defined a variable named `b1` or `b2`, its value would be changed whenever this subroutine is invoked.

As with functions, you can avoid this problem in two ways. First, you could list `b1`, `b2`, and `i` in a **LOCAL** statement within the **SUB** structure, forcing True BASIC to create separate variables named `b1`, `b2`, and `i` for each invocation of the subroutine. Or, you could define `Largest` as an external subroutine where variables are effectively “sealed off” from the rest of the program. The **LOCAL** statement and external procedures are described later in this chapter.

Functions vs. Subroutines

While functions and subroutines play similar roles, there are some fundamental differences.

A function computes a value and that value is used in an expression. Arguments are passed to functions by value — the argument’s value is copied to the parameter and the function cannot change the value of the argument itself.

Subroutines carry out tasks and they can change the value of arguments passed to them. Arguments are passed to subroutines by reference; thus an argument variable is essentially the same variable as its corresponding parameter variable. For instance, in the example above showing an array parameter for a subroutine, `Largest` sends the answer back via the argument that corresponds to the parameter `value`.

The `Gcd` function (defined earlier in this chapter) illustrates the fact that a function does not change the value of its arguments. Suppose that it is invoked as follows:

```
DEF Gcd(a,b)                    ! Greatest common divisor
    DO
       LET r = Mod(a,b)         ! Remainder in division
       IF r = 0 then EXIT DO    ! We are done
       LET a = b                ! Else iterate
       LET b = r
    LOOP
    LET Gcd = b                 ! Value of the function
END DEF
LET a = 121
LET b = 55
PRINT Gcd(a,b); "is the greatest common divisor of"; a; "and"; b
END
```

It prints the following:

```
 11 is the greatest common divisor of 121 and 55
```

The arguments `a` and `b` still have their original values, even though the definition of `Gcd` manipulates them as parameters. True BASIC assures this by copying the values into temporary variables.

Generally, if you want to compute a single value, a function is the best choice. This also ensures that the values of the arguments of the function do not get changed because they are passed by value.

Using array parameters for functions may be wasteful, however, since copying a large array takes a lot of time and space. Although the function `Max_element` and subroutine `Largest` accomplish the same thing, the subroutine `Largest` is probably the better choice. In the subroutine — which passes arguments by reference — the array is not copied. Instead, the argument and parameter variables are associated so that they refer to the same original array.

Subroutines provide greater flexibility than functions. Subroutines can carry out a number of tasks and change the values of any number of variables passed to them. You may change any function into a subroutine simply by adding its value as an extra parameter. But the reverse is not true, as the following routine shows:

```
SUB NextItem (old$, delim$, new$)          ! Find next item
    LET p = Cpos(old$, delim$)             ! Next delimiter
    IF p=0 then LET p = Len(old$) + 1
    LET new$ = old$[1:p-1]                 ! Next item
    LET old$ = Ltrim$(old$[p+1:1000])      ! Remove from old$
END SUB
```

This subroutine identifies items that are separated by delimiters. If `delim$` is given the value `" ,;"` then items may be separated by spaces, commas, or semicolons. This routine not only finds `new$` but also modifies `old$`. A function could not do this, since functions return only a single value and cannot change the values of their parameters.

As stated earlier, you may exit from the middle of either a function or a subroutine using **EXIT DEF** or **EXIT SUB**. However, if you use **EXIT DEF**, take care that you assign a value to the function before the exit, or the function will return the default value (0 or the null string).

Internal and External Procedures: Global and Local Variables

True BASIC programs may consist of one or more ***program units***, the most important of which is the main program. The ***main program*** includes the entire program up to and including the **END** statement. Every True BASIC program must contain a main program. Thus, when a program consists of only one program unit, that program unit must be the main program (which is to say that it must end with the **END** statement).

A True BASIC program may use additional program units in the form of external procedures or modules. This section discusses external procedures; modules are discussed in Chapter 11 "Libraries and Modules."

A ***procedure*** is a defined function, subroutine, or picture (pictures are discussed in Chapter 13 "Graphics"). An ***external procedure*** is not part of another program unit. You may define external procedures in the same file as the main program but after the **END** statement, or in separate files called libraries (see the next chapter). Each external procedure is a separate program unit.

Procedures that are defined within another program unit (modules excepted) are ***internal procedures***. You may define internal procedures within the main program before the **END** statement or within external procedures. Each internal procedure is considered part of the program unit in which it is defined. Thus, internal procedures are not separate program units.

You define internal and external procedures in exactly the same way, you invoke them in the same manner, and you use arguments and parameters in the same way to communicate with them. It is their position in relation to other program units that determines whether they are internal or external. There is one important distinction between them, however: variables that are not used as parameters are treated very differently in external and internal procedures. Also, external functions must be named in **DECLARE DEF** statements within the calling program unit (see the next section on "Internal and External Functions").

Internal procedures share all variables that are not listed as parameters with the program unit in which they are defined. And since internal procedures may be invoked only within the same program unit in which they are defined, we say that non-parameter variables in internal procedures are ***global variables***, because they are shared throughout the scope of that program unit.

Consider, for example, the following program, in which `AddressCode` is an internal subroutine defined before the **END** statement of the main program:

```
! Create address codes for mailing labels
DIM fnames$(3), lnames$(3)
MAT READ fnames$, lnames$
DATA Frank, Peter, James
DATA Hardy, Wimsey, Qwilleran

FOR i = 1 to 3
    LET first$ = fnames$(i)
    LET last$ = lnames$(i)
    CALL AddressCode(fnames$(i),lnames$(i),code$)
    PRINT "The address code for "; first$; " "; last$; " is "; code$
NEXT i

SUB AddressCode(f$,l$,c$)
    LET last$ = Ucase$(l$[1:4])
    LET first$ = Ucase$(f$[1:3])
    LET c$ = last$ & first$
END SUB

END
```

When this program is run, it gives the following results — which are probably not what the programmer intended!

```
The address code for FRA HARD is HARDFRA
The address code for PET WIMS is WIMSPET
The address code for JAM QWIL is QWILJAM
```

What happened? The first time through the **FOR** loop, the program assigns `"Frank"` to `first$` and `"Hardy"` to `last$`. It then calls the `AddressCode` subroutine, passing the first elements in the two arrays to `f$` and `l$`, and associating the argument `code$` with `c$` to receive the address code. When the subroutine ends, the **PRINT** statement uses the values of `first$`, `last$`, and `code$`. But why doesn't it print `"Frank"` and `"Hardy"` for `first$` and `last$`?

The problem is that the subroutine uses the same variable names `first$` and `last$` and it changes the values of those variables. Because the subroutine is internal to the main program those variables are shared throughout the program unit. Hence, the **PRINT** statement, which occurs after the call to the subroutine, uses the changed values for `first$` and `last$`.

Here's the program reorganized so that `AddressCode` is an external subroutine, stored after the **END** statement of the main program:

```
! Create address codes for mailing labels
DIM fnames$(3), lnames$(3)
MAT READ fnames$, lnames$
DATA Frank, Peter, James
DATA Hardy, Wimsey, Qwilleran

FOR i = 1 to 3
    LET first$ = fnames$(i)
    LET last$ = lnames$(i)
    CALL AddressCode(fnames$(i),lnames$(i),code$)
    PRINT "The address code for "; first$; " "; last$; " is "; code$
NEXT i

END

SUB AddressCode(f$,l$,c$)
    LET last$ = Ucase$(l$[1:4])
    LET first$ = Ucase$(f$[1:3])
    LET c$ = last$ & first$
END SUB
```

Now the program gives the intended output as follows:

```
The address code for Frank Hardy is HARDFRA
The address code for Peter Wimsey is WIMSPET
The address code for James Qwilleran is QWILJAM
```

Even though the subroutine uses the same variable names, any changes to those variables do not affect the variables in the main program. Hence, `first$` and `last$` retain the values they had before the **CALL** statement.

This works because external procedures do not treat variables other than parameters in the same way as do internal procedures. All non-parameter variables in external procedures are distinct from those in the program unit that invokes the procedure — even if they have the same name! An external procedure's variables are created when the procedure is invoked and destroyed when it is terminated. We say that non-parameter variables in external procedures are ***local variables***, because they are available only within the procedure that uses them — they cannot affect variables in the calling program unit

The next two sections give additional examples of internal and external functions and subroutines and discuss some important practical issues related to using them.

Internal and External Functions

Just as you must define all internal functions before you can use them, you must name all external functions before you can use them. You don't need to include parameters in the **DECLARE DEF** statement; you list only the names of the functions. You must always define or declare a function before you can use it, so that True BASIC will know that its name refers to a function and not a variable or array. This is illustrated in the third program example below.

As noted in the previous section, you must be aware of what variables you use when you write internal procedures, regardless of whether the procedure is a function or a subroutine. As another less obvious example, consider the following internal function that reverses the order of characters in a string:

```
DEF Reverse$(s$)                        ! Reverse of string
    LET x$ = ""                         ! Empty string to start
    FOR i = 1 to Len(s$)                ! A character at a time
        LET x$ = s$[i:i] & x$           ! Add in reverse order
    NEXT i
    LET Reverse$ = x$                   ! Value of function
END DEF

PRINT Reverse$ ("Hello there!")
GET KEY k                  ! Hold output until a key is pressed
END
```

The `Reverse$` function in this program correctly returns the following to the **PRINT** statement:

```
!ereht olleH
```

But look at what happens if you use the `Reverse$` function as follows to reverse four strings supplied by the user:

```
DEF Reverse$(s$)                        ! Reverse of string
    LET x$ = ""                         ! Empty string to start
    FOR i = 1 to Len(s$)                ! A character at a time
        LET x$ = s$[i:i] & x$           ! Add in reverse order
    NEXT i
    LET Reverse$ = x$                   ! Value of function
END DEF

FOR i = 1 to 4
   LINE INPUT PROMPT "Enter a string: ": string$
   PRINT Reverse$(string$)
NEXT i
PRINT "All done!"
END
```

Here's a sample run:

```
Enter a string: Hello.
.olleH
All done!
```

What happened? The program is supposed to ask for four strings. The problem is that the `Reverse$` function uses two variables that are not parameters: `x$` and `i`. This is a dangerous situation, since either variable might be used by some other part of the program — and indeed the variable `i` is used by the main program as well.

When the program begins, True BASIC creates the function definition (but doesn't execute it) and then begins with the **FOR** loop that uses the function to reverse four strings. This works fine the first time through the loop (where `i` equals 1). The program gets a string from the user and invokes the `Reverse$` function to print that string backwards. The function works properly but notice what it does to `i`. When `Reverse$` finishes it has changed the value of `i` to the length of the string argument plus 1 (or in this example 7). (It has also changed the value of `x$` but `x$` isn't used anywhere else in the program.) Thus, when True BASIC executes the **NEXT**statement after the **PRINT** statement, it increments `i` to 8 so the main **FOR** loop ends. The program prints the final statement and stops.

Now let's look at this same program rewritten with `Reverse$` as an external procedure:

```
DECLARE DEF Reverse$

FOR i = 1 to 4
   LINE INPUT PROMPT "Enter a string: ": string$
   PRINT Reverse$(string$)
NEXT i
PRINT "All done!"
END

DEF Reverse$(s$)                       ! Reverse of string
    LET x$ = ""                        ! Empty string to start
    FOR i = 1 to Len(s$)               ! A character at a time
        LET x$ = s$[i:i] & x$          ! Add in reverse order
    NEXT i
    LET Reverse$ = x$                  ! Value of function
END DEF
```

We did just two things to make this change. We moved the function definition to follow the **END** statement of the main program (we could have moved it to a separate file). And we added the **DECLARE DEF** statement at the beginning of the main program to tell True BASIC that `Reverse$` is a function that is defined elsewhere.

When you run the revised version of the program, it works as expected:

```
Enter a string: Hello.
.olleH
Enter a string: Able was I
I saw elbA
Enter a string: ere
ere
Enter a string: I saw Elba!
!ablE was I
All done!
```

When `Reverse$` is an external function, all variables used in the definition are local to that definition only. The function's variables are totally separate from those in the rest of the program, even if they happen to have the same names. The only information shared with the main program is the string argument passed to the function's parameter and the string value returned by the function.

Internal and External Subroutines

As you've seen, all variables in an internal subroutine are shared with the rest of the program. Variables in an external subroutine are local to that subroutine; the only shared values are those passed between arguments and parameters.

To create an external subroutine, you simply place it after the **END** statement or in a separate file. There is no equivalent to the **DECLARE DEF** statement. Because you must invoke a subroutine with a **CALL** statement, it is always clear to True BASIC that you are referring to a subroutine and not a variable or array.

Although you can use global variables to share information between your program and internal subroutines, we recommend that you use parameters to clearly indicate information passed to and from your subroutines. For example, the following program uses an internal subroutine to get an answer from the user, check it for correctness, and respond appropriately. Although it uses no parameters, it works because all variables are shared with the rest of the program:

```
!State quiz

DIM questions$(10), answers$(10)
LET count, correct = 0
DO WHILE MORE DATA
   LET count = count + 1
   READ questions$(count), answers$(count)
LOOP
FOR i = 1 to count
    DO
       PRINT questions$(i)
       CALL Get_answer
    LOOP UNTIL correct = 1
NEXT i
SUB Get_answer
    INPUT ans$
    IF Lcase$(ans$) = answers$(i) then
       PRINT "Correct!"
       LET correct = 1
    ELSE
       PRINT "Wrong, try again..."
       LET correct = 0
    END IF
END SUB
DATA "What is the 50th state", hawaii
DATA "What is the largest state", alaska
DATA "What state has the nickname 'Old Dominion'", virginia

END
```

There are two potential problems with this program, however. The subroutine would not work if you later decide to make it an external procedure that can be shared with other programs. Also, if the above statements were part of a lengthier program, it might not be clear where the program sets the value for `correct` — used as the exit condition for the DO loop.

You are generally better off using parameters for any values you need to share between the program and the subroutine. For example, you could rewrite `Get_answer` to use two parameters — a string variable and a numeric variable. Notice that the string argument supplied in the **CALL** statement is an element of the string array `answers$`; a single element in an array is treated as a single variable.

Consider the following variation:

```
...
FOR i = 1 to count
    DO
       PRINT questions$(i)
       CALL Get_answer(answers$(i), correct)
    LOOP UNTIL correct = 1
NEXT i

SUB Get_answer(a$,c)
    INPUT ans$
    IF Lcase$(ans$) = a$ then
       PRINT "Correct!"
       LET c = 1
    ELSE
       PRINT "Wrong, try again..."
       LET c = 0
    END IF
END SUB
...
```

This version of the subroutine will now work equally well as an external subroutine. Also, it is clear from reading the program code which values are shared with the subroutine. The first argument supplies the correct answer to the subroutine. After it carries out its tasks, the subroutine passes back an appropriate value to the argument `correct`.

Advantages and Disadvantages of External Procedures

When you use external procedures, you don't have to worry about duplicating variable names. This is particularly important if you wrote a routine a long time ago and have forgotten the names used within it, or if someone else wrote it. To use an external procedure you need to know: (1) its name, (2) what kind of arguments it takes, and (3) what it does. You do not need to know how it is programmed or what variables are used.

You may store external procedures in separate "library files" (see the next chapter). You can therefore use the same procedure from any number of programs by referring to the library file. Thus, you need not copy or rewrite commonly used code into each of your programs.

The price for this protection is that the values of local variables are lost when an external function or subroutine terminates and all local channels are closed. As a consequence, such external procedures cannot 'remember' what they did the last time they were called. The next chapter describes "modules" which give you much greater control over the scope of variables, letting you create procedures that can remember the values of local variables and define items available to more than one program unit.

The LOCAL Statement

The **LOCAL** statement serves two purposes. You may use it to indicate variables or arrays that are to be local to an internal procedure and not available outside that procedure. You may also use it in combination with an **OPTION TYPO** statement to catch misspelled variable names.

The **LOCAL** statement is helpful in internal procedures when you want to keep variables separate from the rest of the program. For example, in the internal subroutine:

```
SUB Xyz
    LOCAL i

    FOR i = 1 to 10
        LET sum = sum + i
    NEXT i
END SUB
```

the variable `i` is local, and hence considered different from `i` in the rest of the main program. In effect, `i` is treated as if `Xyz` were an external routine, yet `sum` is available to the main program. Clearly this option is useful for assuring that a temporary variable like `i` does not conflict with one in the rest of the program.

You may also list arrays — with their dimensions — in a **LOCAL** statement. Because you must give the array dimensions, the **LOCAL** statement takes the place of a **DIM** statement for those arrays.

An **OPTION TYPO** statement tells True BASIC to alert you if it finds an incorrect variable name. You must declare all variables that first appear in your program or program unit after the **OPTION TYPO** statement. True BASIC uses the declared names as a "dictionary" of acceptable names; it also accepts any variable names used before the **OPTION TYPO**. If True BASIC encounters any other variables, such as a misspelled name, an error occurs. For example, if you attempt to run a program that begins as follows:

```
OPTION TYPO
DIM grades(10)                  !  or LOCAL grades (10)
LOCAL name$, sum, average

INPUT PROMPT "Student's name: " : names$
PRINT "Enter grades separated by commas."
MAT INPUT grades
...
```

True BASIC will halt at the misspelled `names$` variable on the **INPUT PROMPT** statement and print the message "Unknown variable."

For this to work best, you should place the **OPTION TYPO** statement early in your program unit and then declare all your variables. You declare variables with **LOCAL** statements, arrays with **LOCAL** or **DIM** statements, and parameters by placing them in the **SUB**, **DEF**, **FUNCTION**, or **PICTURE** statements defining procedures (for information on picture procedures see Chapter 13 "Graphics"). You may also declare variables using the **PUBLIC**, **SHARE**, and **DECLARE PUBLIC** statements described in the next chapter.

You may use the **OPTION TYPO** statement in any program unit or module header (see the next chapter). Once used, it applies to all the variables that first appear after the **OPTION TYPO** statement, and it remains in effect until the end of the file that contains it. Note that **OPTION TYPO** is technically not an executable statement — it is used only when True BASIC compiles the program. For this reason, its effects are related to its position in the file, not to its position in the flow of control. Thus, when the **OPTION TYPO** statement appears in an external procedure stored after the **END** statement of the main program, it applies only to the remainder of that procedure and any other procedures that are stored after it in the file. It does not apply to any portion of the file that precedes it — including those portions of the main program that follow the invocation of the procedure containing the **OPTION TYPO** statement.

CHAPTER

11

Libraries and Modules

The previous chapter shows how procedures break a program into separate tasks, and evaluates the differences between functions and subroutines and between internal and external procedures. In addition, the previous chapter explains the distinctions between local and global variables. To fully understand the concepts presented in this chapter, you should be familiar with all the concepts presented in the previous chapter.

This chapter explains how to further organize your programs using special files called libraries and also how to gain very specific control over the scope of variables and/or procedures using specialized structures called modules. Modules allow you to collect several external procedures into a single program unit and gain specific control over the scope of each and every variable, array, channel number, and/or procedure that they contain. Libraries let you collect several external procedures and/or modules together in a single file that you can then access from any number of programs and/or other libraries.

Libraries

You may place external procedures and modules after the **END** statement of your main program, but you'll usually find it more convenient to store them in one or more separate files or libraries. A ***library*** is a file that contains any number of external procedures (functions, subroutines, or pictures) or modules, but no main program.

Any of your programs can use the procedures and modules in a library without having to duplicate any code. And you can compile libraries separately from the main program. Even your uncompiled programs can use compiled versions of libraries, thus reducing the time it takes your program to begin running by decreasing the amount of code that must be compiled. (See "The True BASIC Environment" chapter in the Introduction for information on compiling programs and libraries.)

Each library file must begin with an **EXTERNAL** statement as its first non-comment line. The **EXTERNAL** statement tells True BASIC not to look for a main program in the current file, allowing the library to be compiled separately from the main program file. Because no main programs are allowed, a library may not contain an **END** statement.

Here's an example of a simple library that contains a function and a subroutine that might be useful for various games of chance:

```
! Making random choices

EXTERNAL

SUB RollDice (sum_dice, num_dice)       ! Roll any number of 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$                               ! Toss a coin
   IF Rnd < .5 then
      LET Coin$ = "heads"
```

```
     ELSE
        LET Coin$ = "tails"
     END IF
  END DEF
```

You can use a library simply by naming the file that contains it in a **LIBRARY** statement. For instance, if a compiled version of the above library is saved in a file named CHANCLIB.TRC (in the currently active directory or folder), you could use any of its procedures by naming it in a **LIBRARY** statement, as in the following program:

```
  ! A game of luck and skill

  LIBRARY "ChancLib.TRC"                     ! Access the library file
  DECLARE DEF Coin$                          ! Declare a library function

  RANDOMIZE
  INPUT PROMPT "Choose heads or tails to see who goes first: " : c$
  IF c$ = Coin$ then                         ! Toss a coin
     LET turn = 1                            ! User goes first
  ELSE
     LET turn = -1                           ! Computer goes first
  END IF

  CALL RollDice (dice,4)                     ! Roll 4 dice
  ...
```

Notice that functions defined in a library are like all other external functions — you must declare them in a **DECLARE DEF** statement before you can use them. The **LIBRARY** statement merely tells True BASIC where to look for any external procedures not defined within the main program file. You still need a **DECLARE DEF** statement to inform True BASIC that `Coin$` is a defined function rather than a variable.

The **LIBRARY** statement tells True BASIC where to look for external procedures by naming the file that contains them. You must give the complete file name, including any extensions, and you must provide enough information to allow True BASIC to locate the file. If you specify a simple file name, True BASIC assumes that the library file is in the current directory (or folder). If the library file is in another directory, then you must provide a valid path to the directory containing the file as part of the file name. That path name may start from the current directory or from the root level of your current disk or a specified disk. For a summary of the rules for defining path names and file names under various operating systems, see the introductory chapter on "The True BASIC Environment."

[!] **Note:** The rules for path names and file names vary between different operating systems. In addition, the paths to specific files can change when you move between computers running the same operating system, or even at different times on the same computer. Therefore, you may need to update a program's **LIBRARY** statements when the program's environment changes.

You may list several library files in a single **LIBRARY** statement:

```
  LIBRARY "ChancLib.TRC", "GamesLib.TRC", "Finance.TRC"
```

If the libraries happen to contain two or more routines with the same name, True BASIC will use the first routine it finds, examining libraries in the order given in the **LIBRARY** statement.

You may name a library file more than once, and you may place **LIBRARY** statements in external procedures. In fact, it is sometimes best to include a **LIBRARY** statement in every procedure that calls another external procedure (even if you are likely to have named the needed library elsewhere). The reason lies in the way True BASIC searches libraries for procedures: it searches libraries in the order named for procedures it "knows" are needed; if one of those procedures then calls another, True BASIC does not "back track" to search earlier libraries. Thus you should name libraries containing higher-level procedures first, or put an appropriate **LIBRARY** statement in the higher procedure itself. Keep in mind, however, that unless all libraries are always kept in the same directory, you must update **LIBRARY** statements if the directory structure containing the files changes. For complex, portable programs, it may be safest to create bound versions. (Bound programs are compiled with all the necessary libraries and run-time resources; see "The True BASIC Environment" in the Introduction for information on binding programs.)

The **OPTION TYPO** statement may help prevent typographic errors within libraries as well as the main program. If you wish to use **OPTION TYPO** for external procedures in library files, you should include the statement at the beginning of the first procedure to which you wish it to apply. Since it is non-executable and its position in the file determines its effect, the **OPTION TYPO** statement will then apply to the procedure that contains it and all those that follow it in the library, forcing variables and arrays used within those procedures to be pre-declared in a **LOCAL**, **DIM**, **SHARE**, **PUBLIC**, or **DECLARE** statement (see the following section on modules).

Modules

A ***module*** is a structure that lets you combine several external procedures into a single program unit and specifically control the availability of those procedures and their variables.

We'll look first at the special properties that make modules particularly useful and then examine how the **MODULE** structure defines modules. The next section gives several examples of modules to help you further understand their structure and usage.

A module can "own" information in the form of shared variables, arrays, and channel numbers. Such ***shared*** items are available throughout the module — they need not be explicitly passed as parameters. Since shared items are "static," they retain their values between invocations of the procedure. Thus, if a procedure sets the value of a shared item, that item will have that value the next time the procedure is invoked (provided, of course, that another procedure in the module hasn't changed the item's value in the interim). Shared items are not available to the main program, other modules, or external procedures outside the module — they are "hidden" from program units outside the module.

In addition to the shared items available to procedures within the module, a module may make some items ***public*** so that any other program unit, such as the main program or other modules, may use them. Within the module that defines them, public items have the same properties as shared items, but they are also available to any other program unit that declares them.

Usually, the external procedures within a module are publicly available to any program unit outside the module. However, modules also let you designate some procedures as ***private*** so that their use is limited to other procedures in the module. Such procedures are "hidden" from other program units.

One final special property of modules is the initialization segment. An ***initialization segment*** is a series of statements preceding the first procedure definition in a module. Each of these statements will be executed in sequence during the initialization of that module, which takes place before the main program is executed.

A module is defined by a **MODULE** structure with the following format:

```
MODULE ModName
       (module header statements)
       (module initialization statements)
       (procedures of the module)
END MODULE
```

where `ModName` is the name given to the module. This name is for your convenience and is used by True BASIC only for error messages.

Module Header Statements

Each module starts with a ***module header***, which is a series of statements that define the nature of various variables, arrays, channel numbers, and procedures used within the module. The module header may consist of **SHARE** statements, **PUBLIC** statements, **PRIVATE** statements, and any of the **DECLARE** or **OPTION** statements.

SHARE statements define the variables, arrays, and channel numbers that will be available to all procedures within the module but not to program units outside the module. Shared items are static and therefore retain their values until the program ends. Likewise, shared channel numbers remain open (once they have been opened) until they are specifically closed or the program ends.

PUBLIC statements define the variables and arrays that will be available to program units outside the module. Of course, public items are also available to all the procedures within the module. Like shared items, public items are static and retain their values until the program ends.

Notice that channel numbers may only be shared, they may not be made public. Also note that arrays that appear in a **PUBLIC** or **SHARE** statement must be dimensioned by that statement — a **DIM** statement is neither required nor allowed for an array defined in a **PUBLIC** or **SHARE** statement.

PRIVATE statements specify certain procedures of the module that may not be accessed from outside the module. You could therefore use **PRIVATE** statements to restrict access to "dangerous" procedures that could destroy the module's data structures. **PRIVATE** statements also let you use the name of the routine for other purposes outside the module. Thus, if you have a private subroutine called `Sum` in the module, you may use `sum` as a variable or procedure name in the main program. If you declare a function in a **PRIVATE** statement, you don't need a **DECLARE DEF** statement in the same module header.

A **DECLARE DEF** statement in a module header makes the functions it names available to all procedures in the module. However, you don't need a **DECLARE DEF** statement in the module header for functions that will be used only from other program units, as the other program unit must have a **DECLARE DEF** statement for that function.

A **DECLARE PUBLIC** statement in a module header will make public items defined elsewhere in the program available throughout the current module. When a **DECLARE PUBLIC** statement lists a public array, the array name must be followed by appropriate bowlegs (with commas for arrays or two or more dimensions) to inform True BASIC how many dimensions that array contains. Public items defined in **PUBLIC** statements need not be listed in **DECLARE PUBLIC** statements within the same module header.

[!] Note: Any program unit that uses public items defined as public elsewhere in the program must have a corresponding **DECLARE PUBLIC** statement listing those items. While **PUBLIC** and **DECLARE PUBLIC** statements may appear in any program unit, a single item may appear in only one **PUBLIC** statement per program. However, that single item may be specified by any number of **DECLARE PUBLIC** statements throughout the program.

While other **DECLARE** statements (see Chapter 18 "Statements and Built-in Functions and Subroutines") may also appear in a module header, there is little point since they have no effect.

Module headers may also contain the various **OPTION** statements, such as **OPTION TYPO**. However, since none of the **OPTION** statements is executable, they all affect the module in terms of their position within the file itself. That is, an **OPTION** statement in a module header will apply to all lines remaining in the file (both in that module and any other modules or procedures that follow it), regardless of the order in which procedures are actually executed.

Here is the beginning of a module that illustrates most of these header statements:

```
MODULE MyMod
   PUBLIC princ, int_rate, values(2,100)    ! Global variables & arrays
   SHARE mo_rate, mo_prin, mo_sum           ! Shared variables & arrays
   SHARE temp(1,1), f
   SHARE #1, #2                             ! Shared channels
   DECLARE DEF Daily                        ! Global & shared function
   PRIVATE Sum                              ! Private procedure
   DECLARE PUBLIC account_no, personal()    ! Public variable & array
                                            !   defined elsewhere
   ...
```

Module Initialization

After the module header and before the procedures of the module, you may include statements that "initialize" the module. This ***initialization segment*** may assign values to variables, open channels, or set certain conditions. In fact, the initialization segment may even invoke procedures in the module. True BASIC executes these initialization statements before it executes the main program.

Typically, you would assign initial values to public and shared variables. These variables would then have the assigned values when the main program begins. It does not make sense to initialize any other variables, arrays, or channels, as they are local to the initialization segment and their values are lost when initialization is completed.

[!] Note: When a module is stored in a library file, the statements in its initialization segment will be executed only if that module is accessed by the program; simply listing the library in a **LIBRARY** statement does not initialize the modules in that library. Modules are accessed when one of their procedures is invoked or when a public variable that they define is declared. When more than one module is defined within a single library, they are initialized in the order in which they appear in the file, but each module will be initialized only if it used in some way by the program. Modules located in the same file as the main program are initialized regardless of whether or not they are accessed.

As an example, here's the beginning of a module showing the module header and initialization segment.

```
MODULE Cards
  SHARE card(0 to 51), n              ! Module header
  SHARE val$(0 to 12), suit$(0 to 3)
  PRIVATE Decipher

  MAT READ val$, suit$                ! Initialization segment
  DATA two, three, four, five, six, seven, eight
  DATA nine, ten, jack, queen, king, ace
  DATA clubs, diamonds, hearts, spades

  RANDOMIZE
  CALL Shuffle
  ...
```

The next section shows the complete `Cards` module plus three other simple, but complete, modules and how they might be used.

Using Libraries and Modules

You may use any procedure defined in a module just as you would use any external procedure, provided that the module has not listed the procedure in a **PRIVATE** statement. If the module that contains the procedure is in a library file, then the program unit that uses it must have an appropriate **LIBRARY** statement. If the procedure is a user-defined function, you must also have a **DECLARE DEF** statement in the program unit that will use the procedure.

To access items defined as public in a module, a program unit must name those items in a **DECLARE PUBLIC** statement. Of course, if the module is in a library, the program unit must also have access to the library. Also, remember that public arrays listed in a **DECLARE PUBLIC** statement must include appropriate bowlegs (with commas for arrays or two or more dimensions) to inform True BASIC how many dimensions that array contains.

To gain a better understanding of modules and their use, consider the following examples.

Defining and Initializing Public Information

The following module exists simply to define some public items and initialize two of them:

```
MODULE Common
      PUBLIC first, sum, p(2,50)       ! Global variables
      LET first = 1                    ! Initialize
      LET sum = 0
END MODULE
```

This module defines the variables `first` and `sum` and the array `p` as public and initializes `first` and `sum`. Any program unit with access to this module may then include a statement:

```
DECLARE PUBLIC first, sum, p(,)
```

to have access to these public items. Once declared, these items will be available throughout the program unit that declares them.

To make a constant globally available, you can define and initialize it in your module:

```
MODULE Common
      PUBLIC first, sum, p(2,50)
      PUBLIC e                         ! Global "constant"
      LET e = Exp(1)                   ! Define the constant
      LET first = 1
      LET sum = 0
END MODULE
```

Then if a program unit names `e` in a **DECLARE PUBLIC** statement, `e` will have the assigned value. This has the disadvantage, though, that some program unit may change the value of `e`. To make sure that the value of `e` is not accidentally changed, you could make it a function by removing `e` from the **PUBLIC** statement and changing the **LET** statement in the module to:

```
DEF e = Exp(1)
```

This protects the value from unauthorized changes since `e` is no longer a variable. Note that the calling program must use a **DECLARE DEF** statement rather than a **DECLARE PUBLIC** statement to gain access to the defined function.

Defining "Data Structures"

You can define and manipulate rudimentary data structures using modules. The advantage to this is that you can make such data structures available to any program without the programmer having to be aware of how they have been implemented. The use of modules for this purpose can also eliminate the need to pass large numbers of arguments between procedures involved in the maintenance of the data structures.

As a simple example, consider the following module that maintains all the necessary information about a deck of cards during a card game. It shuffles, deals, and defines formatted names for the cards.

```
MODULE Cards
  SHARE card(0 to 51), n                ! Deck, number of cards left
  SHARE val$(0 to 12), suit$(0 to 3)    ! Values, suits
  PRIVATE Decipher

  MAT READ val$, suit$                  ! Initialize module
  DATA two, three, four, five, six, seven, eight
  DATA nine, ten, jack, queen, king, ace
  DATA clubs, diamonds, hearts, spades

  RANDOMIZE
  CALL Shuffle

  SUB Shuffle                           ! Set up deck
      FOR i = 0 to 51
          LET card(i) = i
      NEXT i
      LET n = 52
  END SUB
```

```
    SUB Deal(c)                                  ! Deal a card
        LET j = Int(n*Rnd)                       ! Pick random card
        LET c = card(j)
        LET n = n-1
        LET card(j) = card(n)                    ! Fill place
    END SUB

    DEF Name$(c)                                 ! Name of card
        CALL Decipher(c, v, s)                   ! Card, value, suit
        LET name$ = val$(v) & " of " & suit$(s)
    END DEF

    SUB Decipher(c, v, s)
        CALL Divide(c, 13, s, v)
    END SUB

END MODULE
```

Notice how this module uses the three shared arrays `cards`, `val$`, and `suit$` to represent the deck of cards. All these arrays are "owned" by the module — they are available to all the routines in the module, yet they cannot be accessed from outside the module. This means that the data structure may be accessed only through the procedures in the module.

Each of the arrays used to define the deck of cards is initialized in the initialization segment of the module, either through the **MAT READ** statement or the invocation of `Shuffle`, so that the card deck is ready to go even before the program begins executing.

A simple use of this module might be:

```
! Deal 5 cards
LIBRARY "CardLib.TRC"          ! Access library containing Cards module
DECLARE DEF Name$              ! Declare library function

FOR i = 1 to 5
    CALL Deal(c)               ! Use library subroutine
    PRINT Name$(c)             ! Use library function
NEXT i

END
```

Note that the person writing the calling program does not need to know how cards are coded or how `Shuffle` works. Note also that simple external procedures could not achieve this purpose, since the values of the arrays would disappear between their invocations.

Sharing Variables to Imitate Turtle Graphics

Consider another module that imitates the use of the turtle graphics popularized by the programming language Logo™. Turtle graphics work by allowing the user to move a small object, or "turtle," around the screen. As the turtle moves, it leaves a visible trail, drawing a picture on the screen. The turtle can turn left or right a certain number of degrees and can move forward or backward a specified distance. This type of graphics is sometimes called ***relative graphics***, which means that each pen position is measured relative to its previous position.

Normally, True BASIC uses ***absolute graphics***, in which each pen position is measured absolutely from a constant origin. The following simple module lets True BASIC imitate turtle graphics:

```
MODULE Turtle
    SHARE x, y, a                             ! Location, angle
                                              ! Same x, y, and a used
                                              ! throughout the routine
    OPTION ANGLE DEGREES                      ! Initialize
    CALL ClearScreen
```

```
      SUB ClearScreen
         CLEAR
         SET WINDOW -140, 140, -120, 120
         LET x, y = 0                    ! Start at origin
         LET a = 90                      ! Head upward
         PLOT 0, 0;                      ! Start drawing
      END SUB

      SUB Left(da)
         LET a = a + da
      END SUB

      SUB Right(da)
         LET a = a - da
      END SUB

      SUB Forward(d)
         LET x = x + d*Cos(a)
         LET y = y + d*Sin(a)
         PLOT x, y;
      END SUB

      SUB Back(d)
         CALL Forward(-d)
      END SUB

   END MODULE
```

Any of the five procedures may be called from outside or inside the module. In fact, `Back` calls `Forward`, and the initialization segment includes a call to `ClearScreen`. The initialization segment sets degrees as the measure for all angles, and, by invoking `ClearScreen`, it establishes the window coordinates and puts the turtle in the center of the window heading up the screen. (See Chapter 13 "Graphics" for explanations of the **CLEAR**, **WINDOW**, and **PLOT** statements.)

The **SHARE** statement lets the module "remember" the current position and direction of the turtle; the values of the shared variables `x`, `y`, and `a` are never lost when control returns to the calling program. These shared variables may be used by any routine in the module but they cannot be accessed or changed from outside the module. Indeed, the calling program is not even aware of their existence!

The following is a very simple program that uses the module to move the turtle. All it needs to do is name the file containing the module in a **LIBRARY** statement and then pass appropriate arguments to the `Left`, `Right`, `Forward`, and `Back` subroutines.

```
   LIBRARY "TURTLE.TRC"

   PRINT "To turn the turtle type 'R' for right or 'L' for left"
   PRINT "followed by an angle, and then press the return key."
   PRINT
   PRINT "To move the turtle in the current direction, type 'F' or"
   PRINT "'B' followed by a distance to move forward or backward,"
   PRINT "and then press the return key."
   PRINT
   PRINT "Press 'C' to clear the screen or 'S' to stop the program."
   PRINT "Press any key when you are ready to begin."

   GET KEY start
   CLEAR

   DO
      IF KEY INPUT then
         GET KEY how
         LET how$ = Ucase$(Chr$(Mod(how,256)))
```

```
        SELECT CASE how$
        CASE "L"
             INPUT PROMPT how$: angle
             CALL Left(angle)
        CASE "R"
             INPUT PROMPT how$: angle
             CALL Right(angle)
        CASE "F"
             INPUT PROMPT how$: distance
             CALL Forward(distance)
        CASE "B"
             INPUT PROMPT how$: distance
             CALL Back(distance)
        CASE "C"
             CALL ClearScreen
        CASE "S"
             STOP
        CASE ELSE
        END SELECT
     END IF
  LOOP
  END
```

Sharing Channel Numbers

Channel numbers may be shared and passed as parameters, but they cannot be public. Because channel numbers may be shared, modules are often used to isolate the interface to files, logical windows, or a printer. You will find much more information on channel numbers and their use in Chapter 12 “Files for Data Input and Output.”

The following example uses logical windows to illustrate shared channels. Logical windows are described in Chapter 13 “Graphics.” Briefly, however, you need to know that True BASIC defines the default logical window with coordinates 0 to 1 from left to right and 0 to 1 from bottom to top. Thus, the lower left corner of the default window is 0, 0 and the upper right is 1, 1. You can define smaller logical windows within a physical window by specifying the left, right, bottom, and top coordinate limits in an **OPEN** statement, as in the following module. The **WINDOW** statement tells True BASIC where to send subsequent output (see Chapter 13 “Graphics”).

```
  MODULE Timer

      SHARE start, #1, #2                ! Starting time, 2 windows
      OPEN #1: SCREEN 0, .8, 0, 1        ! Working window
      OPEN #2: SCREEN .85, 1, 0, 1       ! Time window
      LET start = Time                   ! Use built-in function
      CALL Clock

      SUB Clock
          LET t = Time
          WINDOW #2                      ! Print to time window
          CLEAR
          PRINT Round(t - start, 1)
          WINDOW #1                      ! Switch back to working window
      END SUB

  END MODULE
```

In this example, True BASIC would open the logical windows and print the initial time in window #2 as part of module initialization before it executes the main program. Output from the main program will be printed in window #1. The main program can update the running time whenever it wishes with the statement:

```
  CALL Clock
```

Note that the `Clock` subroutine in the module can switch logical windows, but the main program cannot itself switch to the timing window (#2) nor can it change either logical window (though it could open other logical windows of its own).

Notice also that the variable `start` is available only within the module. Thus, the timing mechanism is not ruined if the calling program also has a variable called `start`.

Using the Supplied Libraries

Some of the advanced capabilities of the True BASIC language are not really part of "the language" at all, but are included in the form of libraries. Since each built-in function and subroutine makes the language a little bit larger and a little bit slower, much of the advanced functionality is provided in the form of external libraries. The following libraries are included with the True BASIC language.

Libraries Included with True BASIC Language (in TBLIBS)

Library	Functions or Subroutines	Chapter Reference
Mathematical Tools		
MathLib.TRC	trig functions not already built-in	23
HexLib.TRC	bit, octal, and hexadecimal manipulation routines	23
String Tools		
StrLib.TRC	string creation, conversion, formatting, and editing	23
Sorting and Searching Tools		
SortLib.TRC	sorting, searching, and reversing items on arrays	9, 23
Graphics Tools		
BGLib.TRC	pie charts, bar charts, and histograms	23
SGLib.TRC	plotting data and function values	23
SGFunc.TRC	plotting values of functions that you define	23
GraphLib.TRC	simple graphing utilities	23
Interface Tools		
TrueCtrl.TRC	interface elements	14, 22
TrueDial.TRC	dialog boxes	14, 22
File and Directory Tools		
ExecLib.TRC	file and directory control	12, 22

In addition, other toolkits in the form of libraries, are available. See our web site for current listing and prices.

To access any procedure in such a library, simply treat it as you would a procedure in a library that you have created yourself. Name each library in a **LIBRARY** statement using an appropriate path name to precisely locate the library, and declare any functions in **DECLARE DEF** statements.

CHAPTER

12 Files for Data Input and Output

Programs commonly use files to save data for later retrieval or to share data with another program. This chapter describes the use of files to store information produced by programs and to retrieve information stored by other programs. It also describes the ExecLib library of subroutines that give your programs added control over files and directories. Since sending textual output to a printer uses many of the same statements and techniques, this chapter also discusses how to print textual output.

A ***file*** is a unit of information saved on a diskette, a hard disk, or some other "permanent" storage device. A file may contain data, or it may contain a program or a library. Because they continue to exist after your program stops and even after you turn off your computer, files provide long-term storage. A program may access a file to read information from it or write information to it.

Files store information in a variety of ways. Some can contain only the text characters that you commonly enter at the keyboard and display on the screen. Others can store information in more efficient formats that cannot be displayed directly on the screen. True BASIC can create and use ***text files*** and four forms of ***internal files — stream***, ***random***, ***record***, and ***byte***.

[!] Note: Different operating systems use different terminology for the organization of their storage media. Throughout this chapter, the term "directory" refers to the organizational component of a disk that may contain one or more files or other directories. Directories that are contained within another directory are "subdirectories" of that directory. Some operating systems refer to these as "folders" and "subfolders," but this chapter uses the terms directory and subdirectory throughout.

True BASIC programs may use files stored anywhere in any directory or disk accessible to the computer running the program (this includes files and disks that are accessible across a network).

When you need to access a file, you must first open it using a file name that is appropriate to your computer's operating system. That name may be a simple file name, indicating that the file is stored in your current directory, or it may be a complete path name that indicates the file's location. See "The True BASIC Environment" chapter in the introductory section of this manual for information on file and path names appropriate for the different computer operating systems.

The **OPEN** statement opens a file and assigns it a channel number. All other statements that operate on files use the channel number rather than the file name.

The first few sections in this chapter describe statements and operations that apply to all of True BASIC's file types. The later sections discuss each file type and additional statements particular to that file type. Because the internal files share many characteristics, they are described as a group and then individually. The final section describes ExecLib routines that let your programs get information about files and directories and create, rename, and remove directories.

A Summary of File Types

True BASIC uses five kinds of data files. The basic differences between the types lie in how information is stored within the file and how you may access that information.

Text files use ***display format*** **—** they mimic the display that appears on your screen or printer. In text files, both string and numeric values are represented as characters; numeric values are automatically converted to the string of digits used to display them. To get information to and from text files, you use a channel number with the same **PRINT** and **INPUT** statements that you use with the screen, printer, or keyboard.

The other four file types store information in the ***internal format*** that the computer uses to represent values in memory. Each string value is stored as a series of one-byte characters (similar to text files), but each numeric value is stored in the internal IEEE eight-byte format that preserves its full precision. Because of the way information is stored within them, you must use **WRITE** and **READ** statements, with a channel number, to get information to and from internal files.

The five file types may be summarized as follows:

A ***text*** file is one that you may create and save using any text editor capable of saving a text-only file. As such, text files may be displayed on the screen. Every program you create and save with the True BASIC Editor is a text file. Since a compiled program cannot be displayed, it is not a text file. True BASIC views the printer as a text file with the restriction that a program cannot read from it (for obvious reasons). You may, however, write to the printer; hence, you may use file operations to produce hard copy on your printer. Text files are sequential-access files, meaning that you must access each record (or item) in the order in which it appears in the file.

A ***stream*** file is stored in internal format and cannot be displayed on the screen. Like a text file, a stream file is organized sequentially. That is, the elements must be read in exactly the same order in which they were written.

A ***random*** file is stored in internal format and cannot be displayed on the screen. The file is organized into records of fixed length, and you can jump to any record in the file and read it or change it. (Text or stream files do not permit this "random access.") The records may contain any number of string and numeric values, in their internal format, as long as their cumulative length does not exceed the record's maximum length.

A ***record*** file is like a random file except that each record may contain only one numeric or string value.

A ***byte*** file is the most general type of file. You may access any file as a byte file, or simply a sequence of bytes. Approaching a file on the byte level lets you manipulate files created by any application, such as a word processor or spreadsheet, provided you know how that application has represented the information within the file. Because byte files let you operate on each byte individually, they also provide the flexibility required for effectively ***packing*** information to conserve storage space.

Basic File Operations

This section and the next discuss operations and concepts common to all five data-file types. Each file type is then described more fully along with statements and operations particular to that file type.

Opening Files

Before you may use the information stored within a file, you must first open a channel to that file. This process uses an **OPEN** statement as follows:

```
   OPEN #3: NAME "USEFUL"
```

The **OPEN** statement obtains access to the named file (through the current operating system) and associates that file with the specified channel number. ***Channel numbers*** always consist of a pound sign (#) followed by an integer between 0 and 999 (or a numeric expression that evaluates to such a value). Note, however, that channel #0 is reserved for the default logical window, which is always open. (The default logical window is the default output window automatically opened by True BASIC; see Chapter 13 "Graphics" for more information.) After you've opened a file, you must always refer to it by its associated channel number.

The file name in the **OPEN** statement may be a simple name referring to a file in the current directory, or it may include a path name specifically indicating the location of the file. Legal file and path names vary between operating systems, so be sure to check "The True BASIC Environment" chapter in the introductory section for the rules of any operating systems you or your program's users will be using.

You may specify the file name as a string constant, a string variable, or a string expression as long as it evaluates to a legal file name for the current operating system. If a file with the specified name does not exist, an error results.

You may add several options after the file name in an **OPEN** statement to specify how the file should be opened and accessed. Each option consists of an option keyword and an option specifier, and each option is separated from its neighbors by a comma. For example:

```
OPEN #1: NAME "DATA", ORG TEXT, CREATE OLD, ACCESS INPUT
```

The options allowed in the **OPEN** statement may be summarized as follows:

OPEN Statement Options

Option	Effect
ORGANIZATION TEXT	Open as text file
ORGANIZATION STREAM	Open as stream file
ORGANIZATION RANDOM	Open as random file
ORGANIZATION RECORD	Open as record file
ORGANIZATION BYTE	Open as byte file
CREATE NEW	Create a new file
CREATE OLD	Open an existing file (default)
CREATE NEWOLD	Open if exists, else create
ACCESS OUTIN	Allow read/input or write/print (default)
ACCESS INPUT	Allow read or input only
ACCESS OUTPUT	Allow write or print only
RECSIZE n	Set record length for random, record, or byte files

You may abbreviate the option keyword ORGANIZATION as ORG.

If you specify the ORG TEXT, ORG STREAM, ORG RANDOM, or ORG RECORD option when opening an existing file, True BASIC checks the file and gives an error if it does not match the specified type. If you use the ORG TEXT, ORG STREAM, ORG RANDOM, or ORG RECORD option when opening a new or empty file, the file becomes that file type. Any file may be opened with the ORG BYTE option; it will be treated as a byte file as long as it remains open, no matter what type it really is.

If you do not use an ORG option, True BASIC assumes the ORG TEXT option for new or empty files. Otherwise, it checks the file and uses its current type (compiled True BASIC programs are opened as ORG BYTE).

The CREATE options control what happens if the file does or does not already exist. The CREATE NEW option instructs True BASIC to create a new file with the specified name; an error occurs if a file with that name already exists. The CREATE OLD option opens an existing file; an error occurs if the file does not exist. The CREATE NEWOLD option opens the file if it already exists and creates it if it does not. The CREATE NEWOLD option is useful when a program will be run repeatedly; the first time it is run it creates a new file, afterwards it uses the existing one.

Omitting a CREATE option is the same as using the CREATE OLD option; that is, True BASIC looks for an existing file by default.

The ACCESS options let you limit what the program can do with the open file. The ACCESS INPUT option lets the program read from the file but not change its contents — often an important safety feature. The ACCESS OUTPUT option lets the program modify a file, but not read it — this can protect the confidentiality of the file. The default is the ACCESS OUTIN option, which gives the program complete access to the file for both reading and writing.

For example:

```
OPEN #7: NAME "FILE22.TRUE", CREATE NEWOLD, ORG TEXT
```

either will open an existing file called `FILE22.TRU` and make sure that it is a text file, or it will create one. Since an ACCESS option is not specified, both reading and writing will be allowed.

Even if you use ACCESS OUTPUT to limit access to a file to output, the operating system must permit True BASIC to read that file to examine its type.

The RECSIZE option sets the record length for record, random, or byte files to the numeric value given with it. Records are components of random-access files and are discussed more fully in the sections on random and record files later in this chapter. If you use a RECSIZE option when opening a text or stream file, it will be ignored.

Although you must type out the option keywords, you may use string variables and string expressions in place of the option specifiers. This can be extremely useful when writing subroutines. Also, note that channel numbers may serve as parameters to subroutines (but not to functions). The channel-number parameter must consist of a pound sign (#) followed by an integer — not an expression. As usual, the channel number passed as the corresponding argument in the invocation need not be the same as the channel number used as the parameter in the subroutine. For example:

```
SUB FileOpen(org$, cr$, acc$, #9)   ! Open specified file
   PRINT "File name";
   INPUT f$
   OPEN #9: NAME f$, ORG org$, CREATE cr$, ACCESS acc$
END SUB
```

This may be invoked using a statement such as:

```
CALL FileOpen("record", "old", "outin", #1)
```

Invoking `FileOpen` in this way will get the name of a file from the user and open the existing record file with that name with full access privileges. The open file will be associated with channel #1 throughout the program unit that contains the **CALL** statement.

Note that in the above subroutine, errors could occur if the user names a file that does not exist or is not a record file. The following version of the subroutine provides better protection by "trapping" any errors. The **WHEN** structure and other error-handling techniques are discussed in Chapter 16 "Error Handling."

```
SUB FileOpen(org$, cr$, acc$, #9)       ! Protected file opener
    DO
       CLOSE #9                         ! In case channel is open
       INPUT PROMPT "File name: ": fname$
       WHEN ERROR IN
            OPEN #9: NAME fname$, CREATE cr$, ORG org$, ACCESS acc$
            EXIT SUB                    ! Success
       USE
            PRINT "Cannot open that file."
       END WHEN
    LOOP
END SUB
```

There are several reasons why an **OPEN** statement may fail. Therefore, it is generally a good idea to use an error handler whenever appropriate to give the user more than one chance to specify a file name, as in the above example.

Remember that you must open every file before you may access it. If you try to access a channel number that has not been properly opened (or that has been closed), an error will result.

Once a channel has been opened, it will remain open as long as it remains in existence or until you specifically close it, as described in the next section. Channels obey the same scope rules as variables. How long a channel "remains in existence" depends on where the channel is opened.

A channel that is opened in the main program (or in a procedure internal to the main program) remains in existence throughout the remainder of the main program and may only be accessed within the main program. When the program stops running, all open channels are closed and destroyed.

A channel that is opened in an external procedure remains in existence throughout the remainder of that procedure's invocation. However, when the procedure returns to its caller, any channels opened by the procedure (except those passed to it) will be automatically closed and destroyed.

If an external subroutine needs to access a channel opened by its caller, the channel should be passed to the subroutine as a parameter (as in the previous example). When a previously opened channel is passed to a subroutine as a parameter, the subroutine may not contain an **OPEN** statement for that channel. However, if an unopened channel is passed to a subroutine as a parameter, the subroutine may open that channel. In this case, the channel used as the corresponding argument in the **CALL** statement will be open when the subroutine returns to its caller.

Procedures contained in a **MODULE** structure follow the same rules as external procedures. However, if a channel number appears within a **SHARE** statement in that module's header, that channel will be brought into existence when the module is initialized, and it will remain in existence until the program terminates. It will be available only to procedures contained within the module, and of course it must be explicitly opened before it may be used. Once opened, though, a shared channel will remain open until the program terminates (or until it is specifically closed, as described in the next section). (See Chapters 10 and 11 for more information on internal and external procedures and on modules.)

Closing Files

Although you may use any channel number from 1 to 999 (#0 is reserved for the default logical window), True BASIC does not allow more than twenty-five channels to be open at any one time (channel #0 is not counted). Generally, you will not need this many channels, but if your program opens several files (remember that the printer and logical windows require channels too) you may find that you are running into this limit.

It is therefore a good practice to specifically close open channels when you no longer need them. You do this with the **CLOSE** statement. For example:

```
CLOSE #3
```

closes channel #3. If channel #3 was not open, the **CLOSE** statement would simply be ignored. Channels are automatically closed when the program terminates, and channels local to a procedure are closed when that procedure terminates.

Once a channel has been closed, you may reuse the channel number previously associated with it. Note, however, that if you attempt to open a channel that is still open, an error will result.

Erasing Files

True BASIC provides two means of erasing a file. The **ERASE** statement erases a file's contents, and the **UNSAVE** statement destroys the file itself.

The **ERASE** statement erases the entire contents of the file associated with the specified channel number. For example:

```
ERASE #3                ! Erase whole file's contents
```

Of course, the channel must be open with the ACCESS OUTIN option. The **ERASE** statement simply erases the file's contents; the file continues to exist and the channel to it remains open.

The **ERASE** statement does not change any of the file's attributes (as specified by the associated **OPEN** statement); however, once the file is empty some of these attributes (such as file type and record size) can be changed intentionally or incidentally. For example, you may use the **SET RECSIZE** statement to change the record size of an empty random or record file. And if you use a **PRINT** statement to send output to an empty file, that file will become a text file.

A variation of the **ERASE** statement lets you erase the portion of the file following the current position of the file pointer. (File pointers are described fully in the following section.) For example:

```
ERASE REST #3          ! Erase rest of file's contents
```

If you wish to remove a file completely from the storage medium, use the **UNSAVE** statement. For instance, the statement:

```
UNSAVE "FILE22.TRU"         ! Delete the file itself
```

would completely delete the file named `FILE22.TRU`. If the file does not exist, an error occurs. The **UNSAVE** statement requires a file name rather than a channel number. In fact, you must close any channels associated with the file before the **UNSAVE** statement is executed; if a channel to the file is open, an error occurs.

SET and ASK Statements

There is a lot of information involved in the maintenance of files, and True BASIC provides convenient ways to access that file-related information. Several **SET** and **ASK** statements let you manipulate files and get information about them. This section discusses those **SET** and **ASK** statements that work with all files. More specialized **SET** and **ASK** statements are described in the sections about the individual file types.

[!] **Note:** Most of the **ASK** statements described in this section can also provide information about logical windows and printers. For details on opening and using logical windows see Chapter 13 “Graphics.” Details on opening and using printers are provided later in this chapter. You will also find information on the appropriate **SET** and **ASK** statements in those sections.

File Pointers

For each currently open file there is an associated ***file pointer*** that indicates where the next information read from or written to that file is to begin. When you first open a file, True BASIC places the file pointer at the beginning of the file. As your program reads items from the file, or writes information to it, True BASIC automatically moves the pointer to the end of the last item read or written.

In general, you do not need to move the file pointer yourself. However, there are occasions when you will need to move it. For instance, you are allowed to write information only to the end of text and stream files. Since the file pointer is at the beginning of a file when the file is opened, you must move the file pointer to the end of an existing text or stream file before you can add information to it.

You control the position of the pointer with the **SET POINTER** statement, as follows:

```
SET #3: POINTER BEGIN           ! Go to beginning of file
SET #3: POINTER END             ! Go to end of file
```

The following forms of the **RESET** statement are equivalent to these **SET POINTER** statements:

```
RESET #3: BEGIN                 ! Go to beginning of file
RESET #3: END                   ! Go to end of file
```

You can reread a file if you reset the file pointer at the beginning, or append information to the file if you move the file pointer to the end.

For each attribute that can be set with a **SET** statement, there is a corresponding **ASK** statement (the reverse is not always the case). Thus, you may easily find out the current position of a file pointer with the **ASK POINTER** statement, as follows:

```
ASK #3: POINTER ptr$            ! Where is the pointer?
```

In this statement, the variable `ptr$` will be assigned one of the values `"BEGIN"`, `"MIDDLE"`, or `"END"` depending on the current position of the file pointer within the file associated with channel #3.

You can test whether you have reached the end of a file using the logical expressions END or MORE. For example, this program fragment ensures that the file pointer is at the beginning of a text file and then prints the file’s contents:

```
RESET #3: BEGIN
DO
   LINE INPUT #3: line$
   PRINT line$
LOOP UNTIL END #3                 ! Is TRUE if at end of #3
```

The following program fragment is equivalent except that it prevents the error that would occur if the file is empty:

```
RESET #3: BEGIN
DO WHILE MORE #3                  ! Is TRUE if not at end of #3
   LINE INPUT #3: line$
   PRINT line$
LOOP
```

With random-access files such as random, record, and byte files, you can also move the file pointer to individual records and ask for the current record number; see the descriptions for those files.

Names and Directories

You may use the **ASK NAME** statement to find out the name of an open file:

```
ASK #5: NAME filename$          ! Get name of file #5
```

The **ASK NAME** statement will report the full path name of the file associated with that channel number.

[!] **Note:** The **ASK NAME** and **SET NAME** statements without channel numbers are provided for compatibility with earlier versions of True BASIC. In earlier versions, **ASK NAME** without a channel number reported the name of the current program. This version of True BASIC assigns the null string if **ASK NAME** is used without a channel number; **SET NAME** is ignored.

As noted earlier, you may use a path name in the **OPEN** statement to access files that are not stored in the current directory. If you will be opening several files in the same directory, you may prefer to use a **SET DIRECTORY** statement to change the current directory, thus avoiding the need for path names in the **OPEN** statements. For example, the statement:

```
SET DIRECTORY dir$                ! Change directory
```

would set the current directory to the directory specified by the value of `dir$`. The value of `dir$` must be a legal directory name, and it may include a disk name. See "The True BASIC Environment" chapter in the introductory section for information on specifying directories within various operating environments.

The **ASK DIRECTORY** statement lets you find out the name of the current directory. Thus, if you change directories in your program and wish to be able to return to the starting directory before the program ends, you could store the name of your starting directory before changing to a new directory as follows:

```
ASK DIRECTORY old_dir$          ! Starting directory
SET DIRECTORY dir$              ! Change to new directory
```

Then, later switch back to the original directory with the following statement:

```
SET DIRECTORY old_dir$          ! Change to starting directory
```

When the program terminates, you are returned to the directory you were in when you ran the program. The final section in this chapter describes ExecLib library routines that accomplish the same thing as the **ASK DIRECTORY** and **SET DIRECTORY** statements.

File Characteristics

Other **ASK** statements provide information about the file itself or how the file was opened. The following statements may be used regardless of the file type; see the specific file types for additional statements.

The **ASK ORG** statement finds out the type of file that is currently associated with an open channel. For instance, the statement:

```
ASK #3: ORG org$
```

assigns a value of "TEXT", "STREAM", "RANDOM", "RECORD", or "BYTE" to the variable org$. If the channel number refers to a printer, the value "TEXT" is assigned to org$; and if the channel number refers to a logical window, "WINDOW" is assigned to org$.

The **ASK RECTYPE** statement finds out the nature of a file's records. The statement:

```
ASK #3: RECTYPE rectype$
```

assigns a value of "DISPLAY" or "INTERNAL" to the variable rectype$. If the channel number refers to a text file, a printer, or a logical window, the value "DISPLAY" is assigned to rectype$. For all other types of files, "INTERNAL" is assigned to rectype$.

The **ASK ACCESS** statement finds out the access available for the file associated with the specified channel number. For instance, the statement:

```
ASK #3: ACCESS acc$
```

assigns to acc$ a value of "INPUT", "OUTPUT", "OUTIN", "NETIN", "NETOUT", or "NETWORK", as determined by the ACCESS option used when channel #3 was opened. If channel #3 refers to the printer, a value of "OUTPUT" is assigned to acc$. If channel #3 refers to a logical window, "OUTIN" is assigned.

The **ASK FILESIZE** statement lets you find out the size of a file. For example:

```
ASK #3: FILESIZE fs  ! Length in bytes (in records for random & record);
                     !   0 for printer or logical window
```

If the file associated with channel #3 is a text, stream, or byte file, then the number of bytes in the file is assigned to fs. If the file is a record or random file, then the number of records in the file is assigned to fs. If the channel refers to a printer or a logical window, a size of 0 is returned.

The ExecLib routines **Exec_ReadDir** and **Exec_ClimbDir** (described in the last section of this chapter) provide additional information about files and directories including size, date and time last modified, and access permissions.

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.

[!] Note: 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 pre-

cision 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 StrLib 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.

Sending Textual Output to a Printer

You may use a printer as you would a text file opened with the ACCESS OUTPUT option. That is, you may send output to a printer by opening a channel to it and using a **PRINT** statement with that channel number, but you may not use any form of **INPUT** statement with that channel for obvious reasons.

You open a channel to the printer with a special form of the **OPEN** statement, without any options, as follows:

```
OPEN #7: PRINTER
```

After the above statement has been executed, channel #7 will be associated with the printer. Of course, the printer must be turned on, placed on-line, and properly connected to the computer so that True BASIC is able to use it. If a printer is not available, True BASIC will generally generate an error.

If the current operating environment has access to more than one printer, True BASIC opens the channel to the printer that the operating environment identifies as the default choice. Refer to the documentation for your operating environment for more information.

Once you have associated a printer with a channel number, you may send textual output to it with the **PRINT** statement as follows:

```
OPEN #1: PRINTER
FOR i = 1 to 100
    PRINT #1: i             ! Prints to temporary spool file
NEXT i
CLOSE #1                    ! Spool file sent to printer
END
```

The **OPEN** statement associates the specified channel with a special file called a ***spool file***. The spool file is a temporary file that True BASIC creates automatically on the disk. When you send output to the printer channel, that output is stored in the spool file, which continues to accumulate output until the printer channel is closed. Once the printer channel has been closed, True BASIC sends the contents of the complete spool file to the printer and deletes the spool file from the disk. (The channel is closed by a **CLOSE** statement or when the program ends.)

A temporary spool file is used because of the prevalence of page-oriented and networked printers. Page-oriented printers, such as most laser and ink-jet printers, print their output one page at a time, rather than one line at a time, like most dot-matrix printers. Page-oriented printers often do not behave gracefully when they are sent single lines of text at odd intervals. Networked printers often handle the demands of several users at once, and as such they do not cooperate when one program attempts to claim sole ownership for a significant time. By storing your output temporarily in a spool file, True BASIC can send the entire output as a single document, avoiding problems with page-oriented and networked printers.

As with a text file, True BASIC opens a printer with a default margin of 80 characters and a default zonewidth of 16 characters. As with a text file, you may access these settings with the **ASK MARGIN**, **SET MARGIN**, **ASK ZONEWIDTH**, and **SET ZONEWIDTH** statements. Beware, however, that most printers have a physical limitation on the width of the lines that they can print, and setting the margin larger than this value may not have any effect.

Basic Directory Operations

True BASIC statements such as **OPEN**, **READ**, **PRINT**, **SET**, **ASK**, and so on open and access files. For dealing with directories, True BASIC provides the built-in subroutine **SYSTEM**. This subroutine lets a program find out the current directory, change it, create and remove directories, rename files, and get information on the contents of a directory and possibly all its subdirectories.

The **SYSTEM** subroutine, however, is complex and not easy to use. Thus True BASIC also includes the ExecLib library of subroutines that provide easier methods of performing directory operations. This section describes the use of the ExecLib library of subroutines; for information on the built-in **SYSTEM** subroutine, see Chapter 18.

To use these convenience routines, you must include a library statement in your program, such as:

```
LIBRARY "C:\TBSilver\TBLIBS\ExecLib.Trc"   !Use appropriate path name
```

The ExecLib library contains six subroutines that let your programs find out the current directory, change to a new directory, create a new directory, and find out about the contents of a directory including all its subdirectories if desired. A seventh subroutine lets you to rename a file. (Use the True BASIC statement **UNSAVE** to delete a file.)

Identifying and Changing Directories

The **Exec_AskDir** and **Exec_ChDir** subroutines provide the same functionality as the **ASK DIRECTORY** and **SET DIRECTORY** statements. For example:

```
CALL Exec_AskDir (dir$)
```

returns the path name of the current directory in the string variable `dir$`.

Similarly, the **Exec_ChDir** subroutine:

```
CALL Exec_ChDir (newdir$)
```

will change the current directory to the one specified by the contents of new`dir$`. (This should be equivalent to using the CD command on most systems.) If the argument does not specify a valid directory, an error occurs.

As with the **SET DIRECTORY** statement, when the program terminates the current directory returns to what it had been before the run began.

Creating and Deleting Directories

The **Exec_MkDir** subroutine lets your programs create new directories. For example:

```
CALL Exec_MkDir (newdir$)
```

creates a new directory in a location determined by the pathname conventions. If the new directory name contains a path name that starts at root level, the new directory is placed with respect to that root directory. If the new directory path name does not start at root level, the new directory will be placed in the current directory. If the argument does not specify a valid directory name, an error occurs.

The **Exec_RmDir** subroutine removes the named directory:

```
CALL Exec_RmDir (dir$)
```

Some systems may require that the directory be empty before allowing it to be removed. You could use the **Exec_ReadDir** routine described below along with the **UNSAVE** statement to get the names for all the files in a given directory and delete them.

If the argument does not specify a valid directory, an error occurs.

Finding Out About Files in a Directory

The **Exec_ReadDir** and **Exec_ClimbDir** subroutines provide list of names and statistics about the files saved within a directory. **Exec_ReadDir** provides information on the files directly saved in the current directory, while **Exec_ClimbDir** provides information on the files in the designated directory along with those in subdirectories within that directory.

Calls to the routines take the following formats:

```
CALL Exec_ReadDir (template$, name$(), size(), dlm$(), tlm$(), type$(), vname$)
CALL Exec_ClimbDir (dir$, template$, name$(), size(), dlm$(), tlm$(), type$())
```

For the `template$` argument, you may pass a string to select a subset of files (such as `"*.TRU"`). Specify an empty string if you do not wish to limit the search. For **Exec_ClimbDir**, you must specify the topmost directory to search in the `dir$` argument; **Exec_ReadDir** searches the current directory.

Most of the information is returned in a series of one-dimensional arrays as follows:

`name$()`	the names of the files (and possibly directories) in the current or specified directory (for **Exec_ReadDir** the names are simple names of files in the current directory; for **Exec_ClimbDir** the names are returned as full path names)
`size()`	the sizes of the files in bytes
`dlm$()`	the date last modified, in the True BASIC **DATE$** function format `"YYYYMMDD"` where `"YYYY"` is the four-digit year number, `"MM"` is the two-digit month number, and `"DD"` is the two-digit day number
`tlm$()`	the time last modified, in the True BASIC TIME$ function format `"HH:MM:SS"` where `"HH"` is the two-digit 24-hour number, `"MM"` is the two-digit minute number, and `"SS"` is the two-digit second number
`type$()`	the type or access permissions given as a four-character string of the form: `"drwx"` where the first character is `"d"` if the entity is a directory and `"-"` if it is a file; the second character is `"r"` if reading the file or directory is permitted and `"-"` otherwise; the third character is `"w"` if writing or appending to the file or directory is permitted and `"-"` otherwise; and the fourth character is `"x"` if the file is directly executable and `"-"` if not (directories are not executable)

Renaming Files

You can change the name of existing files with the **Exec_Rename** routine:

```
CALL Exec_Rename (oldname$, newname$)
```

The above statement will rename the file specified in `oldname$`, giving it the name specified in `newname$`. If either `oldname$` doesn’t exist or `newname$` is not valid, an error occurs.

An Example

The following example shows how you could use some of the ExecLib subroutines to selectively delete or rename files in a given directory:

```
LIBRARY "C:\TBSilver\TBLIBS\ExecLib.TRC"   ! Use appropriate path name
DIM name$(1), size(1), dlm$(1), tlm$(1), type$(1)
DIM dirnames$(1)

DO
   MAT REDIM dirnames$ (100)
   INPUT PROMPT "Give full path name for directory to be examined": dir$
   CALL Exec_ChDir (dir$)          ! Change to directory to be removed
   CALL Exec_ReadDir ("", name$(), size(), dlm$(), tlm$(), type$(), vname$)

   FOR i = 1 to Ubound(name$)
      IF type$(i)[1:1] = "-" then  ! A file
         PRINT name$(i), size(i), dlm$(i), tlm$(i)
         INPUT PROMPT "Rename (r), delete (d), or continue(c)?": action$
         LET action$ = lcase$(action$[1:1])
         SELECT CASE action$
         CASE "r"                                  ! Rename it
            INPUT PROMPT "New name? ": newname$
            CALL Exec_Rename (name$(i), newname$)
         CASE "d"
            UNSAVE name$(i)                        ! Remove it
         CASE ELSE
         END SELECT
      ELSE                                         ! A directory
         LET num_dirs = num_dirs + 1
         LET dirnames$(num_dirs) = name$(i)        ! Store the name
      END IF
   NEXT i

   IF num_dirs > 0 then            ! Subdirectories were found
      MAT REDIM dirnames$(num_dirs)
      LET num_dirs = 0             ! Reset directory counter
      PRINT "The following subdirectories were found within the directory:"
      MAT PRINT dirnames$
   END IF

   INPUT PROMPT "Examine other directories (y or n)?": more$
   IF lcase$(more$[1:1]) = "y" then let flag = 1      ! Repeat loop

LOOP WHILE flag = 1

END
```

CHAPTER

13

Graphics

One of the many advantages to programming in True BASIC is the power and simplicity of its graphics capabilities. Using True BASIC's various tools, you can easily create complex graphical images to enhance your programs. And unlike those produced by many other programming languages, the graphics you create with True BASIC programs will look the same regardless of the operating environment you use.

True BASIC's **PLOT** statements draw points, lines, curves, and filled regions. You can use any combination of colors your computer provides, and you may freely mix printing and graphics. You define the coordinate system for the graphic statements, and you can create several regions or "logical windows" within the physical output area for graphical or text elements. The user can supply coordinate input to your program as it is running. **BOX** statements can speed up many graphics operations and animate your drawings. ***Pictures*** are subroutines that let you define graphical components that you may combine and transform to create complex drawing.

This chapter introduces all the above elements of True BASIC's graphics statements. It introduces physical windows and logical windows, and describes the coordinate systems available with True BASIC. See Chapter 14 "Interface Elements" for details on creating and manipulating physical windows and on the True Controls library of subroutines for creating graphic and other objects such as menus, scroll bars, radio buttons, and check boxes.

Windows and Coordinate Systems

True BASIC uses two kinds of windows — physical and logical windows — and three coordinate systems — pixel coordinates, screen coordinates, and user coordinates.

For most of your work with True BASIC graphical statements, you will use ***user coordinates***. Graphical statements place elements in the current display area based on user coordinates. You define the limits of these coordinates with the **SET WINDOW** statement described below in the section on "User Coordinates." For quick, simple graphical output, all you need to do is define the limits of the user coordinates and then use the appropriate graphics statements to draw images within those coordinates. In this simple case, True BASIC uses the full content area of the output window to display output in the user-coordinate range you define (or, more technically, True BASIC fits your user-coordinate system into the default logical window that occupies the entire default physical window). If you are new to graphic programming, you can begin by reading the section on user coordinates below and then skipping ahead to "Plotting Points and Lines," returning later to the explanations below.

You can further control the graphical display area, however, by defining specific window areas to display different ranges of user-coordinates. You do this by creating one or more ***logical windows*** within the ***default physical window*** — the standard output window — or by creating additional ***physical windows*** for output, which could in turn have multiple logical windows. The next two sections define physical and logical windows and show how to use ***screen coordinates*** to create logical windows within a physical window.

You create additional physical windows — and may also position user-interface objects within physical windows — with ***pixel coordinates***. Pixel coordinates are introduced below. Chapter 14 "Interface Elements" describes how to create and manipulate physical windows.

Physical Windows vs. Logical Windows

A ***physical window*** in True BASIC is the type of window your operating system uses. These windows occupy a distinct area of the screen and often have features such as title bars, borders, and scroll bars that easily identify them as windows.

When you run a program that produces screen output, True BASIC automatically creates a physical window to display that output. This window is called the ***default physical window***, and it is easily visible on the screen.

What you can't easily see, however, is that True BASIC also creates another window within the default physical window. This second window has no visually identifying features, such as a title or border. This type of window is a ***logical window***. The ***default logical window*** fills the entire content area of the default physical window; your user-coordinate system fills this area if you do not define a specific logical window. (The ***content area*** of a physical window is the region that may contain output; it does not include the title bar and scroll bars.) While your computer's operating system does most of the management of physical windows, True BASIC has exclusive control of logical windows.

True BASIC uses logical windows to partition a physical window. Logical windows serve two important functions: (1) they provide a framework for defining user-coordinate systems, and (2) they define a "clipping region" for graphical output. As described below, you can create several logical windows within a physical window and direct different output elements to different logical windows.

Chapter 14 "Interface Elements" describes how you can use the True Controls library of routines to create and maintain numerous physical windows. In this chapter, however, we assume that you are working with only the default physical window, although you may have several logical windows within that physical window.

[!] Note: Each logical window must exist within a physical window. Although it is possible to use the built-in **Object** routine (see Chapter 19 "Object Subroutine") to create a physical window that does not contain a logical window, it is not possible to define a logical window without a physical window. This chapter defines all logical windows within the default physical window.

Creating Logical Windows with Screen Coordinates

As noted above, if you do not specifically create a logical window, your output will use the default logical window, which fills the content area of the default physical window. Although you can accomplish a lot using only the default logical window, you may sometimes want to define additional logical windows.

You may define any rectangular region of a physical window's content area as a logical window. The **OPEN** statement with the SCREEN keyword creates a logical window within the current physical window, as follows:

```
OPEN #1: SCREEN left, right, bottom, top
```

The **OPEN** statement associates the defined area of the physical window with a specified channel number. ***Channel numbers*** always consist of a pound sign (#) followed by a number from 1 to 999 (or a numeric expression that evaluates to such a value). Channel #0 is reserved for the default logical window, which is always open. After you've opened a logical window, you must always refer to it by its associated channel number.

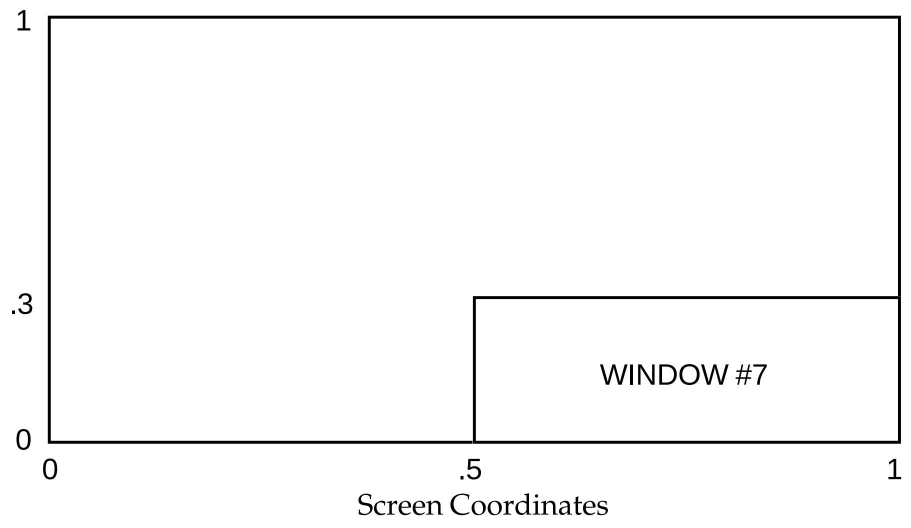
The four numeric values following the keyword SCREEN define an area of the physical window. They use ***screen coordinates*** to represent the positions of the left, right, bottom, and top edges of the logical window within the physical window. Screen coordinates are used exclusively for positioning logical windows. In this coordinate system the point (0, 0) is always in the lower, left corner of the physical window's content region. The upper, right corner is always the point (1,1). Thus, the x-axis (horizontal axis) of the physical window ranges from 0 (at the left end) to 1 (at the right end), and the y-axis (vertical axis) ranges from 0 (at the bottom) to 1 (at the top).

You may define any region of the physical window as a logical window by giving the locations of its edges as values between 0 and 1, inclusive, in an **OPEN** statement. For instance, the example:

```
OPEN #7: SCREEN .5, 1, 0, .3
```

opens a logical window that occupies the lower right corner of the content area of the physical window and associates this logical window with channel #7.

True BASIC Screen Coordinates



The **CLOSE** statement closes channels associated with logical windows (or files or printers) For instance, the statement:

```
CLOSE #7
```

would close the logical window associated with channel #7.

You may not use a channel number that is already open to open a logical window, but you may reuse channel numbers after they have been closed. Also, True BASIC does not allow more than 25 open channels at any one time (not counting #0 which is always the default logical window). Thus, when your program no longer needs a logical window, it should close it to make that channel available for reuse. (Note that files and printers also use channel numbers, see Chapter 12 “Files for Data Input and Output.” A channel number may be associated with only one open item; that channel must be closed before it can be reused for another item.)

Following an **OPEN** statement, the logical window just created will be the ***current logical window*** where True BASIC will send subsequent output (either textual or graphical). Only one logical window can be the current logical window at any one time. To change the current logical window, use the **WINDOW** statement. For instance, to switch to the logical window associated with channel #7, use:

```
WINDOW #7
```

Or, if you have opened four logical windows using channels #1 through #4, the following code:

```
FOR n = 1 to 4     ! Window number
    WINDOW #n      ! Switch window
    PRINT "Window"; n
NEXT n
```

will label each of the logical windows.

True BASIC remembers each logical window’s currently selected options (as described below); when the program switches to a logical window, all options defined for that window are available.

The default logical window is always associated with channel #0 and cannot be closed. Therefore, you can always return to this window with the statement:

```
WINDOW #0
```

If you need to find out the screen coordinates that define the location of the current logical window, you can use the **ASK SCREEN** statement:

```
ASK SCREEN left, right, bottom, top
```

This assigns the screen coordinates of the current logical window to `left, right, bottom, and top`.

User Coordinates

Each logical window has a ***user-coordinate system*** used by True BASIC's various plotting statements to position output within the logical window. This coordinate system is defined by a range of values along the horizontal edge or ***x-axis*** of the window combined with the range of values along the vertical edge or ***y-axis*** of the window. Most graphical operations described in this chapter use user coordinates. You can define any point within the logical window simply by specifying its position in relation to both axes. A point's position along the x-axis is its ***x-coordinate***, and its position along the y-axis is its ***y-coordinate***. Thus, any point may be identified by its x- and y-coordinates.

When a logical window is first opened, it has a ***default user-coordinate system*** in which the x-axis ranges from 0 (on the left) to 1 (on the right) and the y-axis ranges from 0 (at the bottom) to 1 (at the top). Thus, by default, the user-coordinate point (0, 0) is the lower, left corner of a logical window, and the point (1,1) is the upper, right corner of the window.

You may change a logical window's user-coordinate system to anything you wish with the **SET WINDOW** statement. For example:

```
SET WINDOW 0, 2*pi, -1, 1
```

specifies that the values along the x-axis range from 0 (on the left) to 2π (on the right) and the values along the y-axis range from –1 (at the bottom) to 1 (at the top). This user-coordinate system would be suitable for plotting a sine curve. If you wanted to graph population data (in millions) between 1900 and 1990, you might use the following coordinates:

```
SET WINDOW 1890, 2000, -30, 300
```

Notice how these coordinates define an area slightly larger than the graph itself, allowing room for labels.

Remember that each logical window you create has its own user-coordinate system. Thus, if you open several different windows, you may want to specifically set their user-coordinate ranges:

```
OPEN #1: SCREEN .5, 1, 0, .4             ! Lower right portion
SET WINDOW -10, 10, -10, 10              ! Define coordinate system

OPEN #2: SCREEN .5, 1, .5, 1             ! Upper right portion
SET WINDOW -1, 11, -5, 100               ! Define coordinate system
```

You will usually find that your code will be easier to understand if you place the **SET WINDOW** statement immediately after the **OPEN** statement. However, True BASIC does not require this; you may use a **SET WINDOW** statement at any time to change the user coordinates of the current logical window.

You can find the current user-coordinate ranges of the current logical window with the **ASK WINDOW** statement:

```
ASK WINDOW left, right, bottom, top
```

This example would assign the user coordinates of the current logical window to `left, right, bottom,` and `top`.

User coordinates provide great power and flexibility. You may specify any axes range you wish including ranges from larger to smaller values. In fact, the only limitation of the **SET WINDOW** statement is that the values of the left and right ends of the x-axis may not be equal, nor may the values of the top and bottom ends of the y-axis. You will find that most graphical applications are easy to implement when you can choose a coordinate system suited to your needs.

The user-coordinate system adapts itself as the size or shape of the logical window changes. Regardless of the size or shape of the logical window, its user-coordinate range will remain the same — units along the axes are stretched or condensed so that the defined ranges always fill the current logical window. This greatly simplifies graphical programming in varied environments, since programs can draw equivalent images in logical windows of any size and shape on any computer without changes to the source code. This is a significant advantage over the third type of coordinate system available within True BASIC — pixel coordinates.

Pixel Coordinates

Each physical window has a pixel-coordinate system in addition to the screen-coordinate system used to define logical windows.

The word “pixel” is an abbreviation of the phrase “picture element,” and it refers to the units that form images on the computer screen. A computer screen is divided into a very fine grid of a very large number of rectangles. By changing the color of some of these rectangles, the computer displays “pictures” on the screen, thus the rectangles are called picture elements or ***pixels***.

A ***pixel-coordinate system*** is, therefore, another way of identifying points within the content area of a physical window. Each point has two pixel coordinates. The first represents the point’s location as a number of pixels from the left edge of the window, and the second represents the point’s location as a number of pixels from the top edge of the window. (Note that pixel coordinates start from the top edge, while screen coordinates and user coordinates begin at the bottom edge.)

Everything displayed on a computer screen is a pattern of pixels of various colors, but not all computer screens display the same number of pixels. The number of horizontal and vertical pixels determines the screen’s ***resolution***. Pixel-coordinate ranges therefore vary depending upon the resolution of the current computer hardware, the computer’s operating environment, and the size of the physical window in relation to the full screen.

Because they can vary so easily, pixel-coordinate systems are less desirable than user-coordinate systems for most types of graphics. True BASIC graphical statements described in this chapter automatically translate user coordinates into pixel coordinates, thus you need not worry about pixel coordinates when using them. Most of the interface objects described in Chapter 14 “Interface Elements” also let you use user coordinates if you wish, but there are times when you might wish to use pixel coordinates directly.

You may also define a user-coordinate system to mimic pixel coordinates. To do so use the **ASK PIXELS** statement, which reports the number of pixels within the current logical window in the horizontal and vertical directions, along with the **SET WINDOW** statement. For example:

```
ASK PIXELS hpix, vpix
SET WINDOW 0, hpix-1, vpix-1, 0
```

Notice that the `vpix` value sets the *bottom* range of user coordinates. In pixel coordinates “bottom” is always greater than “top” because pixels are counted from the top edge.

Aspect Ratios

You may be disappointed when you ask True BASIC to draw a circle or a square; a square may look like a rectangle, and a circle may look like an ellipse (or oval). Getting square squares and round circles can be tricky as it involves adjusting the aspect ratio of the current logical window.

The ***aspect ratio*** of a window compares the distance of a horizontal line segment to an equivalent vertical line segment — in user-coordinate units. When a window’s aspect ratio is 1, equivalent horizontal and vertical lines will appear to be the same length, and as a result squares will look like squares and circles will look like circles.

Since most computers today have square pixels, you can adjust the aspect ratio of a window by matching the x to y ratios for pixel and user coordinates. For the current logical window for example, the following code would set up a user-coordinate system with the origin (0,0) in the center of the window and with an aspect ratio of 1:

```
ASK PIXELS hpix, vpix
LET pratio = hpix/vpix      ! Find x to y ratio for pixels
LET vrange = 20
LET hrange = 20 * pratio    ! x to y ratio for user coordinates will
                            ! = pixel ratio
SET WINDOW -(hrange/2), (hrange/2), -(vrange/2), (vrange/2)
```

Plotting Points and Lines

The **PLOT POINTS** and **PLOT LINES** statements let you draw points or lines on the screen. For example, the following statement plots the corners of an isosceles triangle:

```
PLOT POINTS: 1,1; 3,1; 2,2
```

And the following draws the sides of the triangle:

```
PLOT LINES: 1,1; 3,1; 2,2; 1,1
```

In the **PLOT LINES** statement the first point must be repeated at the end so that the last point connects to the original one. Notice that both statements have a colon after the keywords, before the list of points.

If the statement is quite long, you can divide it into more than one statement. But for **PLOT LINES**, each statement other than the last must end with a semicolon to indicate that the lines should be connected:

```
PLOT LINES: 1,1; 3,1;
PLOT LINES: 2,2; 1,1
```

The following program draws fifty randomly chosen points:

```
SET WINDOW 0, 1, 0, 1       ! All points between 0 and 1 for each axis
FOR n = 1 to 50
    PLOT POINTS: Rnd, Rnd   ! Random point
NEXT n

END
```

Replacing the **PLOT POINTS** statement with:

```
PLOT LINES: Rnd, Rnd;            ! Random lines
```

will produce a random zig-zag pattern.

You may use the **PLOT** statement as an abbreviation for either the **PLOT POINTS** or **PLOT LINES** statement. If you are plotting unconnected points, however, you must place each coordinate-pair on a separate **PLOT** statement. For example, to draw the points of a triangle you would need three statements:

```
PLOT 1,1
PLOT 3,1
PLOT 2,2
```

A semicolon between or after coordinate pairs on a **PLOT** statement connects the points with lines. Thus, a triangle could be drawn with the statement:

```
PLOT 1,1; 3,1; 2,2; 1,1
```

Similarly, the random points in the loop above could be drawn with:

```
FOR n = 1 to 50
    PLOT Rnd, Rnd                ! Random point
NEXT n
```

and the random lines with:

```
FOR n = 1 to 50
    PLOT Rnd, Rnd;               ! Random lines
NEXT n
```

Notice that there is no colon in a **PLOT** statement.

Although, True BASIC can draw only straight lines between two points, you can plot a curved line as a series of several short lines. As an example, look at the following code segment that prints a table of the sine function:

```
FOR x = 0 to 2*Pi step .1        ! Use built-in functions Pi and Sin
    PRINT x, Sin(x)
NEXT x
```

Just changing the **PRINT** statement to a **PLOT** statement will plot the corresponding points (in a suitable user-coordinate system). And adding a semicolon at the end of the **PLOT** statement will plot the sine curve:

```
SET WINDOW 0, 2*Pi, -1, 1
FOR x = 0 to 2*Pi step .1
    PLOT x, Sin(x);              ! Plot sine curve
NEXT x
PLOT                             ! Stop connecting points

END
```

A **PLOT** statement with no coordinate pair and no punctuation, sometimes called a ***vacuous*** **PLOT** statement, starts a new line or curve. This is analogous to a **PRINT** statement with nothing after it. Thus, the above program uses a vacuous **PLOT** statement after the **NEXT** statement in case it later plots another point or line. Without a vacuous **PLOT**, the last point of the sine curve would be connected to the next point plotted. While it is not essential in this example, it is a good habit to use vacuous **PLOT** statements to avoid stray lines when expanding or maintaining your programs.

In the previous example, the entire curve will be visible. If, however, the user coordinates did not include the entire range, the curve would be ***clipped*** at the logical window boundary. That is, only the part of the curve that lies within the window is drawn. For example, with the user coordinates:

```
SET WINDOW 0, 2*Pi, 0, 1
```

only the top half of the curve would be visible. No error results; the entire curve is "drawn," but that portion outside the bounds of the current logical window is not shown.

Note that all plotting is performed using the current color, as explained later in this chapter.

If you want to plot many points, it may be convenient to compute the coordinates first and store them in an array. The array must be two-dimensional, with one row for each point and exactly two columns. The first column contains the x-coordinates and the second contains the corresponding y-coordinates. The statements **MAT PLOT POINTS** and **MAT PLOT LINES** work like the corresponding **PLOT** statements, plotting points or lines contained in the array named with the statement.

For example, the following program plots a sine curve by first storing the values in an array and then using **MAT PLOT LINES** to plot the points in the array:

```
SET WINDOW 0, 2*Pi, -1, 1
DIM sincurve (100,2)

FOR x = 0 to 2*Pi step .1
    LET point = point + 1
    LET sincurve (point,1) = x      ! Store values in array
    LET sincurve (point,2) = Sin(x)
NEXT x

MAT redim sincurve(point,2)         ! Remove any uncomputed points
MAT PLOT LINES: sincurve            ! Plot values in the array

END
```

Plotting Areas

The **PLOT AREA** statement draws the outline of a region (which may be quite complex) and colors its interior in the current foreground color. It works very much like the **PLOT LINES** statement, but, since the region must be enclosed, it automatically connects the last point to the first. Thus:

```
PLOT AREA: 1,1; 3,1; 2,2
```

will draw a triangle and fill it in. If the boundaries of the region cross each other, it is not obvious which points are on the inside. True BASIC uses a standard mathematical solution of this problem.

You can also color an area with the **FLOOD** statement. After you have drawn the boundaries of a region, you may color a contiguous piece of it with the statement:

```
FLOOD x, y
```

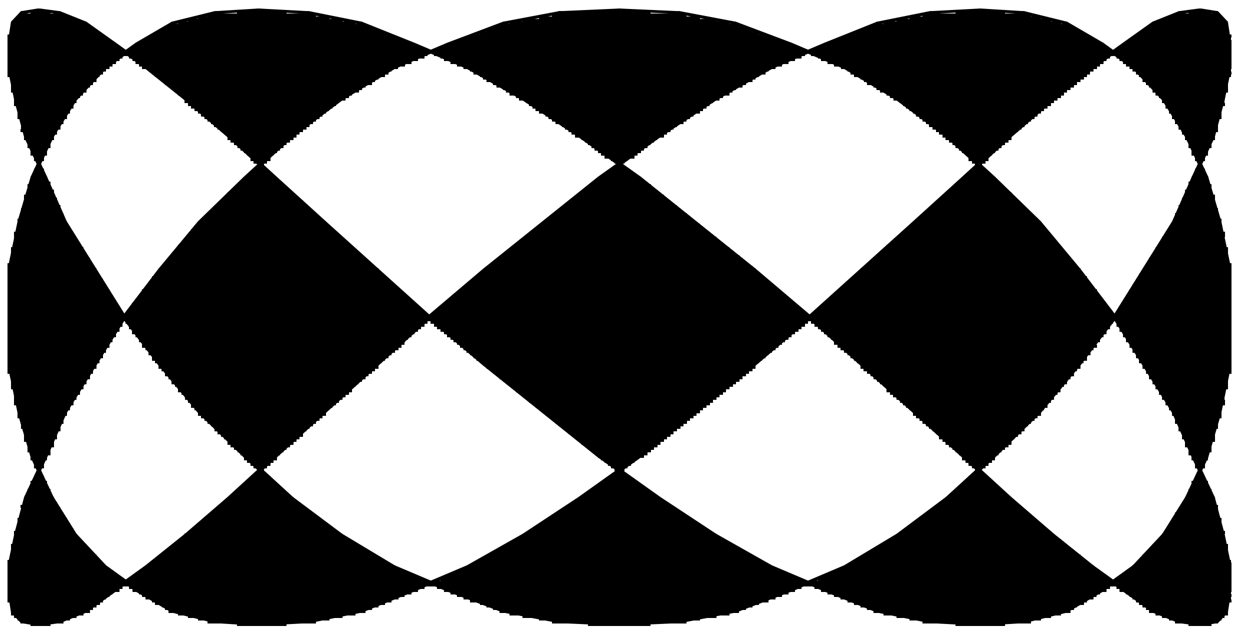
Flooding uses the current foreground color starting from the point `x, y` and continuing out to the boundaries, which are identified by any color different from the original color of the point `x, y`. You may color different areas by using several **FLOOD** statements. To color the exterior, use a coordinate point outside the region.

[!] Note: If the color on the screen is a dithered color, FLOOD will not work correctly. Colors need to be solid (realizing them if necessary) for FLOOD to work correctly.

As mentioned above for plotting points, you can store and plot coordinates in two-dimensional arrays. The first column of the array must contain the x-coordinates and the second, the corresponding y-coordinates. The **MAT PLOT AREA** statement works like the **PLOT AREA** statement for each coordinate pair in the array. The following program produces the picture shown below:

```
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
```

MAT PLOT AREA Example



See the section below on “Box Statements and Animation” for additional statements that can quickly draw or fill simple shapes.

Mixing Text and Graphics

Logical windows may contain text as well as graphics. In fact, they are often used exclusively for text. To print text to a logical window, you may use the standard **PRINT** statement or the more flexible **PLOT TEXT** statement.

Output from a **PRINT** statement goes to the current logical window. Each logical window maintains its own text cursor position, margin, and zone width. Thus, the **SET CURSOR**, **SET MARGIN**, and **SET ZONEWIDTH** statements (plus their associated **ASK** statements and the **ASK MAX CURSOR** statement) apply to the current logical window. As you switch between logical windows, subsequent **PRINT** statements in each window will send output to that window's current text cursor position. (See Chapter 3 "Output Statements" for information on these statements.)

When the text cursor reaches the bottom of a logical window, the contents of that window scroll up to make room for a new line, and the topmost line is lost. Text, like graphics, may be clipped at logical window boundaries if the margin is greater than the width of the logical window. Lines that are too wide to fit within the current margin will be wrapped to the next line; since True BASIC sets an appropriate margin for any logical windows you create, text will normally be wrapped at the window boundary. However, if you reset to a wider margin, that part of a text line that extends beyond the window boundary will be clipped.

[!] **Note:** Operating environments with graphical user interfaces generally do not support automatic text scrolling as efficiently as the text-only environments prevalent during much of True BASIC's evolution. Reliance on True BASIC's automatic text scrolling may not produce fully satisfactory results. If you encounter such a situation, you may be able to produce more pleasing results by handling the scrolling of text yourself (see Chapter 14 "Interface Elements") or avoiding it altogether.

Despite its usefulness for many simple tasks, the **PRINT** statement is limited to specific cursor locations within a logical window. Thus, you may prefer the **PLOT TEXT** statement when combining text with graphics.

The **PLOT TEXT** statement is more convenient because it positions text output using the graphical user-coordinate system. For example, the statement:

```
PLOT TEXT, AT x, y: "Sine curve"
```

places the text label "Sine curve" at the coordinate point `x, y`.

PLOT TEXT can print only string values, but you can easily convert numbers into strings using the **STR$** or **USING$** functions (see Chapter 8 "Built-in Functions"). For example:

```
PLOT TEXT, AT 1990, y: Str$(y)
```

or

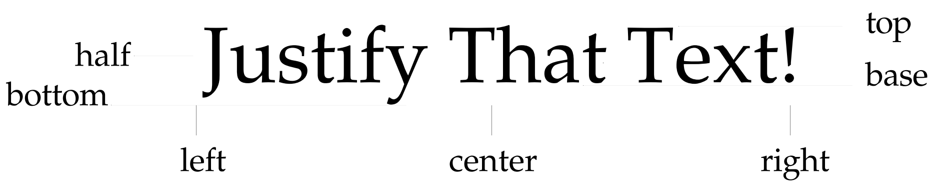
```
PLOT TEXT, AT x, y: Using$("##.##", y)
```

The **PLOT TEXT** statement normally places the lower-left corner of the text at the point defined by `x, y`. However, you can use the **SET TEXT JUSTIFY** statement to control the alignment of the text at the defined point. The general form of the **SET TEXT JUSTIFY** statement is:

```
SET TEXT JUSTIFY horiz$, vert$
```

For `horiz$` you may use one of the values `"LEFT"`, `"CENTER"`, or `"RIGHT"` to indicate a point along the length of the text; for `vert$` you indicate a point in the height of the text as `"TOP"`, `"HALF"`, `"BASE"`, or `"BOTTOM"`. The "bottom" of the text is the lowest point (or descender) of any character, while the "base" of the text refers to its baseline, or the line along the lowest points of uppercase characters.

SET JUSTIFY Values



For example, if you want to center the lowest point of the text at a specified point, you should use:

```
SET TEXT JUSTIFY "center", "bottom"
```

before using the **PLOT TEXT** statement.

The text alignment established by a **SET TEXT JUSTIFY** statement remains in effect for all subsequent **PLOT TEXT** statements until another **SET TEXT JUSTIFY** statement is encountered. The **SET TEXT JUSTIFY** statement controls the alignment of **PLOT TEXT** output only; it has no effect on the alignment of **PRINT** statement output.

The statement:

```
SET TEXT JUSTIFY "left", "base"
```

returns to the default alignment that True BASIC uses, and the statement:

```
ASK TEXT JUSTIFY horiz$, vert$
```

lets your program find the current text alignment.

Consider an example. The following program draws the values of the array `profit` as a bar chart and labels the years. It centers the label at the specified point and uses the **STR$** function to convert numeric values to string. (The **SET COLOR** statement is described below.)

```
SET WINDOW 1975, 1989, -10, 100
SET COLOR "GREEN"
PLOT 1975,0; 1989,0                     ! Axis

SET TEXT JUSTIFY "LEFT", "HALF"         ! Position label
FOR y = 1975 to 1988
    SET COLOR "YELLOW"
    BOX AREA y, y+.5, 0, profit(y)      ! Bar
    SET COLOR "red"
    PLOT TEXT, AT y+.25, 1: Str$(y)     ! Label
NEXT y

END
```

Whether you use text, graphics, or a combination of both, you can clear the contents of a logical window with the **CLEAR** statement:

```
CLEAR
```

The **CLEAR** statement erases the contents of the current logical window, filling it with the current background color and repositioning the window's text cursor in the upper-left corner. The window's margin, zone width, beam state (whether or not a line will be drawn to the next **PLOT** point), and graphics cursor position are not changed.

Using Colors

True BASIC lets you use any color available in your computer's operating environment. At any given time, you may work with two colors — a foreground color and a background color.

The ***foreground color*** is used for objects drawn on the screen including points, lines, and text. By changing the foreground color between plotting or print statements you can produce multi-colored output. The ***background color*** is used behind text produced by the **PRINT** statement and when the window is cleared.

The **SET COLOR** statement establishes the foreground color. There are two forms of this statement; one takes a string and the other takes a number. When used with a string, as in:

```
SET COLOR "RED"
```

the **SET COLOR** statement sets the current foreground color to the named color After the above statement, all drawing and printing will be in red until a new **SET COLOR** statement is executed.

The available color names are:

```
        RED              MAGENTA            YELLOW
       GREEN              BLUE               CYAN
       BROWN              WHITE              BLACK
                        BACKGROUND
```

The value of a string expression used with the **SET COLOR** statement must evaluate to one of these names; otherwise an error occurs. If your computer does not provide all these colors, True BASIC will use another color. For example, red may be used in place of magenta, or vice versa.

Although there are only ten color names, most computers can display many more colors. The second form of the **SET COLOR** statement uses a numeric value to specify the foreground color, as follows:

```
SET COLOR 12
```

The default foreground color is color number -1 (black); the default background color is -2 (white). True BASIC also initially defines color numbers 0 through 15; the rest are set to black. Your computer, however, may likely be able to produce many more colors. The **ASK MAX COLOR** statement will tell you how many colors your computer can simultaneously display. See the next section on "Making Custom Colors" to learn how to define additional color numbers.

You may also specify color numbers as string values. Thus, the following two statements are equivalent:

```
SET COLOR "1"
SET COLOR 1
```

You can find out the current foreground color with the **ASK COLOR** statement. It too has a string form and a numeric form, as follows:

```
ASK COLOR cname$
ASK COLOR cnumber
```

True BASIC assigns the name or number of the current foreground color to the specified variable. If you ask for a string variable and the current color is not one of the official color names, True BASIC assigns a null string to the variable.

The color numbers that correspond to color names vary among computer operating environments. You can use the **ASK COLOR** statement to find out the color number assigned to a particular color name as follows:

```
SET COLOR "RED"
ASK COLOR red
```

These statements first set the current color to `"RED"` and then assign the corresponding color number to the variable `red`.

The **SET BACK** (or **SET BACKGROUND COLOR**) statement establishes the background color. As with the **SET COLOR** statement, it may take a string or numeric value. The same color names and rules for the string expression apply to the **SET BACK** statement as for the **SET COLOR** statement.

The specified background color will be used to surround subsequent printed text and to clear regions of the screen until a new **SET BACK** statement is executed. Any existing background is not affected, however, until it is cleared or printed on.

You can find out the current background color with the **ASK BACK** (or ASK BACKGROUND COLOR) statement, which like **ASK COLOR** has both string and numeric forms:

```
ASK BACK bname$
ASK BACK bnumber
```

If you ask for a string variable and the current color is not one of the official color names, True BASIC assigns a null string to the variable.

The color name `"BACKGROUND"` represents the current background color. For example, the statement:

```
SET COLOR "BACKGROUND"
```

sets the current foreground color to match the current background color. By drawing or redrawing an image in the current background color, you can easily erase or cut a hole in a previously drawn image.

You may simultaneously change current color and background color as follows:

```
SET COLOR "BLUE/WHITE"
```

for drawing or printing in blue on a white background.

To better understand the use of colors, consider the following program that draws blue axes and a red curve on a yellow background:

```
SET WINDOW -1, 10, -3, 3

SET BACK "YELLOW"
CLEAR                          ! Re-paint background with new color
SET COLOR "BLUE"
PLOT 0,0; 10,0                 ! x-axis
PLOT 0,-3; 0,3                 ! y-axis
SET COLOR "RED"
FOR x = .1 to 10 step .1        ! Draw curve
    PLOT x, Log(x);
NEXT x

END
```

The **CLEAR** statement is needed to erase the entire logical window and re-draw it in the new current background color. Without the **CLEAR** statement the axes and curve would be drawn on the default background color.

Making Custom Colors

Your computer can probably display many more colors than the 16 color numbers (0 through 15) initially defined by True BASIC. The **ASK MAX COLOR** statement:

```
ASK MAX COLOR m
```

will assign to `m` the number of colors your computer can display simultaneously. You can use the **SET COLOR MIX** statement to define any available color number.

A computer screen displays colors by directing beams from "color guns" at the phosphor coating on the screen. The nature, intensity, and combination of these beams determine the precise color they produce. There are three such color guns — red, blue, and green — and all can be directed at any single pixel on the screen. By controlling the intensity of the beams from each color gun, you can control the colors displayed on the screen.

The **SET COLOR MIX** statement gives you control of these beam intensities, as follows:

```
SET COLOR MIX (colornum) red, green, blue
```

For `colornum,` you specify a number for the color you want to create. You may choose any number between 0 and the value returned by the **ASK MAX COLOR** statement. Note, however, that a single color number may represent only one color at a time; when you associate it with a new color, any existing color is replaced.

You define the color for `colornum` by specifying the intensity levels of the `red`, `green`, and `blue` color guns. The intensity levels can vary between 0 and 1, where 0 is off and 1 is full intensity. Thus,

```
SET COLOR MIX (14) 0, 0, 0
```

associates pure black with color number 14 (since all the color guns are off), and

```
SET COLOR MIX (13) 1, 1, 1
```

associates pure white with color number 13 (since all the color guns are at full intensity). Likewise, you can use values between 0 and 1 to create different colors:

```
SET COLOR MIX (2) 1, 1/3, 0        ! Color 2 is orange
SET COLOR MIX (5) 0, 0, 1          ! Color 5 is bright blue
```

By varying intensity values, you can create any color your current operating environment can display. If your system cannot display the exact color intensities you specify, True BASIC uses the color closest to the defined mix. Thus, very small changes in the values of `red, green,` and `blue` may not produce different colors.

True BASIC selects a color mix for each legal color number, including (if possible) the nine colors that have names. To find out the current mix for a color number, use the **ASK COLOR MIX** statement:

```
ASK COLOR MIX (colornum) red, green, blue
```

This places the color intensities for color number `colornum` into `red`, `green`, and `blue`.

If you mix your own colors, we advise that you avoid the lower numbers or use color numbers (and not color names) throughout your program. When you use a color name, True BASIC establishes a new color mix for its corresponding color number. Thus, if you have established a custom color and then use a color name that happens to correspond to that same color number, your custom color will be replaced by the color name.

BOX Statements and Animation

You can draw simple shapes quickly and animate your drawings with **BOX** statements. Each **BOX** statement operates on a rectangular region of the screen called its ***bounding rectangle***. This bounding rectangle is specified as four values in user coordinates representing its left, right, bottom, and top edges:

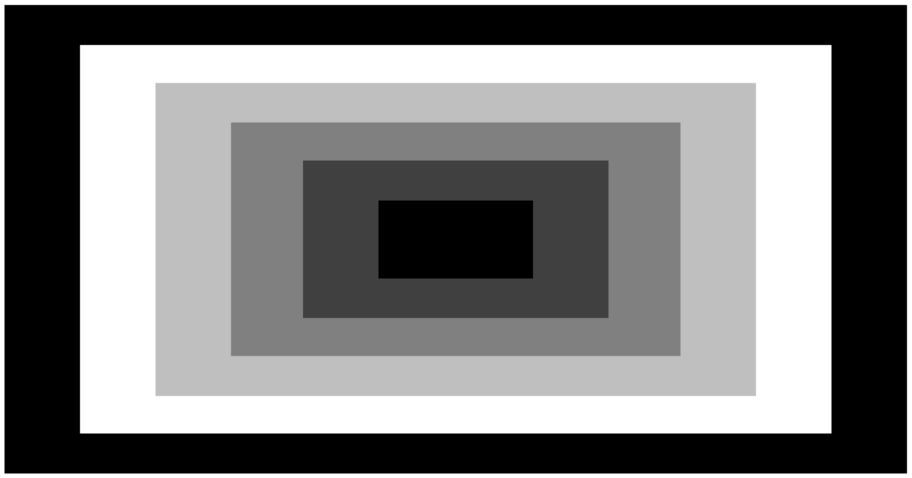
```
BOX LINES left, right, bottom, top   ! Draw rectangle
BOX AREA left, right, bottom, top    ! Draw filled rectangle
BOX CLEAR left, right, bottom, top   ! Erase rectangle
BOX CIRCLE left, right, bottom, top  ! Inscribe an ellipse within rectangle
BOX ELLIPSE left, right, bottom, top ! Inscribe an ellipse within rectangle
BOX DISK left, right, bottom, top    ! Inscribe a filled ellipse within rectangle
```

The **BOX LINES** statement draws the outline of its bounding rectangle in the current foreground color. The **BOX AREA** statement fills its bounding rectangle with the current foreground color. The **BOX CLEAR** statement fills its bounding rectangle with the current background color, effectively erasing its contents. The **BOX CIRCLE** statement (which is identical to **BOX ELLIPSE**) draws the outline of the circle (if the bounding rectangle is a square) or ellipse (if it is not) inscribed within its bounding rectangle. The **BOX DISK** statement fills the circle or ellipse inscribed within its bounding rectangle with the current foreground color.

While many of these **BOX** statements can be reproduced using **PLOT**, **PLOT AREA**, or **FLOOD** statements, the **BOX** statements execute faster and are easier to use. For example, the following program draws six rectangles, each inside the previous one, and each in a different color. If the logical window is square, the result will be six squares.

```
SET WINDOW -6, 6, -6, 6
FOR n = 6 to 1 step -1
    SET COLOR n
    BOX AREA -n, n, -n, n
NEXT n

GET KEY k
END
```

BOX AREA Example



You can use **BOX LINES** to easily "frame" a window:

```
ASK WINDOW left, right, bottom, top     ! Get user coordinates
BOX LINES left, right, bottom, top      ! Draw "frame" around window
```

Note that the same four numbers are used, in the same order. The next series of statements draws a circle (or ellipse) in one color and fills it with a different color:

```
SET COLOR "RED"
BOX CIRCLE 1, 3, 6, 8
SET COLOR "GREEN"
FLOOD 2, 7
```

The **FLOOD** statement uses a point in the middle of the figure to color the area. If you want the outline and the interior to be the same color, the **BOX DISK** statement is faster.

Saving and Showing Screen Images

The **BOX KEEP** and **BOX SHOW** statements let you store and redisplay rectangular regions of the screen. The **BOX KEEP** statement "memorizes" the contents of its bounding rectangle, storing the image in an ***image string***. The **BOX SHOW** statement displays a stored image string (in its original shape and size) at any location in the window. You can produce animation by alternating **BOX SHOW** and **BOX CLEAR** statements to move a drawing or series of drawings across the screen.

For example, suppose that your program has drawn a picture of a dog that you want to display again later in the program. You can use the **BOX KEEP** statement:

```
BOX KEEP 2,4,7,9 IN dog$
```

to save the rectangular area containing the dog picture in the string variable `dog$`. You can then redisplay this image using the **BOX SHOW** statement. The statement:

```
BOX SHOW dog$ AT 5, 8
```

would redisplay the image stored in `dog$` with its lower, left corner at the point (5, 8). The displayed image will be the same size and shape as the rectangular region saved by the **BOX KEEP** statement.

If you combine the **BOX KEEP** and **BOX SHOW** statements with the **BOX CLEAR** statement, you can simulate movement on the screen. As an example, consider the following program, which shoots an arrow across the screen:

```
SET WINDOW 0, 10, 0, 20

PLOT 0,9; 1,9                       ! Draw arrow
PLOT .6,8; 1,9; .6,10
PAUSE 1
BOX KEEP 0, 1, 8, 10 IN arrow$      ! Memorize it

LET x = 0
FOR move = 1 to 50                  ! Move in small steps
    PAUSE 0.1                       ! Slow it down
    BOX CLEAR x, x+1, 8, 10         ! Erase old
    LET x = x + .2
    BOX SHOW arrow$ AT x,8          ! Draw at new position
NEXT move

END
```

You could create more complex animation with several slightly different image strings. For example if you had images of a dog with its legs in different positions, you could save each as a separate image. You could then have the dog walk across the screen by showing and clearing each image in rapid sequence.

You can store **BOX KEEP** images in byte files for use by other programs. For example, you could write the `arrow$` image to a file as follows:

```
OPEN #8: name "arrow.tru", org byte, create newold
ERASE #8                              ! Be sure file is empty
WRITE #8: arrow$
CLOSE #8
```

Another program could then read and display that image as follows:

```
OPEN #4: name "arrow.tru", org byte
ASK #4: FILESIZE fs                   ! Find number of bytes in file
READ #4, BYTES fs: image$             ! Read entire file
CLOSE #4
BOX SHOW image$ AT 0,.5
```

For more information on byte files, see Chapter 12 "Files for Data Input and Output."

BOX SHOW USING Effects

The **BOX SHOW** statement may also take the following extended form:

```
BOX SHOW image$ AT x, y USING option
```

where `option` may be any value from 0 to 15, inclusive. Each value of `option` produces a different result in displaying the designated image. The nature of this result depends both on the contents of the image string being displayed and the current contents of the rectangular region on the screen. These options can produce reverse images and spectacular color effects.

The following table summarizes the 16 available options, which are explained below. The first column shows the option, and the others show the resulting bit-value, depending on the bit in `image$` and the corresponding bit currently displayed on the screen.

Numeric BOX SHOW Options

	Bit in BOX SHOW string:		0	0	1	1
	Bit on screen:		0	1	0	1
	0		0	0	0	0
	1	(AND)	0	0	0	1
	2		0	0	1	0
U	3		0	0	1	1
S	4		0	1	0	0
I	5		0	1	0	1
N	6	(XOR)	0	1	1	0
G	7	(OR)	0	1	1	1
	8		1	0	0	0
C	9		1	0	0	1
O	10		1	0	1	0
D	11		1	0	1	1
E	12		1	1	0	0
	13		1	1	0	1
	14		1	1	1	0
	15		1	1	1	1

Three of these options represent common logical operations and have names. You can use `AND` instead of 1, `OR` instead of 7, and `XOR` instead of 6.

The interpretation is simplest if only one color is available and therefore the "color" of a pixel is represented by one bit, 0 or 1, off or on. Under the `AND` option, a bit is 1 if it is 1 in `image$` and also 1 on the screen. The common part of the two images is displayed. Under the `OR` option a bit is 1 if either bit was 1, therefore a combination of the two images is displayed. The `XOR` (exclusive or) option equals 1 if `image$` or the screen had a 1 in this position, but not both; it produces a combination of the non-overlapping regions of the images.

`USING 3` is equivalent to a **BOX SHOW** statement without any option (i.e. the **BOX SHOW** string bit takes precedence over what was on the screen), while `USING 0` is the same as **BOX CLEAR**. `USING 12` ignores the screen but displays a ***reverse image*** of the show string. For example, if it was black-on-white, it is shown white-on-black.

If more than one color is available, the color value of each pixel is coded by more than one bit. The options still apply, combining corresponding bits of the color codes. Working out the effect of each option is trickier. For example, if there are 16 colors, a four-bit code is used. Say that `image$` has color five for one pixel (coded 0101) while the screen has color six (0110). Then the `AND` option produces color four (0100), the `OR` option produces color seven (0111), the `XOR` option produces color three (0011), and option 12 produces color ten (1010). Some experimentation with the options is recommended when working with colors.

Using Pictures

True BASIC provides special subroutines for graphics, called ***pictures***. For instance, if you want to create several hexagons, you could define a picture and simply draw that picture when needed rather than repeat the **PLOT** statements each time. Pictures are more flexible than either subroutines or **BOX KEEP** images because you can transform them geometrically. When you draw a picture you can rotate it, change its scale, tilt it, or move it anywhere on the screen.

PICTURE structures are defined just like subroutines, except that they begin with a **PICTURE** keyword and end with an **END PICTURE** statement. A picture has a name, may have parameters, may be internal or external, and may be placed in a library file.

A simple example is a picture that draws axes for the current logical window:

```
PICTURE Axes
    ASK WINDOW left, right, bottom, top    ! Find user-coordinate range
    PLOT left,0; right,0                   ! x-axis
    PLOT 0,bottom; 0,top                   ! y-axis
END PICTURE
```

You invoke a picture with a **DRAW** statement:

```
DRAW Axes
```

The **EXIT PICTURE** statement corresponds to the **EXIT SUB** statement and immediately returns control to the line following the **DRAW** statement.

Pictures may include most valid True BASIC statements, including **DRAW** statements to invoke other pictures. However, a picture cannot open a logical window or set user coordinates — this must be done in the invoking program. Pictures can use **WINDOW** statements to switch to existing windows, and you can pass channel numbers to pictures as parameters.

Here's a picture that uses a parameter to draw a regular polygon with `s` sides inscribed in a unit circle. The polygon is centered about the point (0,0).

```
PICTURE Polygon(s)                 ! Polygon of s sides
    OPTION ANGLE DEGREES           ! Use degrees instead of radians
    FOR i = 0 to s                 ! Run through vertices
        LET a = 360*i/s            ! Angle
        PLOT Cos(a), Sin(a);
    NEXT i
    PLOT
END PICTURE
```

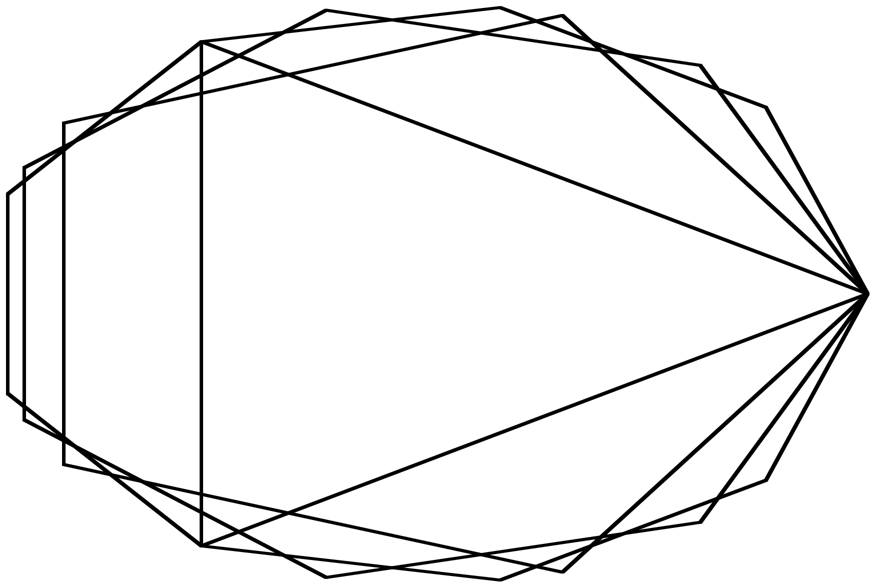
The parameter `s` indicates the number of sides. If the above picture is in a library file, the main program could use:

```
SET WINDOW -1, 1, -1, 1
```

```
FOR n = 3 to 9 step 2
    SET COLOR n
    DRAW Polygon(n)
NEXT n
```

to draw odd-sided polygons with three to nine sides, each in a different color.

Output of the Ploygon Picture



Transformations

The important difference between subroutines and pictures is that you can transform pictures when you invoke them. ***Transformations*** are options applied to the picture to change its appearance. True BASIC's built-in transformations let you resize, move, tilt, and rotate pictures.

For example, the statement:

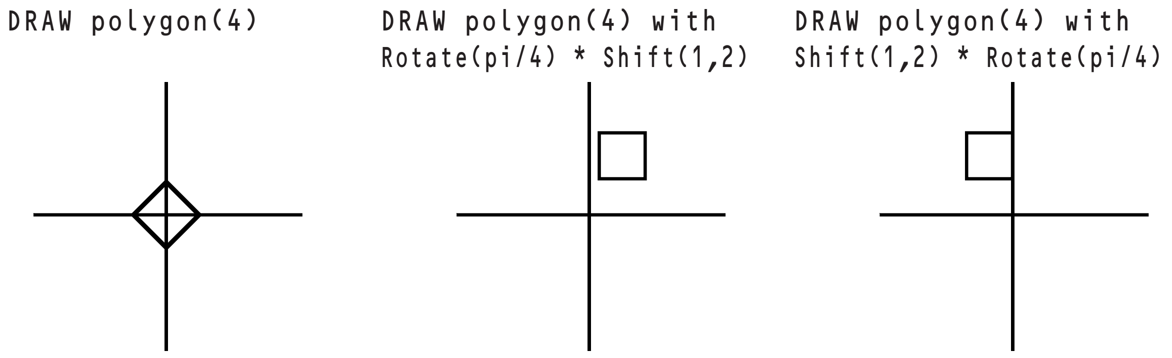
```
DRAW Polygon(4) with Rotate(pi/4) * Shift(1,2)
```

uses the polygon picture defined above to draw a rotated square (or rectangle) that is centered about the point (1,2). First the **DRAW** statement constructs a square centered on the origin as defined by the picture. Before it draws the image, however, it applies the transformations to rotate the square counterclockwise by an angle of Pi/4 (45 degrees) around the origin and then shift the rotated square one unit horizontally and two units vertically, so that its center is at (1, 2).

[!] Note: While the external picture uses degrees, its **OPTION ANGLE** statement has no effect on the main program. Unless otherwise specified, the main program measures in radians. See Chapter 8 "Built-in Functions" for information on specifying degrees or radians with the **OPTION ANGLE** statement.

Multiple transformations — which must be separated by an asterisk — are applied from left to right. Thus, the square above is rotated before it is shifted. Pictures are always rotated about the origin (0,0) which happens to be the square's center. Thus, the square remains centered on the origin until it is shifted. Had the square been shifted first, the result would have been different. The shift would move the center of the square to (1, 2). Then, rotation about the origin would move the square away from its shifted position (as if it were riding on the hand of a clock moving backwards).

Effects of Multiple Transformations



True BASIC defines four transformations that you can use with pictures:

Picture Transformations

Transformation	**Effect**
SHIFT(a,b)	Move by `a` horizontally and by `b` vertically
SCALE(a,b)	Change scale, multiplying horizontal coordinates by `a`, and vertical coordinates by `b`; if `a = b`, then you may specify `SCALE(a)`
ROTATE(t)	Rotate around origin counterclockwise by an angle `t`
SHEAR(t)	Lean vertical lines forward (clockwise) by an angle `t`

In the SHIFT and SCALE transformations, all calculations use user coordinates. In the ROTATE and SHEAR transformations, angles are normally measured in radians, but you may change that with an **OPTION ANGLE** statement.

[!] **Note:** Transformations are applied only to the various forms of the **PLOT** statement within a picture; they are not applied to **BOX** statements.

As mentioned above, a picture may include **DRAW** statements to call other pictures. For example, a picture that draws a house may repeatedly call a picture that draws a window applying different transformations each time to place different-sized windows in various locations. The main program, in turn, might apply a transformation to the house picture. Any transformation applied to the house picture will also affect the window pictures it invokes, maintaining the integrity of the house as a whole. For example you could build a neighborhood of houses by scaling and shifting the houses. For an illustration of this, see the HOUSES.TRU program in the TBDEMOS directory installed with True BASIC.

Constructing Your Own Transformations

This section gives technical details on how transformations work and shows how you can construct additional transformations.

Transformations may be represented by a four-by-four matrix. For example:

$$\texttt{Shift (3,5)} \quad = \quad \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 3 & 5 & 0 & 1 \end{pmatrix}$$

When this transformation is applied to a picture, each plotted point (x, y) is represented by the four-element vector (x, y, 0, 1), which is multiplied on the right by the transformation matrix. The shift yields the result (x + 3, y + 5, 0, 1) and the point is changed into (x + 3, y + 5). With repeated transformations, several multiplications are carried out. Thus, the asterisk separating transformations actually represents matrix multiplication.

True BASIC will accept any four-by-four matrix as a transformation. Thus, you may define your own transformation as a four-by-four matrix and use this matrix as a transformation. For example, you may introduce a reflection around a 45 degree line:

```
DIM Reflect(4,4)                    ! Reflection transformation
MAT READ Reflect
DATA 0, 1, 0, 0
DATA 1, 0, 0, 0
DATA 0, 0, 1, 0
DATA 0, 0, 0, 1

DRAW Polygon(5) with Reflect        ! Pentagon reflected
```

The third and fourth components of (x, y, 0, 1) are currently not used.

Graphical Input

True BASIC provides two simple methods for obtaining graphical input — the **GET MOUSE** and **GET POINT** statements.

The **GET MOUSE** statement returns the current position of the mouse pointer and the state of the leftmost mouse button. The **GET POINT** statement, on the other hand, pauses the program and waits for the user to press the left mouse button; it then returns the position at which the click occurred. Both statements return the position of the mouse pointer in the user coordinates of the current logical window. The mouse pointer must be within the current physical window, but it need not be in the current logical window. Points outside the logical window are returned in appropriate user coordinates as if the coordinate range were extended beyond the window. As usual, however, any lines or points drawn to coordinates outside the user-coordinate range will be clipped at the window edge and not shown.

The following program uses the **GET POINT** statement to draw a figure connecting points selected by the user:

```
DO
   GET POINT x,y
   PLOT x,y;
   IF key input then GET KEY k
LOOP UNTIL k = 27                   ! Use escape key to exit
END
```

There is no special prompt to indicate the program is waiting for **GET POINT** input. You may therefore wish to print instructions to the user before using **GET POINT**.

The **GET MOUSE** statement requires three numeric variables:

```
GET MOUSE x, y, s
```

The current position of the mouse pointer is returned in `x` and `y` and the status of the left mouse button in `s`. The possible values for `s` are:

0	No button down
1	Button is down
2	Button clicked at this point
3	Button released at this point
4	Button shift-clicked at this point

Thus, the above example could use the **GET MOUSE** statement as follows:

```
DO
   GET MOUSE x, y, s
   IF s = 2 then PLOT x,y;
   IF key input then GET KEY k
LOOP UNTIL k = 27                  ! Use escape key to exit
END
```

Note, that omitting the **IF** test before the **PLOT** statement produces a program that draws a curve following every movement of your mouse (as if `s` = 0).

[!] **Note:** The TC_Event routine, described in Chapter 14 "Interface Elements," provides more sophisticated mouse-handling capabilities. The **GET MOUSE** and **GET POINT** statements are primarily for compatibility with earlier versions of True BASIC, and for simpler programs.

Drawing Charts and Graphs

There are certain graphing tasks that many programmers face fairly often. For some of these common tasks, True BASIC includes subroutines that simplify the display of a wide array of charts and graphs.

These routines are not built into the language, but rather are stored in separate library files (see Chapter 11 "Libraries and Modules"). Thus, you must name the library file in a **LIBRARY** statement before you can invoke the subroutines.

The charting and graphing routines are in the library files:

BGLib.TRC	for pie charts, bar charts, and histograms
SGLib.TRC	for plotting data and function values
SGFunc.TRC	for plotting values of functions that you define

which are stored in the TBLIBS subdirectory when you install True BASIC. Remember that the **LIBRARY** statements must use the appropriate "path name" to indicate the location of the library files for the computer you will use to run your program; see "The True BASIC Environment" chapter in the introductory section for information on the correct formats.

See Chapter 23 "Additional Library Routines" for descriptions of the subroutines in these libraries.

CHAPTER

14
Interface Elements

Graphical user interfaces, such as Windows, OS/2, and Macintosh, use menus, buttons, windows, and dialog boxes to make applications easier to use. With True BASIC, you can create and control such objects in your own programs. The easiest way to do this is with the True Controls and True Dials libraries of subroutines provided with the language.

True Controls is a library of subroutines that let you manage windows, menus, graphical objects, buttons, edit areas, and other interface elements from your True BASIC programs. True Dials is a similar library of routines that let you create dialog boxes as part of your programs. This chapter explains how you can create user-interface elements using the routines from these two libraries.

The routines described in this chapter are relatively easy to use, allow control of all user-interface elements, and, in most instances, will meet all of your needs. (Advanced users might want to gain further control of interface elements through direct use of the built-in subroutines **OBJECT** and **TBD**. All of the user-interface "objects" are ultimately controlled by these two extremely powerful and complex subroutines. The **OBJECT** subroutine is used by all the True Controls routines, while True Dials routines call the **TBD** routine to create and display all dialog boxes and return the user's response. Advanced programmers who wish to use the **OBJECT** and **TBD** routines directly should refer to the reference sections (see Chapters 19 and 21). Examining the source code of TRUECTRL.TRU in the TBLIBS folder should help.)

The example programs ARCHERY2.TRU, DAYCALC.TRU, PISTON.TRU, and SURVIVE.TRU (in the TBDEMOS directory) provide a good overview of how the convenience routines described in this chapter may be used to enhance programs. The TBDEMOS directory also contains several programs that illustrate the individual objects; those programs have names that begin with DEM.

User Interface Objects and Controls

The True Controls routines let you create and control the following objects or controls:

Window	Physical window
Menu	Pop-down selections for a window's menu bar
Push button	Push button with text
Radio group	Several radio buttons, only one of which can be on, with text
Check box	Text with check box that may be checked on or off
Group box	Box with or without a title
Static text	One-line piece of text that cannot be edited
Edit field	One-line text input region that can be edited
List box	A (scrollable) selection list of items
List button	Button with a pop-down selection list
List edit button	Edit field with a pop-down selection list
Graphics	Several types of graphics objects
Scroll bar	Horizontal or vertical scroll bar
Text editor	Fully-scrollable text-edit object

Some of the items, such as the graphics and windows, are ***objects***, while others, such as check boxes, buttons, text editors, and radio groups, are ***controls***. For convenience, we may refer to both types as objects, since there is no distinction in how you use the True Controls routines to handle objects or controls.

Dialog boxes are the only control objects not handled by True Controls routines. They are handled by routines in the True Dials library as described later in this chapter. Dialog-box routines are in a separate library because they return information to the program differently than the objects handled by the True Controls routines. Otherwise, dialog boxes are similar to other interface objects.

Program Structure with Interface Elements

By programming with the interface-element routines you can greatly enhance the appearance and "user-friendliness" of your programs. Along with these enhancements, however, you as the programmer must accept the responsibility for processing all the events that may occur as the user responds to the controls and objects in your program. To do this correctly, you may need to re-think the structure of your programs. The two versions of the Archery program (ARCHERY.TRU and ARCHERY2.TRU in the TBDEMOS directory) illustrate the difference in approach needed for work with interface elements.

In programs that don't use interface routines, input statements may typically be scattered about various program structures. In the original ARCHERY.TRU program, the main loop asks if the player wishes another game and responds appropriately, while the MakeShot subroutine gets the user input for angle and velocity. In both instances, the program pauses until there is a user response. Thus, the program could "spend a lot of time" in the MakeShot subroutine. This works fine for that version of the program, because the user has no opportunity for any other input until a shot is completed and control returns to the main loop. The limited order of events is clearly defined in the main loop, as follows:

```
DO
   CALL MakeScene     ! Draws the scene
   CALL PlayGame      ! Calls MakeShot to get velocity and angle input
                      ! and draw resulting shots until success
   PRINT "Play another (y or n)?";
   GET KEY key
   PRINT
LOOP while key = Ord("y") or key = Ord("Y")
```

When you program with interface controls and objects, however, you present the user with a greater possibility of responses and your program must be ready to process those responses or "events" as soon as possible. Such programs generally work best with a main "event processor" loop to handle all potential user input. The main event loop may call subroutines to carry out an appropriate action, but those subroutines should not themselves process input. Thus, control can return immediately to the main event loop and prevent events from piling up.

In ARCHERY2.TRU, control objects let the user input velocity or angle, fire a shot, call for a new game scene, or quit the game. These objects are always available; the user need not use them in any particular order. To prevent the program from waiting in one subroutine, say for a new velocity and angle, when the user may be indicating a different action, such as firing a shot or getting a new game scene, all event handling is placed in the main loop. The subroutines carry out the same actions as before, but they do so with input obtained in the event loop, as follows:

```
DO                                  ! Main event loop
    LET x1, x2 = 0
    CALL TC_Event (0, event$, window, x1, x2)  ! Get next event
    IF event$ = "KEYPRESS" and x1 = 27 then EXIT DO   ! Escape will stop program
    IF x2 = quit then               ! Quit button pressed
       EXIT DO
    ELSE IF x2 = fire and event$ = "CONTROL DESELECTED" then
       CALL MakeShot        ! Fire shot with current angle & velocity
    ELSE IF x2 = newgame and event$ = "CONTROL DESELECTED" then
       CALL MakeScene               ! Draw new scene
    END IF
```

```
      CALL TC_Sbar_GetPosition (angleset, angle)  ! Get current angle setting
      LET angle = 90 - angle
      IF angle <> currentangle then
         LET currentangle = angle
         CALL TC_SetText (angledial, Str$(currentangle))
      END IF

      CALL TC_SBar_GetPosition (speedset, speed)  ! Get current velocity setting
      IF speed <> currentspeed then
         LET currentspeed = speed
         CALL TC_SetText (speeddial, Str$(currentspeed))
      END IF
   LOOP
```

This loop continually looks at the latest user event via the **TC_Event** routine and carries out the appropriate action, if any. Control always returns immediately to this main loop. The loop also updates the values for the current angle and velocity variables; MakeShot uses those same variables. The TC_ subroutines used above are all explained in this chapter.

For additional examples of event processors, examine the DAYCALC.TRU, MATHQUIZ.TRU, PISTON.TRU, and SURVIVE.TRU programs in the TBDEMOS directory. These programs all use True Controls routines, illustrating variations in how a program may be structured to handle events.

Routines Common to all True Controls

Using the True Controls Library

Unlike the underlying and complex **OBJECT** routine, the True Controls routines are not built-in to the True BASIC language. Instead, they are contained in a library called TRUECTRL.TRC. To use any of the routines you must identify the library at the beginning of your program. For example, with OS/2 and Windows, the statement:

```
   LIBRARY "c:\TBSilver\TBLibs\TrueCtrl.trc"       ! Or appropriate path name
```

uses the compiled library. With the Macintosh, you would use something like the following (substituting the appropriate disk and folder names):

```
   LIBRARY "harddisk:TrueBasicSilver:TBLibs:TrueCtrl.trc"
```

[!] **Note:** The language includes both the compiled (.TRC) and source-code versions (.TRU) of the TrueCtrl library so that you may examine the routines to learn more about how they work and how they use the built-in OBJECT routine. However, your programs will start faster if you always use the compiled version of any library routines.

You must also call an initiating routine before you respond to events in an event loop:

```
   CALL TC_Init
```

This routine tells True BASIC that your program will be handling the “events” that occur when a user clicks on a push button, makes a menu selection, or otherwise uses one of the objects you create.

When you are finished using the True Controls routines, you should call the “close-down” routine, as follows:

```
   CALL TC_Cleanup
```

This routine ensures that True BASIC will resume handling any “events” that may occur rather than passing them to your program.

For easy identification, all of the True Controls routines have names that begin with “TC_”. Names of general purpose routines give the “method” or function of the routine after the “TC_”, such as **TC_Init** and **TC_Cleanup**, above, or **TC_Show** and **TC_Event**, described in the next sections. Names of routines specific to a particular object first identify the object and then the function, such as **TC_Win_Create**, **TC_Menu_Set**, **TC_Menu_SetCheck**, **TC_PushBtn_Create**, etc.

The True Controls routines share many public variables, which are all initialized the first time any one of the routines uses any one of the variables. You should therefore be careful not to assign spurious values to those public variables. For the names of the public variables used by the True Controls routines, see the module CONSTANTS located at the end of the file TRUECTRL.TRU (in the directory TBLIBS).

Creating Objects

To create an object and display it on the screen, you generally follow three steps. Each of these steps is explained more fully as the individual objects are described later in the chapter.

1. ***Create the object.*** Separate routines are available for each object type; these are described in the appropriate sections below. True BASIC assigns an ID number to each object that you create; you then use that ID number to identify an object in other True Controls routines. Windows are assigned IDs from 1 to 99 (the default physical window is always numbered 0), menus have IDs from 10001 to 14999, groups (such as radio button groups) have IDs from 15001 and up, while all other controls and objects are assigned IDs from 101 to 9999. All objects and controls, except for windows, are always placed in a physical window, so you must first create a physical window or use the default physical window.

2. ***Specify additional attributes*** of the object, if necessary. There are specific routines for many of the objects. For example, for a select list box you need to supply the list of names using **TC_SetList**:

   ```
   CALL TC_SetList (slbid, list$())
   ```

3. ***Show the object.*** A window is not shown automatically; you must use the **TC_Show** routine to display it:

   ```
   CALL TC_Show (id)
   ```

   All other objects are automatically shown when they are created, as long as the window that contains them is visible. If you are creating several objects within a window, you may wish to create the objects before you explicitly show the window so that all objects are revealed at once.

   You can also change the default so that, even if the containing window is shown, an object is not shown until you specifically use **TC_Show** for it. To change the default, use:

   ```
   CALL TC_Show_Default (flag)
   ```

   If `flag` is set to 0, no new objects or controls are shown until you use **TC_Show** for the specific object. If `flag` has the value 1 (or any non-zero value), all new objects and controls are shown when created as long as the containing window is visible (this is the default setting for `flag`).

   (You can hide an object or a window and all the objects it contains at any time with the **TC_Erase** routine, as explained below in "Erasing and Showing Objects." Note that an erased object or control still exists and may be shown again later.)

Handling Events — Getting Input from Objects

After you've created control objects, such as menus, buttons, and check boxes, you must also make them do something — your program must be able to get appropriate input from the objects. The **TC_Event** routine gets that input for your program.

Each time a user makes some response to your program, such as selecting a menu item, clicking on a push button, or pressing a key, True BASIC stores that action in an ***event queue***. Each action is added to the queue in the order in which it occurs. A call to the **TC_Event** routine from a program returns the first event in the event queue; any additional events stay in the queue until subsequent calls to **TC_Event**.

A call to **TC_Event** requires five arguments:

```
CALL TC_Event (timer, event$, window, x1, x2)
```

The first argument, `timer`, indicates how long the routine should wait for an event. If there are any events in the event queue, the routine always takes the first event and returns immediately. Otherwise, the routine waits for an event up to the number of seconds specified by `timer` before returning. If there are no events in the queue and nothing happens during the `timer` interval (the user has made no response), the routine returns an empty string to `event$`.

If an event has taken place, the remaining three arguments return information about the event. `Window` returns the physical window ID, and `x1` and `x2` return values specific to the event type. These items are described in the sections for the individual controls and are summarized in a table in the "True Controls Events Summary" section later in this chapter.

The recommended way to handle events is within a loop that continually calls **TC_Event** and includes a decision structure to carry out the appropriate action. The **TC_Event** `timer` would be set to 0, since the loop will continually check the queue until an event takes place.

The **TC_Event** routine also performs the following functions in response to certain events:

- It provides automatic handling for scroll bars either connected to text edit controls or if the event returned is an action in the scroll bar of a text edit object, the routine carries out the appropriate action on the screen.
- It toggles check boxes and radio buttons.
- It converts x1 and x2 into menu and item numbers for menu events.

[!] **Note:** After your program calls **TC_Init** so that it may use True Controls routines, the program must use the **TC_Event** routine to handle all user actions until you "turn off" event handling with a call to **TC_Cleanup**. If the user closes or hides a window by clicking in the window's close box (upper left corner), the program must handle the `"HIDE"` event — perhaps by terminating the program then or later re-showing the window so the user may terminate the program some other way. Take care to provide adequate "escape routes" when you program with True Controls routines.

Erasing and Showing Objects

You can ***erase*** or hide any True Controls object, with the **TC_Erase** routine:

```
CALL TC_Erase (wid)
```

This makes the object "invisible"; it cannot be seen on the desktop even if there are no overlapping windows. When a window is invisible, any controls it contains are also invisible. You must use the **TC_Show** routine to make the window, and any objects it contains, visible.

Individual controls may also be erased (hidden) and shown with the **TC_Erase** and **TC_Show** routines. However, if the window is invisible, that status overrides the visible status of any control contained within it. Thus, you can erase (hide) controls within a visible window, but you cannot show controls in an erased (hidden) window.

Note that erasing any True Controls object does not destroy the object; it is merely hidden from view. An erased window still exists, output may be sent to it, and it may be shown again. When you are done with an object (except a menu) or with a window and all its associated objects, you should permanently remove it with the **TC_Free** routine described in the next section on "Removing and Freeing Objects."

Removing and Freeing Objects

When you are finished using an object or a window and the objects contained within it, you can destroy that object or window with the **TC_Free** routine:

```
CALL TC_Free (id)
```

When this routine is used with a window ID, it frees the window and all objects and controls associated with it. If **TC_Free** is used with the ID for another type of object, other than a menu, it frees the object but not the containing window. If you wish to free a menu, use **TC_Menu_Free** as described in the sections on "Creating and Using Window Menus."

In “freeing” an object, the routine first hides the object and then frees the memory associated with it. Freeing a window automatically frees its menu and controls contained in it. Once an object has been freed it no longer exists and it cannot be shown or manipulated.

[!] Note: TC_Erase simply hides an object but does not destroy it; an erased object may be used and later shown again with **TC_Show**. **TC_Free** completely destroys an object so that it no longer exists.

Physical Windows and Coordinate Systems

As introduced in Chapter 13 “Graphics,” True BASIC uses two kinds of windows: (1) ***physical windows***, which are typical of your computer’s operating system and usually have a visible border that may include title bars, menus, and scroll bars, and (2) ***logical windows***, which are invisible partitions in a physical window providing a framework for user-coordinate systems and a clipping region for graphical output.

Physical windows are one of the user-interface elements you can create and control with True Controls routines. All of the other True Controls objects are placed within a specific physical window. You may use the default physical window, which always has the window ID of 0 (zero), or you may create additional physical windows using the **TC_Win_Create** routine as described later in this chapter.

In using True Controls routines, you must keep in mind the distinctions between physical windows and logical windows. These may be summarized as follows:

- The position of physical windows is defined by ***screen coordinates***. referring to the full screen. Other True Controls objects are placed within physical windows according to the current user coordinates, as from SET WINDOW or ASK WINDOW. Logical windows are placed within physical windows with ***screen coordinates*** as described in Chapter 13; positions within logical windows are always defined by ***user coordinates***.
- True Controls routines follow the usual True BASIC order of specifying coordinates (that is, `left`, `right`, `bottom`, `top`).
- True Controls objects are placed in the current ***target*** physical window, which is the physical window designated to receive output from **PRINT** statements, etc. To switch output to another physical window, you must use the routine **TC_Win_Target** or **TC_Win_Switch** (which also moves the window to the front). Regular True BASIC statements such as **PRINT** and the graphics statements (**PLOT LINES**, **BOX AREA**, etc.) are placed in the current logical window. The current logical window is either the default logical window that fills the current target physical window or a defined logical window you have opened (with an **OPEN** statement) and specified with a **WINDOW** statement. For more details about switching between physical and logical windows, see the section below on “Creating and Using Physical Windows.”

You place most True Controls objects on the screen or within a window by specifying a set of rectangular coordinates with the **CALL** to the appropriate routine. These coordinates have the order `left`, `right`, `bottom`, `top`, as in regular True BASIC statements. In the calling sequences described in this chapter, they are designated:

```
xl, xr, yb, yt
```

[!] Note: The True Controls routine that creates a physical window always creates a logical window that fills that physical window, just as there is always a default logical window for the default physical window. The default user coordinates of this logical window are 0, 1, 0, 1 as in the default logical window.

[!] **Note:** It is possible to place windows on the screen and objects and controls within windows using pixels coordinates. This approach is not discussed in this chapter.

Creating and Using Physical Windows

All True Controls objects, except windows, must be placed in a specific physical window, so you must first create and define a physical window or use the default physical window (window ID 0).

Even if you intend to use only the default physical window for object creation, you must make it visible on the screen with a call to **TC_Show** as follows:

```
CALL TC_Show (0)
```

If you do not explicitly show the default window, it will be shown automatically with the first **PRINT** or **PLOT** statement (**CLEAR** does not show the window), but it is *not* shown automatically when a control object is created within it. You may wish to create objects within the window before you show the window (and the objects it contains), but you must explicitly show the window at some point.

To create a new physical window, use the **TC_Win_Create** routine in the format:

```
CALL TC_Win_Create (wid, options$, xl, xr, yb, yt)
```

For this routine, `wid` must be a numeric variable; True BASIC will assign the window object's ID to that variable.

`Options$` is a string variable or expression for setting certain aspects of the window. Separate multiple values in `options$` by spaces or vertical bars "|". If `options$` contains the word `"TITLE"` (case does not matter), the window will be created with a title bar. If `options$` contains the word `"CLOSE"`, the window will include a close box; if it contains the word `"SIZE"`, the window will have a resize box. To create a window with a vertical scroll bar, include `"VSCROLL"` in `options$;` for a horizontal scroll bar, use `"HSCROLL"`. For information on controlling the action of scroll bars attached to windows, see the section "Creating and Using Scroll Bars" later in this chapter.

For additional details on the options available for window creation, see the description of TC_Win_Create in Chapter 22 "Interface Library Routines." The options include: different border types, parent versus child windows, and immune versus nonimmune windows.

As noted earlier, `xl`, `xr`, `yb`, `yt` must be numeric expressions giving the left, right, bottom, and top locations of the window on the full screen in screen coordinates. Screen coordinates are always between 0 and 1, and left < right and bottom < top. True Controls will adjust screen coordinates that are out-of-range, and will also make sure that all portions of the window are visible. (Use pixel coordinates if you have other requirements.)

You can set or change a title to a window with the **TC_Win_SetTitle** routine:

```
CALL TC_Win_SetTitle (wid, title$)
```

For example, if you wish to create a physical window that nearly fills the full screen and contains a close box, a resize box, and a title, you could do the following:

```
LIBRARY "c:\TBSilver\TBLIBS\TrueCtrl.trc" ! Use appropriate path
CALL TC_Init                              ! Initialize

CALL TC_GetScreenSize (ls, rs, bs, ts)
CALL TC_Win_Create (wid, "close size title", .1, .9, .1, .9)
CALL TC_Win_SetTitle (wid, "New Window")

CALL TC_Show (wid)                        ! Display the window
```

The coordinates in the **TC_Win_Create** routine define the ***user-accessible area*** for the window; borders, title bars, and menu bars (if any) are placed outside that area. The user-accessible area is sometimes called the ***client area***. Thus, all the elements of a window created with full-screen coordinates would not be visible. 0.1 should be ample to allow room for all elements, but you may wish to experiment if you want to use the full screen. (Note: if you do not add menus to your window, on certain platforms that space will be added to the client area.)

When you use **TC_Win_Create** to create a new physical window, the routine automatically opens a logical window to fill the content area of that physical window. The logical window is given the default user coordinates of 0,1,0,1. (This is similar to the default logical window, channel #0, that fills the default physical window and has user coordinates 0,1,0,1.)

Creating (and showing) a physical window makes it the active, target window. The ***active physical window*** is always in front of any others on the screen, and its title bar will appear differently than those of other windows; the active physical window will never be partially or completely hidden from view. The ***target physical window*** is where subsequent output and other objects will be placed; it is not necessarily active or even visible. (Creating but not showing a physical window makes it the target window, but does not make the window active because it is not visible.)

Shown, Active, and Target Windows

Physical windows may be ***shown*** (visible) or they may be ***erased*** (invisible or hidden). When one or more physical windows are shown on the screen, one window is the ***active physical window***. If the windows are overlapped, the active window is always on top; it is never partially or completely hidden from view. The active window also has a title bar with a different appearance than those of other windows; thus even if windows are tiled (more than one window is "in front" and not overlapping any others), the active window is easy to identify. (It may occasionally happen that being "in front" and having an "active" title bar will not occur together.) On the Macintosh, it is necessary that a window be active for its menu to appear in the menu bar position at the top of the screen.

The ***target physical window*** is the one that contains the logical window to which subsequent output will be sent. Although it may be the same as the active physical window, it doesn't need to be active or even visible. In fact, a common technique is for a program to fill a hidden window with output and then make it visible.

As noted above, windows must be explicitly ***shown*** (made visible) after they are created with the **TC_Show** routine:

```
   CALL TC_Show (wid)
```

As with all True Controls objects, visible windows may be ***erased*** (hidden) with the **TC_Erase** routine:

```
   CALL TC_Erase (wid)
```

Erased windows may later be shown again with **TC_Show**. As with any True Controls object, erasing a window merely removes it from view; the window still exists, it may be made the target window, and it may later be shown and made active.

Many physical windows may be shown simultaneously (though some may be fully or partially hidden behind other windows), but at any one time there can be only one active window and only one target window. A program can switch among physical windows with the following routines:

```
   CALL TC_Win_Target (wid)       ! To receive output
   CALL TC_Win_Active (wid)       ! Move to the front, if visible
   CALL TC_Win_Switch (wid)       ! Makes target and, if visible, active
```

Calling **TC_Win_Switch** makes the identified window both the target window and, assuming the window is visible, the active window. **TC_Win_Target** and **TC_Win_Switch** also automatically issue a **WINDOW** statement so that subsequent program output is sent to the logical window that fills the target physical window.

Only a shown window may be made active. If a window has not yet been shown after it is created or if it has been erased (hidden), **TC_Win_Active** has no effect and **TC_Win_Switch** makes the window the target but does not show it or make it active. **TC_Show** merely makes a window visible and capable of becoming active; it does not make a window active or the target. Any window, whether shown or erased, may be made the target window.

A shown window may be made active by several means:

- When a program first creates and shows a window with the **TC_Show** routine, that window becomes active.
- The **TC_Win_Switch** routine makes the designated window both the active and target window, as noted above.
- The **TC_Win_Active** routine designates a new active window, but does not change the target window, as noted above.
- If a **TC_Erase** routine erases the active window, another visible window becomes the new active window.
- Using the mouse, the user may click in any visible window to make it active. Similarly, if the user moves or resizes a window on the screen (see below), that window becomes the active window.

A window can become the target window for output as follows:

- The most recently created window is the target (only one window may be a target at any one time).
- The **TC_Win_Switch** routine makes the designated window the target as well as the active window, if visible, as noted above.
- The **TC_Win_Target** routine specifies a new target window as noted above, although it does not make that window active or even visible.
- A **WINDOW** statement, which directs output to a specific logical window, also selects the appropriate physical window as the target for subsequent output, but that physical window does not automatically become active or shown if it is currently erased (hidden). (Generally, if you want to redirect the program's output and insure that it is shown, you should first call the **TC_Show** and **TC_Win_Switch** routines to make the appropriate physical window the active target. Then, if you want a logical window within that physical window other than the default logical window, you can use a **WINDOW** statement.)

Note that the target window is controlled exclusively by the program — there is no way the user can directly change the target window — whereas the active window may be changed either by the program or by the user clicking in it.

[!] Note: In the default physical window, the default logical window is always available as channel #0; you can switch among that and other logical windows with the **WINDOW** statement. That is not the case for additional physical windows you create. Although **TC_Win_Create** automatically creates a logical window to fill any new physical window, that logical window channel is not known outside of True Controls; thus you cannot use a **WINDOW** statement to direct output to such windows. **TC_Win_Switch** does that for you, sending subsequent output to the logical window that fills the physical window you designate, and that may be adequate in most cases.

Placing Objects Within Physical Windows

To place another object in a physical window, you must specify the location of that object. This is normally done in the user coordinates of the current logical window contained in the physical window.

To find the user coordinates of a logical window, use:

```
WINDOW #3
ASK WINDOW left, right, bottom, top
```

To find the user coordinates of an unnumbered logical window that fills a certain physical window, or the logical window that fills the default physical window, use

```
CALL TC_Win_Switch (the_physical_window)
ASK WINDOW left, right, bottom, top
```

(Examples of placing objects in windows are shown in the sections on the various objects below; see Chapter 13 for more on logical windows and user coordinates.)

Redrawing and Resizing Windows

As noted above, physical windows may become partially or fully hidden from view, either by an overlapping active window or by being made invisible with the **TC_Erase** routine. When such a window is re-shown or made active, True BASIC by default redraws the contents of the window; such windows are called ***immune***.

This redrawing of a window may not be a problem in many cases, but it does use memory. If you do not want a window to be immune, use the option "NONIMMUNE" when you create it with TC_Win_Create.

See Chapter 19 on the **OBJECT** routine for more information about immune and non-immune windows.

The size of a window may be changed either by the program or by the user. The **TC_SetRect** routine lets your program change the size of a physical window (or of any other object or control) as follows:

```
CALL TC_SetRectUsers (wid, xl, xr, yb, yt)
```

When you resize a window, any existing contents of the window do not change. If you make the window larger, there will be unused portions of the window; if you make the window smaller, existing contents will be clipped. The full contents of the window are still there, however; making the window larger will show them. When you resize a control or graphical object, you must make sure to be in the correct logical window. This can be done using a WINDOW statement, or, if the logical window is part of a physical window, using TC_Win_Switch.

Note, however, that user coordinates are readjusted to fit the new window size. Thus, any subsequent output that relies on user coordinates (such as **PLOT** or **BOX** or other graphics statements) will be fit to the new window size.

Keep in mind that a user may resize an active window at any time and such a resize has the same effect on existing and subsequent output as the **TC_SetRectUsers** routine. The **TC_Event** routine returns such `"SIZE"` events, so you could include an appropriate decision structure in your program if the user resizes a window.

Changing Window Attributes

You can add or change attributes to physical windows with several **TC_Win** routines or with the general **TC_Set** routine. You can also add menus to windows with the **TC_Menu** routines described in the next section.

You can set or change the ***title*** on a window with:

```
CALL TC_Win_SetTitle (wid, title$)
```

where `title$` contains the new string. If the window is currently visible, the title is changed dynamically. You can also find out the current title of a window with the routine:

```
CALL TC_Win_GetTitle (wid, title$)
```

This returns the current title of window `wid` in the variable `title$`.

You can change the shape of the ***text cursor*** in a window with the **TC_Win_SetCursor** routine:

```
CALL TC_Win_SetCursor (wid, shape$)
```

sets the cursor to the shape given in the string variable `shape$`. Allowable shapes may vary with the computer being used, but they include `"ARROW"`, `"IBEAM"`, `"PLUS"`, `"CROSS"`, or `"WAIT"` symbols.

You can control the ***font*** used in a window with the **TC_Win_SetFont** routine:

```
CALL TC_Win_SetFont (wid, fontname$, fontsize, fontstyle$)
```

The available fonts and styles will vary with the operating system, but some common fonts can be found on all systems. `Fontname$` values common to all systems are:

```
    "FIXED"      "HELVETICA"     "TIMES"     "SYSTEM"
```

You can find out what fonts are available on the current system with the **TC_FontsAvailable** routine.

```
CALL TC_FontsAvailable (fonts$)
```

All currently available font names are returned in the string argument fonts$, separated by vertical bars.

Available `fontstyle$` values are:

```
      "PLAIN"       "BOLD" "ITALIC"      "BOLD ITALIC"
```

The `fontsize` must be a numeric value in points. The default font is 10 points, `"FIXED"`, and `"PLAIN"`.

Three routines let you control the shape and appearance of graphics, lines, and filled objects that are drawn in the window by regular True BASIC statements, such as **PLOT**, etc. These are explained later in this chapter in the section on "Pens, Brushes, and Drawmodes for Windows & Graphical Objects."

The following example creates a nonimmune window with a title bar, close box, and resize box. It will be as large as possible on the screen.

```
LIBRARY "c:\TBVSilver\TBLIBS\TrueCtrl.trc"   ! Use appropriate path name
CALL TC_Init                                          ! Initialize

LET options$ = "close size title nonimmune"
CALL TC_Win_Create (wid, options$, 0, 1, 0, 1)
CALL TC_Win_SetTitle (wid, "My New Window")
CALL TC_Win_SetFont (wid, "helvetica", 12, "plain")   ! Change font
CALL TC_Show (wid)                                    ! Show the window
```

Checking for Valid Windows

As described earlier, you can destroy a window (and the objects within it) when you are finished with:

```
CALL TC_Free (wid)
```

When this routine is used with a window ID, it frees the window and all objects and controls associated with it. The routine first hides the window and then frees the memory associated with it and all associated objects. Once a window has been freed it no longer exists and it cannot be shown or manipulated. Also, its ID number becomes invalid and may be reused later.

You can check that a certain window is valid (i.e., has been created but not "freed") with the routine:

```
CALL TC_Win_Valid (wid)
```

If the window with identifier `wid` is not open, the routine generates an error with the message:

```
Illegal window number: ###  (711)
```

See Chapter 16 on "Handling Errors" for information on handling errors within your programs.

Creating and Using Window Menus

The **TC_Menu** routines let you add menus to any of your physical windows. To create a menu, you must first create a two-dimensional array to contain the text for the menus. The rows represent menus, and the columns contain the items for each menu. The lower bounds for menu array columns must be zero. The first item (0) in each row is the menu title.

Note: On a Macintosh, the menu for the active window always appears on the menu bar at the top of the screen, not attached to the window, but the menu is created and controlled just as with other systems.

So, to create two menus containing up to three choices plus the menu title, you might create an array such as the following (this is similar to the menu in the MATHQUIZ.TRU program in the TBDEMOS directory):

```
DIM menu$ (1:2, 0:3)   ! 2 menus and 4 items (title plus up to 3 choices)
MAT READ menu$
DATA Main, Next Problem, @, Quit@Q               ! Menu 1
DATA Level, Beginner, Intermediate, Advanced     ! Menu 2
```

You can include special characters along with the menu text to place lines between menu items or indicate a key that may be used as a menu shortcut. An ampersand (@) as a separate item places a line before the next menu item. A menu item followed immediately by an @ and one of the characters from the item designates that character as a keyboard equivalent and causes that character to be underlined in the menu item. Keystroke equivalents are shortcuts for menu items. (In Windows and OS/2, the character must be one of the letters in the text of the menu item.) The menu created by the **DATA** statements above is illustrated below.

A menu item may also be followed by two ampersands (@@) to signal the start of a ***hierarchical menu*** — where another menu is associated with that menu. See the section below on hierarchical menus.

Once an array is dimensioned and initialized, you must call the **TC_Menu_Set** routine to pass the `menu$` array to create the menu in the desired window:

```
CALL TC_Menu_Set (wid, menu$)
```

After you have created the menu, you can add check marks to individual items. In the menu example above, the last menu offers three choices of levels: Beginner, Intermediate, and Advanced. To set up the program to show that one level is selected at the beginning, you can use **TC_Menu_SetCheck** to add a check to an item:

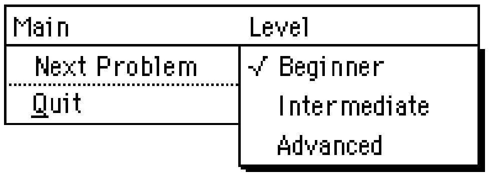
```
CALL TC_Menu_SetCheck (wid, menu, item, flag)
```

`Wid` is the ID for the window that contains the menu. The array subscripts for the menu item to be checked are indicated by `menu` (row) and `item` (column). If `flag` equals 1 (or any value other than 0), the item is checked; if `flag` equals 0, any existing check is removed. By default when you create a menu with **TC_Menu_Set**, no items are checked, but a space for a check is reserved to the left of each item.

As an example, the following program segment sets up the default physical window with a new title and a menu for a simple arithmetic quiz program:

```
! Arithmetic Quiz

LIBRARY "c:\TBSilver\TBLIBS\TrueCtrl.trc"      ! or appropriate path name

DIM menu$ (2, 0:3)    ! 2 menus and 4 items (title plus up to 3 choices)
MAT READ menu$
DATA Main, Next Problem, @, Quit@Q
DATA Level, Beginner, Intermediate, Advanced

CALL TC_Init                              ! Initialize

CALL TC_Win_SetTitle (0, "Arithmetic Quiz")    ! Change window title
CALL TC_Menu_Set (0, menu$)               ! Set the menu
CALL TC_Menu_SetCheck (0, 2, 1, 1)  ! Check menu$(2,1)
CALL TC_Show (0)                          ! Make the default window visible
...
```

The above statements would change the title of the default output window to `"Arithmetic Quiz"` and create the following menus (the Info menu is ignored on Windows and OS2):



Getting Input from Menus

So far, of course, this menu won't do anything; the program won't respond when you select a menu item. You need to create an event processing routine that looks for mouse clicks on menu choices (or keyboard equivalents) and carries out the appropriate action, such as presenting the next problem, ending the program, or resetting the difficulty level (and resetting the check mark in the third menu).

You could do this with a **DO** structure that includes a call to **TC_Event**:

```
CALL TC_Event (timer, event$, window, x1, x2)
```

The `timer` can be set to 0 so that the loop does not wait but takes the first event (if any) in the event list. The event type will be returned as `event$`. The window ID in which the event occurs will be returned as `window`; in this case `window` will be returned as 0 since the program uses only the default physical window. If the `event$` is `"MENU"`, `x1` and `x2` return the subscripts corresponding to the original `menu$` array: `x1` will contain the number or subscript of the menu selected and `x2` will contain the number or subscript of the item selected. The following lines show just such a loop that might occur at the end of the main program:

```
! Difficulty levels:  beginner = 5, intermediate = 11, advanced = 20
LET difficulty = 5            ! Default difficulty is beginner
LET cur_check = 1             ! Current menu item checked

DO
   CALL TC_Event (0, event$, window, x1, x2)

   IF event$ = "MENU" then
      LET menu = x1
      LET item = x2

      IF menu = 1 then              ! Main menu
         IF item = 1 then CALL NextProblem (difficulty, answer, response)
         IF item = 3 then EXIT DO   ! Quit is 3rd item (2nd is separator)
      ELSEIF menu = 2 then          ! Level menu
         CALL TC_Menu_SetCheck (0, 2, cur_check, 0) ! Remove current check
         IF item = 1 then
            LET difficulty = 5      ! Beginner
            CALL TC_Menu_SetCheck (0, 2, 1, 1) ! Add check to first item
            LET cur_check = 1
         ELSEIF item = 2 then
            LET difficulty = 11     ! Intermediate
            CALL TC_Menu_SetCheck (0, 2, 2, 1) ! Add check to 2nd item
            LET cur_check = 2
         ELSEIF item = 3 then
            LET difficulty = 20     ! Advanced
            CALL TC_Menu_SetCheck (0, 2, 3, 1) ! Add check to 3rd item
            LET cur_check = 3
         END IF
      END IF

   END IF
LOOP

CALL TC_Cleanup
END
```

Note that other events may be occurring (and be returned from the event queue), but this loop ignores all except `"MENU"` events. Also, the **DO** and **LOOP** statements contain no tests to end the loop, but the Quit choice in the Main menu leads to an **EXIT DO**, which exits the loop, calls **TC_Cleanup**, and ends the program.

All that remains to be done to make this a workable program is to write a `NextProblem` subroutine to present a random arithmetic problem in the window, using the value of `difficulty` to determine how hard the problem will be.

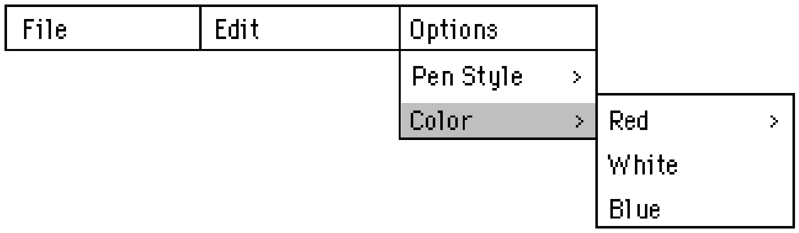
Creating Hierarchical Menus

In a ***hierarchical menu***, one or more menu items may in turn be menus themselves. For example, an Options menu might provide the choices: Pen Style and Color. Each of these in turn could offer additional choices.

You establish hierarchical menus as follows. A trailing double ampersand (@@) indicates that an item is the start of a hierarchical menu; a single ampersand (@) before a menu header, which must match a hierarchical menu item, indicates the menu choices for a hierarchical menu. Consider the following example:

```
LIBRARY "c:\TBSilver\TBLIBS\TrueCtrl.trc"   ! or appropriate path name

DIM menu$ (6, 0:3)   ! 6 menus and 4 items (title plus up to 3 choices)
MAT READ menu$
DATA File, Open, Close, Quit                ! Menu 1
DATA Edit, Cut@T, Copy@C, Paste@P           ! Menu 2
DATA Options, Pen Style@@, Color@@, ""      ! Menu 3
DATA @Pen Style, Solid, Dashed, Dotted      ! Menu 4
DATA @Color, Red@@, White, Blue,            ! Menu 5
DATA @Red, Light Red, Dark Red, ""          ! Menu 6

CALL TC_Init                                ! Initialize TC routines

CALL TC_Win_Switch (0)
CALL TC_Menu_Set (0, menu$)                 ! Set the menu
```

Here the Options menu (menu 3) contains two hierarchical menus: Pen Style and Color. Menus 4 and 5 define the hierarchical menus for those items. Note that the Color menu, in turn, contains a second level hierarchical item for Red, which is defined in Menu 6. When this menu is created and the user selects the Color item under Options, the menus will open appropriately as follows:



Notice also that menus 3 and 6 use empty quotes to indicate that there are fewer items in those menus.

Command Key Equivalents

All platforms provide command key equivalents for menu selection. (The terms “accelerator keys”, “command keys” and “alt mode keys” are often used.) In the above example the ampersand and character following a menu choice (for example: `Cut@T`) indicates such a key. Different platforms have different conventions. On Windows and OS/2, the specified letter must appear in the menu text itself; that letter will be underlined when the menu shows. (On these platforms, one can select a particular menu by holding down the Alt key while pressing the key of the underlined letter. Then, press the key of the underlined letter in the menu item itself.) On the Macintosh, the letter defines a command key combinations. (On this platform, one can select a particular menu item, regardless of which menu it is in, by holding down the command key while pressing the letter key indicated.)

See TC_Menu_Set in Chapter 22 for additional information, including ways to have the menu text correspond to the conventions of particular platforms; that is, where one would use “Exit” on Windows or OS/2, one would used “Quit” on the Macintosh.

Disabling & Editing Menu Items

Just as you can check certain items in a menu, you can disable — or gray out — certain items with **TC_Menu_SetEnable**:

```
CALL TC_Menu_SetEnable (wid, menu, item, flag)
```

The arguments are the same as those for **TC_Menu_SetCheck**. If `flag` is 0, the item is disabled or grayed out; for any other value, the item is enabled or visible. If `item` is 0, then the entire menu can be disabled or enabled. By default when you create a menu with **TC_Menu_Set** all items are enabled. You may find the current state of a given menu item with the **TC_Menu_GetEnable** routine:

```
CALL TC_Menu_getEnable (wid, menu, item, flag)
```

Other routines let you change, add, or delete menu items. You can change the text for a menu item with **TC_Menu_SetText**:

```
CALL TC_Menu_SetText (wid, menu, item, text$)
```

Similarly, you can find out the current text for a menu item with **TC_Menu_GetText**:

```
CALL TC_Menu_GetText (wid, menu, item, text$)
```

Use caution when adding or deleting menu items as this can cause confusion with the array subscripts used to identify menu items returned by events. **TC_Menu_AddItem** lets you add an item at the end of a menu with specified `text$`:

```
CALL TC_Menu_AddItem (wid, menu, text$)
```

The `wid` and `menu` arguments identify the menu as in the routines above. The array subscripts used to identify menu items are automatically updated.

Similarly, **TC_Menu_DelItem** deletes a menu item from the window and menu specified:

```
CALL TC_Menu_DelItem (wid, menu, item)
```

You can delete the last item in a menu without serious confusion about how subscripts identify menu items. But, if you wish to delete an item in the middle of a menu, it would be better to delete the entire structure (with **TC_Menu_Free**) and then rebuild it.

You may also add or delete a menu at the end of the current menu structure, using the routines:

```
CALL TC_Menu_AddMenu (wid, menu$())
```

and

```
CALL TC_Menu_DelMenu (wid)
```

TC_Menu_AddMenu adds an entirely new menu onto the end of the current menu structure. The new menu is given in the list `menu$()`, where item 0 must contain the menu header. Note that these two routines work only on the end of the current menu structure. They are most helpful for adding a special menu that may later be deleted.

Removing Menus

When you no longer need a menu in a window, it is a good idea to delete it and free the memory associated with it. You can do this with the **TC_Menu_Free** routine:

```
CALL TC_Menu_Free (wid)
```

Creating and Using Push Buttons

Push buttons are buttons containing text that the user can click on to indicate a certain action. To see an example of a simple push button, you can examine and run the DEMPUSH.TRU program in the TBDEMOS directory.

As another example, consider the arithmetic quiz being set up in the previous section. If this program presents a problem to the user and gives them a space to enter an answer, you might want to have a push button that the user can click to have the computer check the answer.

The routine that creates a push button is:

```
CALL TC_PushBtn_Create (cid, text$, xl, xr, yb, yt)
```

where `cid` returns the control ID for the button, `text$` is the text that will appear in the button, and the remaining arguments give the left, right, bottom, and top corners of the button in pixel coordinates for the current physical window.

As an example, you could place a push button in the bottom left of the window with the following statements. Remember that the default user coordinates are 0, 1, 0, 1 (unless you change them with a **SET WINDOW** statement as described in Chapter 13 on “Using Graphics”).

```
LIBRARY "c:\TBSilver\TBLIBS\TrueCtrl.trc"   ! or appropriate path name
CALL TC_Init                                ! Initialize TC routines
CALL TC_Show (0)                            ! Show default window

CALL TC_PushBtn_Create (pbid, "Check my answer", .1, .4, .1, -99999)
```

The value of -99999 passed as one of the locations for the push button signals that you want to use the default button height.

The above code merely puts the push button in the window; it cannot do anything yet. To process input from a push button, you would need to test for `event$` of `"CONTROL DESELECTED"` returned from the **TC_Event** routine. For example, you could add an additional test to the **DO** loop that checks for menu input as follows:

```
DO
   CALL TC_Event (0, event$, 0, x1, x2)

   IF event$ = "MENU" then
      <code as shown in earlier section; the Main menu includes choices
      to present a problem or end the loop>

   ELSEIF event$ = "CONTROL DESELECTED" then
      IF x2 = pbid then CALL Check_Answer (answer, response)
      ! ignore x1

   END IF
LOOP

CALL TC_Cleanup
END
```

`"CONTROL DESELECTED"` for a push button means that the user has clicked and released on it. (In the case of a push button, a `"CONTROL SELECT"` event always precedes a `"CONTROL DESELECTED"` event, but the `"CONTROL SELECT"` event is not returned until the object is deselected. Thus, the program can ignore the `"CONTROL SELECT"` event and simply test for the appropriate `"CONTROL DESELECTED"` event.)

For these events, the value returned by `x2` is the ID number for the control that was selected or deselected; `x1` is not used and can be ignored. Thus, if `event$` equals `"CONTROL DESELECTED"` and `x2` equals the ID for the push button (stored in `pbid` in this example), the user has clicked on the push button and the decision structure will carry out the appropriate action (here, checking the user's response).

Creating and Using Groups of Radio Buttons

Another way to offer users a choice of options is to provide a group of radio buttons, in which one button (and only one button) is always checked or selected. For example, in the MathQuiz program, radio buttons could provide the user with a choice of addition, subtraction, multiplication, or division problems. (The DEMRADIO.TRU program in the TBDEMOS directory also provides a simple example of radio buttons.)

The format for the **TC_RadioGroup_Create** routine is:

```
CALL TC_RadioGroup_Create (rid, text$(), xl, xr, yb, yt)
```

The text for each button should be passed in the `text$()` array, whose lower bound must be 1. The ID of the group as a whole is returned in `rid`. Initially, none of the buttons is on. If you wish to set one of the buttons to be "on", use the **TC_RadioGroup_Set** routine:

```
CALL TC_RadioGroup_Set (rid, button)
```

For example, to add a group of four radio buttons to the lower-right corner of the MathQuiz window, you could add the following statements:

```
DIM radio_text$ (4)          ! 4 buttons

MAT READ radio_text$
DATA Addition, Subtraction, Multiplication, Division
```

```
...
CALL TC_RadioGroup_Create (radio_id, radio_text$(), .6, .9, .1, .4)
CALL TC_RadioGroup_Set (radio_id, 1)
LET operation$ = radio_text$(1)
```

These statements indicate the first radio button (Addition) as the “on” button initially. The variable `operation$` is a new parameter to be passed to a NextProblem subroutine that would present an appropriate type of problem.

The above statements create a set of radio buttons but those buttons cannot do anything yet. To test for input from the buttons, you would test for `event$` of `"CONTROL DESELECTED"` returned from the **TC_Event** routine. When `event$` is `"CONTROL DESELECTED"`, `x2` returns the ID for the radio button group (as noted above, you can ignore `x1` for any `"CONTROL DESELECTED"` event).

Your program can find out the currently “on” radio button at any time with the **TC_RadioGroup_On** routine:

```
CALL TC_RadioGroup_On (rid, button)
```

The first argument must be the ID for the radio group as a whole, the second argument returns the ordinal number of the button that is currently on. For example, you might add the following test to the **DO** loop in the MathQuiz program:

```
DO
   CALL TC_Event (0, event$, 0, x1, x2)

   IF event$ = "MENU" then                        ! Menu item selected

      <code to CALL NextProblem or EXIT DO as shown in menu section above,
      adding operation$ parameter to CALL to NextProblem:>

      CALL NextProblem (difficulty, operation$, answer, response)

   ELSEIF event$ = "CONTROL DESELECTED" then      ! Control item selected

      IF x2 = pbid then CALL Check_Answer (answer, response)  ! Push button
                                                  ! ignore x1

      ELSE                                        ! Radio button
         CALL TC_RadioGroup_On (radio_id, on)
         LET operation$ = radio_text$(on)
      END IF

   END IF
LOOP
CALL TC_Cleanup
END
```

If the user clicks a radio button, the above statements reset the value of `operation$` so that the proper value is passed to the subroutine NextProblem (not shown) that will present a problem. The program need not reset the radio buttons as **TC_Event** does that automatically. (Remember that no more than one radio button can be “on”.)

As with push buttons, a `"CONTROL DESELECTED"` event is always preceded by a `"CONTROL SELECT"` event, which is not returned until the object is deselected. Thus, the program need be concerned only with the deselection of the object.

Creating and Using Check Boxes

Check-box objects let you create one or more choices similar to radio buttons. Unlike radio buttons however, in which only one button in the group can be “on”, each check box is a separate object and any one may or may not be checked “on” independently of any other check box. To see an example of a check box, you can examine and run the DEMCHECK.TRU program in the TBDEMOS directory.

The **TC_CheckBox_Create** routine is:

```
CALL TC_Checkbox_Create (cid, text$, xl, xr, yb, yt)
```

You supply the text to go with the check box as `text$`. To indicate if a box is to be checked or not, use the separate routine **TC_Checkbox_Set**:

```
CALL TC_Checkbox_Set (cid, status)
```

If `status` is 0, the box is not checked; if it is 1 (or any non-zero value), the box is drawn with an X in it.

For example, suppose you want to give the user the option of receiving warnings about something (such as misspelled words, numbers outside a certain range, etc.) and you provide three ways of giving warnings. The following statements would create three check boxes, corresponding to three warning methods. Initially, no box is checked, indicating that no warnings are desired:

```
LIBRARY "c:\TBSilver\TBLIBS\TrueCtrl.trc"     ! or appropriate path name
CALL TC_Init                             ! Initialize TC routines
CALL TC_Show (0)                         ! Show default window

DIM check_id (3), check_text$ (3), warn_flag (3)

MAT READ check_text$
DATA "Sound", "Flashing bar", "Message on screen"
MAT warn_flag = 0                        ! All warnings turned off initially

FOR i = 1 to 3
   CALL TC_Checkbox_Create (check_id(i), check_text$(i), .6, .8, 4+.3*i, -99999)
NEXT i
```

The user could then check one or more of the boxes indicating how they wish to receive warnings. The program may test for a "`CONTROL DESELECTED`" event on a check box, but more importantly, it needs to find out the new status of the box (checked or not).

In addition to returning "`CONTROL SELECT`" and "`CONTROL DESELECTED`" events for check boxes, **TC_Event** automatically changes the status of the check box. If the box was not checked before the event, **TC_Event** adds a check and changes the status of the box to 1; if the box had been checked, the check is removed and the status is changed to 0. **TC_Event** does not return the status, however; to find that you must use the **TC_CheckBox_Get** routine, as follows:

```
CALL TC_Checkbox_Get (cid, status)
```

In some cases, a program may not need to test for a check-box event, as long as it checks the status of appropriate check boxes before carrying out related tasks. More commonly, however, a program would check the status of the boxes and set appropriate flags as part of an event processor. For example:

```
DO

   CALL TC_Event (0, event$, window, x1, x2)

   IF event$ = "CONTROL DESELECTED" then
      FOR i = 1 to 3
         IF x2 = check_id (i) then                ! If a check box, ...
            CALL TC_CheckBox_Get (x2, status)     ! Get new status
            LET warn_flag (i) = status            ! Reset flag
         END IF
      NEXT i
   END IF

   <additional event processor code>

LOOP
```

Similarly, a program can change the status of a check box at any time with the **TC_Checkbox_Set** routine:

```
CALL TC_Checkbox_Set (cid, status)
```

passing the desired `status`, 0 or 1, along with the appropriate ID number.

Putting a Box Around a Group of Objects

The **TC_Groupbox_Create** routine puts a box around a group of objects such as radio buttons or related check boxes. The format for the routine is:

```
CALL TC_Groupbox_Create (cid, title$, xl, xr, yb, yt)
```

The second argument, `title$`, lets you place a title on the box; if `title$` is an empty string then no title is added. Thus, to place a simple box around the radio button group above, you could insert this statement before you create the radio button group:

```
CALL TC_Groupbox_Create (box_id, "", .6, .8, .5, .8)
```

If you wish to put a title on the box, you could do so as follows:

```
CALL TC_Groupbox_Create (box_id, "Select Problem Type", .6, .8, .5, .8)
```

Since the group box may be opaque, it must be shown before other controls that it may contain.

No events are returned for group boxes; group boxes merely organize other control objects, .

The DEMGROUP.TRU program in the TBDEMOS directory provides an example of a simple group box.

Adding Titles or Other Static Text Boxes

The group box routine can put a title on a group box. Another routine lets you place a title anywhere in a window, either as part of another object or by itself. The **TC_SText_Create** routine creates an object that contains one line of text. This text cannot be edited by the user; it is called ***static text.*** Your program, however, can change the text in a static text box using the **TC_SetText** routine as described below. (Edit fields, list edit buttons, and text editor objects — all described in this chapter — can be edited by the user.)

You create a one-line static text object as follows:

```
CALL TC_SText_Create (cid, text$, xl, xr, yb, yt)
```

You supply the text for the object in `text$`. If the defined area is too small, the text is truncated.

By default, the text is left-justified in the defined area. If you wish otherwise, you may use the **TC_SetTextJustify** routine, but it must be invoked before the control is shown the first time:

```
CALL TC_SetTextJustify (cid, justify$)
```

In `justify$`, you can specify `"CENTER"`, `"RIGHT"`, or `"LEFT"` to indicate how the text is to be placed in the area defined by `xl`, `xr`, `yb`, and `yt`. (Note: text justification may not work on all systems.)

Alternatively, you can extend the text in the create statement with an appropriate justifer, For example:

```
CALL TC_SText_Create (cid, text$ & "|center", ...)
```

can be used.

Thus, you could add a title to the check boxes created earlier, with or without a group box, as follows:

```
...
DIM check_id (3), check_text$ (3), warn_flag (3)

MAT READ check_text$
DATA "Sound", "Flashing bar", "Message on screen"
MAT warn_flag = 0             ! All warnings turned off initially
LET check_title$ = "Select Warning Type"
CALL TC_Groupbox_Create (box_id, "", .6, .8, .4, .85)
```

```
   FOR i = 1 to 3
      CALL TC_Checkbox_Create (check_id(i), check_text$(i), .6, .8, .4+.1*i, -99999)
   NEXT i

   CALL TC_SText_Create (title_id, check_title$ & "|center", .62, .78, .9, -99999)
   CALL TC_SetTextJustify (title_id, "CENTER")
```

The bottom of the static text box is placed above and slightly inside the edges of the first check box. The final value of -99999 indicates that the default height for static text should be used.

If you do not want a box around the check boxes and their title, you could omit the call to **TC_Groupbox_Create**.

Although users will not be able to edit or select static text items, a program can change the text in a static text object at any time with the **TC_SetText** routine:

```
   CALL TC_SetText (cid, text$)
```

where `cid` is the ID for the static text object and `text$` is the new text for that object. For example, you could change the static text object created above as follows:

```
   CALL TC_SetText (title_id, "Warning Method")
```

If that object is shown on the screen, the text will be updated immediately.

Creating and Using an Edit Field for Text Entry

If you wish to have a single-line field where the user can enter text, you can create an edit field. You can also specify a format for the text to be entered in an edit field. You create an edit field with the **TC_Edit_Create** routine:

```
   CALL TC_Edit_Create (cid, text$, xl, xr, yb, yt)
```

The routine returns the field's ID in `cid`; you specify the initial text to appear in the field as `text$`. If you wish to indicate a desired format for that text, use the routine **TC_Edit_SetFormat**:

```
   CALL TC_Edit_SetFormat (cid, format$)
```

Here, `format$` specifies a format for the text to be entered in an edit field; see the table below. You can check that the text conforms to the format at any time by calling

```
   CALL TC_Edit_CheckField (cid, errormess$)
```

If all is okay, `errormess$` will be the null string; otherwise, `errormess$` will contain a descriptive error message.

For example, if you wanted to create fields for the user to enter a name, phone number, and amount owed, you could create edit fields as follows:

```
   LIBRARY "c:\TBSilver\TBLIBS\TrueCtrl.trc"     ! or appropriate path name
   CALL TC_Init                             ! Initialize TC routines
   CALL TC_Show (0)                         ! Show default window

   LET xl = .2
   LET xr = .6
   LET yb = .8
   CALL TC_Edit_Create (name_id, "First Last", xl, xr, yb, -99999)
   CALL TC_Edit_Create (phone_id, "(000) 000-0000", xl, xr, yb-.1, -99999)
   CALL TC_Edit_SetFormat (phone_id, "phone")
   CALL TC_Edit_Create (balance_id, "000.00", xl, xr, yb-.2, -99999)
   CALL TC_Edit_SetFormat (balance_id, "number")
```

The format string `"PHONE"` allows any of the telephone-number formats shown below; `"NUMBER"` allows any real number. The table below shows other edit-field format strings that are allowed; case does not matter.

Edit Field Format$ Strings

format$ string	Allowable texts (examples)
`"number"`	123.456 (any real number)
`"integer"`	123 (no decimal point)
`"range 123 456"`	any integer in specified range
`"frange 12.3 45.6"`	any real number in specified range
`"zip"`	19096 or 19096-1234
`"phone"`	222-2222 222-222-2222 (222) 222-2222
`"ss"`	123-45-6789
`"date"`	MM-DD-YY DD MMM YY DD MMM YYYY MMM_DD_YYYY YYYYMMDD
`"length 12"`	any number having the specified number of characters
`"format *****"`	customized format string as indicated by any combination of the following codes in place of the *s: A = any character 9 = any digit X = any letter ? = any character at all (other characters are literals)
`list`	a, b, c (must be one of these)

Although you can check the contents of an edit field at any time, you should probably wait until the user has moved on. When **TC_Event** and returns a `"CONTROL DESELECTED"` event for the field (the field ID is returned as `x2`,) the user has finished using the edit field and has selected something else. You can then find out the text entered in the field with a call to **TC_Edit_GetText**:

```
CALL TC_Edit_GetText (cid, text$)
```

You can also check the contents against the format with TC_Edit_CheckField.

Thus, you could get input from the edit fields created above with the following code:

```
DO
   CALL TC_Event (0, event$, window, x1, x2)

   IF event$ = "CONTROL DESELECTED" then
      CALL TC_Edit_CheckField (x2, error$)
      IF error$ <> "" then
         CALL TD_Warn (error$, "Accept|Correct", 1, r)
         IF r = 1 then                         ! "Accept" button pressed
            CALL TC_Edit_GetText (x2, text$)
            IF x2 = name_id then
               LET name$ = text$
            ELSEIF x2 = phone_id
               LET phone$ = text$
            ELSEIF x2 = balance_id
               LET amount_due = Val(text$)   ! Convert to numeric value
            END IF
         ELSE
            CALL TC_Select (x2)
         END IF
      END IF
   END IF
...
LOOP until event$ = "KEYPRESS" and x1 = 27   ! Escape key ends the loop
```

(A description of TD_Warn appears later in this chapter.) Notice that the contents of the edit field are always returned as a string value, `text$`. You may wish to convert to some other form, such as a numeric value or individual values for month, day, and year.

Your program can reset the text or format for an edit field at any time with the two routines:

```
CALL TC_Edit_SetText (cid, text$)
CALL TC_Edit_SetFormat (cid, format$)
```

The new text is shown in the edit field immediately if that object is visible.

For another way to let the user enter a selection, see the list edit button described in the next section.

Creating and Using Selection Lists

There are three types of controls that let you create a list of items the user may select from. A ***list box*** is a box showing the items in a list; if the list of items it too long for the size of the list box, a scroll bar is automatically added and handled by True Controls. ***List buttons*** appear on the screen as a single button with an arrow to the right of the button text. When the user selects the arrow, the list pops down (possibly with a scroll bar) and remains in view until the user selects an item from the list. ***List edit buttons*** are similar to list buttons, but the main difference is that the user may enter a choice not given in the pop-down list. List buttons and list edit buttons allow only single selections, while list boxes may be set to allow for multiple selections on some operating systems.

List Boxes (ListBox)

To see an example of a list box, you can examine and run the DEMLISTS.TRU program in the TBDEMOS directory.

The format for the **TC_ListBox_Create** routine is:

```
CALL TC_ListBox_Create (cid, mode$, xl, xr, yb, yt)
```

`Cid` returns a single ID number that identifies the list box as a whole. The second argument allows you to set the selection mode. Possible values are:

SINGLE	Only single selections are allowed (default)
MULTIPLE	Multiple selections are allowed
READONLY	The list may be read but not selected

If the mode is not recognized or is the null string, the default (SINGLE) mode will be used. Multiple selections may not be available on all operating systems.

The contents of the list box may be set using the subroutine **TC_SetList**:

```
CALL TC_SetList (cid, list$())
```

The array `slist$` contains the text for the items in the list. Its lowest subscript must be <= 1. If the list is too long for the space defined by `yb` and `yt`, a scroll bar is added and handled automatically. For example, the following statements create a list box containing 10 items, although the box cannot display them all:

```
LIBRARY "c:\TBSilver\TBLIBS\TrueCtrl.trc"    ! or appropriate path name
CALL TC_Init                                 ! Initialize TC routines
CALL TC_Show (0)                             ! Show default window

DIM list$ (10), selection (0)                ! Selection array
```

```
MAT READ list$
DATA apple, banana, cranberry, dandelion, eggplant
DATA forsythia, hyacinth, iris, jasmine, kiwi

CALL TC_ListBox_Create (list_id, "SINGLE", .6, .8, .3, .5)
CALL TC_SetList (list_id, list$())
```

Events returned for list boxes are `"CONTROL SINGLE"` or `"CONTROL DOUBLE"` depending on whether the user selects an item with a single click or double click of the mouse button. A `"CONTROL DOUBLE"` is always preceded by a `"CONTROL SINGLE"` event. For these event types, **TC_Event** returns the control ID as `x2` (`x1` can be ignored). To find out what item or items have been selected, you must use the **TC_ListBox_Get** routine, as follows:

```
CALL TC_ListBox_Get (cid, selection())
```

For the designated list box ID, the routine returns one or more numbers in the array `selection()`. The returned numbers correspond to positions in the list — or list-text array subscripts — of any item the user selected. For example, if the user selects “dandelion” from the above list, the `selection()` array would contain the single value 4. (On some operating systems, the user can select only one item.)

First, here’s a sample section of an event handler that would detect an event in a list box and find out what had been selected:

```
DO
   CALL TC_Event (0, event$, window, x1, x2)

   IF event$ = "CONTROL SINGLE" or event$ = "CONTROL DOUBLE" and x2 = list_id then
      CALL TC_ListBox_Get (list_id, selection())   ! Get selected item #
      LET plant$ = list$(selection(LBOUND(selection)))      ! Get name
   END IF
   ...
LOOP
```

A program can redefine the items in a list box at any time with the **TC_SetList** routine:

```
CALL TC_SetList (cid, slist$())
```

For example, you could redefine the list box above to contain a list of animals with the code:

```
DIM animals$ (7)
MAT READ animals$
DATA aardvark, buffalo, cow, dog, elephant, flamingo, giraffe

CALL TC_SetList (list_id, animals$())
```

All previous items in the list box are removed and replaced by the new array of items; the number of items need not be the same. If the box is showing, the list items are changed immediately.

A program can also pre-select one of the items in a list box with the **TC_ListBox_Set** routine:

```
CALL TC_ListBox_Set (cid, selection)
```

The value of selection must be in the range from 1 to the number of items in the list. For example, the following statement would preselect `"dog"` in the list above:

```
CALL TC_ListBox_Set (list_id, 4)
```

Multiple selections, where allowed, are usually made by clicking on several items while holding down the shift key, then using a double click when finished. In this case the event handler would be interested only in the event type `"CONTROL DOUBLE"`, and the array returned by **TC_ListBox_Get** would contain multiple items. (See Chapter 19 on the **Object** Subroutine for more details.)

```
DO
   CALL TC_Event (0, event$, window, x1, x2)
```

```
      IF event$ = "CONTROL DOUBLE" and x2 = list_id then
         CALL TC_ListBox_Get (list_id, selection()) ! Get selected item nos.
         LET plants$ = ""
         FOR i = 1 to UBOUND(selection)
            LET plants$ = plants$ & list$(selection(i)) & " "   ! Get
   corresponding names
         NEXT i
      END IF
      ...
   LOOP
```

List Buttons (ListBtn)

A list button also lets the user select from a list of items, but it initially appears on the screen as a single button with a down arrow. The currently selected item is shown in the button.

When the user clicks on the down arrow, the rest of the list pops down from the button. When the user clicks on an item in the list to select it, that item replaces the selected text in the button and the pop-down list disappears.

To create a list button, use the **TC_ListBtn_Create** routine with a string array to pass the items for the list:

```
   CALL TC_ListBtn_Create (cid, list$(), xl, xr, yb, yt)
```

The first item in the list appears initially in the list button. For example, the following code creates the list button shown above:

```
   LIBRARY "c:\TBSilver\TBLIBS\TrueCtrl.trc"      ! or appropriate path name
   CALL TC_Init                               ! Initialize TC routines
   CALL TC_Show (0)                           ! Show default window

   DIM list$ (10)
   MAT READ list$
   DATA apple, banana, cranberry, dandelion, eggplant
   DATA forsythia, hyacinth, iris, jasmine, kiwi

   CALL TC_ListBtn_Create (listbtn_id, list$, .6, .8, .4, .6)
```

The last two arguments determine the space available for the pop-down list when you select the list button. If there is not enough room for all the items in the list, a scroll bar is added and handled automatically.

The only event returned for a list button is `"CONTROL SINGLE"`, with the control ID returned as `x2` (`x1` can be ignored). To find out what item or items have been selected, you must use the **TC_ListBtn_Get** routine, as follows:

```
   CALL TC_ListBtn_Get (cid, selection)
```

The item's position in the list — its subscript in the `list$()` array — is returned by `selection`. Thus, the following code could handle events in the list button created above:

```
   DO
      CALL TC_Event (0, event$, window, x1, x2)

      IF event$ = "CONTROL SINGLE" and x2 = listbtn_id then
         CALL TC_ListBtn_Get (listbtn_id, selection)! Get selected item #
         LET plant$ = list$ (selection)             ! Get corresponding name
      END IF
      ...
   LOOP
```

To see an example of a list button, you can examine and run the DEMLISTB.TRU program in the TBDEMOS directory installed with TB Silver.

List Edit Buttons (ListEdit)

List edit buttons are similar to list buttons in appearance, but the user may either select an item from the pop-down list or enter a new item in the button. (There is a slight difference in appearance, as list edit buttons have a box around the button text.) To see an example of a list edit button, you can examine and run the DEMLISTE.TRU program in the TBDEMOS directory.

List edit buttons are created by the **TC_ListEdit_Create** routine:

```
CALL TC_ListEdit_Create (cid, list$(), xl, xr, yb, yt)
```

The `list$()` array supplies the items for the pop-down list, and the `list$(0)` string supplies the item to appear initially in the button itself. For example, the following code establishes a list edit button:

```
LIBRARY "c:\TBSilver\TBLIBS\TrueCtrl.trc"     ! or appropriate path name
CALL TC_Init                           ! Initialize TC routines
CALL TC_Show (0)                       ! Show default window

DIM list$ (0:10)
MAT READ list$
DATA Plants                            ! Title for the button
DATA apple, banana, cranberry, dandelion, eggplant
DATA forsythia, hyacinth, iris, jasmine, kiwi

CALL TC_ListEdit_Create (listedit_id, list$(), .6, .8, .2, .4)
```

When the user selects an item not already in the button field, that item moves up to the button field, where it may be edited. When done editing, the user deselects the button by clicking in an area outside the button. A `"CONTROL DESELECTED"` event is then returned with the list edit button's ID.

Thus, to handle list edit button events, the program would first test for a `"CONTROL DESELECTED"` event for the button and then use the **TC_ListEdit_Get** routine to get the new text in the button:

```
CALL TC_ListEdit_Get (cid, text$)
```

The following code segment shows an event handler for the list edit button created above:

```
DO
   CALL TC_Event (0, event$, window, x1, x2)

   IF event$ = "CONTROL DESELECTED" and x2 = listedit_id then
      CALL TC_ListEdit_Get (x2, text$)
      LET plant$ = text$
   END IF
   ...
LOOP
```

You can change the list by using TC_SetList, just as with a list button. But here the 0-th element will be used as the new text for the button itself.

Creating and Using Scroll Bars

As noted in the earlier section on "Creating and Using Physical Windows", you may add vertical or horizontal scroll bars to windows by including `"VSCROLL"` or `"HSCROLL"` as `options$` in the call to the **TC_Win_Create** routine. You may also place horizontal or vertical scroll bars in any window or object with the routine **TC_SBar_Create**:

```
CALL TC_SBar_Create (cid, type$, xl, xr, yb, yt)
```

For a vertical scroll bar `type$` should be `"VSCROLL"`, and for a horizontal scroll bar it should be `"HSCROLL"`. The scroll bar is placed at the indicated location.

Three similar sets of routines let you control the range the scroll bar will cover and the action of the scroll bar and its slider "thumb". Which set you use depends on how you created the scroll bar:

Scroll Bar Routines

TC_SBar_Create	**TC_Win_Create**	**TC_Win_Create**
`type$ = VSCROLL or HSCROLL`	`option$ = VSCROLL`	`option$ = HSCROLL`
TC_SBar_SetRange	TC_WinVSBar_SetRange	TC_WinHSBar_SetRange
TC_SBar_GetRange	TC_WinVSBar_GetRange	TC_WinHSBar_GetRange
TC_SBar_SetPosition	TC_WinVSBar_SetPosition	TC_WinHSBar_SetPosition
TC_SBar_GetPosition	TC_WinVSBar_GetPosition	TC_WinHSBar_GetPosition
TC_SBar_SetIncrements	TC_WinVSBar_SetIncrements	TC_WinHSBar_SetIncrements
TC_SBar_GetIncrements	TC_WinVSBar_GetIncrements	TC_WinHSBar_GetIncrements

The rest of this section describes the **TC_SBar** routines for separate scroll bars created with the **TC_SBar_Create** routine. The **TC_WinVSBar** and **TC_WinHSBar** routines are used in the same way, except that events for window-associated scroll bars are identified by the window id number.

Two routines let you indicate the beginning and end of the range the scroll bar will cover, as well as the "page increment" or range that will be scrolled per "page" when the user clicks on the bar above or below the scroll slider or "thumb":

```
CALL TC_SBar_SetRange (cid, srange, erange, prop)
CALL TC_SBar_SetIncrements (cid, single, page)
```

When the slider is at the left or top, its position is equal to `srange`; when the slider is at the right or bottom, its position is equal to `erange` minus `prop`. On operating systems that let you control the size of the scroll slider, `prop` determines the proportional size of the slider as related to the range of the scroll bar. The `single` value indicates how far the screen should be scrolled when the user clicks the up or down arrow at either end of the scroll bar; the `page` value indicates how far the screen should be scrolled when the user clicks above or below the scroll slider.

By default, the scroll slider is initially set to the `srange` position, but another routine lets you reset a slider's position within the scroll-bar range at any time:

```
CALL TC_SBar_SetPosition (cid, position)
```

The position is always set (or reported in GetPosition) for the top of the slider. Thus, if the slider is moved to the very end of the scroll bar, the `position` will be the value of `erange` minus the value of `prop`.

The DEMVSBAR.TRU program in TBDEMOS provides a simple illustration of a vertical scroll bar. The following code fragments from the ARCERHY2.TRU program in TBDEMOS shows how that program defines the scroll bars that let the user define the angle and velocity of each shot:

```
! Archery2
LIBRARY "..\TBLibs\TrueCtrl.trc"
...
CALL TC_Init                    ! Initialize for True Controls
...
CALL TC_Show (0)                ! Show the default output window
...
! Create speed-setting scroll bar and related controls.

CALL TC_SText_Create (st1, "Force", .42, -20, -38, -99999)
CALL TC_SText_Create (speed0, "0", -12, -5, -38, -99999)
CALL TC_SText_Create (speed200, "200", 170, -185, -38, -99999)
CALL TC_SText_Create (speeddial, "0", -42, -20, -99999, -40)
CALL TC_SBar_Create (speedset, "HSCROLL", -12, 185, -99999, -40)
```

```
! Create angle-setting scroll bar and related controls.

CALL TC_SText_Create (st3, "Angle", 160, 180, -99999, 98)
CALL TC_SText_Create (angledial, "0", 185, 194, -99999, 98)
CALL TC_SText_Create (angle90, "90", 170, 180, -99999, 88)
CALL TC_SBar_Create (angleset, "VSCROLL", 185, -99999, -26, 88)
CALL TC_SText_Create (angle0, "0", 170, 180, -26, -99999)
...
! Set the scroll bar parameters and increments.

CALL TC_SBar_SetRange (angleset, 0, 100, 10)! Range from 0 to 90 (100-10)
CALL TC_SBar_SetIncrements (angleset, 1, 10)! Slider "page" by 10
CALL TC_SBar_SetPosition (angleset, 90)     ! Initial slider position = 90
CALL TC_SBar_SetRange (speedset, 0, 210, 10)! Range from 0 to 200 (210-10)
CALL TC_SBar_SetIncrements (speedset, 1, 10)! Slider "page" by 10
CALL TC_SBar_SetPosition (speedset, 0)      ! Initial slider position = 0
...
```

When initially created, the scroll-bar sliders are placed at the left end or top of the scroll bar, which is position 0 in the scroll-bar range. Because the ARCHERY2 program inverts the vertical scroll bar, a **TC_Sbar_SetPosition** routine resets the slider to the bottom of the range, or 90 (which is inverted to 0). Each time the user clicks in the bar outside the slider, the scroll position is changed by 10 (the value set for the page increment for both scroll bars).

A program can also find out the scroll-bar ranges, page increments, and slider position at any time, with the routines:

```
CALL TC_SBar_GetRange (cid, srange, erange, prop)
CALL TC_SBar_GetIncrements (cid, single, page)
CALL TC_SBar_GetPosition (cid, position)
```

When the user clicks on the scroll bar or arrows associated with it, or moves the slider, **TC_Event** carries out all adjustments to the scroll bar automatically and returns the event type . The `x2` value returns the ID number for the control; `x1` is ignored. Because **TC_Event** carries out most adjustments automatically, a program often needs only to find out the new position of the scroll bar to do something appropriate within the window. The main event loop in the ARCHERY2.TRU program handles scroll-bar events with two calls to **TC_SBar_GetPosition**:

```
CALL TC_SBar_GetPosition (angleset, angle)   ! Vertical scroll bar
LET angle = 90 - angle
IF angle <> currentangle then
   LET currentangle = angle
   CALL TC_SetText (angledial, Str$(currentangle))
END IF

CALL TC_SBar_GetPosition (speedset, speed)   ! Horizontal scroll bar
IF speed <> currentspeed then
   LET currentspeed = speed
   CALL TC_SetText (speeddial, Str$(currentspeed))
END IF
```

Each time through the loop, the slider position is updated, resetting the value for angle and speed of the shot; the current settings are used elsewhere in the game as needed.

The events that may be returned by scroll bars are as follows. Keep in mind, however, that you may not need to use these directly much of the time.

Events Returned by Scroll Bars

`"PAGEDOWN"`	user has clicked on the bar below the slider; position advances by value of the page increment
`"PAGEUP"`	user has clicked on the bar above the slider; position decreases by value of the page increment
`"DOWN"`	user has clicked the arrow at the bottom of the scroll bar; position advances by one
`"UP"`	user has clicked the arrow at the top of the scroll bar; position decreases by one
`"VSCROLL"`	user is in process of moving the scroll-bar slider
`"END VSCROLL"`	user has finished moving the scroll-bar slider; position is final location of slider
`"PAGERIGHT"`	user has clicked on the bar to the right of the slider; position advances by value of the page increment
`"PAGELEFT"`	user has clicked on the bar to the left of the slider; position decreases by value of the page increment
`"RIGHT"`	user has clicked the arrow at the right of the scroll bar; position advances by one
`"LEFT"`	user has clicked the arrow at the left of the scroll bar; position decreases by one
`"HSCROLL"`	user is in process of moving the scroll-bar slider
`"END HSCROLL"`	user has finished moving the scroll-bar slider; position is final location of slider

Creating Graphics Objects

True BASIC's graphics statements described in Chapter 13 provide one method for producing graphical elements. You may also produce such objects via True Controls routines. The main routine that creates a graphics object is:

```
   CALL TC_Graph_Create (gid, type$, xl, xr, yb, yt)
```

As usual, `cid` returns the ID number for the object. For the argument `type$` you may pass any of the strings listed below. The arguments `xl`, `xr`, `yb`, and `yt` are applied differently depending on the object type:

Graphics Object Types

Type$	How coordinates are used
`"RECTANGLE", "CIRCLE", "ARC", "PIE", "ROUNDRECT"`	define rectangular area
`"ALINE", "LINE"`	define start x, end x, start y, end y
`"POLYGON", "POLYLINE"`	ignored (except for scaling)
`"IMAGE"`	define rectangular area; may distort

Graphical objects are not controls; they return no events. They simply provide another way for your programs to create graphical output. They are displayed in their own "layer", which is "above" ordinary True BASIC printed and plotted output. They may also be layered underneath any and all real controls, such as push buttons, although this property is not consistent across all platforms.

RECTANGLE, CIRCLE, ARC, PIE, and ROUNDRECT

For the first five object types, the arguments `xl`, `xr`, `yb`, and `yt` must define the left, right, bottom, and top of a rectangle. A `"RECTANGLE"` is drawn to fill the defined area; the other object types in this group are fit within that rectangle. For example, a `"CIRCLE"` is placed within the defined rectangular area with the edges of the circle touching each side of the area. Thus, a `"CIRCLE"` may appear as an ellipse if the coordinates do not define a square rectangle. Additional routines further define the appearance and placements of arcs, pies, and rounded rectangles.

A `"ROUNDRECT"` is a rectangle with curved corners. The sides of a `"ROUNDRECT"` are drawn just as a similarly defined `"RECTANGLE"` would be, except that the corners are curved. The size of the arcs in the corners may be defined with the **TC_Graph_SetRoundRect** routine:

```
CALL TC_Graph_SetRoundRect (gid, ovalwidth, ovalheight)
```

The `ovalwidth` and `ovalheight` arguments define the size of an oval whose four quadrants will form the corners of the rectangle. Thus, the larger the values for `ovalwidth` and `ovalheight`, the more rounded the corners of the rectangle. A `"ROUNDRECT"` drawn with no call to **TC_Graph_SetRoundRect** will have square corners (the `ovalwidth` and `ovalheight` each equal 0).

An `"ARC"` and a `"PIE"` are segments of circles drawn within the defined rectangular area. An `"ARC"` is a segment of a circle, and a `"PIE"` is an arc with lines from the ends of the arc to the center. The size of the `"ARC"` or the `"PIE"` segment is defined with the **TC_Graph_SetArc** routine:

```
CALL TC_Graph_SetArc (gid, starta, stopa)
```

The `starta` and `stopa` arguments define two angles, in degrees. The arc is defined as the portion of the circumference of the circle (defined by the rectangular area in by **TC_Graph_Create**) that starts at `starta` and ends at `stopa`, proceeding counterclockwise. The angle 0 is the positive x-axis.

To see examples of a arcs and pies, you can examine and run the DEMARC.TRU and DEMPIE.TRU programs in the TBDEMOS directory installed with Silver Edition.

ALINE and LINE (Arrows and Lines)

The next two object types — `"ALINE"` and `"LINE"` — draw an arrow or a plain line. For them, the arguments `xl`, `xr`, `yb`, and `yt` indicate that the line or arrow should begin at the point defined by the first x and y coordinates (`xl` and `yb`) and end at the point defined by the second x and y coordinates (`xr` and `yt`). The following example draws a line that slants down and to the right:

```
CALL TC_Graph_Create (cid, "LINE", .2, .8, .7, .3)
```

The line will be drawn from the coordinate point (.2,.7) to the coordinate point (.8,.3).

Lines with arrowheads are similarly created.

```
CALL TC_Graph_Create (cid, "ALINE", .2, .8, .7, .3)
```

The **TC_Graph_SetAline** routine then is used to define which end will have an arrowhead:

```
CALL TC_Graph_SetAline (gid, start, end)
```

If the value of `start` is non-zero, an arrowhead is placed at the beginning of the line; if `start` equals zero, there is no arrowhead at the start of the line. Similarly, a non-zero value for `end` places an arrowhead at the end of the line. An `"ALINE"` drawn with no call to **TC_Graph_SetAline** will have no arrowheads.

The following example creates an arrow and places an arrowhead at the lower, right end of the line.

```
CALL TC_Graph_Create (cid, "ALINE", .2, .8, .7, .3)
CALL TC_Graph_SetAline (cid, 0, 1)
```

POLYGON and POLYLINE

For the remaining two object types — `"POLYGON"` and `"POLYLINE"` — the coordinates passed by the **TC_Graph_Create** routine have no meaning (unless the object is scaled with **TC_Graph_Scale** described below). The placement of these objects are defined by an array that is passed by the **TC_Graph_SetPoly** routine:

```
CALL TC_Graph_SetPoly (gid, pts(,))
```

The array `pts` must be a two-dimensional numeric array, with each row containing an x-y coordinate pair. The `"POLYLINE"` or `"POLYGON"` will be drawn connecting the points in the array in the order in which they are given. With a `"POLYGON"` object, a line is also drawn from the last point defined to the first, enclosing the polygon.

The following segment of the DEMPOLY.TRU program in TBDEMOS creates a star-shaped polygon:

```
LIBRARY "c:\TBSilver\tblibs\TrueCtrl.TRC"   ! or appropriate path name
CALL TC_Init
```

```
SET WINDOW -3, 3, -2, 2                         ! Define user coordinates
CALL TC_Show (0)                                ! Show default window
...
! Generate points for a star
DIM start (10,2)
OPTION ANGLE degrees
LET i, r = 1
LET short = Cos(72) / Cos(36)
FOR a = 90 to 414 step 36
   LET star (i, 1) = r * Cos(a)                 ! x-coordinate
   LET star (i, 2) = r * Sin(a)                 ! y-coordinate
   LET i = i + 1
   IF r = 1 then LET r = short else LET r = 1
NEXT a

CALL TC_Graph_Create (poly1, "POLYLINE", 0, 1, 0, 1)
CALL TC_Graph_SetPoly (poly1, star(,))
...
```

To see the star shape created by the above code, run the DEMPOLY.TRU program in the TBDEMOS directory.

Images

You may be familiar with BOX KEEP and BOX SHOW, which are described in Chapter 13. These instructions permit extraction of a portion of the True BASIC output screen into a string, called a "box keep string", and later redisplaying it at perhaps a different location. The box keep string keeps the image in pixel format; thus, the resolution is dependent on the resolution of the screen. Furthermore, the box keep string format is different for different platforms.

True BASIC also offers a way to display images in the image layer as a graphics object. The image layer is above the plotting layer (used by box keep) on most platforms. True BASIC also provides a way to convert between images and box keep strings.

Images as stored in files may be any one of several types: JPEG, BMP, and PICT (Macintosh only).

True BASIC provides two subroutines for converting between an image as stored in a file and a box keep string.

```
CALL Read_Image (imagetype$, boxkeepstring$, filename$)
CALL Write_Image (imagetype$, boxkeepstring$, filename$)
```

The first subroutine reads an image from a file, converting it into a box keep string in the local platform format, and stores the result in `boxkeepstring$`. Permissible image types are: `"JPEG"`, `"MS BMP"`, `"OS/2 BMP"`, and `"PICT"`, the last one being valid only for the Macintosh. The type must be specified exactly as shown, although you may use lowercase or mixed case letters. If you don't know the image type, leave that argument a null string; True BASIC will do its best to determine the image type from the contents of the file.

The second subroutine does the reverse; it takes a box keep string, converts it to an image file format, and stores the result in a file. Here you must specify the image type, but type `"JPEG"` is not allowed.

Once in a box keep string, an image may be displayed using the BOX SHOW statement. Or, you can grab part or all of the contents of a window using a BOX KEEP statement, and then save it in a file in an image format.

Three subroutines allow you to deal with the image layer:

```
CALL TC_Graph_SetImageFromFile (gid, filename$, filetype$, adjustflag)
CALL TC_Graph_SetImageFromBox (gid, boxkeepstring$)
CALL TC_Graph_GetImageToBox (gid, boxkeepstring$)
```

The first subroutine allows you to display a graphics image in the image layer. Of course, the graphics object must have been created using TC_Graph_Create, which also defined the id number `gid`. The filetype$ must be one of

`"JPEG"`, `"MS BMP"`, `"OS/2 BMP"`, and `"PICT"`, the last one being valid only for the Macintosh. If you don't know the file type, use a null string; True BASIC will do its best to determine the image type from the contents of the file.

The value of adjustflag tells the subroutine whether you want the image displayed in the rectangle you previously defined (in which case it may be distorted,) in a rectangle of the same size as the image and centered at the center of the rectangle you defined in TC_Graph_Create, or in a rectangle the same size as the image and centered in the center of the logical window.

```
  adjustflag = -1       ! center it in the window, keep original size
  adjustflag = +1       ! center it in the rectangle, but keep original size
  adjustflag = 0        ! use the rectangle, scaling image if necessary
```

If you are displaying a startup logo, you'll probably want to have adjustflag = -1, as that will center the logo and display it without scaling.

The second subroutine allows you to display the contents of a box keep string as an image in the image layer. No provision is made here for adjusting the size of the image to fit the rectangle defined by your call to TC_Graph_Create. You will probably know the exact size of the box keep image, and use rectangular coordinates consistent with that size. Or, you can first write the box keep string into a file using the Write_Image subroutine, and then bring it back using the TC_Graph_SetImageFromFile subroutine, allowing you to display without scaling.

The third subroutine allows you to take an image that has previously been displayed in the image layer and store it in box keep format in a box keep string. This subroutine uses the rectangle of the image to define the limits of the box keep process.

Suppose you wish to construct an image that would normally appear in the graphics layer and combine it with graphics from the plotting layer. The idea is simple. Just read a graphical image from a file into a box keep string using the Read_Image subroutine, display it using BOX SHOW, add additional True BASIC graphics as desired, keep the whole using BOX KEEP, and finally saving the result in a file in image format using the Write_Image subroutine. The following example shows typical code:

```
  CALL Read_Image ("", bks$, imagefile$)     ! Get the bit-mapped image
  BOX SHOW bks$ at .2, .2                    ! Display it in the plot layer
  PLOT TEXT, at .4, .7: "Welcome to ABC Corp."   ! Add other graphics
  BOX KEEP 0, 1, 0, 1 in bks2$               ! Grab the entire window
  CALL Write_Image ("MS BMP", bks2$, outfile$)   ! and save it in MS BMP format
```

If you like, you can now display the combined result in the image layer with

```
  CALL TC_Graph_Create (gid, "IMAGE", .1, .9, .1, .9)
  CALL TC_Graph_SetImageFromFile (gid, outfile$, "MS BMP", 0)
```

Of course, you can always save box keep strings in their local format using the WRITE statement to a byte file, and read them back the same way. But this approach is not platform-independent.

The possibilities can summarized as follows:

From an image in a file to:	**Use**
Box Keep String	Read_Image
Graphics Layer	Read_Image, BOX SHOW
Image Layer	TC_Graph_SetImageFromFile
From a BOX KEEP string to:	**Use**
Image File	Write_Image
Graphics Layer	BOX SHOW
Image Layer	TC_Graph_SetImageFromBox
From an image in the graphics layer to:	**Use**
Image File	BOX KEEP, Write_Image
BOX KEEP String	BOX KEEP
Image Layer	BOX KEEP, TC_Graph_SetImageFromBox

From an image in the image layer to:	**Use**
Image File	TC_GetImageToBox, Write_Image
BOX KEEP String	TC_GetImageToBox
Graphics Layer	TC_GetImageToBox, BOX SHOW

Shifting and Scaling

Graphics objects may be shifted (translated or moved within the window) using the **TC_Graph_Shift** routine:

```
CALL TC_Graph_Shift (gid, xdelta, ydelta)
```

The object indicated by `gid` will be shifted by `xdelta` in the x direction and by `ydelta` in the y direction. `Xdelta` and `ydelta` should use the same coordinate system as the original graphics object.

Graphics objects may also be scaled (expanded or contracted) with **TC_Graph_Scale:**

```
CALL TC_Graph_Scale (gid, xscale, yscale)
```

If `xscale` is greater than 1, the object identified by `gid` will be expanded in the x direction; if `xscale` is less than 1, the objected will be contracted in the x-direction. The same holds true for the y direction. The scaling is relative to the center of the object's defining rectangle. For `"POLYGON"` and `"POLYLINE"` objects, the defining rectangle, though ignored otherwise, is used for scaling the objects.

Pens and Brushes for Windows and Graphics Objects

Two sets of routines let you control the shape and appearance of lines, filled objects, and other graphics drawn in windows. Three **TC_Win** routines set the attributes of objects drawn by regular True BASIC statements such as **PLOT**, etc., while similar **TC_Graph** routines control the same attributes for True Controls graphical objects:

```
TC_Win_SetPen       specify width, color, style and pattern of lines
TC_Graph_SetPen

TC_Win_SetBrush     specify appearance of filled areas
TC_Graph_SetBrush
```

Although the two sets of routines are similar, they act a bit differently. The **TC_Win** routines affect subsequent lines or graphics drawn by True BASIC statements in the designated window; it has no affect on any True Controls objects. **TC_Graph** routines, however, act only on a designated True Controls object; they do not affect any existing or subsequent objects, nor do they affect anything drawn by True BASIC statements.

TC_Win_SetPen and **TC_Graph_SetPen** let you specify the width, color, style, and pattern of any lines drawn by True BASIC graphics statements or a True Controls object, respectively:

```
TC_Win_SetPen (wid, width, color, style$, pattern$)
TC_Graph_SetPen (gid, width, color, style$, pattern$)
```

For **TC_Win_SetPen** you supply the ID for a physical window (`wid`) and for **TC_Graph_SetPen** you supply the ID for a specific graphical object (`gid`). You specify the `width` of the line in pixels; the default `width` is 1 pixel. You may use any of True BASIC's color numbers (see Chapter 13) to specify the pen `color`; the default `color` is -1 (black). The pen `style$` may be one of the following (case does not matter):

`"SOLID"`	solid line (default)
`"DOT"`	dotted line; only if `width` is 1
`"DASH"`	dashed line; only if `width` is 1

If the pen `width` is something other than 1 pixel, the line will be solid regardless of the `style$` setting. The `pattern$` string lets you specify a fill pattern for lines drawn as follows:

`"SOLID"`	solid (default)
`"HOLLOW"`	no visible pattern; overrides `style$` regardless of pen width
`"RUBBER"`	grayish or dappled pattern; occurs only if `style$` is solid and `width` is 1

The pen attributes may be changed at any time. For windows, the attributes affect all subsequent output from True BASIC graphics statements, and existing True BASIC graphics may also be affected if the window is redrawn. For True Control graphic objects, the object is redrawn on the screen to reflect the new attributes; no other objects are affected. If `width` is less than zero, it is not changed; if `color` is less than -2, it is not changed; if `style$` or `pattern$` is the null string, it is not changed. The demonstration program DEMSTYLE.TRU illustrates how pen widths, styles, and patterns interact.

The **TC_Win_SetBrush** and **TC_Graph_SetBrush** routines control the appearance of filled areas created with graphics statements or the area inside the specified graphical object, respectively:

```
   TC_Win_SetBrush (wid, backclr, color, pattern$)
   TC_Graph_SetBrush (gid, backclr, color, pattern$)
```

For **TC_Win_SetBrush** you supply the ID for a physical window (`wid`) and for **TC_Graph_SetBrush** you supply the ID for a specific graphical object (`gid`). The `backclr` and `color` attributes, which may be any valid True BASIC color number, set the background and foreground color for the entire window or the fill pattern, respectively. The default `color` is black (-1) and the default `backclr` is white (-2). The brush `pattern$` string may be any of the following patterns:

`"SOLID"`	solid (default)
`"HOLLOW"`	no visible pattern
`"HORZ"`	horizontal lines
`"VERT"`	vertical lines
`"FDIAG"`	diagonal lines running from lower left to upper right
`"BDIAG"`	diagonal lines running from upper left to lower right
`"CROSS"`	crossing horizontal and vertical lines
`"DIAGCROSS"`	crossing diagonal lines

As with pen settings, **TC_Win_SetBrush** changes affect all subsequent graphics *statements*, and may alter existing graphics, while **TC_Graph_SetBrush** affects only the designated object.

Creating and Using Text Edit Controls

You can include a text editor in your program. This type of control acts on all keypress and mouse events that occur within it. It can handle several different fonts, font styles, and font sizes. The user can select text by clicking and dragging the mouse. If you include scroll bars, they will be automatically synchronized with the text itself. You may specify wrapped text in which lines are folded when they reach the margin. The True Controls library TRUECTRL.TRC includes routines to carry out the cut, copy, and paste functions, and to find certain text sequences.

Text Editor Options

To create a text edit control, make sure you are in the correct target window and use:

```
   CALL TC_Txed_Create (cid, op$, xl, xr, yb, yt)
```

The first argument will be the ID assigned to the control. The four coordinates define the outer limits of the text edit control and include scroll bars and borders, if such are specified. The actual interior size available to the text itself will be slightly smaller.

The options allowed for op$ are given in the following table. Multiple options must be separated by vertical bars (`|`).

Text Edit Control Options

Op$ value	Meaning
`"ATTACHED"`	Embed in the window, resize if the window is resized
`"READONLY"`	The user will not be allowed to change the text
`"WRAP"`	Lines will be folded when they reach the margin
`"MARGIN n"`	The desired margin, ignored unless the text is wrapped
`"BORDER"`	Include a border
`"VSCROLL"`	Include a vertical scroll bar
`"HSCROLL"`	Include a horizontal scroll bar
`"KEY EVENTS"`	Return key events as well as absorbing them
`"MOUSE EVENTS"`	Return mouse events as well as absorbing them

If you include `"ATTACHED"` as an option, the four positioning parameters will be ignored and the text editor will fill the available space in the window. Furthermore, if the window is resized, the text edit control will be resized along with it. And, if you have specified wrapped text with `"WRAP"`, True Controls will reset the `margin` so that all of a line will be visible. Thus, if you use `"ATTACHED"` with wrapped text, you will not need a horizontal scroll bar, as the entire horizontal aspect of the text will be visible. Note also, if you want to use scroll bars with an attached text-edit control, you must specify `"VSCROLL"` or `"HSCROLL"` or both as options when you create the window. True Controls will therefore ignore the `"VSCROLL"` and `"HSCROLL"` options for an attached text edit control, as well as the `"BORDER"` option, since the window itself will provide a border.

`"READONLY"` can be used to present text that the user can not modify, such as help screens. `"WRAP"` specifies that the lines of the text will be folded at the margin specified. If you use `"WRAP"`, you may also specify a margin with `"MARGIN"`, which has the format:

```
   MARGIN 120
```

where the number that follows the word `"MARGIN"` specifies with maximum width of the text *in pixels*. If the option `"ATTACHED"` is used, then the margin is set automatically if the text is wrapped.

You can change the MARGIN setting at any time by using

```
   CALL TC_Txed_SetMargin (cid, margin)
```

Remember that the margin must be expressed in pixels. If you specify a margin < 0, then the margin will be set to the current width of the text edit control.

Include `"BORDER"` as an option if you want a border. Include `"VSCROLL"` if you want a vertical scroll bar; include `"HSCROLL"` for a horizontal scroll bar. True Controls automatically places the scroll bars where they belong, and takes care of synchronizing them with the text. (Do not include these options if the text editor is `"ATTACHED"` to a window.)

Mouse and Key Events in Text Editors

If you need to know about mouse events, in addition to having them acted upon by the text edit control, include `"MOUSE EVENTS"` as an option. For example, you may wish to notify the user of the exact line and character position of the cursor. These will be returned by **TC_Event** as `"TXE MOUSE"` events.

If you need to examine the user's keystrokes, in addition to having them acted on by the text edit control, include `"KEY EVENTS"` as an option. You may need to do this if, for example, you have defined one or more characters as menu equivalents. With the `"KEY EVENTS"` option, all keystrokes will then be returned by **TC_Event** as `"TXE KEYPRESS"` events, and the code (ASCII) of the key will be returned in `x1`.

Instead of specifying `"KEY EVENTS"`, you may wish to specify only certain characters as "trap characters" for special treatment. Occurrences will be returned as `"TXE KEYPRESS"` events with the character number (ASCII code) in `x1`. (The end-of-line character is always returned.) To specify a trap character, use:

```
CALL TC_Txed_SetTrapChar (cid, char, action)
```

`Char` is the (ASCII) code of the character to be trapped. `Action` is a numeric defined as follows:

Text Edit Trap Character Actions

Action	Effect
1	The text edit control is suspended, and the character is ignored by the text edit control
2	The text edit control is not suspended, and the key is acted upon by the text edit control
3	If and only if there is selected text, the text edit control is suspended, and the character is ignored by the text edit control
< 0	The particular character is *unregistered*

All other action codes are ignored.

As examples, if you wish to use the Escape key to exit from the text edit control, give it a stop code of 1. If you wish to readjust the scroll bars whenever the user presses the Enter or Return key, give it a stop code of 2. If you wish to indent selected text when the user enters a ">", give it a stop code of 3. (Note: True Controls always registers the Return key (13) as a trap character with action 2.) In the cases of actions 1 and 3, you will need to have the text edit control resume by issuing:

```
CALL TC_Txed_Resume (cid)
```

Text Input and Output with Text Editors

Once you establish the text edit control, you may wish to supply it with text. And later, if the user has made modifications, you may wish to retrieve the text, perhaps for saving to a file. Six routines are used for these purposes:

```
CALL TC_Txed_ReadTextFromFile (txid, filename$)
CALL TC_Txed_WriteTextToFile (txid, filename$)

CALL TC_Txed_ReadTextFromArray (txid, lines$())
CALL TC_Txed_WriteTextToArray (txid, lines$())
```

The first two read and write the text edit control text from and to a file. The last two read and write the text edit control text from and to a string array. These routines actually use the slightly more primitive routines:

```
CALL TC_Txed_SetText (cid, text$)   ! Supply the text to the editor
CALL TC_Txed_GetText (cid, text$)   ! Retrieve the text from the editor
```

The form of the text in the string variable `text$` will be exactly as the text might be stored in a text file. Lines of the text are assumed to be terminated with the system-dependent end-of-line sequence. The end-of-line character sequence is typically character 13 (Return) or character 13 followed by character 10 (Line feed).

Be aware that what are called *lines* in a text file are called *paragraphs* in the text editor. These consist of strings of ASCII characters terminated by an end-of-line sequence. What the text editor calls *lines* are portions of a paragraph that fit within the specified margin. The way paragraphs are divided into lines depends on the width of the text editor, as well as on the font (name, size, and style) being used.

Fonts, Styles, Sizes, and Colors in Text Editors

The default font is ten-point , plain Helvetica. To specify another font, use:

```
CALL TC_Txed_SetFont (cid, fontname$, fontsize, fontstyle$)
```

Acceptable font names are `"Helvetica"`, `"Times"`, `"Fixed"`, and `"System"`. Acceptable font styles are `"plain"`, `"bold"`, `"italic"`, and `"bold italic"`. Case (upper or lower) doesn't matter. Additional font names and font styles may be available on some systems. The font size is specified in points (a point is approximately 1/72 of an inch). If the font name or the font style is the null string, the previous value will not be changed.

If the font size is a negative number, the previous size will not be changed.

You can find out what fonts are available by calling TC_FontsAvailable.

The default colors are black (-1) on white (-2) , with a black border. If you wish to specify other colors, use

```
CALL TC_Txed_SetColor (cid, forecolor, backcolor, bordercolor)
```

The three colors are numbers that refer to the color mix table currently in use (see Chapter 13 “Graphics”). If you specify a number less than -2, the previous value of that color will not be changed.

Cut, Copy, and Paste with Text Editors

True Controls provides for the usual cut, copy, and paste functions.

```
CALL TC_Txed_Cut (txed)
CALL TC_Txed_Copy (txed)
CALL TC_Txed_Paste (txed)
```

In each case, it is assumed that text has been selected by the user, so that it shows in reversed color. Cut removes the text from the text editor and places it on the system clipboard. Copy just places the selected text on the system clipboard. Paste inserts the contents of the system clipboard at the insertion point, indicated by the insertion cursor; if text has been selected, paste replaces the selected text with the contents of the clipboard.

If you have included menus in the window that contains the text editor, you may wish to create menu items for cut, copy, and paste. You can then use the **TC_Txed_SetCutCopyPaste** routine to have True Controls intercept those menu items and call the appropriate subroutine above. The format for this routine is:

```
CALL TC_Txed_SetCutCopyPaste (wid, cutm, cuti, copym, copyi, pastem, pastei)
```

For `cutm` and `cuti`, you supply the appropriate subscripts for the menu and item choice for Cut, and so on for the remaining arguments. True Controls keeps track of a text edit control attached to a particular window, and invokes appropriate cut, copy, or paste operations. Warning: the text edit control must be attached, and there can be no more than one such attached text edit control.

Find Text in Text Editors

True Controls includes a search utility that works with either wrapped or unfolded text.

```
CALL TC_Txed_Find (cid, case, word, key$, par, ln1, ch1, ln2, ch2, found)
```

If you wish the search to be case-sensitive, set the variable `case` to 1; otherwise, set it to 0. If you wish the search to concentrate on entire words, set the variable `word` to 1; otherwise, set it to 0.

The search key must be supplied in the string variable `key$`. The next five arguments specify where the search should begin. (Note that if the text is not wrapped, then the line number is always 0. Also note that paragraph, line, and character numbering start with 0.) To start the search at the beginning of the text, set all five values to 0.

If the search is successful, then the argument `found` will have the value 1. The matched text will be selected in the text editor, and its position returned in the five arguments. If the search is not successful, `found` will have the value 0, and the prior values of the five arguments will not be changed.

Note that the matched text must be contained within a single paragraph, whose number is returned in `par`.

Selecting Text in Text Editor

Finally, you may wish to select or highlight certain text in the text editor. For example, you may wish to highlight certain portions of a help file. This can be done with

```
CALL TC_Txed_SetSelection (cid, par1, ln1, ch1, par2, ln2, ch2)
```

You must, of course, determine the correct values of the starting and ending paragraphs, lines, and characters. And remember that paragraph, line, and character numbering starts with 0.

Example of Text Edit Control

To see an example of a text edit control, you can examine and run the DEMTXED.TRU program in the TBDEMOS directory.

True Controls Events Summary

The True Controls subroutine **TC_Event** returns the first event on the event queue. The calling sequence is:

```
CALL TC_Event (timer, event$, window, x1, x2)
```

If there is an event in the event queue, **TC_Event** returns immediately reporting the event type in `event$`. If there is no event in the event queue, then **TC_Event** will wait for the number of seconds specified by `timer`. If an event happens during that time, **TC_Event** returns immediately with that `event$`; if no event occurs, the routine returns an empty string to `event$`.

Note that, even if the event is returned by **TC_Event**, that subroutine may already have taken certain actions.

If an event has taken place, the remaining three arguments return additional information about the event. `Window` returns the physical window ID, and `x1` and `x2` return values specific to the event type. The values of `x1` and `x2` returned for each `event$` type are summarized in the table that follows. (The `event$` string is returned in upper case. The notation "—-" means that the value of the variable is ignored.)

Events Returned by TC_Event

Event$	**x1**	**x2**
From mouse activity in windows:		
"SINGLE	x-coord	y-coord
"DOUBLE"	x-coord	y-coord
"EXTEND"	x-coord	y-coord
"SINGLE RIGHT"	x-coord	y-coord
"DOUBLE RIGHT"	x-coord	y-coord
"EXTEND RIGHT"	x-coord	y-coord
"SINGLE MIDDLE"	x-coord	y-coord
"DOUBLE MIDDLE"	x-coord	y-coord
"EXTEND MIDDLE"	x-coord	y-coord
"MOUSE UP"	x-coord	y-coord
"MOUSE UP RIGHT"	x-coord	y-coord
"MOUSE UP MIDDLE"	x-coord	y-coord
"MOUSE MOVE"	x-coord	y-coord
From key press in a window		
"KEYPRESS"	ASCII code	1 if shift key down; 2 if control key down; 3 if both; 0 if neither
From menu selection		
"MENU"	menu number	item number
Events related to windows		
"SIZE"	—-	—-
"REFRESH"	—-	—-
"SELECT"	—-	—-
"HIDE"	—-	—-
From scroll bars		
"UP	—-	ID of scroll bar; —- if attached to a window
"DOWN"	—-	ID of scroll bar; —- if attached to a window
"LEFT"	—-	ID of scroll bar; —- if attached to a window

"RIGHT"	—-	ID of scroll bar; —- if attached to a window
"PAGEUP"	—-	ID of scroll bar; —- if attached to a window
"PAGEDOWN"	—-	ID of scroll bar; —- if attached to a window
"PAGELEFT"	—-	ID of scroll bar; —- if attached to a window
"PAGERIGHT"	—-	ID of scroll bar; —- if attached to a window
"VSCROLL"	—-	ID of scroll bar; —- if attached to a window
"HSCROLL"	—-	ID of scroll bar; —- if attached to a window
"END VSCROLL"	—-	ID of scroll bar; —- if attached to a window
"END HSCROLL"	—-	ID of scroll bar; —- if attached to a window
Events from list boxes and list buttons		
"CONTROL SINGLE	—-	control ID
"CONTROL DOUBLE	—-	control ID
From push buttons, radio buttons, check boxes, edit fields, list edit buttons, & text edit controls.		
"CONTROL SELECT	—-	control ID
"CONTROL DESELECTED	—-	control ID
From text edit controls.		
"TXE KEYPRESS	char	ID
"TXE MOUSE	0	ID

Creating and Using Dialog Boxes (True Dials)

The True Dials routines let you create warning dialog boxes, yes-no response dialog boxes, one-line and multiple-line dialog boxes, file open and file save dialogs, and list selection boxes. The dialog box routines are saved in a separate library from the other user interface items because they act and are used a bit differently than the other objects.

All dialog boxes are ***modal***; that is, no action can occur outside the dialog box until the dialog box activity is completed or has "timed out". By default, dialog boxes are placed in the center of the active window by True Dials. Thus, the True Dials routines are a bit easier to use than the True Controls routines.

All of the True Dials routines call on the powerful **TBD** built-in subroutine. Users who want direct control of dialog boxes should refer to the **TBD** routine in Chapter 21. Additional control over placement and size is provided by the **TBDX** subroutine.

The True Dials routines have names that begin with TD_ and are saved in the TRUEDIAL.TRC library, which must be named at the beginning of any program that will call the routines. Your programs will start faster if you use the compiled version of the library. The following statement uses the compiled library in Windows or OS/2:

```
LIBRARY "c:\TBSilver\TBLIBS\TrueDial.trc"     ! or appropriate path name
```

On the Macintosh, the statement might be:

```
LIBRARY "hdisk:TBSilver:TBLIBS:TrueDial.trc" ! use appropriate disk & folder names
```

There is no initialization routine that must be called and no need to "clean up" after you've used dialog boxes.

Different types of dialog boxes are set up by different routines as described below. These routines share many of the same arguments:

Arguments Used by True Dials Routines

`title$`	specifies the title that appears at the top edge of some of the dialog boxes. On some platforms, such as the Macintosh, the title will not show for any of the dialog boxes.
`message$`	specifies the message that is to appear in the dialog box. The message may contain several lines, which should be separated in the `message$` string by vertical bars (|). If there is not enough room for the message, it will be truncated.

`button$`	specifies from one to four buttons that may be displayed in the dialog box. The texts for the buttons should be separated in the string `button$` by vertical bars (|). If there is not enough room in a button to display the text, it will be truncated.
`default`	specifies which button, if any, is to be outlined. An outlined or selected button can be activated by pressing the Return or Enter key.
`result`	specifies which button was selected to terminate the dialog box. If timeout has occurred, result = 0.

When a dialog box is created, it remains on the screen and no other activity can occur until the user responds, or until it has “timed out.” You can set the timeout parameter, which by default is 0 (which means no timeout), using the following routine:

```
CALL TD_SetTimeout (seconds)
```

Dialog boxes will be displayed for the specified number of `seconds`. If `seconds` is 0 (the default), there is no timeout and the dialog box will remain until the user responds. This statement must be executed to set the timeout before a dialog box is displayed; it remains in effect until another call to **TD_SetTimeout**. A similar routine may be used to find out what the current timeout is:

```
CALL TD_GetTimeout (seconds)
```

The True Dials routines that create the various kinds of dialog boxes are described in the following sections.

Warning Box

```
CALL TD_Warn (message$, button$, default, result)
```

TD_Warn displays the message in `message$`. The `button$` string may contain text for up to four buttons, with vertical bars separating the buttons. The box remains on the screen until the user presses a button or until timeout occurs. (Note: it is not possible in this version of True BASIC to display special icons along with the warning message.) The message may contain up to ten lines with the vertical bars “|” separating the lines.

The DEMWARN.TRU program in TBDEMOS illustrates a simple warning box:

```
LIBRARY "c:\TBSilver\TBLIBS\TRUEDIAL.TRC     ! or appropriate path name

DO
   CALL TD_Warn ("message from ET", "Read it|Ignore it|Quit", 1, result)
   IF result = 3 then EXIT DO                             ! Quit
   IF result = 1 then                                     ! Show message
      LET title$ = "Here is a message from ET"
      LET message$ = "From outer space:|Hello, down there."
      CALL TD_Message ( title$, message$, "Again|Quit", 1, result)
      IF result = 2 then EXIT DO
   ELSE
      PAUSE 1
   END IF

LOOP
END
```

Run the DEMWARN.TRU program to see the warning box created by this code.

Message Box with Title

```
CALL TD_Message (title$, message$, button$, default, result)
```

TD_Message displays the message in `message$`. The dialog box is slightly larger than the one used for **TD_Warn**, and has a title bar. Again, the box remains on the screen until the user presses a button or until timeout occurs. As with **TD_Warn**, the message may contain up to ten lines; the vertical bar “|” is the line separator.

Note: On the MacOS, message boxes cannot have titles.

The DEMWARN.TRU example shown above uses a message box to print the message if requested

Yes-No Box

```
CALL TD_YN (message$, default, result)
```

TD_YN displays the message in `message$` along with two buttons, one labeled "Yes" (button 1) and the other "No" (button 2). The box remains on the screen until the user clicks on one of the boxes or until timeout occurs. As in **TD_Warn** and **TD_Message**, the message may contain up to ten lines, with multiple lines separated by vertical bars (|).

Yes-No-Cancel Box

```
CALL TD_YNC (message$, default, result)
```

TD_YNC displays the message in `message$` along with three buttons, one labeled "Yes" (button 1), the next "No" (button 2), and the last "Cancel" (button 3). The box remains on the screen until the user clicks on one of the boxes or until timeout occurs. As in **TD_Warn** and **TD_Message**, the message may contain up to ten lines, with multiple lines separated by vertical bars (|).

The program DEMYNC.TRU in TBDEMOS illustrates a yes-no-cancel box:

```
LIBRARY "c:\TBSILVER\TBLIBS\TRUEDIAL.TRC    ! or appropriate path name

DO
   CALL TD_YNC ("Do you want to quit?", 1, result)
   SELECT CASE result
   CASE 1                    ! Yes
      PRINT "Quitting"
      PAUSE 1
      EXIT DO
   CASE 2                    ! No
      PRINT "Continuing"
      PAUSE 3
   CASE 3                    ! Cancel
      Print "Canceling"
      PAUSE 1
   END SELECT
LOOP
END
```

Run the DEMYNC.TRU program to see the box created by this code.

Input Box

```
CALL TD_Input (message$, button$, text$, default, result)
```

TD_Input displays a one-line text field that may be edited; the initial and final values are in `text$`.

The program DEMINPUT.TRU in TBDEMOS illustrates this dialog box:

```
LIBRARY "c:\TBSilver\TBLIBS\TRUEDIAL.TRC   ! or appropriate path name

LET message$ = "Enter your name."
LET buttons$ = "OK|Cancel|Quit"

DO
   LET name$ = "  "                  ! Initially must be non-blank
   CALL TD_Input (message$, buttons$, name$, 1, result)
   SELECT CASE result
```

```
      CASE 1                           ! OK
         PRINT "You just entered: "; name$
         PAUSE 1
      CASE 2                           ! Cancel
         PRINT "You just canceled"
         PAUSE 1
      CASE 3                           ! Quit
         PRINT "You just quit"
         PAUSE 1
         EXIT DO
       END SELECT
   LOOP
   END
```

Run the DEMINPUT.TRU program to see the box created by this code.

Multiple Input Box

```
   CALL TD_InputM (title$, message$, button$, name$(), text$(), start, default, result)
```

TD_InputM displays a multiple-line set of text edit fields. The names of each line appear to the left of the editable portion and are in the string array `name$()`. The initial and final values of the text lines are in the string array `text$()`. `Start` specifies the line in which the editing cursor initially appears. The arrays `name$()` and `text$()` should have the same size; if they do not, the shorter one will be padded with blanks. Note: on the Macintosh, input boxes cannot have titles.

The program DEMINPTM.TRU in TBDEMOS illustrates a multiple-line input box.

Line Input Box

```
   CALL TD_LineInputM (message$, text$)
```

TD_LineInput displays a single-line input box with the message provided. There is but a single button – "OK". The returned text$ may consist of the null string.

File Open Box

```
   CALL TD_GetFile (type$, filename$, changedir)
```

TD_GetFile displays a typical file open dialog box. The list of file names displayed may be limited with the first argument. Unfortunately, this argument is used differently on different platforms. On Windows and OS/2, it specifies an extension (e.g., "tru") that may be used to limit the file names displayed; if `extension$` is the null string, all file names are displayed. The extension may be specified in lower- or uppercase, but the period (.) must not be included. On the Macintosh, the first argument specifies the Macintosh FILE TYPE for the file names to be displayed. The types "TEXT" and "TEXTTRUE" will result in all True BASIC files being displayed. The selected file name is returned in `filename$`. If `changedir` = 0, the user is not allowed to change directories in the course of searching for the desired file name. If `changedir` = 1, the user may change directories.

Save File Box

```
   CALL TD_SaveFile (type $, filename$)
```

TD_SaveFile displays a typical file save dialog box, which is similar to a file open box with an additional line containing the suggested file name. See the description of TD_GetFile, just above, for the user of the first argument to limit the file names displayed. The suggested file name is supplied in `filename$`, and the selected file name is returned in `filename$`. The user is allowed to change directories.

Selection List Box

```
   CALL TD_List (message$, button$, list$(), choice, default, result)
```

TD_List displays a scrollable list of choices, which are supplied in the array `list$()`. The number of the user's selection is returned in `choice`. The box remains on the screen until the user clicks on one of the buttons or until timeout occurs.

The DEMSLIST.TRU program in the TBDEMOS directory illustrates the selection list dialog box.

CHAPTER

15

Sound and Music

With True BASIC you can enhance your programs with a wide variety of sounds and music. The **PLAY** statement lets you play music using codes close to Western musical notation. The **SOUND** statement lets you generate a wider range of noises, with finer control over the output.

Using the PLAY Statement

You can play simple melodies on your computer with a statement such as:

```
PLAY melody$
```

where `melody$` is a special music string. Here's an example:

```
! Play "Amazing Grace"

LET a1$ = "t100 ml o4 d4 g2 b8 g8 b2"
LET a2$ = "a4 g2 e4 d4. r8"
LET a3$ = "d4 g2 b8 g8 b2 a8 mn >d8 d2. ml r2"
LET b1$ = "<b4 >d4. <b8 >d8 <b8 g2"
LET b2$ = "d4 e4. mn g8 g8 ml e8 d4. r8"
LET b3$ = "d4 g2 b8 g8 b2 a4 g2. r2"
LET m$  = a1$ & a2$ & a3$ & b1$ & b2$ & b3$
FOR times = 1 to 3
    IF times = 3 then                    ! Last time
       LET ln = Len(m$)
       LET m$[ln-1:ln] = "g2"            ! Repeat last note
    END IF
    PLAY m$
NEXT times

END
```

Normally, True BASIC waits for the melody to end before moving to the next statement. However, you can also play music "in the background" while computing continues if you use the MB option (which works on all versions except for Windows 3.x) described below.

The music string may contain codes for:

- the notes
- the lengths of notes
- the tempo
- whether to play in the foreground or the background

You may enter letters in either upper or lowercase, and you may insert spaces anywhere in the string to enhance readability.

The following table lists all the codes allowed in a music string for the **PLAY** statement:

Codes for PLAY Statement Music Strings

Notation	Meaning
A through G	Name of note
# or +	Sharp
–	Flat
R or P	Rest
O n	Octave number n
>	Next octave up
<	Next octave down
L n	Notes are length n
T n	Tempo n
ML	Legato
MN	Normal
MS	Staccato
MF	Play in foreground (default)
MB	Play in background (except on Windows 3.x)

Thus, `C#` represents a C-sharp and `B`   represents a B-flat.

Octaves start with C and end with B, and they are numbered 0 through 7. Middle C is the beginning of octave 4. If no octave if specified, the default is octave 5. You may specify a new octave either by O (the letter "oh") followed by the octave number, or by using > or < to move up or down an octave. In the example above, "Amazing Grace" is played in `O4` except for a few notes that are one octave higher in strings `a3$` and `b1$`.

A positive integer indicates the length of a note — 1 stands for a full note, 2 is a half note, 3 is a triplet, 4 a quarter note, 8 an eighth, etc. You may use the L code to specify the length of notes; for example, `L2` means that the following notes are half notes. Or you may attach the integer to the name of a note or rest, as in `A2` or `R4`. You may also use these two methods in combination. If you specify `L4`, notes that follow are quarter notes unless they are followed by an integer. You may also indicate a "dotted note," as in `A4.`, which multiplies the length by 3/2. If no length is indicated, the default is a quarter note, or `L4`.

You specify tempo with the letter T and an integer indicating the number of quarter notes to play in a minute. The default is `T120`, the standard speed of a metronome. To play the melody faster, increase the integer; to play slowly, decrease the integer. "Amazing Grace" uses the code `T100` to play a bit slower than standard speed.

You can also modify the way notes are played with ML, MN, and MS for legato, normal, and staccato. With "legato" mode, each note is played for the full length of time specified by the L code, which makes the melody sound slower and more sweeping. In "normal" mode a note plays 7/8 the specified time with a little break after each note to give crispness to the melody. In "staccato" mode notes are played to 3/4 of their length, making the melody quite brisk. The "Amazing Grace" example plays primarily in legato, but switches to normal mode for a few notes (see the strings `a3$` and `b2$`).

Finally, the MF code plays the melody in foreground (the default), while the MB code plays the melody in the background. When the melody is being played in the foreground, True BASIC executes no other statements until the **PLAY** statement is done. When the melody is being played in the background, other statements are executed while the melody is playing; the one exception is the execution of another **PLAY** statement. If you want to cut off the background melody at some point, include the statement:

```
SOUND 0, 0
```

Using the SOUND Statement

The **SOUND** statement is harder to use than **PLAY**, but gives you complete flexibility. For example, the statement:

```
SOUND 440, 10
```

plays concert A, which has a frequency of 440 Hertz, for 10 seconds. The **SOUND** statement requires two numeric values: the first specifies the frequency of the sound in Hertz, and the second gives the duration in seconds.

Very short sounds repeated rapidly may not be reproduced properly.

CHAPTER

16

Error Handling

If when compiling or running your program True BASIC encounters a problem it cannot handle, it stops the compilation or run and prints an error message identifying the type of error and where it occurred.

Errors that True BASIC detects during compilation before it begins to execute the code are called ***compile-time errors***. These are often caused by typing errors or statements that do not follow the rules described in previous chapters. The True BASIC Editor reports such errors in the debug window; the Language System prints compile-time errors in an output window. In both cases, True BASIC indicates the problem and line containing the error. "The True BASIC Environment" chapter in the introductory section describes how these errors are reported, and Appendix C includes explanations for these messages; refer to the appropriate sections in the manual for help on correcting the errors.

Errors that occur when a program is running are called ***run-time errors*** or ***exceptions***. When an exception occurs, True BASIC assigns it an error number and an error message. Some errors, such as division by zero, are ***fatal*** and will stop the program. True BASIC is able to continue the program after other, ***non-fatal errors***, such as incorrect user input.

You can add ***error handlers*** to your programs to prevent fatal errors from stopping your programs or to handle non-fatal errors your own way. The **WHEN** structure and built-in error functions let you intercept errors and error messages.

A program may also create its own specialized error, if, for example, it requires very specific input formats or if certain values must remain within a designated range. The **CAUSE** statement generates an error and assigns it an error number and an error message.

This chapter discusses True BASIC's built-in errors, the **CAUSE** statement for defining additional errors, and the built-in functions and **WHEN** structure that let you prevent errors from stopping your program or handle non-fatal errors your own way.

About Errors

Whenever a run-time error or exception occurs, True BASIC assigns it an error number and an error message. It also notes where the error occurred. Error handlers and error functions, described in the sections below, let you use this information in your programs. Appendices B and C list all the error numbers and messages generated by True BASIC.

If an error is ***fatal***, True BASIC stops the program and prints the error message in the Error Window. If the error occurs in a procedure outside the main program unit, True BASIC identifies both the offending line in the procedure and the line in the main program that invoked the procedure that caused the error. Examples of fatal errors include attempts to divide by zero, to calculate a number larger than the computer can handle, or to open a file that doesn't exist or is the wrong type You can use an error handler to intercept such errors and handle them in your own way as shown in the next section.

If the error is ***non-fatal***, the True BASIC corrects the error (or asks the user to correct the error) and continues. Most non-fatal errors are input mistakes. For those, True BASIC prints an error message and requires the user to re-enter the information. For other non-fatal errors, True BASIC makes an adjustment or uses a previous value and continues the program. (Both Appendixes B and C identify the non-fatal errors; the error-message explanations in Appendix C describe what happens after non-fatal errors.) As with fatal errors, you can intercept non-fatal errors with an error handler and handle them in your own way.

True BASIC's ***error numbers*** all have absolute values of 1000 or higher (some error numbers are negative). The numbers with absolute values of 1 through 999 are therefore available for you to use when generating specialized errors. Many of the True BASIC libraries also use the lower numbers. These numbers are helpful when you use an error handler to protect against errors. As you'll see in the sections below, you can use the error number to identify the type of error that occurred.

You might want to create errors specific to your program's function. For example, if your program plays a game in which certain moves are prohibited at certain times, the program could generate an error when the player attempts an illegal move:

```
...
INPUT nextmove
IF level < 4 and nextmove > 10 then
   CAUSE ERROR 100, "Moves greater than 10 prohibited below level 4"
END IF
....
```

With the **CAUSE** statement you must define an error number, and you may also define an error message. These errors are always fatal. In the simple example above, if the **CAUSE** statement is executed the program will stop at that line and display the defined error message. This ability to create errors is most helpful when you also use an error handler to cope with the error. The final section of this chapter illustrates the use of the **CAUSE** statement within **WHEN** structures to handle some very specific input requirements.

Using the WHEN Structure

The **WHEN** structure protects a block of code from errors and lets you specify what the program should do if an error occurs within that block. This process is called "handling an error," and for this reason the **WHEN** structure is often referred to as an ***error handler***.

Here's an example that shows a common use of an error handler. This subroutine opens a file. It asks the user for the name of a file and attempts to open it with the type and access specified by the calling program:

```
SUB FileOpen(org$, acc$, #9)       ! Protected file opener
    DO
       CLOSE #9                    ! In case file still open
       PRINT "File name";
       INPUT fname$
       WHEN ERROR IN
            OPEN #9: name fname$, org org$, access acc$
            EXIT SUB               ! Success
       USE
            PRINT "Cannot open that file."
       END WHEN
    LOOP
END SUB
```

The error handler starts with the line **WHEN ERROR IN** (or **WHEN EXCEPTION IN**), followed by the ***protected code***. The **USE** statement separates the block of protected code from the block of ***handler code***, in which you specify what to do in case of an error. The error handler must end with an **END WHEN** statement, which also serves to mark the end of the handler code. Normally, only the protected code is executed, but if an error occurs during the execution of the protected code, the program jumps to the line immediately following the **USE** statement and executes the handler code.

In this example, the **WHEN** structure protects the **OPEN** statement. If an error occurs, it is most likely because the file does not exist or is of the wrong type. Control then goes to the handler code, which prints a message. The program returns to the beginning of the loop and gives the user another chance. If the **OPEN** is successful, **EXIT SUB** is executed, exiting both the enclosing **DO** and **WHEN** structures.

The **WHEN** structure can help you identify non-fatal errors that True BASIC would normally handle itself. For example, if the argument to the **TAB** function is less than one:

```
PRINT TAB(-2); "Hello, out there."
```

True BASIC assumes the argument to the **TAB** function is 1, and the program continues. This could be the result of a programming error, however — especially if the argument to the **TAB** function is a variable calculated elsewhere. You can intercept such non-fatal errors by placing the potentially offending line in the protected part of a **WHEN** structure:

```
WHEN ERROR IN
   PRINT Tab(tabstart); "Value"
USE
   PRINT "Tab is set to"; tabstart
END WHEN
```

This will reveal when the value of `tabstart` is less than one.

Any statement, except for procedure definitions, may occur in the protected code. There can be calls to procedures, in which case any error occurring in the invoked procedure is also intercepted. However, if the called procedure has its own error handler, it can handle its own errors or it can “pass them up.” Passing up errors is discussed below.

The handler code may also consist of any block of code except for procedure definitions. This gives you great flexibility in handling errors. The next section describes two additional statements that are permitted only in the handler code of an error handler.

Using RETRY and CONTINUE

The handler code of an error handler may contain two special statements: the **RETRY** statement, which transfers control back to the statement that caused the error, and the **CONTINUE** statement, which transfers control to the statement that “logically follows” the statement that caused the error. These statements are not allowed outside an error handler.

As an example, suppose you don’t like True BASIC’s messages for faulty responses to a form of the **INPUT** statement (all of which are non-fatal errors). You can substitute your own message, as follows:

```
WHEN EXCEPTION IN
   INPUT age, ht, wt
USE
   PRINT "Enter your age, height, and weight,"
   PRINT "on the same line, separated by commas,"
   PRINT "as in '? 27,71.5,185'"
   RETRY
END WHEN
```

As another example, consider the **SET TEXT JUSTIFY** statement. If the program specifies an improper value for the **SET TEXT JUSTIFY** statement (see Chapter 13 “Graphics”), True BASIC normally ignores the improper value and uses the previous one. If you want the program to continue but not necessarily use the previous value, you could use the following:

```
WHEN ERROR IN
   SET TEXT JUSTIFY horiz$, vert$
   CALL Instructions
USE
   PRINT "Improper TEXT JUSTIFY values.  I'll center the text for you."
   PAUSE 2
   CLEAR
```

```
      SET TEXT JUSTIFY "center", "half"
      CONTINUE
   END WHEN
```

Using Error Functions

There are four functions that provide information about an error. These error functions let you work with the error number, error message, and information about the location of the error. They always refer to the most recent error. Since fatal errors stop the program unless intercepted by an error handler and non-fatal errors are ignored unless intercepted by an error handler, these error functions are generally used only in the handler code of an error handler.

The **EXTYPE** function returns the error number of the most recent error. It is 0 if no error has occurred. Knowing that the numbers of True BASIC's built-in errors have absolute values 1000 or higher and that errors in True BASIC libraries (and perhaps those you created) have error numbers with absolute values of 1 through 999, you may use the **EXTYPE** number to separate types of errors.

The **EXTEXT$** function returns the string that True BASIC would have printed as an error message. You might wish to have an error handler print this, showing the user what error occurred, even though the program will continue. If no error has yet occurred, the null string is returned.

The following example uses the **EXTYPE** and **EXTEXT$** functions to respond appropriately to either a True BASIC error or one defined by the program:

```
   DO
      WHEN ERROR IN
         INPUT PROMPT "Enter your next move: ": nextmove
         IF level < 4 and nextmove > 10 then
            CAUSE ERROR 100, "Moves greater than 10 prohibited below level 4"
         END IF
         EXIT DO                         ! Success; exit the handler
      USE
          IF Abs(Extype) < 1000 then      ! Program-defined error
             PRINT Extext$
             PRINT "To review the rules, press I,"
             PRINT "To re-enter your next move, press any other key."
             GET KEY k
             IF k = Ord("i") or k = Ord("I") then CALL Instructions
          ELSE                            ! True BASIC error
             PRINT Extext$
          END IF
      END WHEN
   LOOP
```

If the absolute value of the **EXTYPE** function indicates a program-defined error, the user gets the option of reviewing the rules for entering moves. No matter what the type of error, the program uses **EXTEXT$** to print the error message and then loops back to the **INPUT** statement.

The **EXLINE** function returns the line number where the error occurred. This number is either the sequential position of the line in the file containing the program unit, or, if the program uses line numbers, it is the line number of the offending line.

The **EXLINE$** function returns a detailed description of the location of the most recent error. The result of the **EXLINE$** function is a string that describes the "path" from the location of the intercepted error to the error handler that intercepted it. This path begins with the number of the line and the name of the procedure where the error occurred, followed by the name and line number of the procedure that invoked the procedure containing the error, and so on. Thus, by tracing the line numbers and subroutine names in this "lineage" you can trace the sequence of procedure calls that resulted in the error. If the total number of procedures involved in the lineage is greater than ten, only the first five and the last five will be listed.

Passing Errors

You can use several layers of nested error handlers to protect a given block of code. For example, the main program may protect some code that calls a subroutine. The subroutine may have its own error handler and may invoke a function that has its own error handler. If an error cannot be handled conveniently at a given level, it can be “passed up” to the calling procedure.

Any error that occurs in a procedure and is not handled there (i.e., is not contained in the protected part of a **WHEN** structure) is automatically passed back to the calling procedure. The error is then deemed to have occurred at the **CALL** statement that invoked the offending procedure. Thus, in the following example:

```
CALL Test
...
SUB Test
   LET x = 1/0
END SUB
```

error 3001 (division by zero) occurs at the **LET** statement but is not handled there. Therefore, the error is “passed back” to the **CALL** statement. If the **CALL** statement is contained within a **WHEN** structure, the error is handled there. If there is no **WHEN** structure and the **CALL** statement is itself within a procedure, the error is passed back to that calling procedure, and so on. If there is no higher error handler, an error results and the error message is printed.

You can also explicitly “pass up” an error with an **EXIT HANDLER** statement or with a **CAUSE ERROR** statement. The simplest method is to use the **EXIT HANDLER** statement within the handler code of an error handler.

For example, suppose you wish to intercept a potential overflow (calculation of a number larger than the computer can handle), but let the calling procedure handle any other errors:

```
SUB Calculate (a, b, c, result)
   WHEN ERROR IN
        LET result = a*b/c
   USE
        IF Extype = 1002 then     ! Overflow
           LET result = Maxnum    ! Make it very large
        ELSE
           EXIT HANDLER          ! Let someone else handle
        END IF
   END WHEN
END SUB
```

In place of the **EXIT HANDLER** statement, you could use a **CAUSE** statement to specify a number and message of your own to be passed up to the calling procedure:

```
SUB Calculate (a, b, c, result)
   WHEN ERROR IN
        LET result = a*b/c
   USE
        IF Extype = 1002 then          ! Overflow
           LET result = Maxnum         ! Make it very large
        ELSE
           CAUSE ERROR 888, "This calculation is not possible"
        END IF
   END WHEN
END SUB
```

Used inside the handler code, the **CAUSE** statement generates an error that is not intercepted by the error handler containing it. Instead, the error number specified in the **CAUSE** statement is assigned to the **EXTYPE** function, the error message specified in the **CAUSE** statement (if present) is assigned to **EXTEXT$**, and that information is passed back to the calling procedure. It is therefore up to the calling procedure to handle the error, or the error will be passed up to the next higher routine. This process continues until the error has been passed up as far as it can go (which is the main program), and if the error has still not been handled, it stops the program run.

Note that although the **CAUSE** statement may occur anywhere, the **EXIT HANDLER** statement may be used only in handler code blocks. Also, the **EXIT HANDLER** statement does not change the values of **EXTYPE** and **EXTEXT$**.

In either case, the error is passed up to the next higher error handler. For example, an error in a procedure might be passed up to the program unit that called the procedure. If there is no higher error handler, an error results and the current error message is printed.

Using Detached Handlers

WHEN structures normally have two parts. The part between the **WHEN** line and the **USE** statement is called the ***protected code***; and the part between the **USE** statement and the **END WHEN** statement is called the ***handler code***.

The handler code, however, may also be defined in a separate structure and given a name, like a subroutine. Such a structure is called a ***detached handler***, and it is defined using the **HANDLER** structure. With detached handlers, two or more protected parts may use the same handler code. Note, however, that the detached handler must be in the same program unit as the protected part that will use it. In this regard it is like an internal subroutine.

Consider these two **WHEN** structures:

```
WHEN ERROR IN
   OPEN #1: name infile$
USE
   PRINT "Can't open the file."
END WHEN
..
WHEN ERROR IN
   OPEN #2: name outfile$
USE
   PRINT "Can't open the file."
END WHEN
```

They could be changed to:

```
HANDLER CantOpen
   PRINT "Can't open the file."
END HANDLER
. . .
WHEN ERROR USE CantOpen
   OPEN #1: name infile$
END WHEN
...
WHEN ERROR USE CantOpen
   OPEN #2: name outfile$
END WHEN
```

You must be careful that the detached handler will properly handle all the errors that might be referred to it. For instance, in the above example the detached handler cannot print the name of the file that could not be opened, since the value of the offending file name is contained in two different variables.

Examples of Error Handlers

This section builds a library of utility routines, each useful in itself but also used by later routines. Each illustrates a strategy for error handling, and the last shows how to sort out various types of errors.

The first procedure converts a string into a number:

```
SUB Convert(n$,n)   ! Protected number converter
   WHEN ERROR IN
       LET n = Val(n$)
   USE
      CAUSE ERROR 100, "Not a number"
   END WHEN
END SUB
```

If `n$` does not represent a number, then the **VAL** function causes an error. The True BASIC error message about a "VAL string" would confuse the user who does not know that the program used the **VAL** function. Hence the subroutine intercepts it and issues a simple error message.

The next subroutine calls the `Convert` routine defined above and makes sure the number is an integer.

```
SUB Integer(n$,n)   ! Must be an integer
   CALL Convert(n$,n)
   IF n <> Int(n) then CAUSE ERROR 200, "Not an integer"
END SUB
```

Either error 100 (from `Convert`) or 200 could occur, and each error message is appropriate.

Finally these two subroutines could be used by an input routine that asks the user to type a fraction, such as "17/64", and returns the numerator and denominator. This routine intercepts any error, sends a relevant message, and gives the user another chance:

```
SUB Get_fraction(prompt$,n,d)           ! numerator, denominator
   DO
      WHEN ERROR IN
         PRINT prompt$;                 ! Prompt user
         LINE INPUT x$
         LET p = Pos(x$,"/")            ! Find /
         IF p = 0 then CAUSE ERROR 300
         CALL Integer(x$[1:p-1],n)      ! Numerator
         CALL Integer(x$[p+1:1000],d)   ! Denominator
         IF d = 0 then CAUSE ERROR 400
         EXIT SUB                       ! All ok
      USE
         IF Extype = 400 then
            PRINT "Denominator cannot be 0"
         ELSE
            PRINT "Type: integer / integer"
         END IF
         END WHEN
   LOOP
END SUB
```

The loop gives the user repeated chances if needed. A **WHEN** structure protects the body of the code. First, the **LINE INPUT** accepts any input. Next the routine looks for "/" and causes an error if there is no "/"; this error in turn causes the program to jump to the handler code, and the rest of the protected code is skipped. If "/" is found, the routine calls `Integer` twice, which in turn calls `Convert`. Either of these may cause an error and jump into the handler code. Finally, the routine causes an error if the denominator is zero.

The **EXIT SUB** statement is reached only if everything is correct. Since this is the only way out of the loop, the routine repeats until the user enters a legal fraction.

If an error is intercepted the handler code decides what error message to issue. Errors 100, 200, 300 all have to do with incorrect format, so they can use same message. Error 400, however, needs a different error message. Note that the first error detected throws the program into the handler code; only one error message can result.

CHAPTER

17 Constants, Variables, Expressions, and Program Units

This chapter defines concepts used in the rest of this manual. It uses a special notation to describe the correct grammatical use, or ***syntax***, of these terms. Higher-level concepts are defined in terms of lower-level concepts, which are defined in terms of still lower-level concepts, and so on. The lowest-level concepts are defined directly in keywords, letters, digits, or in English sentences. The names of the concepts appear in italics and may contain hyphens. (This approach to specifying the correct syntax is widely used in computing, and is sometimes called ***BNF***; the names of the concepts are sometimes called ***metanames***.)

Here is an example of how to read the notation, using the definition for ***signed-integer.***

signed-integer::	*integer*
	+integer
	–integer
integer::	*digit ...digit*

This may be read as: “A *signed-integer* consists of either an *integer*, or a plus sign (+) followed by an *integer*, or a minus sign (–) followed by an *integer*. The concept *integer*, used to define the concept *signed-integer*, is now defined as consisting of one or more *digits*.” (A *digit* is one of the ten characters “0”,“1”, ... “9”.)

Symbols Used in the Snytax

The Symbol	Is Read As
::	“consists of”
...	“followed by zero or more”

If the definition of a concept contains a list (as, for example, the list *integer*, *+integer*, *–integer* in the definition above), it means that the concept can be any one of the items in the list.

[!] **Note:** Do not confuse a double colon (::) that is part of the special notation, with a single colon (:) that appears in certain True BASIC statements. Also, do not confuse the ellipsis (...) with a decimal point (.), which can appear in numeric constants.

Constants

Constants are sequences of characters. They can be used to represent numbers, or they can merely represent themselves. That is, `123` may represent the number one hundred twenty three, in which case it is a *numeric constant*, or it may represent the character `"1"` followed by the character `"2"` followed by the character `"3"`, in which case it is a *string constant*. Usage determines which of the two cases is meant.

Constants can be used in *expressions*, in **DATA** statements, as responses to **INPUT** statements, and in True BASIC commands. In program statements, *numeric-constants* are *unquoted-strings* while *string-constants* must be *quoted-strings*. In **DATA** statements and **INPUT** responses, *string-constants* may be either *quoted-* or *unquoted-strings*, while *numeric-constants* must be *unquoted-strings*.

Numeric Constants

Numeric-constants are sequences of characters that represent numbers. The rules for forming *numeric-constants* are:

numeric-constant::	*unsigned-constant*
	+unsigned-constant
	–unsigned-constant
unsigned-constant::	*decimal-constant*
	decimal-constant exponent-part
decimal-constant::	*integer*
	integer.
	integer.integer
	.integer
exponent-part:	*E signed-integer*
	e signed-integer
signed-integer::	*integer*
	+integer
	-integer
integer::	*digit ...digit*

Spaces or commas are not allowed in *numeric-constants*. The E in *exponent-part*, which can be either upper case (E) or lower case (e), stands for "times ten to the power"; thus, 123.45e6, 1234.5e5, and 123450000 represent the same number: 123,450,000.

The above rules allow such constants as `1.e3` and `.4e5`, but do not allow `.e3`, `1e`, or `e1`. (The last will be construed to be a variable name if it appears unquoted in a program statement.) In other words, if there is a decimal point, there must be at least one digit either before or after the decimal point. If there is no decimal point, there must be at least one digit before the `e`. The *exponent-part* cannot contain a decimal point, and must contain at least one digit.

Unsigned-constants are used in numeric expressions, *signed-integers* are used in **OPTION BASE** and **DIM** statements, and *numeric-constants* are used in **DATA** statements and as responses to **INPUT** statements.

String Constants

String-constants are simply strings of characters. Like *numeric-constants*, they can be *quoted-strings* or *unquoted-strings*. The rules for forming *quoted-* and *unquoted-strings* are given below in English rather than BNF.

A *quoted-string* consists of zero or more characters surrounded by quote marks (" "). The quote marks surrounding the string are not part of the string, but all the characters inside the quote marks are part of the string. If there are no characters inside the quote marks, the *quoted-string* represents the ***null string***, i.e., a string containing no characters, not even a space.

Any printable characters from the ASCII character set (see Appendix A), except for a quote mark, can be placed inside the quote marks. To represent a quote mark in a *quoted-string*, it must be doubled. For example, in:

```
      "He said, ""Hello."""
```

the first and last quote marks merely surround the string. The second and third quote marks stand for a single quote mark. The fourth and fifth quote marks also stand for a single quote. If this *string-constant* were printed, the result would be

```
      He said, "Hello."
```

Similarly, the *quoted-string* """" stands for a single quote mark.

Quoted-strings can include the printable characters in the ASCII character set and can also include certain other characters and control characters as long as they do not have a special system meaning.

Unquoted-strings, which are used in **DATA** statements and as **INPUT** responses, can contain any of the printable characters in the ASCII character set except the comma (,) or quote mark ("). In addition, an *unquoted-string* used in **DATA** statements cannot contain an exclamation mark (!). Finally, neither the first nor the last character of an *unquoted-string* can be a blank space, although interior spaces can be included. Thus, *unquoted-strings* can't be null – you have to use the *quoted-string* ("").

If you wish to use a *string-constant* that contains one of the prohibited characters, or includes leading or trailing spaces, make it a *quoted-string*. For example, the *unquoted-string*:

```
        ab c
```

consists of four characters: the letter `a`, the letter `b`, one space, and the letter `c`; leading and trailing spaces are omitted. On the other hand, the *quoted-string*:

```
        "   ab c  "
```

consists of nine characters: three spaces, the letter `a`, the letter `b`, one space, the letter `c`, and two spaces.

Quoted-strings can appear in string expressions, as well as in **DATA** statements and **INPUT** statement responses. *Unquoted-strings* cannot appear in string expressions but can appear in numeric expressions if they represent *unsigned-constants*.

Quoted-strings and *unquoted-strings* can be used in **DATA** statements and in **INPUT** responses as strings. If the matching **READ** or **INPUT** variable is a string variable, then the *quoted-string* or *unquoted-string* is assigned to it. *Quoted-strings* that represent *numeric-constants* can be received by a *numeric-variable* in an **INPUT** statement, but *not* in a **READ** statement.

Identifiers

Identifiers are names that refer to items such as variables, arrays, and subroutines. They are defined as follows:

identifier::	*letter …ident-char*
ident-char::	*letter* *digit* *underline*

Thus, an *identifier* consists of a *letter* followed by any number of *letters*, *digits*, and *underlines*. *Letter* stands for an uppercase or lowercase letter, *digit* stands for one of the ten digits, and *underline* stands for the underline character (_).

String-identifiers are *identifiers* to which a final dollar sign ($) is attached. The formal definition is:

string-identifier::	*identifier$*

Identifiers name numeric variables and arrays, numeric defined functions, subroutines, pictures, modules, and programs. *String-identifiers* name string variables and arrays, and string defined functions.

Certain *identifiers* may not be used for certain purposes. Such identifiers are called ***reserved words***. The names of the no-argument numeric functions and array constants CON, DATE, EXLINE, EXTYPE, IDN, MAXNUM, PI, RND, RUNTIME, TIME, and ZER may not be used to name a simple numeric variable, a numeric array, or a numeric function. The names of the no-argument string functions and array constant DATE$, EXLINE$, EXTEXT$, TIME$ and NUL$, may not be used to name a simple string variable, a string array, or a string function. The keywords ELSE, NOT, PRINT, and REM may not be used to name a numeric variable, function, subroutine, or picture. Finally, if you use:

```
        OPTION NOLET
        DATA = 3
```

the `DATA = 3` statement will be treated as a **DATA** statement, not an assignment statement.

Expressions

There are three types of expressions: numeric expressions, string expressions, and logical expressions.

Numeric Expressions

Numeric expressions are formulas created from numeric variables, array elements, unsigned numeric constants, or numeric function values, together with the arithmetic operators +, –, *, /, ^, and parentheses.

The notation *numex* stands for numeric expression.

numex::	*term …addop term*
	addop term …addop term
addop::	+
	–

In other words, a numeric expression *numex* consists of one or more *terms* joined by + or – signs, and possibly starting with a + or – sign. As usual, + stands for addition while – stands for subtraction. For example, `1-2+3` is a *numex*.

term::	*factor …multop factor*
multop::	*
	/

Thus, a *term* consists of one or more *factors* joined by * or / signs. Here, * stands for multiplication while / stands for division. For example, 2*3/5 is a *term*. *Term* refers to *factor*, which we now define.

factor::	*primary …^primary*

A *factor* consists of one or more *primaries* joined with ^ signs. The sign (^) stands for "raised to the power." For example, `10^2` is a *factor*.

primary::	*unsigned-constant*
	numvar
	numeric-function
	numeric-function (arg …, arg)
	(numex)
arg::	*numex*
	strex
	numarr
	strarr

Therefore, a *primary* is either an unsigned numeric constant, a numeric variable, a numeric function value (the numeric function may or may not require arguments), or a *numex* contained within parentheses. Numeric functions may be either supplied by True BASIC or provided by the programmer through *defined-functions*. The names and argument types for the supplied functions are given in Chapters 8 and 18. The programmer may choose any *identifier* as the name of a *defined-function* as long as there is no conflicting use of that name.

numvar::	*simple-numvar*
	numarr (rnumex …, rnumex)
rnumex::	*a numex that is rounded before use*

In other words, a *numvar* is either a simple numeric variable or a numeric array element. A *rnumex* is a numeric expression that is rounded to the nearest integer before use. For example, if `a` is an array, `a(1.7)` is the same as `a(2)` and `a(3.3)` is the same as `a(3)`.

Finally, note that both *simple-numvars* and *numarrs* are denoted by *identifiers* .

The order of evaluation implied by the above rules is as follows:

- expressions inside parentheses are evaluated first,
- then exponentiations (^),
- then multiplications (*) and divisions (/),
- and finally additions (+) and subtractions (–).

Operators of the same level are evaluated from left to right. For example, `(7-4-2)` means `((7-4)-2) = 1` and not `(7-(4-2)) = 5`. Similarly, `(4^3^2)` means `((4^3)^2) = 4,096` and not `(4^(3^2)) = 262,144`.

As another example:

```
  3 + (-6 ^ 2 / 4 / 3) * (4 / 3 ^ 2 ^ 3 / 3 )
```

is evaluated in the order:

```
        6^2                     to yield        36
        36/4                    to yield        9
        9/3                     to yield        3
        -3                      to yield        -3(saved)
        3^2                     to yield        9
        9^3                     to yield        729
        4/729                   to yield        5.48695e-3
        5.48695e-3/3            to yield        1.82899e-3
        -3*1.82899e-3           to yield        -5.48695e-3
        3 + (-5.48695e-3)       to yield        2.99451
```

The evaluation of numeric expressions can lead to the following ***runtime errors***:

Exceptions:	1001	Overflow in numeric constant.
	1002	Overflow.
	2001	Subscript out of bounds.
	3001	Division by zero.
	3002	Negative number to non-integral power.
	3003	Zero to a negative power.

[!] Note: ***Runtime errors***, also called ***exceptions***, can occur only while a program is running and may depend on the data provided to the program. ***Syntax errors*** are caused by one or more lines disobeying the established grammar rules of True BASIC. A program containing syntax errors will not compile or run.

Other errors can arise from misuse of numeric functions. (See Chapter 18 and Appendix C.)

String Expressions

A string expression is a formula created from string variables, string array elements, quoted string constants, or string function values, together with the ampersand (&), substring extraction, and parentheses. (The ampersand stands for ***concatenation***, which involves joining two strings to create one longer string.)

strex:: *str-factor … & str-factor*

A string expression *strex* consists of one or more string factors joined by concatenation signs (&). When two strings are concatenated, the first character of the second string comes immediately after the last character in the first string.

str-factor:: *str-primary*
str-primary substrex

substrex:: *[rnumex : rnumex]*

A string factor *str-factor* consists of either a string primary, or a string primary followed by a substring expression *substrex*. You can use parentheses instead of square brackets in a *substrex*.

The string expressions `s$[a:b]` has a value consisting of the `a`-th through the `b-`th characters of the string `s$`.

For example:

```
LET a$ = "abcdefghijk"
! Then a$[3:7] = "cdefg"
```

If a or b falls outside the string, then these substitution rules apply:

```
! For s$[a:b]
LET ls = LEN(s$)          ! Length of the string
IF a < 1  then LET a = 1
IF a > ls then LET a = ls + 1
IF b < 1  then LET b = 0
IF b > ls then LET b = ls
```

Finally, if `b < a,` the null string results.

A string primary is defined as:

str-primar::	*quoted-string*
	strvar
	string-function
	string-function (arg …,arg)
	(strex)

In other words, a *str-primary* consists of a quoted string constant, or a string variable, or a string function value, or a string expression contained within parentheses. String functions may be either supplied by True BASIC or provided by the programmer through *defined-functions*. The names and argument types for True BASIC's supplied functions are given in Chapters 8 and 18. The programmer may choose any *string-identifier* as the name of a string *defined-function* as long as there is no conflicting use of that name.

A *strvar* is either a simple string variable or a string array element:

strvar::	*simple-strvar*
	strarr (rnumex …, rnumex)

Finally, note that both *simple-strvars* and *strarrs* are denoted by *string-identifiers*.

The order of evaluation implied by the above rules is as follows:

- string expressions inside parentheses are evaluated first
- then substring expressions, and
- finally concatenations (&).

For example, if `s$ = "abcdefghij"`, then `s$ & ("xyz" & s$)[5:10]` is evaluated in the order:

`"xyz" & s$`	to yield	`"xyzabcdefghij"`
`"xyzabcdefghij"[5:10}`	to yield	`"bcdefg"`
`s$ & "bcdefg"`	to yield	`"abcdefghijbcdefg"`

String variables and arrays may be given a maximum length in a **DECLARE** statement. An attempt to assign a string longer than that maximum will result in an exception. For example, after:

```
DECLARE STRING str$*10, str_array(10)*5
```

the following will cause exceptions:

```
LET strvar$ = "0123456789x"
LET str_array$(3) = "ABCDEF"
```

while:

```
LET strvar$ = "ABCDEF"
```

will not. Note that it is the maximum length that is fixed, not the actual length. In the last example, the length of `strvar$` is 6; not `10`.

You can find out the maximum string length of a particular string variable or array with the **MAXLEN** function. (See the **MAXLEN** function and the **DECLARE STRING** statement in Chapter 18.)

The evaluation of string expressions can lead to the following runtime errors:

Exceptions:	1051	String too long.
	1106	String too long in assignment.

Exception 1051 can occur only on computers whose operating systems limit the length of strings. Exception 1106 will occur if you attempt to assign a string that is too long for the string variable or array element that has been given a maximum length.

Other exceptions may arise from misuse of string functions. (See Chapter 18 and Appendix C.)

Logical Expressions

A logical expression, denoted *logex*, consists of a combination of relational expressions and special logical clauses, together with the logical operators AND, OR, NOT, and parentheses. A logical expression is one that takes on the value "true" or "false." (Logical constants and variables do not exist in True BASIC.)

The formal definition of *logex* is:

logex:: *log-term ... OR log-term*

In other words, a logical expression consists of one or more *log-terms* joined by OR. The *log-terms* are examined from left to right. As soon as a "true" *log-term* is found, no further *log-term* is evaluated and the *logex* as a whole is "true"; otherwise (if *none* of the *log-terms* is true), the *logex* as a whole is "false." A *logex* with one or more ORs is sometimes called a ***disjunction***.

A *log-term* is defined as:

log-term:: *log-factor ... AND log-factor*

A *log-term* consists of one or more *log-factors* joined by AND. The *log-factors* are examined from left to right. As soon as a "false" *log-factor* is found, no further *log-factor* is evaluated and the *log-term* as a whole is "false;" otherwise (if every *log-factor* is true), the *log-term* as a whole is "true." A *log-term* with one or more ANDs is sometimes called a ***conjunction***.

The process whereby *disjunctions* and *conjunctions* are evaluated from left to right but only far enough to determine truth or falseness is known as ***short-circuiting***.

The *log-factor* from which a *log-term* is built is defined as:

log-factor:: *log-primary*
NOT log-primary

A *log-factor* is either a *log-primary* or a *log-primary* preceded by NOT. This rule precludes more than one NOT in front of a *log-primary*. A *log-factor* without NOT is "true" if and only if the *log-primary* is true. A *log-factor* with NOT is "true" if and only if the *log-primary* is "false." A *log-factor* with NOT is sometimes called a ***negation***.

log-primary:: *relational-expr*
special-clause
(logex)

A *log-primary* consists of a *relational expression*, or a *special logical clause*, or a *logex* contained within parentheses. A *log-primary* is true if and only if the *relational-expr*, the *special-clause*, or the *logex* within parentheses is "true."

The order of evaluation for logical expressions implied by the above rules is as follows:

- expressions inside parentheses are evaluated first,
- then relational expressions and special logical clauses,
- then negations (NOT),
- then conjunctions (AND),
- and finally disjunctions (OR).

Relational expressions are formally defined as:

relational-expr::	*numex relop numex*
	strex relop strex

Where the allowed relational operators are:

relop::	<
	<= or =<
	=
	>= or =>
	>
	<> or ><

A numeric relational expression *numex relop numex* is "true" if and only if the order relation specified by the *relop* between the two *numexs* is "true." The usual real number ordering relation is used.

A string relational expression *strex relop strex* is "true" if and only if the order relation specified by the *relop* between the two *strexs* is "true."

The ordering relation for strings is determined by the numeric values of the ASCII code. (For those characters not in the ASCII code, the numeric values are determined by the operating system; see Appendix A.) When two strings are compared, the first character of each is examined. If the first character in the first string occurs earlier in the numeric code sequence than the first character in the second string, then the first string as a whole is considered less than (<) the second string. If the first character in the first string occurs later in the numeric code sequence than the first character in the second string, then the first string as a whole is considered greater than (>) the second string. If the first characters are the same, then the second characters are examined in a like manner. If the strings have the same number of characters and the characters are the same, position by position, the two strings are considered equal (=). If the first string is shorter than the second string, but the characters match up through the number of characters in the first string, then the first string is considered less than (<) the second string, and vice versa.

The logical *special-clause* is defined as:

special-clause::	END #*rnumex*
	MORE #*rnumex*
	END DATA
	MORE DATA
	KEY INPUT

END #*rnumex* is "true" if #*rnumex* refers to an opened file whose file pointer is at the end of the file, or if the file is empty. END #*rnumex* is "false" if the associated file pointer is not at the end of the file or if #*rnumex* refers to a window (#0 is always a window). See Chapter 12 for a discussion of file pointers.

MORE #*rnumex* is "true" if #*rnumex* refers to an opened file whose file pointer is not at the end of the file or if #*rnumex* refers to a window (#0 is always a window). MORE #*rnumex* is "false" if the associated file pointer is at the end of the file or if the file is empty.

END DATA is "true" if the *data-list* of the current invocation of the *program-unit* has become exhausted, and is "false" otherwise. (See later in this chapter for a definition of *program-unit*.)

MORE DATA is "true" if the *data-list* of the current invocation of the *program-unit* has not become exhausted, and is "*false*" otherwise.

For both *data-lists* and files, MORE is the same as NOT END, and END is the same as NOT MORE.

If the END clause is "false" or the MORE clause is "true," it is not necessarily true that additional input statements (such as INPUT and READ) can be executed without causing an exception. For example, with READ and DATA statements, the READ statement may contain two variables, but there may be only one data item remaining in the *data-list*. The same situation can arise with files.

KEY INPUT is “true” if there is at least one character in the keyboard input buffer that can be supplied to a GET KEY statement, and is “false” otherwise.

The evaluation of logical expressions can lead to the following runtime error:

Exceptions:	7004	Channel isn’t open.

I/O Recovery Clauses

There are two I/O recovery clauses that let you protect many types of input and output operations from missing data items and overwriting existing items. These can be used only as part of **READ** statements, file **READ** and **WRITE** statements, and file **INPUT** and **PRINT** statements, and their **MAT** statement equivalents.

if-missing::	IF MISSING THEN *action*
if-there::	IF THERE THEN *action*
action::	EXIT DO
	EXIT FOR
	line-number

If the *action* is **EXIT DO**, the input or output statement must be contained within a **DO** loop. If the *action* is **EXIT FOR**, the input or output statement must be contained within a **FOR** loop. If the *action* is a *line-number*, the action is a **GO TO** to that *line-number* and must follow the rules for valid **GO TO** statements. (See Chapter 18.)

If an *if-missing* is used in a data **READ** statement, the indicated *action* is taken if there is no data value for any one of the variables in the **READ** statement.

If an *if-missing* is used with a file statement, the indicated *action* is taken if there is no file record available at that point in the file. If an *if-there* is used with a file statement, the indicated *action* is taken if there is a file record available at that point in the file. The *action* is taken regardless of the contents of the file record.

The logical *special-clauses* interact with *if-missing* and *if-there* as follows. MORE DATA and END DATA detect if there is or is not at least one more data element available, whereas the IF MISSING clause attached to the **READ** statement is in effect if there are not enough data to satisfy all the variables in the **READ** statement. For example:

```
DO WHILE MORE DATA
   READ x
   DATA 1, 2, 3
LOOP
```

and

```
DO
   READ IF MISSING THEN EXIT DO: x
   DATA 1, 2, 3
LOOP
```

are equivalent; as soon as there are no more items in the data pool, the **DO** loop is exited.

In the first of the following examples, the **FOR** loop is exited when the data pool is exhausted because the MORE DATA clause checks only if there is a data element for the variable `a`. In the second example, the **DO** loop is exited when the data pool is exhausted because MORE DATA is checked before the IF MISSING clause is carried out.

```
FOR i = 1 TO 3
    DO WHILE MORE DATA
       READ IF MISSING THEN EXIT FOR: a, b
       DATA 1, 2, 3
    LOOP
NEXT I

FOR i = 1 TO 3
```

```
      DO WHILE MORE DATA
         READ IF MISSING THEN EXIT FOR: a, b
         DATA 1, 2
      LOOP
   NEXT i
```

Similar relationships hold for MORE #n, END #n, and the IF MISSING and IF THERE clauses attached to file statements.

Array Terms

Arrays are denoted by *identifiers* or *string-identifiers*. Other terms associated with the use of arrays are: *arrayparms*, *arrayargs*, *bowlegs*, *bounds*, and *redims*:

arrayparm::	*array bowlegs*
arrayarg::	*array*
	array bowlegs
array::	*numarr*
	strarr
bowlegs::	( )
	(, … ,)

Bowlegs, which consist of parentheses containing zero or more commas, tell True BASIC that the names to which they are attached are *arrays* having a certain number of dimensions. The number of commas in a *bowlegs* must be one less than the number of dimensions in the corresponding array. For example, a one-dimensional numeric array `a` is written `a()`, while a one-dimensional string array `a$` is written `a$()`; similarly, a three-dimensional numeric array `b` is written `b(,,)`, while a three-dimensional string array `b$` is written `b$(,,)`.

Arrayparms, or arrays with *bowlegs*, are used in **SUB**, **PICTURE**, **DECLARE PUBLIC,** and **DEF** statements. The *bowlegs* are required for those statements because there is no **DIM** or similar statement in the subroutine to tell True BASIC how many dimensions the array has. *Arrayargs*, or arrays with or without *bowlegs*, are used in **CALL** and **DRAW** statements, in function references, and with certain supplied functions such as **UBOUND**, **LBOUND**, and **SIZE**. The *bowlegs* are optional for those statements. *Bowlegs* are not allowed in **MAT** statements.

Suppose the subroutine `Total` takes two arguments, one a numeric array and the other a simple variable. Then the **CALL** and **SUB** statements might look like this:

```
   DIM x(50)
   ...
   ! The bowlegs are optional here
   CALL Total (x(), t)
   ...
   END

   ! The bowlegs are required here
   SUB Total (a(), b)
   ...
   END SUB
```

Bounds provide the initial dimensions of arrays in **DIM**, **LOCAL**, **PUBLIC**, or **SHARE** statements.

bounds::	*(bounds-range …, bounds-range)*
bounds-range::	*signed-integer*
	signed-integer TO *signed-integer*
	signed-integer : *signed-integer*

If the keyword TO or the colon (:) is present, the *bounds-range* establishes both the lower and upper bounds for a particular subscript. If TO and the colon are absent, the *bounds-range* establishes the upper bound for the subscript; the default lower bound currently in effect establishes the lower bound. If the upper bound is one less than the lower bound, the subscript range is empty, and the array has *no* elements. The upper bound is not allowed to be smaller than one less than the lower bound.

Redims establish new subscript ranges for arrays in **MAT READ**, **INPUT**, **REDIM** and similar statements.

redim::	*(redim-range …, redim-range)*
redim-range::	*rnumex*
	rnumex TO *rnumex*
	rnumex : rnumex

The number of *redim-ranges* must be the same as the original number of *bounds-ranges* for a particular array. That is, the number of dimensions, once set, cannot be changed.

The new ranges may be smaller or larger than the original ranges. If the keyword TO or the colon (:) is present, the *redim-range* establishes new lower and upper bounds for the subscript. If TO and the colon are absent, the *redim-range* establishes the new upper bounds for the subscript; the default lower bound currently in effect establishes the new lower bound. (The default lower bound is initially 1, but an **OPTION BASE** statement can change it to any desired value.) The upper bound is not allowed to be smaller than one less than the lower bound.

Exceptions:	5000	Out of memory.
	6005	Illegal array bounds.

Program Units

A True BASIC program consists of a main program together with any number of modules and external procedures. The precise definitions of *procedure*, *program-unit*, and *program* are as follows:

procedure::	*defined-function*
	subroutine
	picture

Thus, *procedure* stands for a *defined-function*, *subroutine*, or *picture*. A *procedure* that is inside a *program-unit* is said to be *internal*; otherwise, it is *external*. The *procedures* that form a *module* are *external*.

The term *program-unit*, used extensively in the rest of the manual, refers to either the *main-program* or an *external-procedure*.

program-unit::	*main-program*
	external-procedure

A *program unit* may contain *internal-procedures*, but an *internal-procedure* may not contain other *internal-procedures*.

A *program* consists of a *main-program* together with its associated *external-procedures* and *modules*. The *external-procedures* and *modules* can be contained in the *main-program* file or in any number of **LIBRARY** files. (The order in which the **LIBRARY** files are processed may dictate how the *external-procedures* and *modules* are distributed among the files.)

CHAPTER

18

Statements, Built-in Functions and Subroutines

This chapter describes all of True BASIC's built-in functions, subroutines, and statements, and is organized alphabetically.

The following ***built-in functions*** are covered. Function names that end with a dollar sign ($) are string-valued; that is, they yield values that are strings. The others are numeric-valued.

ABS	ACOS	ANGLE	ASIN
ATN	CEIL	CHR$	CON
COS	COSH	COT	CPOS
CPOSR	CSC	DATE	DATE$
DEG	DET	DOT	EPS
EXLINE	EXLINE$	EXP	EXTEXT$
EXTYPE	FP	IDN	INT
INV	IP	LBOUND	LCASE$
LEN	LOG	LOG10	LOG2
LTRIM$	MAX	MAXLEN	MAXNUM
MAXSIZE	MIN	MOD	NCPOS
NCPOSR	NUL$	NUM	NUM$
ORD	PI	POS	POSR
RAD	READPIXEL	REMAINDER	REPEAT$
RND	ROUND	RTRIM$	RUNTIME
SEC	SGN	SIN	SINH
SIZE	SQR	STR$	STRWIDTH
TAB	TAN	TANH	TIME
TIME$	TRIM$	TRN	TRUNCATE
UBOUND	UCASE$	UNPACKB	USING$
VAL	ZER		

The MAT functions and constants (CON, IDN, INV, NUL$, TRN, and ZER) can appear only in MAT assignment statements. TAB can appear only in **PRINT** statements. The picture transformations (SHIFT, SCALE, ROTATE, and SHEAR) can appear only in **DRAW** statements and are not included here. Other functions are available through libraries and are described in Chapters 22 "Interface Library Routines" and 23 "Additional Library Procedures."

The following ***built-in subroutines***, which must be invoked with **CALL** statements, are covered:

ADD_POSTSCRIPT	BEGIN_POSTSCRIPT	CLIPBOARD	COMLIB
COMOPEN	DIVIDE	END_POSTSCRIPT	OBJECT
PACKB	READCPIXEL	READ_IMAGE	SOCKET
SQL	STRWIDTH	SYS_EVENT	SYSTEM
WRITE_IMAGE	TBD	TBDX	

(ADD_POSTSCRIPT, BEGIN_POSTSCRIPT, END_POSTSCRIPT, SOCKET and SQL are available only in Gold Edition)

The following True BASIC statements are described:

ASK	BOX	BREAK	CALL
CASE	CAUSE	CHAIN	CLEAR
CLOSE	CONTINUE	DATA	DEBUG
DECLARE	DEF	DIM	DO
DRAW	ELSE	ELSEIF	END
ERASE	EXIT	EXTERNAL	FLOOD
FOR	FUNCTION	GET KEY	GET MOUSE
GET POINT	GOSUB	GOTO	HANDLER
IF IMAGE	INPUT	LET	LIBRARY
LINE INPUT	LOCAL	LOOP	MAT
MODULE	NEXT	ON GOSUB	ON GOTO
OPEN	OPTION	PAUSE	PICTURE
PLAY	PLOT	PRINT	PRIVATE
PROGRAM	PUBLIC	RANDOMIZE	READ
REM RESET	RESTORE	RETRY	RETURN
SELECT	SET	SHARE	SOUND
STOP	SUB	TRACE	UNSAVE
USE	WHEN	WINDOW	WRITE

(The statements LOCK and UNLOCK [often essential in data base programs] are available in the Gold Edition.)

Most of the above are single statements, such as **LET**. Several have numerous variations provided by additional keywords, such as the **ASK**, **MAT**, and **SET** statements. And several are the beginning keywords of multi-line structures, such as **FOR** and **DO**.

For some functions, subroutines, and statements, certain values may be illegal for arguments, causing a ***runtime error*** or ***exception***. The programmer can use a **WHEN** structure (see Chapter 16 "Error Handling") to "intercept" such errors and take corrective action. If no **WHEN** structure is present, the program will halt. We give the number of the exception (returned by the **EXTYPE** function) and the error message (returned by the **EXTEXT$** function) for each function that can generate an exception.

The evaluation of expressions may cause such exceptions as "Overflow (1002)" or "Division by zero (3001)." These exceptions are listed in Chapter 17, where numeric and string expressions are defined, and are omitted here.

The computation of some functions and subroutines may require additional memory that is not available. This chapter does not include this exception (5000). Nor does it include exceptions that may arise from the evaluation of the arguments of the function that are numeric or string expressions. See Chapter 17 for these errors.

The accuracy of the trigonometric and transcendental functions is at least 10 decimals; that is, the ***absolute error*** should be less than 10^(-10) in absolute value. (The TAN and EXP functions are computed to an accuracy of 10 significant figures; that is, their ***relative error*** should be less than 10^(-10) in absolute value.) If your computer has an arithmetic coprocessor, the accuracy will be that provided by the coprocessor.

Some of the statements deal with graphical input and output or the management of the terminal display screen. If used on a computer that does not offer graphical input and output, exceptions will occur.

In what follows, we use the following terms, which are defined in Chapter 17 "Constants, Variables, Expressions, and Program Units:"

numex	numeric expression
rnumex	rounded numeric expression
strex	string expression
redim	array redimensioning expression
arrayarg	array argument (array name with optional bowlegs)

ABS Function

ABS(*numex*)

Returns the absolute value of the argument.

ABS can be defined in terms of other True BASIC statements as follows:

```
DEF ABS(x)
    IF x<0 then
       LET ABS = -x
    ELSE
       LET ABS = x
    END IF
END DEF
```

ACOS Function

ACOS(*numex*)

Returns the value of the arccosine function. If OPTION ANGLE DEGREES is in effect, the result is given in degrees. If OPTION ANGLE RADIANS (default) is in effect, the result is given in radians.

For example, if OPTION ANGLE DEGREES is in effect, `ACOS(.5)` is 60; if OPTION ANGLE RADIANS is in effect, then `ACOS(.5)` is approximately 1.04720...

ACOS may be defined in terms of other True BASIC functions as follows:

```
DEF ACOS(x) = PI/2 - ASIN(x)
```

Exception:	3007	ASIN or ACOS argument must be between 1 and -1.

Add_Postscript Subroutine *available only in Gold Edition; see Chapter 27 of Gold manual.*

ANGLE Function

ANGLE(*numex*, *numex*)

`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). For example:

`ANGLE(1,1)`	= 45 degrees (`pi`/4 radians)
`ANGLE(0,1)`	= 90 degrees (`pi`/2 radians)
`ANGLE(1,0)`	= 0 degrees
`ANGLE(-1, 1)`	= 135 degrees (3*`pi`/4 radians)

ANGLE can be defined in terms of the ATN and SGN function as follows (assume OPTION ANGLE DEGREES):

```
DEF ANGLE(x,y)
   IF x > 0 then
      LET ANGLE = ATN(y/x)
   ELSE IF x < 0 and y <> 0 then
      LET ANGLE = ATN(y/x) + SGN(y) * 180
   ELSE IF x < 0 and y = 0 then
      LET ANGLE = 180
   ELSE IF y <> 0 then
       LET ANGLE = SGN(y)*90
   ELSE
      CAUSE EXCEPTION 3008, "Can't use ANGLE(0,0)."
    END IF
END DEF
```

Exception:	3008	Can’t use ANGLE(0,0).

ASIN Function

ASIN(*numex*)

Returns the value of the arcsine function. If the OPTION ANGLE is DEGREES, then the result is given in degrees. If the OPTION ANGLE is RADIANS (default), then the result is given in radians.

For example, with OPTION ANGLE DEGREES then `ASIN(.5)` is 30; with OPTION ANGLE RADIANS then `ASIN(.5)` is approximately .523549...

ASIN may be defined in terms of other True BASIC functions as follows:

```
DEF ASIN(x)
   IF abs(x) < 1 then
       LET Asin = Atn(x/Sqr(1 - x*x))
   ELSEIF x = 1 then
       LET Asin = 1
   ELSEIF x = -1 then
      LET Asin = -1
   ELSE
      CAUSE EXCEPTION 3007, "ASIN or ACOS argument must be between 1 and -1."
   END IF
END DEF
```

Exception: 3007 ASIN or ACOS argument must be between 1 and -1.

ASK Statements

A running program can get information about its current environment (for example, how many colors are available or how much free memory is left) through the ASK statements. Some ASK statements require a channel expression (which can refer to a file or a logical window), others forbid it, and a few can have it or not. For those that allow or require a channel expression, channel #0 always refers to the default logical window. If the channel expression refers to neither a file nor a logical window, then an exception occurs.

Exception: 7004 Channel isn't open.

ASK and SET work together. The program can ASK about any parameter that can be SET. The reverse is not true, as there are parameters beyond the control of the program. It is important to realize that ASK gives the actual values of the parameters, which may not necessarily be what previous SET statements assigned to them.

For the ASK statements that receive string information, the string variable may be followed by a substring expression. Such a variable is referred to as a *starget*, that is, a "string target."

starget::	*strvar*
	strvar substrex

If the *strvar* in an ASK statement is followed by a *substrex*, the appropriate keyword replaces the character positions given by the *substrex*.

For example, with:

```
ASK COLOR col$[1:2]
```

if the color is GREEN, that string will replace the characters in `col$` in position 1 through 2, and will incidentally lengthen the string `col$` by three characters (since the five-character word GREEN is replacing only two characters).

Here is an alphabetical list of the ASK statements; each is described below:

ASK ACCESS
ASK BACK
ASK COLOR
ASK COLOR MIX
ASK CURSOR
ASK DATUM

ASK DIRECTORY
ASK ERASABLE
ASK FILESIZE
ASK FILETYPE
ASK FREE MEMORY
ASK MARGIN
ASK MAX COLOR
ASK MAX CURSOR
ASK MODE
ASK NAME
ASK ORGANIZATION
ASK PIXELS
ASK POINTER
ASK RECORD
ASK RECSIZE
ASK RECTYPE
ASK SCREEN
ASK SETTER
ASK TEXT JUSTIFY
ASK WINDOW
ASK ZONEWIDTH

Several ASK statements may be combined:

ASK *io-ask-item …, io-ask-item*

io-ask-item:: MARGIN *numvar*
ZONEWIDTH *numvar*

ASK #*rnumex:* *file-ask-item …, file-ask-item* *file-ask-item::* ACCESS *starget*
DATUM *starget*
ERASABLE *starget*
FILESIZE *numvar*
FILETYPE *starget*
MARGIN *numvar*
NAME *starget*
ORGANIZATION *starget*
POINTER *starget*
RECORD *numvar*
RECSIZE *numvar*
RECTYPE *starget*
SETTER *starget*
ZONEWIDTH *numvar*

See the individual ASK statements that follow for the details of each *io-ask-item* or *file-ask-item*.

ASK ACCESS Statement

ASK #*rnumex*: ACCESS *starget*

Determines the access mode available to the file referred to by #*rnumex*, and assigns into *starget*:

INPUT	if the file is available only for input (input, read)
OUTPUT	if the file is available only for output (print, write)
OUTIN	if the file is available for both input and output (default), or if the channel is a logical window

ASK BACK Statement

ASK BACK *numvar*
ASK BACK *starget*
ASK BACKGROUND COLOR *numvar*
ASK BACKGROUND COLOR *starget*

ASK with BACK (or BACKGROUND COLOR) and *numvar* assigns to *numvar* the background color number of the current screen. If none has been set, -2 is the default. ASK with BACK (or BACKGROUND COLOR) and *starget* assigns into *starget* the background color name, in capital letters, of the current screen. If none has been set, "WHITE" is the default. If the current background color does not have a name, then the null string is assigned. (See also SET BACK for more details.)

ASK COLOR Statement

ASK COLOR *numvar*
ASK COLOR *starget*

ASK COLOR with *numvar* assigns to *numvar* the foreground color number of the current screen. If none has been set, -1 is the default. ASK COLOR with *starget* assigns into *starget* the foreground color name of the current screen. If none has been set, "BLACK" is the default. If the current foreground color does not have a name, then the null string is assigned. (See also SET COLOR)

ASK COLOR MIX Statement

ASK COLOR MIX (*rnumex*) *numvar*, *numvar*, *numvar*

Assigns to each *numvar*, respectively, the red, green, and blue components of the color whose color number is the value of *rnumex*. (See also SET COLOR MIX for more details.)

ASK CURSOR Statement

ASK CURSOR *starget*
ASK CURSOR *numvar, numvar*

ASK CURSOR with *starget* assigns into *starget* the cursor state in the current logical window:

ON	if the cursor is on, or if graphics is not available
OFF	if the cursor is off

ASK CURSOR with two *numvars* assigns to each *numvar*, respectively, the current line and column locations of the cursor in the current window. The cursor locations are in character coordinates. That is, the top line in the window is numbered 1, as is the left-most column. (See also SET CURSOR.)

ASK DATUM Statement

ASK #*rnumex*: DATUM *starget*

If #*rnumex* refers to a STREAM file, ASK DATUM assigns into *starget*:

NUMERIC	if the next item in the file is a number
STRING	if the next item in the file is a string
NONE	if there is no next item in the file
UNKNOWN	if the type of the next item cannot be determined.

For all other file types and organizations, and for logical windows, UNKNOWN is assigned.

ASK DIRECTORY Statement

ASK DIRECTORY *starget*

Assigns into *starget* the string of characters that defines the current directory being used for reading and writing files. (See Chapter 12 "Files for Data Input and Output" for details; also SET DIRECTORY.)

ASK ERASABLE Statement

ASK #*rnumex*: ERASABLE *starget*

Determines whether or not the file referred to by #*rnumex* can actually be erased and assigns into *starget*:

YES	if the ERASE statement can actually erase file elements
NO	in all other cases

ASK FILESIZE Statement

ASK #*rnumex*: FILESIZE *numvar*

Determines the size of the file referred to by #*rnumex*, and assigns to *numvar* the current number of records in a RANDOM or RECORD file and the current number of bytes in all other file types. If #*rnumex* refers to a logical window, 0 is assigned.

ASK FILETYPE Statement

ASK #*rnumex*: FILETYPE *starget*

Assigns into *starget*:

FILE	if #*rnumex* refers to a disk file
DEVICE	in all other cases

ASK FREE MEMORY Statement

ASK FREE MEMORY *numvar*

Assigns to *numvar* the number of bytes available in free memory. On virtual memory systems, may assign 256K.

ASK MARGIN Statement

ASK MARGIN *numvar*
ASK #*rnumex*: MARGIN *numvar*

Assigns to *numvar* the margin associated with the current logical window. If the channel expression is present and corresponds to a file or a logical window, assigns the current margin of a TEXT file or a logical window; for RANDOM, STREAM, RECORD, and BYTE files, 0 is assigned. (See also SET MARGIN.)

ASK MAX COLOR Statement

ASK MAX COLOR *numvar*

Assigns to *numvar* the maximum color number of foreground colors that may be shown at the same time. For example, if this number is 3, you may assign foreground colors in the range 1 through 3. (The maximum color number depends on the computer system being used.)

ASK MAX CURSOR Statement

ASK MAX CURSOR *numvar, numvar*

Assigns to each *numvar*, respectively, the maximum line (row) and character (column) positions to which the text cursor may be set in the current window.

ASK MODE Statement

ASK MODE *starget*

Assigns into *starget* the current screen mode, in uppercase letters, which will always be `"GRAPHICS"` in Version 5. This statement is provided for compatibility with earlier versions of True BASIC.

ASK NAME Statement

ASK NAME *starget*
ASK #*rnumex*: NAME *starget*

If the channel expression is present and refers to a file, the name of that file is assigned to *starget* . If the channel expression refers to a logical window, the null string is assigned.

The form without a channel expression is provided for compatibility with earlier versions of True BASIC. In this version, this form of the statement always assigns the null string into *starget*.

ASK ORGANIZATION Statement

ASK #*rnumex*: ORGANIZATION *starget*
ASK #*rnumex*: ORG *starget*

Determines the organization of the file referred to by #*rnumex*, and assigns into *starget*:

TEXT	if the file is a text file
STREAM	if the file is a stream file
RANDOM	if the file is a random file
RECORD	if the file is a record file
BYTE	if the file has been opened as a byte file
WINDOW	if #*rnumex* refers to a logical window

ASK PIXELS Statement

ASK PIXELS numvar, numvar

Assigns to each *numvar*, respectively, the number of pixels in the current window in the horizontal and vertical directions.

ASK POINTER Statement

ASK #*rnumex*: POINTER *starget*

Determines the pointer position of the file referred to by #*rnumex* and assigns into *starget:*

BEGIN	if the pointer is at the start of the file
END	if the pointer is at the end of the file, or if the file is empty
MIDDLE	if the pointer is not at the start or the end of the file, or if #*rnumex* refers to a logical window

(See also the SET POINTER statement.)

ASK RECORD Statement

ASK #*rnumex*: RECORD *numvar*

Assigns to *numvar* the current position of the file pointer of the file referred to by #*rnumex*. The units are records for RANDOM and RECORD files, bytes for TEXT and BYTE files, and 0 for STREAM files. For logical windows, -1 is assigned. (See also SET RECORD.)

ASK RECSIZE Statement

ASK #*rnumex*: RECSIZE *numvar*

Determines the record size parameters of the file referred to by #*rnumex* and assigns to *numvar* the record size, measured in bytes, for RANDOM, RECORD and BYTE files, and 0 for other file types. If none has yet been set, 0 is assigned. (See also SET RECSIZE.)

ASK RECTYPE Statement

ASK #*rnumex*: RECTYPE *starget*

Determines the record type of the file referred to by #*rnumex* and assigns into *starget:*

DISPLAY	if the file is a text file, or is a logical window, or is the printer
INTERNAL	if the file is of internal type, or is a device

ASK SCREEN Statement

ASK SCREEN *numvar*, *numvar*, *numvar*, *numvar*

Assigns to each *numvar*, respectively, the left, right, bottom, and top edges of the current logical window within its physical window. The values are in screen coordinates that range from 0 to 1 in both directions. (See also OPEN SCREEN.)

ASK SETTER Statement

ASK #*rnumex*: SETTER *starget*

Determines whether or not the record pointer of the file referred to by #*rnumex* can be set to any record and assigns into *starget*:

YES	if the file is a RANDOM or RECORD file
NO	in all other cases

ASK TEXT JUSTIFY Statement

ASK TEXT JUSTIFY *starget, starget*

Assigns into the first *starget* one of the values “LEFT” (default), “RIGHT”, or “CENTER” according to the current horizontal text position. Assigns into the second *starget* one of the values “TOP”, “BOTTOM”, “BASE” (default), or “HALF” according to the current vertical text position. (See also SET TEXT JUSTIFY.)

ASK WINDOW Statement

ASK WINDOW *numvar*, *numvar*, *numvar*, *numvar*

Assigns to each *numvar*, respectively, the left, right, bottom, and top edges of the current logical window. The values are in user coordinates, which are the ones used for PLOT statements, BOX statements, etc. (See SET WINDOW statement.)

ASK ZONEWIDTH Statement

ASK ZONEWIDTH *numvar*
ASK #*rnumex*: ZONEWIDTH *numvar*

Assigns to *numvar* the zonewidth of the current logical window. (When a comma appears in a PRINT statement, subsequent printing starts in the next print zone, which could be on a new line. The zonewidth is the number of character positions in each zone.) If the channel expression is present, this statement assigns to *numvar* the zonewidth of the TEXT file referred to by #*rnumex*. For BYTE, STREAM, RANDOM, and RECORD files, 0 is assigned. For logical windows, the zonewidth of the window is assigned. (See also SET ZONEWIDTH.)

ATN Function

ATN(*numex*)

`ATN(x)` returns the arctangent of *x*, which is the angle whose tangent is `x`. The angle will be given in radians or degrees according to whether the current OPTION ANGLE is RADIANS (default) or DEGREES. The angle will always be in the range -90 `< ATN(x) <` 90 (assuming that the current OPTION ANGLE is DEGREES).

For example:

`ATN(1)`	= 45 degrees (`pi`/4 radians)
`ATN(-1)`	= -45 degrees (-`pi`/4 radians)

Begin_Postscript Subroutine *available only in Gold Edition; see Chapter 27 of Gold manual.*

BOX Statements

All BOX statements except BOX SHOW use *boxcoords*.

boxcoords:: *numex*, *numex*, *numex*, *numex*

The values of the four *numex* refer, respectively, to the left edge, right edge, bottom edge, and top edge of the box, in user coordinates. The left edge need not be less than the right edge, or the bottom less than the top. The four coordinates will be taken simply as the coordinates of a rectangular region; the first two being *x*-coordinates and the second two *y*-coordinates.

For BOX AREA, BOX CIRCLE, BOX CLEAR, BOX ELLIPSE, and BOX LINES, only the part of the *box* within the current logical window is affected. Furthermore, BOX statements are unaffected by PICTURE transformations. This means that rectangles always look like rectangles of the same physical size, unless, of course, part of the rectangle extends beyond the current logical window and is therefore "cropped" or "clipped." Finally, all the BOX statements are graphics statements and cannot be used in a text-only mode.

On systems that do not support graphics, attempts to execute any of the BOX statements will cause an exception.

BOX AREA Statement

BOX AREA *boxcoords*

Draws the rectangle specified by the *boxcoords* and fills it with the current foreground color. The result is the same as using a BOX CLEAR and a BOX LINES statement, and then FLOODing the interior, but BOX AREA is faster.

BOX CIRCLE Statement

BOX CIRCLE *boxcoord*s

Draws an ellipse (or circle) inscribed in the rectangle specified by the *boxcoords* in the current foreground color. Whether or not the ellipse looks like a circle depends on the aspect ratio, which depends on the user coordinates, the screen coordinates, and the dimensions of the physical screen.

BOX CLEAR Statement

BOX CLEAR *boxcoords*

Clears the rectangular region specified by the *boxcoords*; that is, it fills that region with the current background color.

BOX DISK Statement

BOX DISK *boxcoords*

Draws an ellipse (or circle) inscribed in the rectangle specified by the *boxcoords* and fills it with the current foreground color. The result is the same as using a BOX CLEAR and a BOX CIRCLE or BOX ELLIPSE statement, and then FLOODing the interior, but BOX DISK is faster.

BOX ELLIPSE Statement

BOX ELLIPSE *boxcoords*

BOX ELLIPSE is the same as BOX CIRCLE.

BOX KEEP Statement

BOX KEEP boxcoords IN starget

Stores the entire rectangular region specified by the *boxcoords* into the *starget* in pixel form for subsequent use in a BOX SHOW statement. The format of the bits in the string is highly mode- and machine-dependent, and such strings are not portable. (See Chapter 13 for more details.)

If the rectangle is partly or completely outside the logical window, only that portion of the region within the window will be stored. The portion outside the logical window will be ignored. Assuming the user and box coordinates are not reversed, the BOX KEEP statement:

```
BOX KEEP l, r, b, t in a$
```

will have the same effect as:

```
ASK WINDOW lw, rw, bw, tw
BOX KEEP max(l,lw),min(r,rw),max(b,bw),min(t,tw) in a$
```

BOX LINES Statement

BOX LINES *boxcoords*

Draws the outline of a rectangle specified by the *boxcoords* in the current foreground color.

BOX SHOW Statement

BOX SHOW *strex* AT *numex*, *numex*
BOX SHOW *strex* AT *numex*, *numex* USING *showword*
BOX SHOW *strex* AT *numex*, *numex* USING *rnumex*

showword::	"AND"
	"OR"
	"XOR"

Note that *showword* must be one of the three *quoted-strings* shown, although the words can be spelled using lowercase or uppercase letters.

BOX SHOW restores the image stored in *strex* to the rectangular position whose lower left corner is specified by the pair of values that give, respectively, the *x*- and *y*-coordinates of the lower left corner of the rectangle, in user coordinates. The size of the rectangle, in pixels, is determined by the BOX KEEP statement.

The string expression will normally be the string variable or array element in which the image was previously stored using a BOX KEEP statement. (It is possible to build an image from scratch, but the details will differ from machine to machine and mode to mode, depending on the number of pixels on the screen, the number of colors, and so on.)

If the rectangle is partly or completely outside the logical window, only that portion of the image that is within the window will be shown.

If the USING clause is present, the bits of the "show string" will interact with the bits in screen memory as described in Chapter 13 "Graphics."

BREAK Statement

BREAK

If debugging is active for the program unit containing the BREAK statement, it causes an exception. Otherwise, the BREAK statement is ignored. This statement is included only for compatibility with the ANSI Standard. Its use is not recommended.

Exception:	10007	Break statement encountered.

CALL Statement

CALL *identifier*
CALL *identifier*(*subarglist*)

subarglist::	*subarg* …, *subarg*
subarg::	*numex*
	strex
	arrayarg
	#rnumex

The CALL statement invokes the *subroutine* given by the SUB statement with the same name. The subroutine can be internal if it is within the *program-unit* containing the CALL statement, or it can be external. If the subroutine is in a separate *module*, it must not be named in a PRIVATE statement in the *module-header*.

The arguments in the CALL statement are matched with the parameters in the SUB statement, the first with the first, and so on. The types of the parameters must correspond to the types of arguments as follows:

Examples of Argument/Parameter Relationship

Argument (CALL)	**Parameter (SUB)**
numex	*simple-numvar*
strex	*simple-strvar*
arrayarg	*arrayparm*
#rnumex	*#integer*

(The "#" signifies that the argument or parameter is a file reference number.)

A *numex* must match a simple numeric variable; a subscripted array element is not acceptable. The types and number of dimensions of arrays must match. (The distinction between *arrayarg* and *arrayparm* permits *bowlegs* to be optional in the CALL statement, although they are required in the SUB statement.)

Parameter passing is ***by reference***. That is, the variables and arrays in the SUB statement actually refer to the matching variables and arrays in the *program-unit* of the CALL statement. Changes to them in the subroutine will cause simultaneous changes to the variables in the calling *program-unit*.

If the argument is an expression that is not a simple-variable or array element, then that expression is evaluated and stored in a location not available to the programmer. This evaluation is done for each such argument, left to right. The parameters of the subroutine are then matched, by reference, to these private locations. Thus, `CALL S(x)` allows `S` to change `x`, while `CALL S((x))` does not. In this way, passing "by value" can be achieved.

Arrays are always passed to subroutines by reference. (In contrast, arrays are always passed to functions by value.)

Files and logical windows are also passed by reference. That is, any changes to them, or to their states, will be made to the corresponding files and windows in the calling *program-unit*.

Exception: 5000 Out of memory.

CASE Statement

The CASE statement can occur only as part of a SELECT CASE structure. See the SELECT CASE structure.

CAUSE Statement

CAUSE EXCEPTION *rnumex*
CAUSE EXCEPTION *rnumex*, *strex*
CAUSE ERROR *rnumex*
CAUSE ERROR *rnumex*, *strex*

The keyword ERROR is a synonym for the keyword EXCEPTION.

This statement causes a program-generated error (a runtime exception). The value of *rnumex* becomes the error number, which will be the value of a subsequent reference to the function EXTYPE. Like True BASIC's runtime errors, these errors can be intercepted with WHEN structures. It is recommended that you use error numbers in the range 1 through 999, as some of the numbers 1000 and above correspond to True BASIC's error messages. (See Appendix B for a complete list of these errors.)

If *strex* is present as the second argument and the error is intercepted by a WHEN structure, the value of *strex* can be obtained using:

EXTEXT$

If the error is not intercepted with a WHEN structure, then the value of *strex* followed by the value of *rnumex* in parentheses will be printed on the screen. If *strex* is absent, EXTEXT$ will return the null string.
The CAUSE statement can be used to generate errors in user defined functions, to pass control from a low-level subroutine back to a top-level WHEN structure, and for other purposes.

CEIL Function

CEIL(*numex*)

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

CEIL may be defined in terms of other True BASIC functions as follows:

```
DEF CEIL(x) = -INT(-x)
```

(See also INT, IP, and ROUND.)

CHAIN Statement

CHAIN *strex*
CHAIN *strex* WITH (*funarglist*)
CHAIN *strex*, RETURN
CHAIN *strex* WITH (*funarglist*), RETURN

funarglist::	*arg* …, *arg*
arg::	*numex*
	strex
	numvar
	strarr

The CHAIN statement stops the current program and starts the program in the file named in *strex*. If the target program is not accessible, an exception occurs.

If the WITH clause is present, and there is a PROGRAM statement with matching parameters as the first executable statement in the target file, the arguments will be passed to the corresponding parameters in the target program. The parameter passing mechanism is by value, the same as for defined functions. (See the PROGRAM statement.) If the PROGRAM statement does not have parameters, or they do not match in number and type, an exception occurs.

If the RETURN clause is missing, all storage associated with the first program is released, allowing the target program to occupy more memory. If the RETURN clause is present, the first program is retained; when the target program finishes, control is returned to the statement following the CHAIN statement in the first program, and that program continues.

If the string-expression after the word CHAIN begins with a `"!"`, the rest of it will be taken as a command to the operating system. If the string-expression begins with a `"!&"`, and a RETURN clause is present, the program will continue immediately; the command will be executed in background.

NOTE: On some Windows systems, `"!"` and `"!&"` behave effectively the same. There will still be a small difference, as the latter will return immediately, while the former will wait until the CHAINed application is launched before returning.

When the target program starts, all modules associated with it are initialized, even if the target program had been chained to previously. However, loaded modules are not re-initialized.

On personal computer systems, chaining from either source or compiled programs is permitted to either source or compiled targets. Chaining to or from executable (bound) programs is restricted. On Unix systems, chaining from either source or executable programs is permitted for either source or compiled targets; chaining from an executable program to a source program requires the location of the compiler to be known within the current directory.

A target program can itself chain to another program. This process can continue indefinitely, limited only by the amount of memory on your system.

If the RETURN clause is present, any runtime error (exception) that occurs in the target program but is not handled there by a WHEN USE structure will be sent back to the original program. If the target program is in source form and contains syntax errors, the exception number will be 10005 but the message will describe the actual error.

If the RETURN clause is absent, any runtime error in the target program not handled by a WHEN USE structure will be handled and reported by the system.

Exceptions:	4301	Mismatched parameters for CHAIN/PROGRAM.
	4302	Mismatched dimensions for CHAIN/PROGRAM.
	5000	Out of memory.
	10005	Program not available for CHAIN.

CHR$ Function

CHR$(*rnumex*)

Returns the character whose number is *rnumex*. If *rnumex* is not in the range 0 to 255, inclusive, then an exception is caused. See Appendix A for the ASCII characters that correspond to numbers in the range 0-255. For example:

`CHR$(65)`	= "A"
`CHR$(48)`	= "0" (the digit)
`CHR$(47.9)`	= "0" (the digit)
`CHR$(304)`	causes exception

Exception:	4002	CHR$ argument must be between 0 and 255.

CLEAR Statement

CLEAR

Clears the current logical window. The text cursor is reset to the (default) position, i.e. row 1, column 1. The position of the graphics cursor and/or mouse pointer is not affected.

CLIPBOARD Subroutine

CALL Clipboard (*strex*, *strex*, *strex*)

The Clipboard subroutine provides access to the system clipboard. The contents of the clipboard can contain text or images:

```
CALL CLIPBOARD (operation$, type$, item$)
```

`Operation$` must be one of `"GET"` or `"PUT"`.

`Type$` must be `"TEXT"`, `"PICT"`, or `""` (null).

`Item$` is the string that contains (or is to contain) the text or picture in the form of a BOX KEEP string.

The GET operation transfers the contents of the system clipboard to the string variable `item$`. The PUT operation places the contents of the string expression in the third argument onto the system clipboard, erasing the previous contents of the clipboard. If you PUT images of the PICT type, you need to supply a BOX KEEP string as the third argument of the routine. Or, if you GET images of the PICT type, you will receive a BOX KEEP string as the third argument of the routine.

NOTE: For Windows and OS/2, the native type of image that will be placed or expected on the clipboard will be a standard device-independent bitmap, which all PC paint programs should be able to handle easily.

On the MacOS, the native type of image that will be placed or expected on the clipboard will be the Macintosh PICT format.

The TEXT type is used for simple text. The PICT type is used for pictures or images, the format of which is system-dependent. The null type defaults to the TEXT type.

Example

```
CALL Clipboard ("PUT", "TEXT", string$)
```

will put the text that is in `string$` onto the system clipboard.

Exceptions:	-11210	Invalid option for SUB Clipboard.
	-11211	Invalid type for SUB Clipboard.
	-11212	Error opening clipboard for reading.
	-11213	Error closing clipboard.
	-11214	Error opening clipboard for writing.
	-11215	Error putting text onto clipboard.

See also: BOX CLEAR, BOX AREA

CLOSE Statement

CLOSE #*rnumex*

If the channel number refers to a file, this statement closes the file, allowing the channel number to be reused or the file to be reopened.

If the channel number refers to a logical window, the CLOSE statement closes the window and frees the channel number for reuse. It also eliminates the logical window's coordinate system, text cursor, margin, zone width, and so on; but it does not erase its current contents. If you close the currently active logical window, the default logical window (#0) becomes the currently active logical window.

Attempts to close the default logical window (#0) cause no action, but can be trapped as a nonfatal exception.

If the channel number is not associated with a currently open file or screen, no action occurs and no error results.

Exception:7002 Can't use #0 here. (nonfatal)

ComLib Subroutine

CALL ComLib (method, p1, p2, options$)

The ComLib subroutine provides access to the communications ports on your computer. It is a low-level subroutine and is documented in Chapter 22. Most users will find that the convenience subroutines found in ComLib.trc (source code in ComLib.tru) will be adequate for most purposes.

ComOpen Subroutine

CALL ComOpen (method, #1, port, speed, options$)

The ComOpen subroutines allows you to open and close a communications port on your computer. It is a low-level subroutine and is documented in Chapter 22. Most users will find that the convenience subroutines found in ComLib.trc (source code in ComLib.tru) will be adequate for most purposes.

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. The dimensions of the array of ones are determined in one of two ways. If the *redim* is present, then an array of those dimensions will be generated; the array being assigned to in the MAT assignment statement will be resized (see the MAT Assignment statement) for these new dimensions. If the *redim* is absent, then the dimensions will match those of the array being assigned to in the MAT assignment statement.

Exceptions:6005 Illegal array bounds.

(See also IDN, NUL$, and ZER.)

CONTINUE Statement

CONTINUE

The CONTINUE statement can appear only in the *handler-part* of a WHEN or HANDLER structure. If the line being executed when the exception occurred contains a statement (not a structure or loop), then CONTINUE transfers to the following line. If the line being executed is a required part of a loop or structure, then CON-

TINUE transfers to the line following the closing statement for that structure. That is, if the exception occurred in a DO statement or LOOP statement, CONTINUE will transfer to the line following the LOOP statement. Similarly, if the exception occurred in an IF line of an IF structure, an ELSEIF line, or an ELSE line, then CONTINUE transfers to the line following the END IF.

See the WHEN and HANDLER structures and the EXIT HANDLER and RETRY statements.

COS Function

COS(*numex*)

Returns the value of the cosine function. If the OPTION ANGLE is DEGREES, then the argument is assumed to be in degrees. If the OPTION ANGLE is RADIANS (default), then the argument is assumed to be in radians.

For example, if OPTION ANGLE DEGREES is in effect, then `COS(45)` is approximately 0.707107...; if OPTION ANGLE RADIANS is in effect, then `COS(1)` is approximately 0.540302...

COSH Function

COSH (*numex*)

Returns the value of the hyperbolic cosine function. For example, `COSH(1)` = 1.54308...

COSH may be defined in terms of other True BASIC functions as follows:

```
DEF COSH(x) = (EXP(x) + EXP(-x))/2
```

Exception:	1003	Overflow in numeric function.

COT Function

COT(*numex*)

Returns the value of the cotangent function. If OPTION ANGLE DEGREES is in effect, the argument is assumed to be in degrees. If OPTION ANGLE RADIANS (default) is in effect, the argument is assumed to be in radians.

For example, if OPTION ANGLE DEGREES is in effect then `COT(45)` is 1; if OPTION ANGLE RADIANS is in effect, then `COT(1)` is approximately .642093...

COT may be defined in terms of other True BASIC functions as follows:

```
   DEF COT(x) = 1/TAN(x)
```

Exception:	1003	Overflow in numeric function.

CPOS Function

CPOS(*strex*, *strex*)
CPOS(*strex*, *strex*, *rnumex*)

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 is the null string, 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.

For example:

`CPOS ("banana", "mno")`	returns 3
`CPOS ("banana", "pqr")`	returns 0
`CPOS ("banana", "mno", 4)`	returns 5
`CPOS ("banana", "mno", 10)`	returns 0

CPOS can be defined more precisely in terms of other True BASIC statements as follows:

```
   DEF CPOS(s1$,s2$,start)
       LET start = MAX(1,MIN(ROUND(start), LEN(s1$) + 1))
       FOR c = start TO LEN(s1$)
```

```
         FOR j = 1 to LEN(s2$)
            IF s1$[c:c] = s2$[j:j] THEN
               LET CPOS = c
               EXIT DEF
            END IF
         NEXT j
      NEXT c
      LET CPOS = 0
   END DEF
```

(See also POS, POSR, CPOSR, NCPOS, and NCPOSR.)

CPOSR Function

CPOSR(*strex*, *strex*)
CPOSR(*strex*, *strex*, *rnumex*)

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 is the null string, 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. For example:

`CPOSR ("banana", "mno")`	returns 5
`CPOSR ("banana", "pqr")`	returns 0
`CPOSR ("banana", "mno", 4)`	returns 3
`CPOSR ("banana", "mno", 10)`	returns 5

CPOSR can be defined more precisely in terms of other True BASIC statements as follows:

```
   DEF CPOSR(s1$,s2$,start)
      LET start = MAX(0,MIN(ROUND(start),LEN(s1$)))
      FOR c = start TO 1 STEP -1
         FOR j = 1 to LEN(s2$)
            IF s1$[c:c] = s2$[j:j] THEN
               LET CPOSR = c
               EXIT DEF
            END IF
         NEXT j
      NEXT c
      LET CPOSR = 0
   END DEF
```

(See also POS, POSR, CPOS, NCPOS, and NCPOSR.)

CSC Function

CSC(*numex*)

Returns the value of the cosecant function. If OPTION ANGLE DEGREES is in effect, the argument is assumed to be in degrees. If OPTION ANGLE RADIANS (default) is in effect, the argument is assumed to be in radians.

For example, if OPTION ANGLE DEGREES is in effect, then `CSC(45)` is approximately 1.41421...; if OPTION ANGLE RADIANS is in effect, then `CSC(1)` is approximately 1.18840...

CSC may be defined in terms of other True BASIC functions as follows:

```
DEF CSC(x) = 1/SIN(x)
```

Exception:	1003	Overflow in numeric function.

DATA Statement

DATA *datum* …, *datum*
datum:: *quoted-string*
unquoted-string

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. DATA statements in internal procedures will be added to the data list for the *program-unit* containing the internal procedure. There is a separate data list for each *program-unit*, and for each module header. In addition, the data list for an external procedure is supplied afresh to each invocation of that procedure.

If a *datum* is a *quoted-string*, leading and trailing spaces are included as part of the *datum*. If a *datum* is an *unquoted-string*, leading and trailing spaces are ignored. As with string constants, a null string is represented by a double quote (""). The surrounding quote marks of a *quoted-string* are not part of the *datum* that is added to the data list. If quote marks appear as part of the datum, they must be doubled.

The READ statements in the program unit assign to their variables the next available *datum*. If the variable is a string variable, then it receives the *datum* as is. If the variable is a numeric variable, then the *datum* must be an *unquoted-string* and must represent a *numeric-constant*.

When the data list has become exhausted, further READ statements will cause a runtime error (exception), unless an IF MISSING clause is present.

A RESTORE statement can be used to reset the data pointer to the beginning of the data list, allowing the data list to be reused. A RESTORE to line-number statement can be used to reset the data pointer to some intermediate point in the data list, provided the entire file is line-numbered.

(See also the READ and RESTORE statements.)

DATE Function

DATE

DATE, a no-argument function, returns the current date in the decimal numeric form YYDDD, where YY is the year (more exactly, 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. (Values of the DATE function can be sorted; that is, an earlier date will correspond to a smaller number.) For example,

For February 1, 1990, DATE returns	90032
For November 9, 1989, DATE returns	89313

DATE$ Function

DATE$

DATE$, a no-argument string-valued function, returns the current date in the character string form “YYYYMMDD” – here YYYY is the year, MM is the month number (from 1 to 12), and DD is the day number within the month (from 1 to 28, 29, 30, or 31). If your computer cannot tell the date, then DATE$ returns “00000000”. (Values of the DATE$ function can be sorted; that is, an earlier date will correspond to a string that occurs earlier in “alphabetical” order.)

For example:

For February 1, 1990, DATE$ returns	“19900201”
For November 9, 1988, DATE$ returns	“19891109”

DEBUG Statement

DEBUG ON
DEBUG OFF

The DEBUG ON statement, when executed in a *program-unit*, activates debugging for that *program-unit*. Debugging remains active until a DEBUG OFF statement is executed in that *program-unit*. Exiting and reentering a *program-unit* does not change the debugging status for that *program-unit*. If debugging is active in a *program-*

unit, the BREAK and TRACE statements have an effect; otherwise, they are ignored.

The DEBUG OFF statement, when executed in a *program-unit*, deactivates debugging for that *program-unit*.

This statement is included only for compatibility with the ANSI Standard. Its use is not recommended.

DECLARE Statements

The DECLARE statements are used to provide information on the identifiers — variables, defined functions, arrays, and subroutines — for the program. Some of them are required in certain situations, others are optional, and still others are ignored.

Here is an alphabetical list of the DECLARE statements.

```
DECLARE DEF
DECLARE FUNCTION
DECLARE NUMERIC
DECLARE PUBLIC
DECLARE STRING
DECLARE SUB
```

DECLARE DEF Statement

See the DECLARE FUNCTION statement.

DECLARE FUNCTION Statement

DECLARE FUNCTION *funname ..., funname*
DECLARE INTERNAL FUNCTION *funname ..., funname*
DECLARE EXTERNAL FUNCTION *funname ..., funname*

funname::	*identifier*
	string-identifier

(The keyword DEF is a synonym for the keyword FUNCTION and may be substituted for it.)

All external defined functions used in the given *program-unit* must be named in a DECLARE DEF, DECLARE FUNCTION, DECLARE EXTERNAL DEF, or DECLARE EXTERNAL FUNCTION statement that appears lexically before the first reference to that external defined function.

Internal defined functions whose definitions occur later in the *program-unit* than the first reference must be named in a DECLARE DEF, DECLARE FUNCTION, DECLARE INTERNAL DEF, or DECLARE INTERNAL FUNCTION statement that appears lexically before such first reference.

Other internal defined functions and external defined functions, including nonexistent ones, may also be named in DECLARE FUNCTION statements without adverse effect, except to preclude other uses of those names within the *program-unit*.

DECLARE NUMERIC Statement

DECLARE NUMERIC *numeric-dec ..., numeric-dec*

numeric-dec::	*simple-numvar*
	numarr bounds

The appearance of a *simple-numvar* in a DECLARE NUMERIC statement has no effect other than to preclude other uses of its name within the *program-unit*. The appearance of a *numvar* with *bounds* in a DECLARE NUMERIC statement has the same effect as if it appeared in a DIM statement. (Note: DECLARE NUMERIC used in an internal procedure does not make a *simple-numvar* LOCAL.)

DECLARE PUBLIC Statement

DECLARE PUBLIC *publicname ..., publicname*

publicname::	*simple-numvar*
	simple-strvar
	arrayparm

All public variables defined elsewhere but used in a *program-unit* must be named in a DECLARE PUBLIC statement that appears lexically before such use. Any variable or array named in a DECLARE PUBLIC statement must be defined (in some other *program-unit* or *module*) with a PUBLIC statement. The syntax and semantics for such variables is similar to the syntax and semantics for subroutine parameters. Simple variables, either numeric or string, or *arrayparms* (which must contain the *bowlegs*) can be used. The association with the public variable or array itself is ***by reference***; that is, a change to a public variable or array in any program unit is immediately reflected in all other program units referring to that variable or array.

DECLARE STRING Statement

DECLARE STRING *stringdec* …, *stringdec*

DECLARE STRING *length-max* *stringdec* …, *stringdec*

stringdec::	*simple-strvar*
	simple-strvar length-max
	strarr bounds
	strarr bounds length-max
length-max::	**integer*

If the DECLARE STATEMENT begins with a *length-max*, the effect is to force the string variables and arrays to have a maximum length (maximum number of characters) given by the *integer,* unless a particular *stringdec* also includes a *length-max,* which takes precedence. If a DECLARE STATEMENT does not begin with a *length-max*, the string variable and arrays will not have a maximum length, unless a particular *stringdec* contains a *length-max*.

The appearance of a *strarr* with *bounds* has the same effect as if it appeared in a DIM statement. (Note: DECLARE STRING used in an internal procedure does not make a *simple-strvar* LOCAL.)

A *length-max* for an array is applied to each individual string element in that array.

The appearance of a *simple-strvar* in a DECLARE STRING statement, both without a *length-max*, has no effect other than to preclude other uses of its name within the *program-unit*. (Note: DECLARE STRING used in an internal procedure does not make a *simple-strvar* LOCAL.)

For example, with:

```
DECLARE STRING a$, b$*10, c$(15), d$(5)*8
DECLARE STRING *10 e$, f$(25), g$*13, h$(20)*17

! Variable  Maximum length (characters)
!   a$        unlimited
!   b$        10
!   c$()      unlimited, for each element
!   d$()      8, for each element
!   e$        10
!   f$()      10, for each element
!   g$        13
!   h$()      17, for each element
```

(See also the MAXLEN function.)

DECLARE SUB Statement

DECLARE SUB *subname* …, *subname*
DECLARE INTERNAL SUB *subname* …, *subname*
DECLARE EXTERNAL SUB *subname* …, *subname*

subname::	*identifier*

The DECLARE SUB statement and it variations has no effect. (This statement serves no useful purpose in this version of True BASIC but is included for compatibility with ANSI.)

DEF Statement

The keyword DEF is a synonym for the keyword FUNCTION throughout the language. See the FUNCTION statement.

DEF Structure

The keyword DEF is a synonym for the keyword FUNCTION throughout the language. See the FUNCTION structure.

DEG Function

DEG(*numex*)

Returns the number of degrees in *numex* radians. This function is not affected by the current OPTION ANGLE. For example:

```
DEG(PI/2) = 90.
```

DEG can be defined in terms of other True BASIC statements as follows:

```
DEF DEG(x) = 180*x/PI
```

(See also PI and RAD.)

DET Function

DET (*numarr*)
DET

Returns the value of the determinant for the square numeric matrix named as its argument. For example, if `A` is:

$$\begin{bmatrix} 1 & 2 \\ 3 & 4 \end{bmatrix}$$

then `Det(A)` yields -2.

DET with no argument returns the value of the determinant of the matrix most recently inverted with the INV array function.

Exceptions:	1009	Overflow in DET or DOT.
	6002	DET needs a square matrix.

DIM Statement

DIM dimitem ..., dimitem

dimitem:: *numarr bounds*
strarr bounds

Except for function or subroutine parameters, each array in a *program-unit* must be dimensioned in a DIM statement (or a LOCAL, PUBLIC, or SHARE statement) that occurs lexically before the first reference to that array.

DIM is not executable; instead array storage is created when the containing program-unit is invoked.

The actual bounds for an array may be changed later by a MAT REDIM statement or a MAT statement with a *redim*. The changed ranges for the subscripts can be larger than the original ranges, but the number of dimensions must be the same.

Exception:	5000	Out of memory.

DIVIDE Subroutine

CALL DIVIDE (numex, numex, numvar, numvar)

`CALL DIVIDE (dvend, dvsor, q, r)` divides `dvend` by `dvsor` to give quotient `q` and remainder `r`. More specifically, `q` and `r` are solutions to `dvend = dvsor*q + r`, where `q = INT(dvend/dvsor)` and `r = MOD (dvend, dvsor)`. For example:

```
! DIVIDE with these arguments    ! Returns
CALL DIVIDE (2.5, 1.5, q, r)     ! q = 1, r = 1
CALL DIVIDE (1, 10, q, r) ! q = 0, r = 1
CALL DIVIDE (7, 3, q, e)  ! q = 2, r = 1
CALL DIVIDE (-7, 3, q, r) ! q = -3, r= 2
CALL DIVIDE (7, -3, q, r) ! q = -3, r = -2
CALL DIVIDE (-7, -3, q, r)       ! q = 2, r = -1
```

Exceptions:	1002	Overflow.
	3001	Division by zero.

DO Loop Structure

do-loop::	*do-statement*
	. . .
	loop-statement
do-statement::	DO
	DO WHILE *logex*
	DO UNTIL *logex*
loop-statement::	LOOP
	LOOP WHILE *logex*
	LOOP UNTIL *logex*

When the DO statement is reached, the next statement to be executed will be the first statement inside the loop if (1) the DO statement has no WHILE or UNTIL condition, (2) the WHILE condition is present and *logex* is "true," or (3) the UNTIL condition is present and *logex* is "false." Otherwise, the next statement to be executed will be the first statement after the associated LOOP statement.

When the LOOP statement is reached, the next statement to be executed will be the associated DO statement if (1) the LOOP statement has no WHILE or UNTIL condition, (2) the WHILE condition is present and *logex* is "true," or (3) the UNTIL condition is present and *logex* is "false." Otherwise, the next statement to be executed will be the first statement after the LOOP statement.

In other words, the WHILE condition keeps the loop going if *logex* is true, and the UNTIL condition keeps the loop going if *logex* is false.

If an EXIT DO statement is encountered while executing the statements inside the loop, the next statement to be executed will be the first statement following the LOOP statement. Such an EXIT DO statement must be lexically contained within the loop.

The action of the WHILE and UNTIL clauses on the DO and LOOP statements can be obtained with an IF statement containing an EXIT DO statement, as follows:

Forms of DO LOOP

```
This Form:              Is Equivalent to this Form:
DO WHILE logex          DO
...                     IF NOT logex THEN EXIT DO
...                     ...

DO UNTIL logex          DO
...                     IF logex THEN EXIT DO
...                     ...

...                     ...
...                     IF NOT logex THEN EXIT DO
LOOP WHILE logex        LOOP
```

```
        ...                    ...
        ...                    IF logex THEN EXIT DO

        LOOP UNTIL logex       LOOP
```

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.

For example, if `A = (1 2 3)` and `B = (4 5 6)`, then `DOT(A,B)` will return `1*4+2*5+3*6 = 32`.

Exceptions:	1009	Overflow in DET or DOT.
	6001	Mismatched array sizes.

DRAW Statement

DRAW *identifier*
DRAW *identifier* (*subarglist*)
DRAW *identifier* WITH *transform*
DRAW *identifier* (*subarglist*) WITH *transform*

transform::	*trans-item* ... * *trans-item*
trans-item::	SCALE (*numex*)
	SCALE (*numex*, *numex*)
	ROTATE (*numex*)
	SHIFT (*numex*, *numex*)
	SHEAR (*numex*)
	numarr

See the CALL statement for definitions of *subarglist* and *subarg*.

The DRAW statement causes the picture whose name is *identifier* to be drawn on the screen, just as if the DRAW statement were replaced by the code of the picture definition. The DRAW statement and the associated PICTURE definition are like the CALL statement and the associated SUB definition. The picture named in the DRAW statement can consist entirely of non-graphics statements, in which case it acts exactly like a subroutine.

If the *subarglist* is present, the rules are identical to those for subroutines, i.e., the parameter passing mechanism is ***by reference***. (See the CALL statement for a more complete discussion.)

If the WITH clause is present, then the transformation specified in the WITH clause is applied to PLOT, FLOOD, and MAT PLOT statements (but *not* BOX statements) in the picture before drawing it. If a picture contains DRAW statements also with WITH clauses, then the final transformation is the "product" of the transformations along the way. Similarly, the inverse transformation is applied to the point determined by a GET POINT or GET MOUSE statement. The transformation consists of shifts, rotations, shears, or changes of scale, or any sequence thereof.

The transformation applied to the picture can be represented by a four-by-four matrix. When a graphics statement is executed using coordinates (`x, y`), the *transform* matrix is pre-multiplied by a row vector (`x`, `y`, 0, 1). The first two elements of the resulting row vector are the transformed coordinates. The first two rows and columns of the *transform* matrix correspond to the x- and y-coordinates. The fourth row and column provides for "homogeneous coordinates," allowing shifts to be represented by matrix multiplications. The third row and column corresponds to the z-coordinate, which is not currently used.

A *trans-item* can consist of any four-by-four numeric matrix. Its effect will be determined by matrix multiplication, just as with the four named *trans-items* SCALE, ROTATE, SHIFT, and SHEAR.

SCALE with two arguments causes the x-coordinates of the picture to be scaled by a factor of the first argument and the y-coordinates to be scaled by a factor of the second argument. For example, `SCALE(2,1)` will turn a circle into an ellipse, elongated in the x-direction. 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 around the origin of the coordinate system by an amount equal to *numex*. The angle is measured in radians unless OPTION ANGLE DEGREES is in effect, in which case the angle is measured in degrees.

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. The two arguments will be added to the x- and y-coordinates, respectively.

SHEAR causes the picture to be tilted *clockwise* through an angle given by the argument. That is, SHEAR leaves horizontal lines horizontal, but tilts *vertical* lines through the specified angle. The angle is measured in radians unless OPTION ANGLE DEGREES is in effect, in which case the angle is measured in radians.

More precisely, `SHEAR(a)` causes the new x- and y-coordinates to be related to the old x- and y-coordinates as follows:

```
LET xnew = xold + yold*Tan(a)
LET ynew = yold
```

If there are several *trans-items*, they are applied left to right. For example:

```
DRAW square WITH SHIFT(2,0) * ROTATE(45)
```

will first move the picture of the square 2 units to the right (the x-direction), and then rotate the entire scene 45 degrees (assuming OPTION ANGLE DEGREES to be in effect) counterclockwise about the origin. Contrast this with:

```
DRAW square WITH ROTATE(45) * SHIFT(2,0)
```

which will first rotate the square about the origin to produce a diamond-shaped object, and then will move that object 2 units to the right.

On systems that do not support graphics, executing a DRAW statement will not cause an exception, as long as the picture it refers to does not contain any graphics statements.

These transforms can be represented as four-by-four numeric matrices:

Graphics Transformations (DRAW)

Function	Transformation
`SHIFT(a,b)`	Translates (x,y) to (x+a,y+b). Returns: $\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ a & b & 0 & 1 \end{pmatrix}$
`SCALE(a,b)`	Scales (x,y) to (a*x,b*y). Returns: $\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & b & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$
`SCALE(a)`	Scales (x,y) to (a*x,a*y). Returns: $\begin{pmatrix} a & 0 & 0 & 0 \\ 0 & a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$

`ROTATE(a)` Rotates the picture counterclockwise about the origin of the coordinate system by angle `a`.

Returns: $$\begin{pmatrix} \cos(a) & \sin(a) & 0 & 0 \\ -\sin(a) & \cos(a) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

`SHEAR(a)` Shears non-horizontal lines to lean to the right by angle `a`, that is, by mapping (x,y) into (x+y*tan(a),y).

Returns: $$\begin{pmatrix} 1 & 0 & 0 & 0 \\ \tan(a) & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

The ROTATE and SHEAR functions work with arguments in radians unless OPTION ANGLE DEGREES is in effect, in which case they use degrees.

Exception: 6001 Mismatched array sizes.

ELSE Statement

The keyword ELSE can appear only as part of an IF statement or an IF structure. See the IF statement and the IF structure.

ELSEIF Statement

The ELSEIF statement can appear only as part of an IF structure. ELSEIF may also be spelled ELSE IF. See the IF structure.

End_Postscript Subroutine *available only in Gold Edition; see Chapter 27 of Gold manual.*

END Statements

END statements are used to end the main program, all procedures, and several structured constructs. They are, alphabetically:

END
END DEF
END FUNCTION
END HANDLER
END IF
END MODULE
END PICTURE
END SELECT
END SUB
END WHEN

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. The program can also be stopped with the STOP statement. (See also the STOP statement.)

END DEF Statement

The END DEF statement is the same as the END FUNCTION statement.

END FUNCTION Statement

The END FUNCTION statement can appear only as the last statement of a multi-line defined function and is required. See the FUNCTION structure.

END HANDLER Statement

The END HANDLER statement can appear only as the last statement of a HANDLER structure and is required. See the HANDLER structure.

END IF Statement

The END IF statement can appear only as the last statement of an IF structure and is required. See the IF structure.

END MODULE Statement

The END MODULE statement can appear only as the last statement of a module and is required. See the MODULE structure.

END PICTURE Statement

The END PICTURE statement can appear only as the last statement of a picture and is required. See the PICTURE structure.

END SELECT Statement

The END SELECT statement can appear only as the last statement of a SELECT structure and is required. See the SELECT structure.

END SUB Statement

The END SUB statement can appear only as the last statement of a subroutine and is required. See the SUB structure.

END WHEN Statement

The END WHEN statement can appear only as the last statement of a WHEN structure and is required. See the WHEN structure.

EPS Function

EPS(*numex*)

`EPS(x)` returns the smallest positive number that can "make a difference" when added to or subtracted from x. More precisely, `EPS(x)` is `MAX(x x', x"-x, sigma)`, where `x'` is the immediate predecessor of `x` (in floating point representation), `x"` is the immediate successor of `x`, and `sigma` is the smallest positive number that can be represented, which is given by `EPS(0)`.

For example, on an IBM-compatible PC, without a numeric coprocessor, `EPS(1e13)` = 1.953125e-3 and `EPS(0)` = 2.2250739e-308. In other words, if a value is 1e13 (10^13), then the smallest amount that can change that value through addition or subtraction is 1.953125e-3. Similarly, the smallest positive number that can be represented on such a machine is 2.2250739e-308 (2.2250739 * 10^(-308).)

ERASE Statement

ERASE #*rnumex*
ERASE REST #*rnumex*

If *rnumex* refers to a file opened with access OUTIN, the ERASE statement erases the contents of a file. None of the attributes associated with the file are changed. For example, the organization of the file (TEXT, RECORD, etc.) is not affected. However, certain attributes can be changed; a PRINT statement to an empty file forces its organization to be changed to TEXT. Similarly, if the file is a RECORD file, the record size remains, but can be changed since the file is now empty.

If *rnumex* refers to a file opened with access OUTIN, the ERASE REST statement erases the contents of the file from the item or record currently pointed to through the end of the file. (This statement may be impossible to execute on certain systems, in which case an exception will be generated. Even when possible, this statement may require excessive time or storage space.)

If *rnumex* refers to a logical window, the ERASE statement clears the window.

Exceptions:	7002	Can't use #0 here. (nonfatal)
	7004	Channel isn't open.
	7301	Can't ERASE file not opened as OUTIN.
	9100	Can't open temporary file.

EXIT Statements

EXIT statements are used to exit from loops and procedures other than at their ends. They are, alphabetically:

EXIT DEF
EXIT DO
EXIT FOR
EXIT FUNCTION
EXIT HANDLER
EXIT PICTURE
EXIT SUB

EXIT DEF Statement

EXIT DEF

The EXIT DEF statement is the same as the EXIT FUNCTION statement. See the EXIT FUNCTION statement.

EXIT DO Statement

EXIT DO

The EXIT DO statement can appear only within a DO loop. If this statement is encountered during the execution of a DO loop, the next statement to be executed will be the one following the LOOP statement of the inner-most DO loop containing the EXIT DO. For example, in:

```
DO
   ...
   DO
      ...
      EXIT DO
      ...
   LOOP
   REM end of inner loop
   ...
LOOP
REM end of outer loop
```

the EXIT DO statement will cause a jump to the REM statement at the end of the inner loop.

EXIT FOR Statement

EXIT FOR

The EXIT FOR statement can appear only inside a FOR loop. If this statement is encountered during the execution of a FOR loop, the next statement to be executed will be the one following the NEXT statement of the inner-most FOR loop containing the EXIT FOR. For example, in:

```
FOR i = 1 to 20
    ...
    FOR j = 5 to 10
        ...
```

```
            EXIT FOR
            ...
        NEXT j
        ...
    NEXT i
```

the EXIT FOR statement will cause a jump to the statement immediately following `NEXT j`. (After the inner FOR loop has been exited, the value of `j` can be examined. If `j` is equal to 11 [the first value not used] the loop was exited normally; if `j` is in the range 5 to 10, the loop was exited through the EXIT FOR statement.)

EXIT FUNCTION Statement

EXIT FUNCTION

The EXIT FUNCTION statement can appear only inside a multi-line defined function. When this statement is encountered during the execution of a defined function, the result is as if the END FUNCTION statement had been reached.

EXIT HANDLER Statement

EXIT HANDLER

The EXIT HANDLER statement can appear only in the *handler-part* of a WHEN or HANDLER structure.

The EXIT HANDLER statement causes the exceptions to “recur” as if the WHEN structure were not present. The values of EXTYPE, EXLINE, EXLINE$, and EXTEXT$ are not changed by the EXIT HANDLER statement.

EXIT PICTURE Statement

EXIT PICTURE

The EXIT PICTURE statement can appear only inside a picture definition.

The EXIT PICTURE statement causes a jump to the END PICTURE statement of the innermost picture that contains it.

EXIT SUB Statement

EXIT SUB

The EXIT SUB statement can appear only inside a subroutine definition. The EXIT SUB statement causes a jump to the END SUB statement of the innermost subroutine that contains it.

EXLINE Function

EXLINE

A no-argument function, 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, e.g., the 17th.

EXLINE$ Function

EXLINE$

A no-argument function, EXLINE$ returns a string giving the location in your program where the most recent error occurred. If an error has occurred, EXLINE$ gives the erroneous line and the routine in which the error occurred. If the program has line numbers, these are used to identify the line. If the error occurred several levels deep in nested subroutine calls, EXLINE$ returns the genealogy of the error, except that only the first five and the last five subroutines are given. If the program does not have line numbers, then EXLINE$ assumes that the first line is 1, the second 2, and so on. In this case, the line numbers refer to a file, which may contain several routines, and not to individual routines within the file. If no error has occurred since the program was started or chained to, EXLINE$ gives the null string.

EXP Function

EXP(*numex*)

Returns the *natural* exponential of the argument. That is, `EXP(x)` calculates `e^x`, where `e` = 2.718281828..., the base of the natural logarithms. For example, `EXP(0)` = 1 and `EXP(1)` = 2.718281828....

Exception: 1003 Overflow in numeric function.

EXTERNAL Statement

EXTERNAL

The EXTERNAL statement, when occurring prior to any procedure declaration in a library file, designates all the procedures in that file as being ***external***. If the EXTERNAL statement is absent, the compiler will assume that the procedure definitions are ***internal*** and part of a main program; it may then complain about the lack of an END statement.

The keyword EXTERNAL can also appear in front of any SUB, FUNCTION (or DEF), or PICTURE statement that begins an external subroutine, defined function, or picture. If all the subroutines, defined functions, and pictures in a particular file contain the keyword EXTERNAL as described here, then the initial keyword EXTERNAL in the file need not appear.

EXTEXT$ Function

EXTEXT$

A no-argument function, EXTEXT$ returns the error message associated with the most recent error or CAUSE EXCEPTION statement, provided that the error was *trapped* by an error handler (see Chapter 16 “Error Handling”). If no error has occurred, then EXTEXT$ returns the null string. If an error is not trapped by an error handler, then the True BASIC system prints the error message and stops the program.

EXTYPE Function

EXTYPE

A no-argument function, EXTYPE returns the error number of the most recent error, provided that the error was *trapped* by an error handler (see Chapter 16 “Error Handling”). If the error was not trapped by an error handler, then the True BASIC system prints the error message (see EXTEXT$) and stops the program. If no error has occurred since the program was started or chained to, EXTYPE returns 0.

True BASIC error numbers lie in the range 1000 and up in absolute value. (See Appendix B for a complete list.) Numbers in the range 1 to 999 are therefore available for your use. (See the CAUSE ERROR statement.)

FLOOD Statement

FLOOD *numex*, *numex*

FLOOD will fill, with the current foreground color, the closed graphical region containing the point whose x-coordinate is the first *numex* and whose y-coordinate is the second *numex*, in user coordinates.

The closed region consists of the pixel identified by the *x*- and *y*-coordinates and all adjacent pixels in the horizontal and vertical directions that have the same value (i.e., color, if there is more than one bit in the pixel) and so on. FLOOD sets the pixels so identified to the current color. FLOOD does not change the value (color) of pixels that have a different value from the original point.

Normally, FLOOD is used to fill a region surrounded by a closed boundary. If this boundary has openings, the flooding will seep through and may extend to the edges of the logical window.

You should also be aware that if the color on the screen is a dithered color, FLOOD will not work correctly. Colors need to be solid (realizing them if necessary) for FLOOD to work correctly.

If colors are being used with the FLOOD statement on a machine or in a graphics mode that does not allow colors, unexpected results may occur. The reason is that colors in such cases may be represented by patterns whose boundaries are not necessarily closed. The flooding will then seep through the gaps in the boundary and extend beyond the intended region.

[!] **NOTE: FLOOD will not work correctly if the color on the screen is dithered. Dithered colors can sometimes be made solid by "realizing" the palette.** *(See TC_Win_Realize Palette in Chapter 22.)*

FOR Loop Structure

for-loop::	*for-statement*
	. . .
	NEXT *simple-numvar*
for-statement::	FOR *simple-numvar* = *numex* TO *numex*
	FOR *simple-numvar* = *numex* TO numex STEP *numex*

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 FOR loop may be described in terms of other statements as follows:

```
FOR v = initialvalue TO limit STEP increment
   ...
NEXT v
is equivalent to
LET own1 = limit
LET own2 = increment
LET v = initialvalue
DO UNTIL (v - own1)*(0.5 + SGN(own2)) > 0
   ...
   LET v = v + own2
LOOP
```

Here, `v` is a simple numeric variable (not a numeric array element), and `own1` and `own2` are variables associated with the particular FOR loop and not available to the programmer.

The reason for the `(0.5 + SGN(own2))` is as follows: When the step size is 0, the loop is infinite if the limit is larger than or equal to the initial value but is executed zero times if the limit is smaller than the initial value.

If the STEP clause is missing, the increment is 1.

Upon normal exit from the FOR loop (i.e., other than through the EXIT FOR statement) the value of the FOR variable is *the first value not used*. The following examples illustrate the various cases:

```
FOR i = 2 TO 3
   ...
NEXT i
! i is now equal to 4
FOR i = 6 TO 3 STEP -2
   ...
NEXT i
! i is now equal to 2
FOR i = 3 TO 1
   ...
NEXT i
! i is now equal to 3, since 3 was not used
FOR i = 2 TO 3 STEP -1
   ...
NEXT i
! i is now equal to 2, since 2 was not used.
```

(See also the EXIT FOR statement.)

FP Function

FP(*numex*)

Returns the fractional part of the argument. For example, `FP(1.9) = .9, FP(-1.3)` = -.3, and `FP(-17)` = 0.

FP can be defined in terms of the IP function as follows:

```
DEF FP(x) = x - IP(x)
```

FUNCTION Statement

FUNCTION *identifier* = *numex*
FUNCTION *identifier* (*funparm* …, *funparm*) = *numex*
FUNCTION *string-identifier* = *strex*
FUNCTION *string-identifier* (*funparm* …, *funparm*) = *strex*

funparm::	*simple-numvar*
	simple-strvar
	arrayparm

The keyword FUNCTION may be replaced by the keyword DEF.

The FUNCTION statement allows the programmer to define new one-line functions.

The arguments in the invocation are matched with the parameters, the first argument with the first parameter, and so on. Each argument must match the corresponding parameter in type, numeric or string. Arrays must agree in the number of dimensions. The arguments are evaluated in left to right order and assigned to the corresponding *funparms*. (This parameter passing mechanism is called ***passing by value*** and contrasts with ***passing by reference***, the mechanism used with subroutines and pictures. It should be noted that passing an array ***by value*** requires that it be copied completely before use, which can be time- and memory-consuming for large arrays.)

The expression on the right is then evaluated. If the defined function is internal, the variables and arrays that appear in the expression, but that are not *funparms*, refer to variables and arrays in the containing *program-unit*. If the defined function is external, then all such variables and arrays that are not shared will have their default initial values (i.e., 0 or null).

A FUNCTION statement can be replaced by a FUNCTION structure. For example, the following two definitions of the function *f* are equivalent.

```
FUNCTION f(a,b) = expr

FUNCTION f(a,b)
    LET f = expr
END FUNCTION
```

(See also the FUNCTION structure.)

FUNCTION Structure

function-structure::	*funopener*
	. . .
	END FUNCTION
funopener::	FUNCTION *identifier*
	FUNCTION *identifier* (*funparmlist*)
	FUNCTION *string-identifier*
	FUNCTION *string-identifier* (*funparmlist*)
funparmlist::	*funparm* …, *funparm*
funparm::	*simple-numvar*
	simple-strvar
	arrayparm

The keyword FUNCTION may be replaced by the keyword DEF.

The FUNCTION structure allows the programmer to define new multi-line functions.

The arguments in the invocation are matched with the parameters, the first argument with the first parameter, and so on. Each argument must match the corresponding parameter in type, numeric or string. Arrays must agree in the number of dimensions. The arguments are evaluated in left to right order and assigned to the corresponding *funparms*. (This parameter passing mechanism is called ***passing by value*** and contrasts with ***passing by reference***, the mechanism used with subroutines and pictures. It should be noted that passing an array ***by value*** requires that it be copied completely before use, which can be time- and memory-consuming for large arrays.)

The statements of the function are then executed. The defined function is assigned its value through the execution of one or more LET statements with the name of the function as a LET variable. If no such LET statement is executed, then the value of the function will be the default initial value (i.e., 0 or null).

If the defined function is internal, the variables and arrays that appear in the expression, but that are not *funparms*, refer to variables and arrays in the containing *program-unit*. If the defined function is external, then all such variables and arrays that are not shared will have their default initial values (i.e., 0 or null).

The defined function can also contain DECLARE PUBLIC, DECLARE DEF, LOCAL, SHARE, and PUBLIC statements.

GET KEY Statement

GET KEY *numvar*
GET KEY: *numvar*

The GET KEY statement assigns to *numvar* the numerical equivalent of the next character in the keyboard input buffer. If no character is in the buffer, the program waits until the user presses a key.

If the character is an ASCII character, the numerical equivalent of that character is assigned to *numvar* (see Appendix A). Otherwise, the value assigned to *numvar* depends on the particular machine.

The logical clause KEY INPUT can be used in conjunction with the GET KEY statement. If there is a character in the input buffer, then KEY INPUT is “true,” and the first character will be assigned to *numvar* by the GET KEY statement without delay.

GET MOUSE Statement

GET MOUSE *numvar*, *numvar*, *numvar*
GET MOUSE: *numvar*, *numvar*, *numvar*

The GET MOUSE statement returns the current position of the mouse and its current state. On mice with multiple buttons, only the state and position of the left-most button is reported. The current *x*- and *y*-coordinates of the current position, in user coordinates, are assigned to the first two *numvars*, respectively. If one or more picture transformations are in effect, their inverse is applied before assigning to the variables. The current state of the mouse is assigned to the third *numvar* according to the following table:

Mouse State

Value	**Mouse state**
0	No button Down
1	Button Down
2	Button clicked at this point
3	Button released at this point
4	Button shift-clicked at this point

The values assigned to the two variables may be outside the current logical window; they are not “clipped” and are given in terms of the current user coordinates.

The state of the mouse is determined within a certain time interval that is dependent on the particular machine. Thus, even if the GET MOUSE statement is executed continuously, as within a tight loop, it may nonetheless miss the exact moments when the button is being clicked or released.

GET MOUSE is provided primarily for compatibility with earlier versions of True BASIC. This version of the language provides more reliable and flexible means of getting mouse input via the Sys_Event routine (see Chapter 20 “Sys_Event Subroutine”) or the TC_Event library routine (see Chapter 14 “Interface Elements”).

GET POINT Statement

GET POINT *numvar*, *numvar*
GET POINT: *numvar*, *numvar*

When the GET POINT statement is executed, the program will stop to allow the user to indicate a point on the screen, in current user coordinates. A graphics cursor indicates the current point. The form of the graphics cursor depends on the system; for instance, it may be small cross hairs or an arrow. The user is then allowed to move the graphics cursor via cursor keys, the mouse, or other means depending on the particular machine.

When the new location is correct, the user signals by pressing the RETURN key or the left-most mouse button, or by some other means; the x- and y-coordinates of the graphics cursor are assigned to the two *numvars* respectively, and the program is continued. If one or more picture transformations are in effect, their inverse is applied before assigning to the variables.

The values assigned to the two variables may be outside the current logical window; they are not “clipped” and are given in terms of the current user coordinates.

GET POINT is provided primarily for compatibility with earlier versions of True BASIC. This version of the language provides more reliable and flexible means of getting input via the Sys_Event routine (see Chapter 20,“Sys_Event Subroutine”) or the TC_Event library routine (see Chapter 14 “Interface Elements”). The GET POINT statement will not work if program-controlled event handling (via Sys_Event or TC_Event) is in effect.

GOSUB Statement

GOSUB *line-number*
GO SUB *line-number*

The GOSUB statement causes a jump to the *line-number* specified. It also places the number of the line following the GOSUB statement on the top of a RETURN stack. A subsequent RETURN statement will jump to the line whose number is at the top of this stack. There is a separate return stack for each invocation of a program-unit.

The GOSUB statement can appear only in a line-numbered program. The target *line-number* must be within the scope of the GOSUB statement and must contain an allowable statement.

Exception:	5000	Out of memory.

GOTO Statement

GOTO *line-number*
GO TO *line-number*

The GOTO statement causes a jump to the *line-number* specified.

The GOTO statement can appear only in a line-numbered program. The target *line-number* must be within the scope of the GOTO statement and must contain an allowable statement.

HANDLER Structure

handler-structure	HANDLER *handler-name* *handler-part* END HANDLER
handler-name::	*identifier*
handler-part::	… *statement*

The HANDLER structure, a ***detached handler***, is used in conjunction with the WHEN EXCEPTION USE form of the WHEN structure and must be located within the same program-unit. (The HANDLER structure may be viewed as an *internal-procedure*.)

When an exception occurs in the *when-part* of a WHEN EXCEPTION USE structure, control transfers to the HANDLER structure named in the WHEN EXCEPTION USE line.

The effect of the statements in the *handler-part* is the same as if the statements were in the USE part of a WHEN EXCEPTION IN structure.

If an EXIT HANDLER or CAUSE EXCEPTION statement is not executed and the END HANDLER statement is reached, control is transferred to the line following the END WHEN line of the WHEN structure that invoked the handler. The action is as if the handler were a subroutine and the END WHEN line were a CALL to that subroutine.

An error (exception) that occurs while the statements of the HANDLER structure are being executed will be treated as a new error.

An EXIT HANDLER statement will cause the exception to "recur" as if the WHEN structure that caught the exception were not present. A RETRY statement will transfer to the statement being executed when the exception occurred. A CONTINUE statement will transfer to the statement following the statement being executed when the exception occurred, unless the offending statement is an essential part of loop or choice structure, when the transfer will be made to the statement following the end of the choice structure. See the WHEN structure, EXIT HANDLER, RETRY, and CONTINUE statements.

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. The dimensions of the identity matrix are determined in one of three ways. If the *redim* is present and represents a square matrix, then an array of those dimensions will be generated. If the *redim* represents a one-dimensional matrix, then IDN will generate a square matrix with the given *redim* applying to both dimensions. In these two cases, the array being assigned to in the MAT assignment statement will be resized (see the MAT Assignment statement) to these new dimensions. If the *redim* is absent, then the dimensions will match those of the array being assigned to in the MAT assignment statement.

For example:

$$\texttt{IDN (1 to 3, 2 to 4)} \quad = \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

Exceptions:	6004	IDN must make a square matrix.
	6005	Illegal array bounds.

(See also CON, NUL$, and ZER.)

IF Statement

IF *logex* THEN *simple-statement*
IF *logex* THEN *simple-statement* ELSE *simple-statement*

If the value of *logex* is "true," then the *simple-statement* following the keyword THEN will be executed, following which control will pass to the next line.

If *logex* 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.

The *simple-statement* can be replaced by a *line-number*, in which case a GOTO to that *line-number* will result, but only in a line-numbered program. See the GOTO statement.

The *simple-statements* in True BASIC are the ones beginning with the following keywords: ASK, BOX, CALL, CAUSE, CHAIN, CLEAR, CLOSE, DRAW, ERASE, EXIT, FLOOD, GET, GO, GOSUB, GOTO, INPUT, LET, LINE, MAT, ON, OPEN, PAUSE, PLAY, PLOT, PRINT, RANDOMIZE, READ, RESET, RESTORE, RETURN, SET, SOUND, STOP, UNSAVE, WINDOW, and WRITE.

IF Structure

```
if-structure::     IF logex THEN
                        . . .
                     ELSEIF logex THEN
                        . . .
                     ELSEIF logex THEN
                        . . .
                     ELSE
                        . . .
                     END IF
```

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

If the value of *logex* in the first line of the IF structure 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 the value of *logex* in the first line of the IF structure is “false,” control passes to the first ELSEIF part following the IF line. If the value of *logex* in the ELSEIF part 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 *logex* in the ELSEIF part 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.

IMAGE Statement

IMAGE: *format-string*

The IMAGE statement provides an alternate way to specify the *format-string* for a PRINT USING statement. The *format-string* is taken to consist of all characters starting with the character after the colon (even if it is a space) up through the last nonspace character prior to the on-line comment symbol “!” or the actual end of the line. The IMAGE statement may be used only with line-numbered programs, and is referred to by line-number in the PRINT USING statement.

The following two program fragments will produce identical results:

```
100 IMAGE        : The answer is ###### percent
110 PRINT USING 100   : x

100 LET image$ = " The answer is ###### percent"
110 PRINT USING image$: x
```

Exceptions:	8201	Badly formed USING string.
	8202	No USING item for output.

INPUT Statement

INPUT *inputlist*
INPUT *inputlist*,
INPUT *input-option* …, *input-option*: *inputlist*
INPUT #*rnumex*: *inputlist*
INPUT #*rnumex*, *file-input-option* …, *file-input-option*: *inputlist*

inputlist::	*var* …, *var*
var::	*numvar*
	strvar
	strvar substrex

file-input-option::	*input-option*
	IF MISSING THEN *action*
input-option::	PROMPT *strex*
	TIMEOUT *numex*
	ELAPSED *numvar*
action::	EXIT DO
	EXIT FOR
	line-number
input-response::	*input-item* …, *input-item*
	input-item …, *input-item*,
input-item::	*quoted-string*
	unquoted-string

When the INPUT statement without #*rnumex* is executed, the program awaits an *input-response* from the user. The *input-response* consists of *quoted-strings* and *unquoted-strings*, separated by commas, possibly followed by a trailing comma. (Only certain characters are allowed in *unquoted-strings* used in *input-responses*..)

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*. If the *strvar* is followed by a *substrex,* the *input-item* is assigned to the character positions given by the *substrex*.

The rules are the same as for READ and DATA statements. That is, leading and trailing spaces are included for *quoted-strings*, but omitted for *unquoted-strings*. The null string must be represented by the *quoted-string* ("").

If there are *input-options* present, no more than one of each type may be present.

If there is a PROMPT clause, the *strex* is displayed as the prompt, before awaiting the user's response. If there is no PROMPT clause, the default prompt "?", is used.

If the user does not supply enough *input-items* to satisfy the *inputlist*, the message "Too few input items. Please reenter input line." is displayed. The user can then retype the *input-items* to satisfy the *inputlist*. (This is a nonfatal exception, and can be intercepted.)

If the user supplies more *input-items* than are needed to satisfy the *inputlist*, the message "Too many input items. Please reenter input line." is displayed. The user can then retype the *input-items* to satisfy the *inputlist*. (This is a nonfatal exception and can be intercepted.)

If an *input-variable* is numeric and the corresponding *input-item* is not a valid numeric representation, the message "String given instead of number. Please reenter input line." is printed and a prompt issued. The user can then retype the *input-response*. (This is a nonfatal exception and can be intercepted.)

If the *inputlist* ends with a comma, the remainder of the *input-response*, if any, is retained for use by the next INPUT statement, and the error message is not displayed.

If the *input-response* ends with a comma and the *inputlist* requires additional *input-items*, then the default prompt "?" is issued and the user given a chance to enter additional *input-items*.

To illustrate these ideas, for the following sequence,

```
INPUT a, b,
INPUT c, d
```

each of the following is a valid *input-response*:

```
    ? 1,2,3,4
```

or

```
    ? 1,2,
    ? 3,
    ? 4
```

For an INPUT statement with #*rnumex*, the variables in the *inputlist* receive their values from the TEXT file associated with #*rnumex*.

If there are more items on a text line in the TEXT file than are needed to satisfy the *inputlist*, an exception occurs.

If there are fewer items on a text line in the TEXT file than are needed to satisfy the *inputlist*, an exception occurs.

If the *inputlist* ends with a comma, the remainder of the line in the TEXT file, if any, is retained for use by a subsequent INPUT statement with channel, and no exception occurs.

If a line in a TEXT file ends with a comma and the *inputlist* requires additional *input-items*, then the next line from the TEXT file is used.

If a TIMEOUT clause is present, then the user is given *numex* seconds, possibly fractional, to supply a valid *input-response*; otherwise, an exception (8401) occurs. If the TIMEOUT value is 0, exception (8401) will immediately occur, but values may still be assigned to the input variables if there are characters available in the input buffer of the operating system.

If the ELAPSED clause is present, then the value of *numvar* will contain the number of seconds, possibly fractional, that it took the user to supply a valid *input-response*. The elapsed time is measured by the program and may be longer than the elapsed time as perceived by the user if, for example, there are network delays.

If an INPUT statement with *#rnumex* contains a *file-option-list*, no more than one of each type may be present. If the *action* of an IF MISSING clause is EXIT FOR or EXIT DO, then the INPUT statement must be contained within a loop of that type. If the *action* is a *line-number*, it must follow the same rules as a GOTO statement with that *line-number*.

Before actual input occurs, the *action* of the IF MISSING clause, if present, is carried out if the file-pointer is at the end of the file.

The PROMPT, TIMEOUT, and ELAPSED *input-options* have no effect if the *channel* refers to a disk file. If the *channel* refers to the interactive terminal, then these options have the same effect as if the *channel* were missing.

Exceptions:	1007	Overflow in INPUT. (nonfatal)
	1008	Overflow in file INPUT.
	1054	String too long in INPUT. (nonfatal)
	1105	String too long in file INPUT.
	7004	Channel isn't open.
	7303	Can't input from OUTPUT file.
	7318	Can't INPUT from INTERNAL file.
	8002	Too few input items. (nonfatal)
	8003	Too many input items. (nonfatal)
	8011	Reading past end of file.
	8012	Too few data in record.
	8013	Too many data in record.
	8101	Data item is not a number.
	8102	Badly formed input line. (nonfatal)
	8103	String given instead of number. (nonfatal)
	8105	Badly formed input line from file.
	8401	Input timeout.
	8402	TIMEOUT value < 0.
	-8450	Nested INPUT statements with TIMEOUT clauses.
	9001	File is read or write protected.

INT Function

INT(*numex*)

Returns the greatest integer that is less than or equal to *numex*. INT is sometimes called the ***floor*** function. For example, `INT(1.9)` = 1, `INT(13)` = 13, and `INT(-2.1)` = -3.

(See also CEIL, IP and ROUND.)

INV Array Function

INV(*numarr*)

Returns the inverse of its argument, which must be a square two-dimensional numeric matrix. If the matrix is singular, an error will occur, and the value of DET with no argument will be set to 0. If the matrix is non-singular, the value of DET with no argument will be set to the determinant of the matrix.

For example:

$$\text{If } A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix} \text{, then INV(A)} = \begin{pmatrix} -2.0 & 1.0 \\ 1.5 & -0.5 \end{pmatrix}$$

Exceptions:	3009	Can't invert singular matrix.
	6003	INV needs a square matrix.

IP Function

IP(*numex*)

Returns the greatest integer that is less than or equal to *numex* without regard to sign, that is, towards zero. For example, `IP(1.9) = 1, IP(13)` = 13, and `IP(-2.1)` = -2.

(See also CEIL, INT and ROUND.)

LBOUND Function

LBOUND(*arrayarg*, *rnumex*)
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 *rnumex*. If there is no second argument, *arrayarg* must be a vector, and LBOUND returns the lowest value (lower bound) for its subscript. For example:

```
! For these OPTION and DIM statements
OPTION BASE 0
DIM A(2:5, -3:10), V(10)
! LBOUND takes on these values:
! LBOUND(A,1) = 2
! LBOUND(A,2) = -3
! LBOUND(V)   = 0
```

Exception:	4008	LBOUND index out of range.

(See also SIZE and UBOUND.)

LCASE$ Function

LCASE$(*strex*)

Returns the value of *strex* with all ASCII uppercase letters (see Appendix A) converted into lower case. Characters outside the range of the ASCII uppercase letters are left unchanged. For example:

```
LCASE$("Mr. Smith is 65.")
```

returns the value `"mr. smith is 65."`

(See also UCASE$.)

LEN Function

LEN(*strex*)

Returns the length (that is, the number of characters) of the argument *strex*. All characters count, including control characters and other non-printing characters. For example:

`LEN("a;sldkfjg")`	returns 9
`LEN("""")`	returns 1

LET Assignment Statement

LET *numvar* …, *numvar* = *numex*
LET *starget* …, *starget* = *strex*

starget::	*strvar*
	strvar substrex

The keyword LET may be omitted if your program contains OPTION NOLET. *Substrex* refers to a substring expression and is defined in Chapter 17.

The LET statement causes the expression on the right of the equal sign to be evaluated and then assigns the result to the variables (simple variables or array elements) on the left of the equal sign. More precisely, first the subscript and substring expressions on the left are evaluated, from left to right. Second, the expression on the right is evaluated. Third, this value is assigned to the variables and array elements on the left, in left to right order.

For example:

```
LET i, j = 2
LET i, a(i) = 1
```

will assign the value 1 to both `i` and `a(2)`; that is, the subscript expression is calculated first, and thus uses the old value of `i`.

For string assignments, if a *strvar* on the left does not have a *substrex*, then the *strex* on the right becomes the new value of that *strvar*. If a *strvar* has a *substrex*, then the *strex* on the right replaces only the substring defined by the *substrex*.

The rules for *substrex* when the "from" and "to" positions extend beyond the ends of the string are described in Chapter 17. If the *substrex* defines a null string, then the assignment of the *strex* on the right is made as an insertion immediately in front of the "from" position; no characters are removed. If the "from" position is less than or equal to 0, then the insertion is made to the front of the string. If the "from" position is greater than the length of the string, then the *strex* is appended to the end of the string. The resulting string being assigned into may become longer or shorter as a result.

For example:

```
LET a$, b$, c$, d$ = "abc"
LET a$[2:2] = "x"
LET b$[2:1] = "x"
LET c$[-4:-2] = "x"
LET d$[4:6] = "x"
PRINT a$, b$, c$, d$
```

will yield:

```
axc         axbc          xabc          abcx
```

Inside a multiple-line function definition, the list of variables to the left of the equal sign may include the function's name. The value of the expression then becomes the function's return value. The function's name may contain a substring expression.

Exceptions:	1106	String too long in assignment.
	5000	Out of memory.

LIBRARY Statement

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

The LIBRARY statement names the file or files containing external routines needed by the entire program. Normally, the LIBRARY statements are all in the main program, but this is not necessary. The library files can be compiled or uncompiled.

The order in which the files are named in the collection of LIBRARY statements may be critical. Generally speaking, if a higher-level routine calls a lower-level routine, the file containing the higher-level routine should be named before the file containing the lower-level routine, or the higher-level routine should contain a LIBRARY statement that tells where to find the lower-level routine.

The procedure is as follows:

1. The loader constructs a *need-list* of the names of all external routines that are needed by the main program. (A user-defined function is assumed to be needed if its name appears in a DECLARE FUNCTION statement.) It also constructs a *library-list* of all the files named in the LIBRARY statements.
2. The loader loads *all* routines following the END statement in the main program file. The names of all additional lower-level routines needed by any of the loaded routines are added to the *need-list*. In addition, libraries named in such routines are added to the end of the *library-list*.
3. The loader examines the first file named in the *library-list*. All routines that are known to be needed are loaded; the others are discarded. The names of any additional routines that are needed are added to the *need-list*. If a loaded routine contains LIBRARY statements, their library file names are added to the end of the *library-list*.
4. The loader discards this file name from the head of the *library-list*. Thus, True BASIC never looks back at earlier libraries.
5. This process is continued with the rest of the files named in the LIBRARY statements.

For example, suppose there are two files of external routines, and that their names are `"file1"` and `"file2"`. Suppose `"file1"` contains a routine `x` and `"file2"` contains a routine `y`. Further suppose that the main program calls (or invokes) `x` but not `y`, that the routine `x` calls (or invokes) `y`, and that none of the other routines in the two files calls `x` or `y`. Then the first LIBRARY statement below will work, but the second will not.

```
LIBRARY "file1", "file2"  ! Will work
LIBRARY "file2", "file1"  ! Will not work
```

It would also be correct if the main program contained:

```
LIBRARY "file1"
```

while routine `x` in “file1” contained:

```
LIBRARY "file2"
```

In fact, a safe procedure is to include a LIBRARY statement in every routine that calls another external routine.

It is permissible to name a LIBRARY file more than once, as follows:

```
LIBRARY "file2", "file1", "file2"
```

This technique allows subroutines in either file to call subroutines in the other file and will work for the previous example.

LINE INPUT Statement

LINE INPUT *strvarlist*
LINE INPUT *input-option* ..., *input-option*: *strvarlist*
LINE INPUT #*rnumex*: *strvarlist*
LINE INPUT #*rnumex*, *file-input-option* ..., *file-input-option*: *strvarlist*

strvarlist::	*starget* ..., *starget*
starget::	*strvar*
	strvar substrex

(See the INPUT statement for an explanation of the *input-options* and the *file-input-options*.)

A LINE INPUT statement without #*rnumex* requests one or more lines of input from the user. The first line is supplied to the first *strvar*, 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. The final *end-of-line* is not included.

A LINE INPUT statement with a PROMPT input-option issues *strex* as the prompt for the first response, but uses the default prompt “?” for subsequent responses.

A LINE INPUT statement with #*rnumex* obtains lines of text from the associated file and assigns them in order to the *stargets* in the *strvarlist*.

An unquoted null string is a valid response to a LINE INPUT statement.

Exceptions:	1054	String too long in INPUT. (nonfatal)
	1105	String too long in file INPUT.
	7004	Channel isn't open.
	7303	Can't input from OUTPUT file.
	7318	Can't INPUT from INTERNAL file.
	8011	Reading past end of file.
	8401	Input timeout.
	8402	TIMEOUT value < 0.
	-8450	Nested INPUT statements with TIMEOUT clauses.

LOCAL Statement

LOCAL *local-item* …, *local-item*

local-item::	*simple-numvar*
	simple-strvar
	array bounds

A LOCAL statement specifies that the variables named in it are ***local*** to the routine containing the statement. This statement is normally irrelevant in external routines, since all variables except parameters are automatically local, but it can be important in internal routines. All variables and arrays named in LOCAL statements but used in different invocations of the internal routine are different and are initialized each time the internal routine is invoked. The appearance of an *array bounds* in a LOCAL statement causes the array to be dimensioned, as in a DIM statement. In addition to creating local variables in internal routines, the LOCAL statement can be used in conjunction with the OPTION TYPO statement to avoid typographical errors in variable names.

An OPTION TYPO statement that occurs early in a *program-unit* or a *module-header* requires subsequent variables and arrays to be declared. This may be done in one of the following ways: naming them in a LOCAL statement; having them appear in the SUB, DEF, FUNCTION, or PICTURE statements that initiate the program unit; naming them in DECLARE PUBLIC statements; declaring arrays in DIM statements; or naming them in SHARE or PUBLIC statements in the program unit or in the *module-header* containing the program unit.

A variable or array appearing in a LOCAL statement in a module overrides the definition of a variable or array having the same name and appearing in a SHARE or PUBLIC statement in the *module-header*.

An OPTION TYPO statement that appears in a *module-header* applies to all the routines of the module. Thus, all routines in the module must name in a LOCAL statement their variables and arrays that are not included in previous SHARE or PUBLIC statements. An OPTION TYPO statement that appears in an external procedure in a library file applies to the rest of the procedure containing it and to all subsequent procedures in that library file. (See the OPTION TYPO statement.)

LOG Function

LOG(*numex*)

Returns the natural logarithm of *numex*, 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.... For example:

`LOG(1)`	returns 0
`LOG(10)`	returns 2.30259...

Exception:	3004	LOG of number <= 0.

(See also LOG10 and LOG2.)

LOG10 Function

LOG10(*numex*)

Returns the common logarithm of *numex*, which must be greater than 0. The common logarithm of `x` is defined as that value `v` for which `10^v = x`. For example:

`LOG10(100`	returns 2
`LOG10(2)`	returns .30103...

LOG10 can be defined in terms of LOG as follows:

```
DEF LOG10(x) = LOG(x)/LOG(10)
```

Exception: 3004 LOG of number < = 0.

(See also LOG and LOG2.)

LOG2 Function

LOG2(*numex*)

Returns the logarithm to the base 2 of *numex*, 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`. For example:

`LOG2(8)`	returns 3
`LOG2(10)`	returns 3.32193 ...

LOG2 can be defined in terms of LOG as follows:

```
DEF LOG2(x) = LOG(x)/LOG(2)
```

Exception: 3004 LOG of number < = 0.

(See also LOG and LOG10.)

LOOP Statement

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

LTRIM$ Function

LTRIM$(*strex*)

Returns the value of *strex* but with leading blank spaces removed. Trailing spaces, if any, are retained. For example:

`LTRIM$("   a b c    ")` returns "`a b c    `"

(See also RTRIM$ and TRIM$.)

MAT Statements

There are a number of MAT statements, which deal with computation, input, output, and graphics. They are as follows; each is described separately below:

MAT Assignment
MAT INPUT
MAT LINE INPUT
MAT PLOT
MAT PRINT
MAT READ
MAT REDIM
MAT WRITE

MAT Assignment Statement

MAT *assignment*

assignment::	*numarr = numarrex*
	strarr = strarrex
	strarr substrex = strarrex
numarrex::	*numarr*
	numarr numarrop numarr
	numarrconst
	numarrconst redim

	primary
	primary * *numarr*
	primary * *numarrconst*
	primary * *numarrconst redim*
	numarrfunction(*numarr*)
numarrop::	+ or – or *
numarrconst::	CON or IDN or ZER
	CON or IDN or ZER *redim*
numarrfunction::	INV or TRN
strarrex::	*strarrprim*
	strarrprim & *strarrprim*
	str-factor & *strarrprim*
	strarrprim & *str-factor*
	str-factor
	strarrconst
	str-factor & *strarrconst*
strarrprim::	*strarr*
	strarr substrex
strarrconst::	NUL$
	NUL$ *redim*

(A *primary* is, for example, a numerical expression within parentheses. A *str-factor* is, for example, a quoted-string or a *string-variable* followed by a *substrex*. See Chapter 17 for complete definitions of these and other syntax items and the exceptions that can occur if they are misused.)

A MAT assignment evaluates the array expression (*numarrex* or *strarrex*) on the right and then assigns it to the array on the left. If the array being assigned to is a string array followed by a substring expression, the array elements from the array expression on the right replace only the characters specified by the substring expression in the array on the left.

The number of dimensions of the array expression and the array on the left must agree. (If the array expression is a *numarrconst* or *strarrconst* without a *redim*, the number of dimensions of the array expression is determined from the array on the left.)

The result of array addition (or subtraction) is an array, each of whose elements is the sum (or difference) of the corresponding elements in the arrays of the *numarrex*.

The result of array multiplication depends on what is being multiplied. If both arrays are two-dimensional matrices, the result is a two-dimensional matrix. If one array is one-dimensional and the other is two-dimensional, then the result is a one-dimensional array, i.e., a vector. If both arrays are one-dimensional, then the result is a one-dimensional array, i.e., a vector; in this case, the vector will have only one element whose value will be the DOT product of the two vectors being multiplied. (Note: array multiplication is *not* the element-by-element product.)

Below is a list of array functions and their effects:

Array Constants and Functions

Function	Effect
CON	Produces an array each of whose elements is 1.
ZER	Produces an array each of whose elements is 0.
IDN	Produces a square matrix having 1s on the main diagonal and 0s elsewhere.
INV	Produces the mathematical inverse of a square matrix.
TRN	Produces the transpose of a two-dimensional matrix.
NUL$	Produces an array each of whose elements is the null string.

(See Chapter 9 *Arrays and Matrices,* for descriptions of DET, DOT, LBOUND, SIZE, and UBOUND, which require array arguments but return numeric values.)

The presence of a *primary* (which is a *numeric variable*, a *numeric-constant*, or a *numeric expression* in parentheses) implies ***scalar multiplication***; that is, each element of the array is multiplied by the numeric value of the *primary*. If the *primary* exists by itself, it generates a *numeric* array, each of whose elements has the value of the *primary* and with dimensions determined from the array on the left. (For example, `MAT A = 17` is the same as `MAT A = 17*CON.`) The *primary* is evaluated and stored in a temporary location before the MAT assignment begins.

The presence of a *str-primary* (which is a *string variable*, a *quoted-string* constant, or a *string expression* in parentheses) generates a string array, each of whose elements has the value of the *str-primary*. If a *strarr* is followed by a *substrex*, that substring applies to each element of the *strarr*.

For addition and subtraction of arrays, the two arrays must be numeric and must have the same sizes, but they need not have the same subscript ranges.

For multiplication of two arrays, the two arrays must be one- or two-dimensional, must be numeric, and must have sizes that conform with the mathematical rules for matrix multiplication. (One-dimensional arrays will be treated as either row or column vectors in order to obey the rules for matrix multiplication.) The argument of the INV function must be a two-dimensional square numeric array. The argument of the TRN function must be a two-dimensional numeric array.

For the MAT assignment statements, the actual subscript ranges for the array on the left are inherited from the array expression on the right according to the following rule:

```
new LBOUND(left array) = old LBOUND(left array)

new UBOUND(left array) = new LBOUND(left array) + SIZE(array expression) - 1
```

Thus, the lower bounds remain the same, while the upper bounds are adjusted to be consistent with the size of the array expression on the right.

For those *numarrex* and *strarrex* whose dimensions and subscript ranges are determined from the array on the left (such as `MAT a = ZER`, `MAT b = 17`, `MAT c = 17*CON`, or `MAT d$ = "Hello"`), the dimensions and subscript ranges for the array on the left do not change.

Exceptions:	1005	Overflow in MAT operation.
	1052	String too long in MAT.
	3009	Can’t invert singular matrix.
	5000	Out of memory.
	6001	Mismatched array sizes.
	6003	INV needs a square matrix.
	6004	IDN must make a square matrix.
	6005	Illegal array bounds.
	6101	Mismatched string array sizes.
		(Other numeric and string exceptions may occur.)

(See also the CON, IDN, ZER, INV, TRN, and NUL$ functions and the MAT REDIM statement.)

MAT INPUT Statement

MAT INPUT *matinlist*
MAT INPUT *input-option …, input-option: matinlist*
MAT INPUT #*rnumex*: *matinlist*
MAT INPUT #*rnumex, file-input-option …, file-input-option*: *matinlist*

matinlist::	*inputarray …, inputarray*
inputarray::	*array*
	array redim
	array (?)

(See the INPUT statement for an explanation of the *input-options* and the *file-input-options* and their effects.)

If an *inputarray* is followed by a (?), it must be one-dimensional.

MAT INPUT without #*rnumex* assigns values from the *input-response* to the elements of the arrays, in order. There must be a separate *input-response* for each array in the *inputlist*. For each array, the elements are assigned values in "odometer" order. (That is, if `A` is a 2-by-2 array, odometer order is `A(1,1)`, `A(1,2)`, `A(2,1)`, `A(2,2)`.) The *input-response* must contain a sufficient number of values of the appropriate type (numeric or string), separated by commas, in a single *input-response* or in a collection of *input-responses* with all but the last ending with a comma. (See the INPUT statement for details of *input-responses*.)

If there are insufficient values in the *input-response*, True BASIC will print "Too few input items. Please reenter input line," issue the default prompt (?), and await further input. If there are too many values in the *input-response*, True BASIC will print "Too many input items. Please reenter input line," issue the default prompt, and await further input. In these cases, it is necessary only to reenter the *last* input line. Previously entered input lines that end with commas will have been stored in the array.

If a *redim* is present on an array in the *matinlist*, then that array is redimensioned before the values are assigned to it. The new dimensions are precisely those of the *redim*.

If the PROMPT clause is present as an *input-option*, then that prompt is used for the initial *input-response*. Subsequent *input-responses*, if needed, will use the default prompt.

If the *matinlist* contains a vector with a (?) as the *redim* expression, then as many values are assigned to the elements of the vector as are supplied by the user in the *input-response*, or in the collection of *input-responses*, with all but the last ending with a comma. The lower bound of the vector is unchanged, but its upper bound is adjusted to accept the values. Thus, the new upper bound of the vector is (*lower-bound* + *number-of-values-supplied* – 1).

If #*rnumex* is present, then the values assigned are taken from the associated TEXT file. If the required input extends over several lines of text in the file, all lines but the last must end with a comma. The PROMPT clause is not permitted with #*rnumex*.

Exceptions:	1008	Overflow in file INPUT.
	1054	String too long in INPUT. (nonfatal)
	1105	String too long in file INPUT.
	6005	Illegal array bounds.
	7004	Channel isn't open.
	7303	Can't input from OUTPUT file.
	7318	Can't INPUT from INTERNAL file.
	8002	Too few input items. (nonfatal)
	8003	Too many input items. (nonfatal)
	8011	Reading past end of file.
	8012	Too few data in record.
	8013	Too many data in record.
	8101	Data item is not a number.
	8102	Badly formed input line. (nonfatal)
	8103	String given instead of number. (nonfatal)
	8105	Badly formed input line from file.
	8401	Input timeout.
	8402	TIMEOUT value < 0.
	-8450	Nested INPUT statements with TIMEOUT clauses.

MAT LINE INPUT Statement

MAT LINE INPUT *lineinlist*
MAT LINE INPUT *input-option* …, *input-option*: *lineinlist*
MAT LINE INPUT #*rnumex*: *lineinlist*
MAT LINE INPUT #*rnumex*, *file-input-option* …, *file-input-option*: *lineinlist*

lineinlist::	*redimstrarray* …, *redimstrarray*
redimstrarray::	*strarr*
	strarr redim

(See the INPUT statement for an explanation of the *input-options* and the *file-input-options* and their effects.)

MAT LINE INPUT without #*rnumex* assigns *response-lines* to the elements of the arrays in the *redimarraylist*, in order from left to right, and within each array in odometer order. The entire line of input is assigned to an array element, including leading and trailing spaces and embedded commas.

If a *redim* is present, that array is redimensioned before values are assigned to its elements. The new dimensions are precisely those of the *redim*.

If the PROMPT clause is present as an *input-option*, that prompt is used for the first line of input. Subsequent lines of input will be prompted with the default prompt (?).

If #*rnumex* is present, then the lines of input are taken from the associated TEXT file. The PROMPT clause is not permitted with #*rnumex*.

Exceptions:	1054	String too long in INPUT. (nonfatal)
	1105	String too long in file INPUT.
	6005	Illegal array bounds.
	7004	Channel isn't open.
	7303	Can't input from OUTPUT file.
	7318	Can't INPUT from INTERNAL file.
	8011	Reading past end of file.
	8401	Input timeout.
	8402	TIMEOUT value < 0.
	-8450	Nested INPUT statements with TIMEOUT clauses.

MAT PLOT Statement

MAT PLOT POINTS: *matplotarray*
MAT PLOT LINES: *matplotarray*
MAT PLOT AREA: *matplotarray*

The MAT PLOT statements all require a single *matplotarray*, which must be a two-dimensional numeric array that has two (or more) columns. Each of the rows is interpreted as a single point. The first column is the x-coordinate of the point, and the second column is the y-coordinate. (If the array has more than two columns, the last column is the *y*-coordinate.)

MAT PLOT POINTS plots the points in the array, just as if the points were supplied with a MAT PLOT statement naming the array elements. Thus, the following two statements are equivalent:

```
MAT PLOT POINTS: x
PLOT POINTS: x(1,1),x(1,2); ...; x(n,1),x(n,2)
```

MAT PLOT LINES plots the line segments connecting the points in the array. Thus, the following two statements are equivalent:

```
MAT PLOT LINES: x
PLOT LINES: x(1,1),x(1,2); ...; x(n,1),x(n,2)
```

The line segments plotted may fail to represent a closed polygon if, for example, the last point does not repeat one of the earlier points.

MAT PLOT AREA plots the polygon represented by the points and fills it with the current foreground color. Thus, the following two statements are equivalent:

```
MAT PLOT AREA: x
PLOT AREA: x(1,1),x(1,2); ...; x(n,1),x(n,2)
```

It is not necessary for the last point to repeat the first point, as the MAT PLOT AREA statement supplies the line segment joining the first and last points.

Exception:	6401	Must have two or more columns for MAT PLOT.
	-11000	Can't do graphics on this computer.

MAT PRINT Statement

MAT PRINT *matprintlist*
MAT PRINT *using-option*: *matusinglist*
MAT PRINT #*rnumex*: *matprintlist*
MAT PRINT #*rnumex*, *file-print-option* …, *file-print-option*: *matprintlist*
MAT PRINT #*rnumex*, *file-using-option* …, *file-using-option*: *matusinglist*

matprintlist::	*array* …, *separator array*
	array …, *separator array separator*
matusinglist::	*array* …, *array*
	array …, *array* ;
separator::	, or ;

(See the PRINT statement for an explanation of the *print-options* and the *file-print-options* and their effects.)

The MAT PRINT statement prints the elements of each array in its *matprintlist* to the screen or, if #*rnumex* is present, at the end of the associated file or into the associated logical window. 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.

The *separators* following the *array* determine the spacing of the printed array values. If the separator is a comma, the values are printed in successive print zones; if it is a semicolon, the values are printed immediately following each other. If there is no *separator* following the last *array*, a comma is assumed. The effects of the current zonewidth and margin are the same as for the ordinary PRINT statement.

If the USING clause is present, the *separators* must be commas. The values are then printed according to the format specified, instead of being printed in zones or being packed. The effect is exactly as if you used a normal PRINT USING statement containing all the array elements listed in odometer order. (See the PRINT statement for what happens if a field in the format is inadequate for a particular value.) A possible final semicolon is ignored.

If a single row has more elements that the format string has fields, the format string is re-used from the beginning, which includes starting a new line. This process continues without regard to ends of rows until all the elements of the array have been printed. Thus, if you wish to display two-dimensional arrays, you should have as many numeric-fields in the format-string as there are columns in the array. A subsequent array in the same statement starts at the beginning of the format string.

Exceptions:	7004	Channel isn't open.
	7302	Can't output to INPUT file.
	7308	Can't PRINT or WRITE for middle of this file.
	8201	Badly formed USING string.
	8202	No USING item for output.
	8203	USING value too large for field. (nonfatal)
	8204	USING exponent too large for field. (nonfatal)

MAT READ Statement

MAT READ *readarraylist*
MAT READ IF MISSING THEN *action*: *readarraylist*
MAT READ #*rnumex*: *readarraylist*
MAT READ #*rnumex*, *read-option* …, *read-option*: *readarraylist*

readarraylist::	*readarray* …, *readarray*
readarray::	*array*
	array redim

(See the READ statement for details of the *read-options* and their effects.)

MAT READ, with or without #*rnumex*, assigns values to the elements of each of the arrays in the *readarraylist*, in order. For each array in the *readarraylist*, the values are assigned in "odometer" order – that is, the last subscript changes most rapidly, then the next to last, and so on. If a *redim* is present, the array being read into is first redimensioned. The new dimensions are precisely those of the *redim*.

The MAT READ statement without #*rnumex* assigns to its variables the next series of *data* from the DATA list in the current invocation of the *program-unit* containing the READ statement. (See the DATA statement.)

A *strvar* can receive any valid *datum*. A *numvar* can receive only a *datum* that happens to be a valid and unquoted *numeric-constant*.

If the IF MISSING clause is present, then its action will take effect if and when the DATA become exhausted while the reading is still in progress.

The MAT READ statement with #*rnumex* assigns to its *readarrays* values obtained from a file. If the BYTES clause is present as a *read-option*, then the file must be of type BYTE.

For a STREAM file, values from the file are assigned to the elements of the arrays in the READ statement in odometer order. The file pointer will advance to the next item after reading each value.

For a RANDOM file, values from the record in the file at the file-pointer are assigned to the elements of the arrays in the READ statement in odometer order. There must be the same number of elements as there are values in the record. The file pointer will advance to the next record after reading each record.

For a RECORD file, the next array element is read from the file at the current position. The file pointer will advance to the next value (record) after reading each value (record). The number of records remaining in the file must be sufficient to supply each element in the array.

For a BYTE file, if the BYTES clause is present as a *read-option*, you may read only into string arrays. A number of bytes equal to the second *rnumex* is read from the BYTE file and assigned to the next array element. The array elements are accessed in odometer order. If there are fewer bytes remaining in the file than are in the second *rnumex*, but there is at least one, those remaining are assigned, but there must be sufficient bytes in the file to supply all the array elements. The number of characters specified in the BYTES clause overrides the record size specified in a previous SET RECSIZE statement or in a RECSIZE OPEN option.

For a BYTE file, if the BYTES clause is absent, you may read into string or numeric arrays, or any combination thereof. The number of bytes assigned to each element of a string array is determined by a SET RECSIZE statement or an OPEN RECSIZE option. The number of bytes assigned to each element of a numeric array is always eight, regardless of the current RECSIZE. These eight bytes must be in the IEEE floating point representation of a number, equal to that produced by the NUM$() function.

The file pointer will be advanced to the next byte after the reading is completed.

Exceptions:	1006	Overflow in file READ.
	1053	String too long in READ.
	7004	Channel isn't open.
	7303	Can't input from OUTPUT file.
	8001	Reading past end of data.
	8011	Reading past end of file.
	8012	Too few data in record.
	8013	Too many data in record.
	8101	Data item isn't a number.
	-8104	Data item isn't a string.
	8302	Input item bigger than RECSIZE.
	-8503	Can't use READ or WRITE for TEXT files.

MAT REDIM Statement

MAT REDIM *arrvar redim* …, *arrvar redim*

MAT REDIM redimensions the array according to the *redim* provided. In the redimensioning, some of the original values may be retained. In particular, if the array is one-dimensional and the new dimensions lead to a smaller size, then the obvious original values will be discarded. If the new dimensions lead to a larger size, then all the original values will be retained, and the new elements will be filled with zero or the null string, depending on the type of the array. If the new dimensions change the lower and upper bounds but leave the size intact, then all of the original values will be retained.

If the array is two-dimensional, and the number of rows is changed (made smaller or larger), then some of the original rows will be discarded (if smaller), or new rows of zero or null strings will be added (if larger). If the number of columns is changed, then the elements will be reorganized. The rule is that the values in the original array stay put. The new dimensions may associate the old values with new array positions, which can be determined through applying odometer order.

If the array has more than two dimensions, the rules are an extension of the rules for two-dimensional arrays.

Exceptions:	5000	Out of memory.
	6005	Illegal array bounds.

MAT WRITE Statement

MAT WRITE #*rnumex*: *matwritelist*
MAT WRITE #*rnumex*, *write-option* …, *write-option*: *matwritelist*

matwritelist::	*array* …, *array*
array::	*numarr*
	strarr

(See the WRITE statement for a detailed description of the *write-options* and their effects.)

MAT WRITE writes the elements, in odometer order, of the arrays in the *matwritelist* to the file specified.

For a STREAM file, the values from all the arrays in the MAT WRITE statement are written to the file in odometer order without regard to records.

For an RANDOM file, the values from all the arrays in the MAT WRITE statement are written, in odometer order, to the same record in the file. The record must be large enough to contain all the values. New records may be added to the end of the file by using SET #*n*: POINTER END or by using the *record-setter* END.

For a RECORD file, the values from all the arrays in the MAT WRITE statement are written, in odometer order, one to a record, starting with the current record. The record must be large enough to contain the largest single element. New records may be added to the end of the file by using SET #*n*: POINTER END or by using the *record-setter* END.

For a BYTE file, the elements of each array are written as bytes to the file. For a numeric array, eight bytes in the IEEE floating point representation are written for each array element; that is, the eight-byte string produced by the function NUM$ is written. For a string array, the characters of each string are written without headers. It is important to remember that, unlike STREAM, RANDOM, or RECORD files, a BYTE file does not keep track of whether particular bytes were originally part of a number or part of a string.

Exceptions:	7004	Channel isn't open.
	7302	Can't output to INPUT file
	8301	Output item bigger than RECSIZE.
	-8304	Must SET RECSIZE before WRITE.

(In addition, see the exceptions for SET POINTER.)

MAX Function

MAX (*numex*, *numex*)

Returns the larger of the values of the two arguments. (Note: -1 is larger than -2.) MAX can be defined in terms of other True BASIC statements as follows:

```
DEF MAX (a,b)
    IF a > b THEN LET MAX = a ELSE LET MAX = b
END DEF
```

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. (The maximum length may have been set by a DECLARE STRING statement or imposed by the operating system.) 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. For example, on an IBM-compatible PC without a numerical coprocessor, MAXNUM = 1.7976931e+308.

MAXSIZE Function

MAXSIZE (*arrayarg*)

Returns 2^31. (This function serves no useful purpose in this version of True BASIC, but does in the ANSI-compatible version.)

MIN Function

MIN (*numex*, *numex*)

Returns the smaller of the values of the two arguments. (Note: -2 is smaller than -1.) MIN can be defined in terms of these other True BASIC statements:

```
DEF MIN (a,b)
    IF a < b then LET MIN = a ELSE LET MIN = b
END DEF
```

MOD Function

MOD(*numex*, *numex*)

`MOD(x,y)`	returns `x` modulo `y`, provided `y` is not equal to zero. For example:
`MOD(7, 3)`	returns 1
`MOD(-7, 3)`	returns 2
`MOD(7, -3)`	returns -2
`MOD(-7, -3)`	returns -1

MOD can be defined in terms of INT as follows:

```
DEF MOD(x,y) = x - y*INT(x/y)
```

Exception: 3006 MOD and REMAINDER can’t have 0 as 2nd argument.

(See also REMAINDER.)

MODULE Structure

module-structure::	MODULE *identifier*
	…*module-header*
	…*procedure-part*
	END MODULE
module-header::	…*modstatement*
modstatement::	*public-statement*
	share-statement
	private-statement
	other-statement
procedure-part::	… *procedure*

A module consists of a collection of external routines together with a module header. The module header may contain *public-statements*, *share-statements*, and *private-statements*, as well as ordinary True BASIC statements (which we denote here by *other-statement*).

Public-statements name variables and arrays that are accessible throughout the program, including the procedures in the module. To be accessible from a given *program-unit* not included in the module, a public variable or array must be named in a DECLARE PUBLIC statement in that *program-unit*. Public arrays must be dimensioned in the *public-statement* in which they are named, not in a separate DIM statement.

Share-statements name variables, arrays, and channels (files and logical windows) that are accessible to each routine in the module, but not to program units outside the module. Shared arrays must be dimensioned in the *share-statement* in which they are named, not in a separate DIM statement.

A variable or array cannot appear in more than one of the following: a PUBLIC statement, a SHARE statement, a DIM statement, or a LOCAL statement in a module header. A variable or array cannot appear in more than one PUBLIC statement in the program. Every variable or array named in a DECLARE PUBLIC statement must be named somewhere in a PUBLIC statement. In other words, one module or program unit "owns" a public variable, but other program units can use it.

Private-statements name those procedures in the module that are *not* accessible from outside the module. Such procedures may be described as ***private external procedures***. All other external procedures in the module are ***public***.

Each of the external procedures of the module may contain any number of *internal* procedures.

The *other-statements* in the module header are executed at program startup and serve to initialize the module. The order in which the modules are initialized at program startup is determined by the order in which they are loaded. *Warning: a module in a library will not be initialized unless it contains an external procedure that is used.* Thus, if you wish to use a module header to initialize public variables, you must include in the module at least one procedure and invoke it, even if it is a dummy procedure or even if the invoking code is never executed.

All variables, arrays, and channels named in PUBLIC and SHARE statements retain their existence throughout the life of the program, though their values may change.

A LOCAL or DIM statement in an external procedure of a module can be used to "override" the meaning of a variable or array that is also named in a PUBLIC or SHARE statement in the module header.

A DECLARE PUBLIC statement in a module header makes the variables and arrays it names available to the entire module.

A DECLARE PUBLIC statement cannot be used to override the meaning of a shared variable or array.

An OPTION ANGLE or OPTION BASE statement that appears in a module header applies to the subsequent portion of the *module-header* and to all procedures in the module, although its effect can be overridden for a given procedure by a similar OPTION statement inside that procedure. An OPTION TYPO or OPTION NOLET statement that appears in the *module-header* or any procedure of the module applies to the rest of that procedure and to the entire rest of the file containing the module. (See also the PUBLIC, SHARE, PRIVATE, DECLARE PUBLIC, and the various OPTION statements.)

NCPOS Function

NCPOS(*strex*, *strex*)
NCPOS(*strex*, *strex*, *rnumex*)

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 the null string, then NCPOS returns 0. If the second argument is null 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 null 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.

For example:

```
     NCPOS ("banana", "mno")             returns 1
     NCPOS ("banana", "pqr")             returns 1
     NCPOS ("banana", "mno", 4)          returns 4
     NCPOS ("banana", "mno", 10)         returns 0
```

NCPOS can be defined more precisely in terms of other True BASIC statements as follows:

```
  DEF NCPOS(s1$,s2$,start)
      LET start = MAX(1,MIN(ROUND(start),LEN(s1$)+1))
      FOR c = start TO LEN(s1$)
          FOR j = 1 to LEN(s2$)
              IF s1$[c:c] = s2$[j:j] THEN EXIT FOR
          NEXT j
          IF j = LEN(s2$)+1 THEN
             LET NCPOS = c
             EXIT DEF
          END IF
      NEXT c
      LET NCPOS = 0
  END DEF
```

(See also POS, POSR, CPOS, CPOSR, and NCPOSR.)

NCPOSR Function

NCPOSR(*strex*, *strex*)
NCPOSR(*strex*, *strex*, *rnumex*)

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 the null string, then NCPOSR returns 0. If the second argument is null 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 null 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.

For example:

```
     NCPOSR ("banana", "mno")            returns 6
     NCPOSR ("banana", "pqr")            returns 6
     NCPOSR ("banana", "mno", 4)         returns 4
     NCPOSR ("banana", "MNO", 10)        returns 6
```

NCPOSR can also be defined more precisely with these True BASIC statements:

```
  DEF NCPOSR (s1$,s2$,start)
     LET start = MAX(0,MIN(ROUND(start),LEN(s1$)))
     FOR c = start TO 1 STEP -1
        FOR j = 1 to LEN(s2$)
           IF s1$[c:c] = s2$[j:j] THEN EXIT FOR
        NEXT j
        IF j = LEN(s2$)+1 THEN
           LET NCPOSR = c
           EXIT DEF
        END IF
     NEXT c
```

```
      LET NCPOSR = 0
   END DEF
```
(See also POS, POSR, CPOS, CPOSR, and NCPOS.)

NEXT Statement

The NEXT statement can be used only as part of a FOR loop and is required. See the FOR structure.

NUL$ Array Constant

NUL$ *redim*
NUL$

NUL$ is an array constant that yields a string array consisting entirely of null strings. NUL$ can appear only in a MAT assignment statement. The dimensions of the array of null strings are determined in one of two ways. If the *redim* is present, then an array of those dimensions will be generated; the array being assigned to in the MAT assignment statement will be resized (see the MAT Assignment statement) to these new dimensions. If the *redim* is absent, then the dimensions of the array will match those of the array being assigned to in the MAT assignment statement.

Exceptions:	6005	Illegal array bounds.

(See also CON, IDN, and ZER.)

NUM Function

NUM(*strex*)

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.

Exception:	-4020	Improper NUM string

(See also NUM$.)

NUM$ Function

NUM$(*numex*)

Returns a string of length 8 that contains the numerical value using the IEEE eight-byte format, whether or not your machine has an IEEE-compatible numeric coprocessor. This gives a way to store numbers on disk in a machine-independent format. Normally, the NUM function must be used to convert the string back to a number.

(See also NUM.)

Object Subroutine

CALL Object (*numex*, *numex*, *strex*, *strex*, *numarr*)

The subroutine Object provides access to the creation and manipulation of physical windows, controls, and selected graphics objects. A single calling sequence is used:

```
   CALL Object (method, id, attributes$, values$, values())
```

`Method` is a number between 0 and 26 (usually represented by a variable name), and denotes the method to be applied to the object or control. `Id` is the identification number of the object or control. `Attributes$` is a string expression that contains one or more attributes for which values need to be set (SET method) or interrogated (GET method); if there is more than one, the items in the list are separated by vertical bars "`|`". Additional string information is communicated through `values$`; again, multiple items are separated by vertical bars "`|`". Additional numeric information is communicated through a numeric list `values()`.

For a complete and detailed discussion of the Object subroutine, see Chapter 19 "Object Subroutine."

ON GOSUB Statement

ON *rnumex* GOSUB *line-number-list*
ON *rnumex* GOSUB *line-number-list* ELSE *simple-statement*

line-number-list::	*line-number* …, *line-number*

The keyword GOSUB may also be written as GO SUB.

The *rnumex* is evaluated. If this value is in the range from 1 to the number of *line-numbers* in the *line-number-list*, a GOSUB jump is made to the *line-number* whose position in the list is that value. If the value is outside the range and the ELSE part is missing, an exception occurs. If the value is outside the range and the ELSE part is present, then its *simple-statement* is executed, following which control passes to the next line.

The *simple-statement* may be replaced by a *line-number*, in which case it becomes a GOTO statement. (See the IF statement for a list of *simple-statements*.)

Following the completion of the GOSUB subroutine (i.e., upon execution of an appropriate RETURN statement), a jump is made to the first statement following the ON GOSUB statement.

The program must be *line-numbered*, and the target *line-numbers* must lie within the scope of the ON GOSUB statement.

Exception:	10001 ON index out of range, no ELSE given.

ON GOTO Statement

ON *rnumex* GOTO *line-number-list*
ON *rnumex* GOTO *line-number-list* ELSE *simple-statement*

line-number-list::	*line-number* ..., *line-number*

The keyword GOTO may also be written as GO TO.

The *rnumex* is evaluated. If this value is in the range from 1 to the number of line numbers in the *line-number-list*, a jump is made to the *line-number* whose position in the list is that value. If the value is outside the range and the ELSE part is missing, an exception occurs. If the value is outside the range and the ELSE part is present, then its *simple-statement* is executed, following which control passes to the next line.

The *simple-statement* may be replaced by a *line-number*, in which case it becomes a GOTO statement. (See the IF statement for a list of *simple-statements*.)

The program must be *line-numbered*, and the target *line-numbers* must lie within the scope of the ON GOTO statement.

Exception:	10001	ON index out of range, no ELSE given.

OPEN Statement

OPEN #*rnumex*: NAME *strex*
OPEN #*rnumex*: NAME *strex*, *open-list*
OPEN #*rnumex*: PRINTER
OPEN #*rnumex*: SCREEN *screen-coords*

open-list::	*open-clause* …, *open-clause*
screen-coords::	*numex*, *numex*, *numex*, *numex*
open-clause::	ACCESS INPUT
	ACCESS OUTPUT
	ACCESS OUTIN
	ACCESS *strex*
	CREATE NEW
	CREATE OLD
	CREATE NEWOLD
	CREATE *strex*

```
ORGANIZATION TEXT
ORGANIZATION STREAM
ORGANIZATION RANDOM
ORGANIZATION RECORD
ORGANIZATION BYTE
ORGANIZATION strex
RECSIZE rnumex
```

The keyword ORGANIZATION may be replaced by the keyword ORG. The keywords for the ACCESS, CREATE, and ORGANIZATION clauses may be in upper, lower, or mixed.

The *open-list* may contain no more than one *open-clause* of ACCESS type, no more than one of CREATE type, no more than one of ORGANIZATION type, and no more than one of the RECSIZE type.

If a file is opened with ACCESS INPUT, only INPUT (or READ) statements may be subsequently used to access that file. If a file is opened with ACCESS OUTPUT, only PRINT (or WRITE) statements may subsequently be used to access that file. If a file is opened with ACCESS OUTIN (default), both input and output may occur with the file. If a *strex* is used with an ACCESS clause, it must evaluate to one of the ACCESS keywords. If no ACCESS clause occurs, ACCESS OUTIN is assumed.

Besides the restrictions that may be placed on file operations by the ACCESS clause, there may be additional restrictions placed on the file by the operating system.

If CREATE NEW occurs in a file OPEN statement, then the file must not already exist, but a new one will be created. If CREATE OLD occurs, then the file must already exist. If CREATE NEWOLD occurs, then the file will be opened if it exists, or a new one will be created and opened if it does not already exist. If the CREATE clause has a *strex*, it must evaluate to one of the CREATE keywords. If no CREATE clause occurs, CREATE OLD is assumed.

Using ORGANIZATION (ORG) TEXT for an empty file that has been erased will make it a TEXT file, regardless of the file's previous organization. If the file is not empty, it must be of TEXT type; otherwise, an exception will occur.

A newly created file opened without an ORG clause will have the default organization TEXT. A newly created file opened without a RECSIZE clause will have the default record size 0.

Using ORGANIZATION (ORG) STREAM for an empty file will make it a STREAM file, regardless of the file's previous organization. If the file is not empty, it must be of STREAM type; otherwise, an exception will occur.

Using ORGANIZATION (ORG) RANDOM for an empty file will make it a RANDOM file, regardless of the file's previous organization. If the file is not empty, it must be of RANDOM type; otherwise, an exception will occur.

Using ORGANIZATION (ORG) RECORD for an empty file will make it a RECORD file, regardless of the file's previous organization. If the file is not empty, it must be of RECORD type; otherwise, an exception will occur.

Using ORGANIZATION (ORG) BYTE for any file specifies that that file will subsequently be accessed with BYTE file operations.

If the ORGANIZATION (ORG) clause contains a *strex*, it must evaluate to one of the ORGANIZATION keywords.

If no ORGANIZATION (ORG) clause occurs, then ORGANIZATION RANDOM will be assumed if the file is a random file, ORGANIZATION RECORD will be assumed if the file is a record file, ORGANIZATION STREAM will be assumed if the file is a stream file, ORGANIZATION BYTE will be assumed if the file is a True BASIC compiled file.

The RECSIZE clause establishes the record size for a RANDOM, RECORD or BYTE file, provided the file is empty or is newly created. (If no RECSIZE clause occurs, then a SET RECSIZE statement must be executed before the first write to the file.) If a RANDOM or RECORD file is not empty, then the RECSIZE clause, if present, must agree with the record size of the file. If a BYTE file is not empty, then the RECSIZE clause, if present, establishes a new record size that will be used in subsequent READ statements that do not have BYTE clauses. (WRITE to a BYTE file is not affect by RECSIZE.)

OPEN with PRINTER assigns the default printer, if it exists, to that channel. The printer is treated like an OUTPUT only TEXT file.

OPEN with SCREEN assigns to the channel a logical window which occupies that portion of the physical window denoted by the *screen-coords*. The *screen-coords* refer to, respectively, the left edge, the right edge, the bottom, and the top of the logical window. The full physical window is (0, 1, 0, 1), with fractional values denoting portions of the physical window. Thus, (.5, 1, .5, 1) specifies that the logical window occupies the upper right quadrant of the physical window. The *screen-coords* must follow the following rules:

```
0 <= left edge < right edge <= 1
0 <= bottom edge < top edge < = 1
```

Exceptions:	7001	Channel number must be 1 to 1000.
	7002	Can't use #0 here. (nonfatal)
	7003	Channel is already open.
	7100	Unknown value for OPEN option.
	7102	Too many channels open.
	7103	File's record size doesn't match OPEN RECSIZE.
	7104	Wrong type of file.
	9001	File is read or write protected.
	9002	Trouble using disk or printer.
	9003	No such file.
	9004	File already exists.
	9007	Too many files open.
	9101	Can't open PRINTER.
	-11002	Screen minimum >= maximum.
	-11003	Screen bounds must be 0 to 1.

(See also the ERASE statement.)

OPTION Statements

There are seven OPTION statements, which are described below. None of them is executable; their effect is governed by their lexical position in a program or file and not by whether they are actually encountered in the course of execution.

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

Several OPTION statements may be combined:

OPTION *option* …, *option*

option:: ANGLE RADIANS
ANGLE DEGREES
ARITHMETIC DECIMAL
ARITHMETIC NATIVE
BASE *signed-integer*
COLLATE NATIVE
COLLATE STANDARD
NOLET
TYPO
USING TRUE
USING ANSI

If several *options* of the same type appear, the last one takes effect.

The definitions of each of the *options* are given in the definitions of the individual OPTION statements that follow.

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 (SIN, COS, TAN, SEC, CSC, COT, ASIN, ACOS, ATN, ANGLE, ROTATE, and SHEAR). The OPTION ANGLE statement affects all evaluations of trigonometric functions and graphics transforms that follow it lexically, until either the end of the *program-unit* or another OPTION ANGLE statement is encountered. In the absence of an OPTION ANGLE statement, the default angle measure is RADIANS.

An OPTION ANGLE statement that occurs in a MODULE header or a procedure of the module applies to the trigonometric functions and transforms that follow it lexically in that module, including those in later procedures, until the end of the module or another OPTION ANGLE statement is encountered.

OPTION ARITHMETIC Statement

OPTION ARITHMETIC NATIVE
OPTION ARITHMETIC STANDARD

The OPTION ARITHMETIC statement has no effect and is included for compatibility with ANSI.

OPTION BASE Statement

OPTION BASE *signed-intege*r

The OPTION BASE statement allows you to specify the lower bound of all arrays and array constants whose lower bounds are not explicitly set in DIM, LOCAL, PUBLIC, or SHARE statements, or in certain *redims*. The OPTION BASE statement specifies the default lower bound for array declarations and *redims* that follow it lexically, until either the end of the *program-unit* or another OPTION BASE statement is encountered. In the absence of an OPTION BASE statement, the default lower bound is 1.

An OPTION BASE statement that occurs in a MODULE header or a procedure of the module applies to all array declarations that follow it lexically in that module, including those in later procedures, until the end of the module or another OPTION BASE statement is encountered.

(See also the MAT statement.)

OPTION COLLATE Statement

OPTION COLLATE NATIVE
OPTION COLLATE STANDARD

The OPTION COLLATE statement has no effect and is included for compatibility with ANSI.

OPTION NOLET Statement

OPTION NOLET

The OPTION NOLET statement allows LET statements to be written without the keyword LET.

An OPTION NOLET statement can appear in a *module-header* or in any *program-unit*. It then applies to all assignment statements which occur lexically after the OPTION NOLET statement and continues to be in effect to the end of the file containing it.

Certain reserved words cannot be used as variable or array names, whether OPTION NOLET is in effect or not. These reserved words are as follows:

Keywords:	ELSE, NOT, PRINT, REM
Numeric functions:	DATE, EXLINE, EXTYPE, MAXNUM, PI, RND, RUNTIME, TIME
String functions:	DATE$, TIME$
Array constants:	CON, IDN, NUL$, ZER

All other keywords of True BASIC can be used as variables whether OPTION NOLET is in effect or not. The only restriction is that IF and ELSEIF cannot be used as variables on the left in a LET statement if the keyword LET is omitted under OPTION NOLET.

The keyword DATA is not a reserved word and can be used in a LET statement if LET is included. It cannot, however, be used as a *numvar* in a LET statement if the LET is omitted under OPTION NOLET, because it will be interpreted as a DATA statement. For example, in

```
OPTION NOLET
DATA = 3
```

the second line will be interpreted as a DATA statement with a *data-item* equal to "= 3" rather than as a LET statement without the LET.

OPTION TYPO Statement

OPTION TYPO

The OPTION TYPO statement requires that all non-array variables that appear after it be declared explicitly. They must be declared in a LOCAL, PUBLIC, SHARE, or DECLARE PUBLIC statement, or by appearing as *parms* in a SUB, DEF, FUNCTION, or PICTURE statement.

An OPTION TYPO statement can appear in a MODULE header or in any *program-unit*. It then applies to all variables whose first appearance occurs after the OPTION TYPO statement and continues to be in effect to the end of the file containing it.

OPTION USING Statement

OPTION USING TRUE
OPTION USING ANSI

The OPTION USING statement determines which form of the PRINT USING statement or USING$ built-in function you wish to use. Selecting OPTION USING ANSI will use the ANSI standard version of these statements, while OPTION USING TRUE will use the traditional True BASIC version. OPTION USING TRUE is the default.

ORD Function

ORD(*strex*)

Returns the ordinal position in the ASCII character set of the character given by *strex*, which must be either a single character or an allowable two- or three-character name of certain ASCII characters as described in Appendix A, except that `ORD("")` = -1. ORD is the opposite of the CHR$ function in that `ORD(CHR$(n)) = n` for all n in the range 0 to 255. However, `CHR$(ORD(a$)) = a$` only if the value of `a$` is a single ASCII character.

For example:

`ORD(CHR$(1))`	returns 1
`ORD("A")`	returns 65
`ORD("BEL")`	returns 7
`ORD("cr")`	returns 13

Exception:	4003	Improper ORD string.

PACKB Subroutine

True BASIC provides two routines, PACKB and UNPACKB, for packing and unpacking numeric integer values as bits in a string. PACKB is a callable subroutine, while UNPACKB is a function.

The action of `CALL PACKB (s$, bstart, blen, value)` may be described as follows:

1. `Bstart`, `blen`, and `value` are first rounded.
2. If `value` is greater than 2^31 - 1, it is replaced by 2^31 - 1. If `value` is less than -2^31, it is replaced by -2^31.

3. `Blen` should be in the range -32 to 32, inclusive. If `blen` is not in this range, the results will be machine-dependent.
4. If `blen` is less than zero, it is replaced by ABS(`blen`).
5. If `bstart` is less than 1, it is replaced by 1.
6. If `bstart` is greater than the number of bits in the string `s$` (which is always a multiple of eight) then it is replaced by the actual number of bits in `s$` plus 1.
7. If the adjusted `value` falls in the range -2^(`blen-1`) to 2^(`blen-1`)-1, inclusive, the two's-complement binary representation of the adjusted `value` is stored in the field of length `blen` bits starting at bit position `bstart`. The string is extended, if necessary, but always by a whole number of eight-bit characters.
8. If the adjusted `value` falls outside the range defined in step 7, only the least significant bits are retained and stored.

For example:

```
CALL PACKB (s$, 1, 8, 12)
! Places the bits "00001100" into the 1st byte of s$.
CALL PACKB (s$, 1, 16, 12)
! Places the bits "0000000000001100" into
!      the first two bytes of s$.
CALL PACKB (s$, 9, 8, 0)
! Places the bits "00000000" into the 2nd byte of s$.
CALL PACKB (s$, 33, 1, 0)
! Places the bit "0" into the 33rd bit of s$.
CALL PACKB (s$, 33, 1, 1)
! Places the bit "1" into the 33rd bit of s$.
```

(See also UNPACKB.)

PAUSE Statement

PAUSE *numex*

The PAUSE statement will cause the execution of the program to stop for a number of seconds given by *numex* and then continue. The pausing will be as accurate as possible on a given system. Negative values will be treated as 0.

PI Function

PI

A no-argument function, PI returns the value of *pi*, the ratio of a circle's circumference to its diameter (approximately equal to 3.14159265). It gives as much accuracy as possible on your computer, but in any case at least ten decimal places.

PICTURE Structure

picture-statement
. . .
END PICTURE

picture-statement::	PICTURE *identifier*
	PICTURE *identifier* (*subparmlist*)

See the SUB statement for the definition of *subparmlist*.

A PICTURE is invoked with a DRAW statement. Other than that, a PICTURE acts exactly like a subroutine. The parameter passing mechanism is that of subroutines. In fact, PICTURES can be used in all cases in place of a subroutine, except that they may not execute SET WINDOW statements.

If the PICTURE contains PLOT statements (PLOT, MAT PLOT, FLOOD, GET POINT, or GET MOUSE) or contains CALL or DRAW statements to other pictures or subroutines, then the final picture will reflect all the transforms applied through all the DRAW statements.

If the system does not support graphics, the PICTURE structure is legal and acts like a subroutine although any graphics statements used in it will generate an exception.

Exception: 11004 Can't SET WINDOW in picture.

PLAY Statement

PLAY *strex*

The *strex* defines the tune and is made up of characters that have the following meanings:

A to G	Play the note in the current octave, in the current tempo, and with the current dynamics. Optional # or + may follow to indicate a sharp; or a - may follow to indicate a flat.
A `n` to G `n`	Play the note in the current octave but with the length of this note given only by `n` (see L `n`, below.) Optional #, +, or - signs may be added, as in A#2.
O `n`	Set current octave to that indicated by `n`. Octaves run from 0 to 7. A given octave runs from C to the next higher B. Middle C is the first note of octave 4. If no "O" command has yet been executed, the default is octave 5.
> and <	Move up (>) or down (<) one octave.
L `n`	Set the length (duration) of the subsequent notes and rests. The lengths are the reciprocals of `n` – that is, 1 stands for a whole note, 2 for a half note, 4 for a quarter note, and so on. Note that `n` must be a whole number. The default value for `n` is 4 – a quarter note.
.	A dot after a note indicates that the note is to be played as a dotted note; that is, 1.5 times its normal length. Thus, `"A4."` means the note A played as a dotted quarter. Several dots may be used to increasingly lengthen the note. Rests may also be dotted.
R `n` or P `n`	Rest (pause) for length `n`. Lengths are measured as in the L command above.
T `n`	Set the tempo. Here, n stands for the number of quarter notes in one minute. The default tempo is 120, or 120 quarter notes to the minute.
MF	Subsequent music will be played "in the foreground"; that is, no other statement will be executed until the entire music string has been played. This is the default.
MB	Play subsequent music "in the background" – that is, your program will continue to run while the remainder of the music string is played.
MN	Play the music with "normal" dynamics, as opposed to "legato" or "staccato." This means that each note plays for 7/8 of the time specified by the L command; the remaining 1/8 is silence. This is the default.
ML	Play the music with "legato" dynamics. This means that each note plays for the full time specified by the L command.
MS	Play the music with "staccato" dynamics. This means that each note plays for 3/4 of the time specified by the L command. (Not available for Windows 3.x)

Uppercase and lowercase letters are equivalent in play strings. Spaces may be inserted to improve legibility, but True BASIC ignores them. No characters, other than those listed above, are allowed in play strings.

If the music is being played in the foreground, the program waits until the play string is finished before executing the next statement. If the music is being played in the background, the music continues until it is completed or until it encounters another PLAY or SOUND statement.

If the system does not support sound, the PLAY statement is ignored.

Exception: -4501 Error in PLAY string.

PLOT Statements

There are five types of PLOT statements:

PLOT POINTS
PLOT LINES
PLOT AREA
PLOT TEXT
Vacuous PLOT

The terms *pointslist* and *point* are used with these statements and are defined as:

pointslist::	*point*; *point*
point::	*numex*, *numex*

All PLOT statements in pictures are subject to the effects of the current transform. (See the DRAW statement.)

All PLOT statements, except for PLOT TEXT, are *clipped* at the edges of the current logical window. That is, the portion of the drawing that is inside the logical window is shown, while the portion outside the logical window is not. No error occurs if part or all of the drawing lies outside the logical window.

PLOT Statement (Vacuous PLOT)

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.

PLOT AREA Statement

PLOT AREA: *pointslist*

PLOT AREA plots the region defined by the *pointslist* 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 LINES Statement

PLOT LINES: *pointslist*
PLOT *pointslist*
PLOT LINES: *pointslist*;
PLOT *pointslist*;

PLOT LINES plots the line consisting of the line-segments that connect the points in the *pointslist*. A line is drawn from the previous point to the first point in the *pointslist* 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
```

PLOT LINES and PLOT statements may end with a (;). In this case, the beam is left on so that subsequent PLOT LINES or PLOT statements will continue plotting the line without a break. Otherwise, the beam is turned off.

PLOT POINTS Statement

PLOT POINTS: *pointslist*
PLOT *point*

PLOT POINTS plots the points (x-y pairs) in the *pointslist* as dots in user-coordinates. `PLOT x,y` is an abbreviation for `PLOT POINTS: x,y`.

PLOT TEXT Statement

PLOT TEXT, AT *point*: *strex*

PLOT TEXT plots the text string in the current color at the *point* specified in the AT clause. Which part of the text string will fall at that point is determined by the current state of TEXT JUSTIFY.

(See also the SET TEXT JUSTIFY statement.)

POS Function

POS(*strex*, *strex*)
POS(*strex*, *strex*, *rnumex*)

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 null while the second is not, then POS returns 0. If the second string is null, 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 null, POS returns the starting position.

The first form of POS is equivalent to the second form with the third argument equal to one.

For example:

`POS ("banana", "ana")`	returns 2
`POS ("banana", "nan")`	returns 3
`POS ("banana", "n", 4)`	returns 5
`POS ("banana", "xyz", 10)`	returns 0

POS can be defined more precisely in terms of other True BASIC statements as follows:

```
DEF POS(s1$,s2$,start)
    LET start = MAX(1,MIN(ROUND(start),LEN(s1$)+1))
    FOR c = start TO LEN(s1$)-LEN(s2$)+1
       IF s1$[c:c+LEN(s2$)-1] = s2$ THEN
           LET POS = c
           EXIT DEF
       END IF
    NEXT c
    LET POS = 0
END DEF
```

(See also POSR, CPOS, CPOSR, NCPOS, and NCPOSR.)

POSR Function

POSR(*strex*, *strex*)
POSR(*strex*, *strex*, *rnumex*)

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 null but the second is not, POSR returns 0. If the second string is null, 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 null, 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. For example:

`POSR ("banana", "anan")`	returns 2
`POSR ("banana", "nan")`	returns 3
`POSR ("banana", "n", 4)`	returns 3
`POSR ("banana", "xyz", 10)`	returns 0

POSR can be defined more precisely in terms of other True BASIC statements as follows:

```
DEF POSR(s1$,s2$,s)
   LET difflen = LEN(s1$)-LEN(s2$)+1
   LET start = MAX(0,MIN(ROUND(s),difflen)
   FOR c = start TO 1 STEP -1
      IF s1$[c:c+LEN(s2$)-1] = s2$ THEN
         LET POSR = c
         EXIT DEF
       END IF
   NEXT c
   LET POSR = 0
END DEF
```

(See also POS, CPOS, CPOSR, NCPOS, and NCPOSR.)

PRINT Statement

PRINT
PRINT *print-list*
PRINT *using-option*: *using-list*
PRINT #*rnumex*
PRINT #*rnumex*:
PRINT #*rnumex*: *printlist*
PRINT #*rnumex, file-print-option* ..., *file-print-option*: *print-list*
PRINT #*rnumex, file-using-option* ..., *file-using-option*: *using-list*

printlist::	*printitem* ... *separator printitem*
	printitem ... *separator printitem separator*
using-list::	*usingitem* ..., *usingitem*
	usingitem ..., *usingitem* ;
separator::	, or ;
printitem::	*numex*
	strex
	tab-call
	null
usingitem::	*numex*
	strex
tab-call::	TAB(*rnumex*)
	TAB(*rnumex, rnumex*)
file-using-option::	*using-option*
	file-print-option
using-option::	USING *strex*
	USING *line-number*
file-print-option::	IF THERE THEN *action*
action:	EXIT DO
	EXIT FOR
	line-number

The *printitems* are printed on the screen or, if #*rnumex* is present, at the end of the associated file, provided the file pointer is at the end of the file. Numeric values are printed according to the semantics for the STR$ function, except that a trailing space is always provided and, for positive numbers, a leading space is provided. 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, except for the trailing and possibly leading space surrounding a numeric value. If the separator is a comma, then the next item is printed in the next print zone.

A *printitem* can be omitted entirely, in which case nothing is printed, but the *separator* has its usual effect. Thus,

```
      PRINT ,  ,"*"
```

will skip the current print zone and the next and print the asterisk in the zone after that.

If the IF THERE clause is present, its *action* is carried out if the file-pointer is not at the end of the file.

If a USING clause is present, the values are then printed according to the format specified, without regard to print zones. If a *strex* follows the word USING, it will be used to determine the format. If a *line-number* follows the word USING, the line-number must be that of an IMAGE statement; the characters following the colon of the IMAGE statement then determine the format. (See the IMAGE statement.)

If there are not enough fields in the format string, the format string is re-used from its beginning, including starting a new line. If there are left over fields at the end of the printing, they are ignored.

If a field in the using string is inadequate (too short, not enough "^" for an exponent, no "–" for a negative number), then what happens depends on whether or not the PRINT statement is contained in the *protected-part* of a WHEN structure. If it is, an exception occurs. Otherwise, the following actions occur:

1. The field is filled with asterisks and printed.
2. The value is printed unformatted on the next line.
3. Printing continues on the following line.

The margin is ignored if a USING clause is included in a PRINT statement. That is, characters may be printed beyond the margin.

The zonewidth is 16 characters wide by default. The zones begin with character position 1, character position 17, etc. You can change the zonewidth with a SET ZONEWIDTH statement.

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.

For items whose printing would extend beyond the right margin, two rules apply:

1. If the starting print position is not at the left margin, then the item will start printing on the next line.
2. If the item is too long to fit on the current line and the starting print position is at the left margin, then the item will be printed "folded" – that is, only what fits on the line will be printed there, with the rest being printed on the next line, and so on. The margin is set initially to 80 characters. You can change the margin with a SET MARGIN statement.

TAB with one argument causes the next item to be printed, starting at the column position specified by the argument. If the cursor is already at that position, no action occurs. If the printed material on that line already extends past the position specified, the new item is printed at that position but on the next line. If the TAB argument is less than 1, it is replaced by 1, unless the PRINT statement is contained lexically within the *protected-part* of a WHEN structure, when an exception occurs. If the TAB argument is greater than the margin, it will be reduced "modulo" the margin. That is, in TAB(`n`), if `n` > margin, `n` will be replaced by MOD(`n` – 1, margin) + 1. The effect of TAB with one argument is the same whether or not the PRINT statement includes a channel, and whether the channel refers to a file or a logical window.

TAB with two arguments may be used in a PRINT statement only when printing to a logical window. It causes the next item to be printed starting at a row number given by the first argument and a column given by the second argument. The row and column are in cursor coordinates; the top row of the window is numbered 1, and the left column is numbered 1. The row and column must be within the limits for the current logical window, or the logical window specified.

TAB with two arguments is equivalent to the SET CURSOR statement; that is, the following two sequences are equivalent:

```
   PRINT TAB(r,c); "x"
```

```
SET CURSOR r, c
PRINT "x"
```

(See the TAB function and SET CURSOR and IMAGE statements.)

Exceptions:	4005	TAB column less than 1. (nonfatal)
	7004	Channel isn't open.
	7302	Can't output to INPUT file.
	7308	Can't PRINT or WRITE to middle of this file.
	8201	Badly formed USING string.
	8202	No USING item for output.
	8203	USING value too large for field. (nonfatal)
	8204	USING exponent too large for field. (nonfatal)
	9601	Cursor set out of bounds.
	-11000	Can't do graphics on this computer.

PRIVATE Statement

PRIVATE *procname* …, *procname*

procname::	*identifier*
	string-identifier

The PRIVATE statement, which can appear only as part of a module header, specifies that the external procedures (subroutines, functions, or pictures) named are private to that module. That is, they can be accessed by the other procedures in the module, but not by procedures outside the module. The procedures named need not actually exist. A module's procedures can't be named more than once in PRIVATE statements.

PROGRAM Statement

PROGRAM *identifier*
PROGRAM *identifier* (*funparmlist*)

funparmlist::	*funparm* … , *funparm*
funparm::	*simple-numvar*
	simple-strvar
	arrayparm

The PROGRAM statement, if used, must be the first statement of the main program, other than comment lines. The *identifier* following the keyword PROGRAM must be present, although it is not used.

If the program is the target of a CHAIN statement containing a WITH clause, and the PROGRAM statement contains the (*funparmlist*), then the parameters in the *funparmlist* receive the arguments sent in the CHAIN statement. If the *funparmlist* is missing, or the number and type of the parameters do not match with the arguments, an exception will occur at the CHAIN statement.

If there is no WITH clause in the CHAIN statement, the PROGRAM statement must not contain a *funparmlist*; if it does, an exception will occur at the CHAIN statement.

If the program is run directly, the PROGRAM statement, if included with a parameter list, can be used to receive data from the operating system command line.

The parameter passing mechanism is that of defined functions – by value. Thus, even if the CHAIN statement contains a RETURN clause, the parameters cannot carry information back to the original program.

PUBLIC Statement

PUBLIC *publicitem* …, *publicitem*

publicitem::	*simple-numvar*
	simple-strvar
	array bounds

The PUBLIC statement specifies that each variable and array named can be accessed from outside the *module* or *program-unit*, provided that a *program-unit* desiring to access the variable or array specifies it in a DECLARE PUBLIC statement. The appearance of an *array bounds* in a PUBLIC statement causes the array to be dimensioned, as in a DIM statement.

A given *publicitem* cannot appear more than once in the PUBLIC statements of a program.

PUBLIC variables and arrays are similar to external variables and arrays (as in PL/I). The purpose and use of PUBLIC variables and arrays is similar to the purpose and use of COMMON (as in FORTRAN). All these mechanisms allow variables and arrays to be shared throughout the program without passing them as parameters to the external procedures of the program.

RAD Function

RAD(*numex*)

`RAD(x)` returns the number of radians in `x` degrees. This function is not affected by the current OPTION ANGLE. For example:

```
RAD(90)     = PI/2 = 1.5708...
```

RAD can be defined in terms of other True BASIC statements as follows:

```
DEF RAD(x) = x*PI/180
```

(See also DEG and PI.)

RANDOMIZE Statement

RANDOMIZE

The RANDOMIZE statement produces a new seed for the random number generator. The new seed is calculated from system parameters that are changing constantly, such as the date and the time. However, you should realize that it is remotely possible that the same seed could be produced if the conditions under which the program is run are identical, including the date and the time. The random numbers produced by the generator can be accessed through the RND function.

It is not necessary, nor is it desirable, to use the RANDOMIZE statement more than once in the running of a program.

ReadCPixel Subroutine

```
CALL ReadCPixel (x, y, r, g, b)
```

where x and y are the user coordinates of the pixel you wish to read, and r, g, and b are the red, green, and blue intensities of the pixel, in the range 0-1 to match SET COLOR MIX values.

ReadPixel Function

```
DEF ReadPixel (x, y)
```

where x and y are the user coordinates of the pixel you wish to read, and the result is the True BASIC color number of the pixel. Note that the color number is found by matching the intensities of the pixel with those in the color mix table, so that if there is more than one color number with the same color mix set, the function will return the lowest-numbered color. Note, too, that in some situations, there may be no match. This could be because a) the system has modified the intensities slightly from what was requested in SET COLOR MIX; b) the pixel is part of a dithered pattern (if the palette is unrealized); c) the color mix table has been modified since the pixel in question was painted. For this reason, you should probably use ReadCPixel and compare the intensity on each gun to determine if a color matches or not.

READ Statement

READ *var* …, *var*
READ IF MISSING THEN *action*: *var* …, *var*
READ #*rnumex*: *var* …, *var*
READ #*rnumex*: *var* …, *var* SKIP REST
READ #*rnumex*, *read-option* …, *read-option*: *var* …, *var*
READ #*rnumex*, *read-option* …, *read-option*: *var* …, *var* SKIP REST

read-option::	*record-setter*
	IF MISSING THEN *action*
	BYTES *rnumex*
record-setter::	BEGIN
	END
	NEXT
	SAME
action::	EXIT DO
	EXIT FOR
	line-number
var::	*numvar*
	strvar
	strvar substrex

If the *action* of an IF MISSING clause is EXIT FOR or EXIT DO, then the READ statement must be contained within a loop of that type. If the *action* is a *line-number*, it must follow the same rules as a GOTO statement with that *line-number*.

The READ statement without #*rnumex* assigns to its variables the next *datum* from the DATA list in the current invocation of the *program-unit* containing the READ statement. (See the DATA statement.)

A *strvar* can receive any valid *datum*. A *numvar* can receive only a *datum* that is unquoted and is a valid *numeric-constant*. If the *strvar* is followed by a *substrex*, the string *datum* replaces the characters specified in the *substrex*.

If the IF MISSING clause is present, then its action will take effect if and when the data become exhausted while the reading is still in progress.

The READ statement with #*rnumex* assigns to its *vars* values obtained from the associated file. If the BYTES clause is present, then the file must be of type BYTE. If the *record-setter* is present, the file-pointer is repositioned before the actual reading begins. If the *record-setter* is BEGIN, the file-pointer is positioned at the beginning of the file. If END, it is positioned at the end of the file. If SAME, it is positioned to where it was at the beginning of the previous READ or similar operation carried out on that file. If NEXT, the file-pointer is not changed unless it was left in the middle of a record because an exception occurred; in this case, NEXT causes the file-pointer to move to the beginning of the next record.

After the *record-setter*, if any, has been applied, the *action* of the IF MISSING clause, is carried out if the file-pointer is at the end of the file.

For a STREAM file, values from the file are assigned to the variables in the READ statement. The file pointer is set to the next value after the READ is completed.

For a RANDOM file, values from the record in the file at the file-pointer are assigned to the variables in the READ statement. There must be the same number of variables as there are elements in the record; if the SKIP REST clause is present, there may be more values in the record than there are variables – the unused ones are skipped. The file pointer is set to the next record after the READ is completed.

For a RECORD file, values from successive records in the file starting at the file pointer are assigned to the variables in the READ statement. The file pointer is set to the next record after the READ is completed.

For STREAM, RANDOM, and RECORD files, strings and numbers may be mixed in any order in the file (or in the records of a RANDOM file), but they must be read into the proper type of variable, that is, strings into string variables and numbers into numeric variables.

For a BYTE file, if the BYTES clause is present, you may read only into *strvars*. A number of bytes equal to the BYTES *rnumex* is read from the BYTE file and assigned to the next *strvar*. If there are fewer bytes remaining in the file than are requested by the second *rnumex*, but if there is at least one, the remaining are assigned. The number of characters specified in the BYTES clause overrides the record size specified in a previous SET RECSIZE statement or in a RECSIZE OPEN option.

If the BYTES clause is absent, you may read into *strvars* or *numvars*, or any combination thereof. The number of bytes assigned to each *strvar* is determined by a previous SET RECSIZE statement or an OPEN RECSIZE option. If there are fewer bytes in the file, but there is at least one, the remaining are assigned. The number of bytes assigned to each *numvar* is always eight, regardless of the current RECSIZE. These eight bytes must be in the IEEE floating point representation of a number, as if produced by the NUM$() function.

The file pointer will be advanced to the next byte after the READ is completed.

Exceptions:	1053	String too long in READ.
	7004	Channel isn’t open.
	7303	Can’t input from OUTPUT file.
	-7351	Must be BYTE file for READ BYTES.
	8001	Reading past end of data.
	8011	Reading past end of file.
	8012	Too few data in record.
	8013	Too many data in record.
	8101	Data item isn’t a number.
	-8104	Data item isn’t a string.
	8105	Badly formed input line from file.
	-8503	Can’t use READ or WRITE for TEXT file.
	9001	File is read or write protected.

Read_Image Subroutine

CALL Read_Image (sourcetype$, imagedata$, filename$)

The Read_Image subroutine reads a graphics image from the file whose name is specified, converting it into the local BOX KEEP format, which is then stored in the string imagedata$. The user should specify the type of the image file in sourcetype$, which may contain one of: “JPEG”, “PICT” (Macintosh only,) “MS BMP”, possibly “OS/2 BMP”, and possibly “PCX”. These file types must be supplied exactly as shown, although they may be lower- or mixed case. If the sourcetype$ is the null string, True BASIC will do its best to determine the actual type of the file and act accordingly. (Note: The software behind this subroutine is based in part on the work of the Independent JPEG Group.)

Exceptions:	7104	Wrong type of file.
	9002	Trouble using disk or printer.
	9003	No such file.

REM Statement

REM *... character*

The REM statement allows you to add comments to your program. You can use any characters you want in the REM statement except an *end-of-line*. REM statements are ignored.

A REM statement is equivalent to a comment line that begins with an exclamation mark (!). In addition, an exclamation mark (!) can be used to place comments on the same lines as other True BASIC statements.

REMAINDER Function

REMAINDER(*numex, numex*)

`REMAINDER(x,y)` returns the remainder obtained by dividing `x` by `y`; `y` must not be equal to 0. For example:

```
REMAINDER(17,5)          returns 2
REMAINDER(-17,5)         returns -2
REMAINDER(-17,-5)        returns -2
```

REMAINDER is equal to MOD when both arguments are positive or both arguments are negative, or if the remainder is zero. Otherwise, it will be `MOD(x,y) - y`. A simple rule is that, unless the remainder is zero, REMAINDER will have the same sign as its first argument, while MOD will have the same sign as its second argument.

`REMAINDER(x,y)` can be defined in terms of the IP function as follows:

```
DEF REMAINDER(x,y) = x - y*IP(x/y)
```

Exception: 3006 MOD and REMAINDER can't have 0 as 2nd argument.

(See also MOD.)

REPEAT$ Function

REPEAT$(*strex*, *rnumex*)

Returns the string consisting of *rnumex* copies of *strex*. For example:

```
REPEAT$ ("Hi", 2.7)          returns "HiHiHi"
REPEAT$ ("---", 2)           returns "------"
```

Exception: 4010 REPEAT$ count < 0.

RESET Statement

RESET #*rnumex*: BEGIN
RESET #*rnumex*: END
RESET #*rnumex*: NEXT
RESET #*rnumex*: SAME
RESET #*rnumex*: RECORD *rnumex*

These statements are equivalent to, respectively,

SET #*rnumex*: POINTER BEGIN
SET #*rnumex*: POINTER END
SET #*rnumex*: POINTER NEXT
SET #*rnumex*: POINTER SAME
SET #*rnumex*: RECORD *rnumex*

Exceptions:	7004	Channel isn't open.
	7202	Must be RECORD or BYTE file for SET RECORD.

RESTORE Statement

RESTORE
RESTORE *line-number*

The RESTORE statement resets the data pointer to the start of the *data-list*. It lets you reuse the *data-list* in the current invocation of the *program-unit* containing the RESTORE statement. The data pointers in other *program-units*, or in other invocations of the containing *program-unit*, are not affected.

If a *line-number* is present, then the *program-unit* must be line-numbered. The data-pointer is reset to that point in the *data-list* represented by the DATA statements at that line-number and beyond in the *program-unit*.

(See also the DATA statement.)

RETRY Statement

RETRY

The RETRY statement can appear only in the *handler-part* of a WHEN or HANDLER structure. It transfers directly to the line that was being executed when the exception occurred.

(See the WHEN and HANDLER structures, and the EXIT HANDLER and CONTINUE statements.)

RETURN Statement

RETURN

The RETURN statement is used in conjunction with the GOSUB statement. It transfers control to the first line following the line containing the most recently executed GOSUB statement. There must be at least one GOSUB statement in the current invocation of the current *program-unit* for which a RETURN has not yet been executed.

Exception:	10002	RETURN without GOSUB.

RND Function

RND

A no-argument function, RND returns the next "pseudo-random" number in the sequence. These numbers, which have no obvious pattern, fall in the range 0 < = RND < 1. If the program containing RND is rerun, True BASIC produces the same sequence of RND values. If you want your program to produce unpredictable results, use a RANDOMIZE statement before you use the RND function.

(See also the RANDOMIZE statement.)

ROUND Function

ROUND(*numex*, *rnumex*)
ROUND(*numex*)

`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)`. For example:

```
ROUND(123.4567, 3)          returns 123.457
ROUND(123.4567, -2)         returns 100
ROUND(123.4567)             returns 123
```

`ROUND(x,n)` can be defined in terms of the INT function as follows:

```
DEF ROUND(x,n) = INT(x*10^n + 0.5)/10^n
```

Exceptions:	1003	Overflow in numeric function.

(See also TRUNCATE and INT.)

RTRIM$ Function

RTRIM$(*strex*)

Returns the value of *strex* but with the trailing blank spaces removed. Leading spaces, if any, are retained. For example:

```
RTRIM$("    abc     ")          returns "    abc"
```

(See also LTRIM$ and TRIM$.)

RUNTIME Function

RUNTIME

A no-argument function, RUNTIME returns the number of seconds of processor time used since the start of execution. It is useful only in time-sharing systems where the TIME function can be used to measure elapsed time only. On personal computers, RUNTIME may return -1.

(See TIME.)

SEC Function

SEC(*numex*)

Returns the value of the secant function. If OPTION ANGLE DEGREES is in effect, the argument is assumed to be in degrees. If OPTION ANGLE RADIANS (default) is in effect, the argument is assumed to be in radians.

For example, if OPTION ANGLE DEGREES is in effect, then `SEC(45)` is approximately 1.41421...; if OPTION ANGLE RADIANS is in effect, then `SEC(1)` is approximately 1.85082...

SEC may be defined in terms of other True BASIC functions as follows:

```
DEF SEC(x) = 1/COS(x)
```

Exception: 1003 Overflow in numeric function.

SELECT CASE Structure

The SELECT CASE structure, which allows multi-way branches depending on the value of a particular expression, has the following form:

```
select-case-structure::  SELECT CASE selectex
                         CASE case-specifier
                         . . .
                         CASE case-specifier
                         . . .
                         . . .
                         CASE ELSE
                         . . .
                         END SELECT
selectex::               numex
                         strex
case-specifier::         case-part  …, case-part
case-part::              constant
                         constant TO constant
                         IS relop constant
constant::               numeric-constant
                         quoted-string
```

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 *selectex* in the SELECT CASE statement.

The *selectex* in the SELECT CASE statement is evaluated. The *case-specifier* in the first CASE part is examined. If the *selectex* 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.

The *selectex* satisfies a case-part if (1) the *case-part* consists of a single value and the *selectex* equals it exactly, (2) the *case-part* consists of a pair of values separated by the word TO and the *selectex* lies in the range, including end points, defined by the two values, or (3) the *case-part* consists of an IS clause and the value of the *selectex* satisfies the relation:

```
selectex relop constant
```

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.

Exception: 10004 No CASE selected, but no CASE ELSE.

SET Statements

A running program can set certain graphics and files parameters through the SET statement. Examples are changing the color mix for a certain color, and setting the record size for a RANDOM file. Some SET statements require a channel expression (which can refer to a file or a logical window), others forbid it, and a few can have it or not. For those that allow or require a channel expression, channel #0 always refers to the default logical window. If the channel expression refers to neither a file nor a logical window, then an exception occurs.

Exception: 7004 Channel isn't open.

SET and ASK work together. All quantities that can be SET can be asked about. The reverse is not true, as there are parameters beyond the control of the program. (See the ASK statement.)

Here is an alphabetical enumeration of the types of SET statements.

SET BACK
SET COLOR
SET COLOR MIX
SET CURSOR
SET DIRECTORY
SET MARGIN
SET MODE
SET NAME
SET POINTER
SET RECORD
SET RECSIZE
SET TEXT JUSTIFY
SET WINDOW
SET ZONEWIDTH

SET BACK Statement

SET BACK *rnumex*
SET BACKGROUND COLOR *rnumex*
SET BACK *strex*
SET BACKGROUND COLOR *strex*

SET BACK (SET BACKGROUND COLOR) with *rnumex* sets the background color to the color corresponding to *rnumex*. Valid color numbers are those in the range 1 through the maximum color number. You can determine the maximum color number through an ASK MAX COLOR statement. Numbers outside this range will have effects that are dependent on your particular machine. If your machine does not support color, True BASIC may supply a suitable patterm. The default background color is -2

SET BACK (SET BACKGROUND COLOR) with *strex* sets the background color to the color named in the *strex* (upper, lower, or mixed case may be used.)

SET BACK (SET BACKGROUND COLOR) sets the color for the current physical window, including any logical windows it may contain. However, the background color is generally only used when a portion of a window needs to be cleared; therefore, it is possible to limit its apparent effect by carefully controlling use of the CLEAR and BOX CLEAR statements.

See the SET COLOR statement for a list of legal color names.

Exception: -11008 No such color.

SET COLOR Statement

SET COLOR *rnumex*
SET COLOR *strex*

SET COLOR with *rnumex* sets the foreground color to the color that corresponds to *rnumex*. Valid color numbers are those in the range 0 through the maximum color number. You can determine this maximum color number with an ASK MAX COLOR statement. 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. The default color is -1.

SET COLOR with *strex* sets the foreground color of the current logical window to the color named in the *strex*, which must evaluate to one of the following (upper, lower, or mixed case may be used):

MAGENTA	CYAN	WHITE	
RED	BLUE	GREEN	BACKGROUND
YELLOW	BROWN	BLACK	

A *strex* that names two colors separated by a slash can set the foreground and background colors at the same time. For example:

```
   SET COLOR "RED/GREEN"
```

will set the foreground color to red and the background color to green.

True BASIC may substitute for a color not available. For example, MAGENTA may be substituted for RED, if RED is not available. The value BACKGROUND refers to the current background color. Set COLOR 0 will set the color to that of color mix table entry 0, which is not necessarily the background color.

Exceptions: -11008 No such color.

SET COLOR MIX Statement

SET COLOR MIX (*rnumex*) *numex*, *numex*, *numex*

Sets the intensities of the color whose number is given by (*rnumex*). The three remaining arguments represent, respectively, the intensities of the red, green, and blue components of the color. Values < 0 will be replaced by 0, and values > 1 will be replaced by 1. Other values will be adjusted to an available intensity. On some machines, or in some modes, this statement has no effect.

For example, suppose that a particular graphics mode allows each of the three colors to take on the following four intensities: 0, 1/3, 2/3, and 1. There then would be 4*4*4 = 64 different colors or hues, although they might not all be available simultaneously. The value of the *numex* for each color will be converted to one of these four intensities as shown below; these values will be returned in a subsequent ASK statement.

```
   ! The SET COLOR intensity value is x
   If x < .25 THEN
      LET intensity = 0
   ELSE IF x < .50 THEN
      LET intensity = 1/3
   ELSE IF  x < .75 THEN
      LET intensity = 2/3
   ELSE
      LET intensity = 1
   END IF
```

Exception: -11000 Can't do graphics on this computer.

SET CURSOR Statement

SET CURSOR *strex*
SET CURSOR *rnumex, rnumex*

SET CURSOR with *strex* sets the mode of the text cursor as follows:

`"OFF"`	Turns the cursor off
Anything else	Turns the cursor on

The value `"OFF"`can be in upper, lower, or mixed case.

SET CURSOR with numeric arguments sets the text cursor to the text cursor position whose row (line) and column (character) are given, respectively, by the two arguments. The top row of the logical window is 1, and the same for the left most column.

Exceptions: 8601 Cursor set out of bounds.
-11000Can't do graphics on this computer.

SET DIRECTORY Statement

SET DIRECTORY *strex*

SET DIRECTORY allows the program to change to another directory. The *strex* must contain the new directory name using the conventions of the operating system.

Exceptions: 9002 Trouble using disk or printer.
9008 No such directory.

SET MARGIN Statement

SET MARGIN *rnumex*
SET #*rnumex*: MARGIN *rnumex*

SET MARGIN without a channel changes the right margin for PRINT statements for the current logical window or standard output. The margin must not be less than the current zonewidth.

SET MARGIN with a channel that refers to a text file sets the right margin for PRINT statements outputting to that file. The effect of the margin with text files is identical to that with the screen. The margin for a text file must be no less than the current zonewidth for that text file.

If you wish to output strings of arbitrary length to a file or printer, you can prevent unwanted *end-of-lines* by using SET MARGIN MAXNUM.

SET MARGIN with a channel that refers to a logical window sets the margin for that window. SET MARGIN #0 refers to the default logical window.

Exceptions:	4006	Margin less than zonewidth.
	7312	Can't set ZONEWIDTH or MARGIN for this file.
	7313	Can't set ZONEWIDTH or MARGIN for INPUT file.

SET MODE Statement

SET MODE *strex*

There are several modes. They may be expressed in uppercase, lowercase, or mixed case. Other modes will be ignored.

COLORDOS16	This is the default mode, and conforms to the colors that have always been present in the EGA or VGA modes of our DOS product. (There are still "maxcolor" entries in the color mix table; this simply fills in the first 16.
COLOR16	This is another 16-color mode, with the colors arranged in a rainbow fashion. (There are still "maxcolor" entries in the color mix table; this simply fills in the first 16.)
COLOR256	This is like COLOR16 but with 256 colors instead of 16 colors.
COLORSYSTEM	This fills the color mix table with the *currently defined* system colors.
COLORSTANDARD	This fills the color mix table with the *default* system colors.

SET NAME Statement

SET NAME *strex*

SET NAME is provided for compatibility with earlier versions of True BASIC; it is ignored by this version of the language.

SET POINTER Statement

SET #*rnumex*: POINTER *record-setter*
SET #*rnumex*: *io-recovery*
SET #*rnumex*: POINTER *record-setter*, *io-recovery*

record-setter::	BEGIN
	END
	NEXT
	SAME
io-recovery::	IF MISSING THEN *action*
	IF THERE THEN *action*
action::	EXIT DO
	EXIT FOR
	line-number

If the POINTER clause is present, the file pointer is set according to the *pointer-keyword*, which must be one of the following (these may be in upper, lower, or mixed case):

BEGIN	Sets the pointer to the beginning of the file
END	Sets the pointer to the end of the file
NEXT	Sets the pointer to the next record of the file
SAME	Sets the pointer to the record just processed

BEGIN and END can be used with all types of files. Use END if you want to append to the end of the file.

NEXT and SAME can be used only with record files. NEXT does not move the file pointer forward one record except when the previous file operation resulted in an exception. SAME moves the file pointer backward one record. Thus, if you wish to read the record you have just written, use

```
WRITE #1: x
SET #1: POINTER SAME
READ #1: samex
```

If the IF MISSING THEN clause is present, the indicated *action* is taken if the file-pointer is at the END of the file. If the IF THERE clause is present, the indicated *action* is taken if the file-pointer is not at the END of the file.

Exceptions:	7002	Can't use #0 here. (nonfatal)
	7202	Must be RECORD or BYTE for SET RECORD.
	7204	Can't use SAME here.
	-7252	File pointer out of bounds.
	9002	Trouble using disk or printer.

SET RECORD Statement

SET #*rnumex*: RECORD *rnumex*

Sets the pointer to the desired record in a RANDOM or RECORD file, or to the desired byte in a BYTE file. The value must be in the range (1, filesize + 1). To add a record to the end of a record file #1, you can use:

```
ASK #1: FILESIZE n
SET #1: RECORD n+1
```

or:

```
SET #1: POINTER END
```

Exceptions:	7002	Can't use #0 here. (nonfatal)
	7202	Must be record or byte file for SET RECORD.
	-7252	File pointer out of bounds.

SET RECSIZE Statement

SET #*rnumex*: RECSIZE *rnumex*

This statement sets the record size for a RANDOM, RECORD or BYTE file. If the file is a RANDOM or RECORD file, then it must be newly created or empty. The RECSIZE must be set before you can WRITE to an empty RANDOM or RECORD file. SET RECSIZE is not needed for a BYTE file if you use the BYTES clause on the READ statement.

Each record in a RANDOM file holds an arbitrary number of numeric or string values, which can be freely mixed. SET RECSIZE for a RANDOM file must include the size of the items being stored plus all the hidden bytes.

The space required within a RANDOM file record is as follows:

ITEM	**LENGTH (bytes)**	**RECSIZE (bytes)**	**ACTUAL (bytes)**
File header	5	0	5
Number	8	9	9
String	length	length + 4	length + 4
End of record	1	1	1

Thus, for a RANDOM record to contain two numbers and three strings of length 12 characters, its RECSIZE must be at least

```
   recsize >= 2*(8 + 1) + 3*(12 + 4) + 1 = 67 bytes.
```

(The actual total storage used will be `5 + filesize*recsize` bytes.)

Each record in a RECORD file holds a single number or a single string. RECSIZE must reflect only the length of a number (8) or the length (LEN) of the string. Numbers and strings can be freely mixed within the same RECORD file, so that the RECSIZE must reflect the largest of these.

The space required within a RECORD file record is as follows:

ITEM	**LENGTH (bytes)**	**RECSIZE (bytes)**	**ACTUAL (bytes)**
File header	5	0	5
Number	8	8	12
String	length	length	length + 4
End of record	0	0	0

Thus, the RECSIZE for a RECORD file must be, for numbers, at least 8, and, for strings, at least the maximum length of the strings. (The actual total storage used will be 5 + filesize*(recsize + 4) bytes.

Notice that RECSIZE is different for RANDOM and RECORD files. For RANDOM files, RECSIZE must include the extra characters. For RECORD files, RECSIZE includes only the length of the item to be stored.

If you are working with a BYTE file, the SET RECSIZE statement can be used to specify the size of each READ statement. The BYTE file need not be empty, and its RECSIZE can be changed at any time. (See the READ statement.)

Exceptions:	7002	Can’t use #0 here. (nonfatal)
	-7250	Can’t SET RECSIZE on non-empty RECORD file.
	-7251	Must be BYTE file or empty for SET RECSIZE.

SET TEXT JUSTIFY Statement

SET TEXT JUSTIFY *strex, strex*

Sets the text placement point for subsequent PLOT TEXT statements.

The first *strex* specifies the horizontal position and must take on one of the following values, which may be in upper, lower, or mixed case:

`"LEFT"`	The point is at the left edge of the text
`"RIGHT"`	The point is at the right edge of the text
`"CENTER"`	The point is at the center of the text

The second *strex* specifies the vertical position and must take on one of the following values:

`"TOP"`	The point is at the top of the text
`"BOTTOM"`	The point is at the bottom of the text, i.e., the lowest pixel
`"BASE"`	The point is at the baseline of the text, i.e., the lowest pixel of uppercase letters
`"HALF"`	The point is at the center of the text

Exception:	4102	Improper TEXT JUSTIFY value.
	-11000	Can’t do graphics on this computer.

SET WINDOW Statement

SET WINDOW *numex*, *numex*, *numex*, *numex*

Sets the user coordinates for graphics in the current logical window. The coordinates are specified in the order: left edge, right edge, bottom edge, top edge. The edges may be out of order for picture reversal, but may not be equal. That is, the left edge can be greater numerically than the right edge, and the same for the bottom and top edges.

Exceptions:	-11000	Can’t do graphics on this computer.
	-11001	Window minimum = maximum.
	11004	Can’t SET WINDOW in picture.

SET ZONEWIDTH Statement

SET ZONEWIDTH *rnumex*
SET #*rnumex*: ZONEWIDTH *rnumex*

SET ZONEWIDTH sets the zonewidth for PRINT statements in the current logical window. The zonewidth must be greater than zero but not greater than the current margin.

SET ZONEWIDTH used with a channel that refers to a text file sets the zonewidth for PRINT statements outputting to that file. The effect of the zonewidth with text files is identical to that with the screen. The zonewidth for a text file must be greater than zero but not greater than the margin for that file.

SET ZONEWIDTH with a channel that refers to a logical window sets the zonewidth for that window. SET ZONEWIDTH #0 refers to the default logical window.

Exception: 4007 ZONEWIDTH out of range.

SGN Function

SGN(*numex*)

`SGN(x)` returns the “sign” of `x`. SGN can be defined in terms of other True BASIC statements as follows:

```
DEF SGN(x)
    IF x < 0 THEN LET SGN = -1
    IF x = 0 THEN LET SGN =  0
    IF x > 0 THEN LET SGN = +1
END DEF
```

SHARE Statement

SHARE *share-item* …, *share-item*

share-item::	*simple-numvar*
	simple-strvar
	array bounds
	#integer

The SHARE statement can occur as a part of a *module-header* and is used to name the variables, arrays, and channels that can be shared among the procedures of the module. A *share-item* cannot also appear in a PUBLIC or DECLARE PUBLIC statement. A shared *array* cannot also appear in a DIM statement in the same *module-header* or *program-unit*. The appearance of an *array bounds* in a SHARE statement causes the array to be dimensioned, as in a DIM statement.

A *share-item* that appears in a SHARE statement in the module header retains its value between calls to the procedures of the module. The appearance of the *share-item’s* name in a procedure of the module shall be a reference to the module’s *share-item* itself, provided that the name does not appear as a *parameter* in the SUB, DEF, FUNCTION, or PICTURE statement for that procedure, and does not appear in a DIM or LOCAL statement in that procedure.

A *share-item* that appears in a SHARE statement in a procedure retains its value between invocations of that procedure.

SIN Function

SIN(*numex*)

Returns the sine of the angle *numex*. The angle is measured in radians unless OPTION ANGLE DEGREES is in effect, in which case the angle is measured in degrees. For example, if OPTION ANGLE DEGREES is in effect, then `SIN(45)` is 0.707107...; if OPTION ANGLE RADIANS is in effect, then `SIN(1)` = 0.841471...

SINH Function

SINH(*numex*)

Returns the value of the hyperbolic sine function. For example, `SINH(1)` is 1.17520...

SINH may be defined in terms of other True BASIC functions as follows:

```
DEF SINH(x) = (EXP(x) - EXP(-x))/2
```

Exception: 1003 Overflow in numeric function.

SIZE Function

SIZE(*arrayarg*, *rnumex*)
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 *rnumex*. If there is no second argument, or if the second argument is zero, then SIZE returns the total number of elements in the entire array. For illegal second arguments, SIZE returns -1. `SIZE(A,n)` can be defined in terms of the UBOUND and LBOUND functions as follows:

```
DEF SIZE(A,n) = UBOUND(A,n) - LBOUND(A,n) + 1
```

For example:

```
! If the OPTION and DIM statements are
OPTION BASE 1
DIM A(2:5, -3:10), V(10)
! then the SIZE function will have these values
! SIZE(A,1) =  4
! SIZE(A,2) = 14
! SIZE(A)   = 56
! SIZE(V)   = 10
```

Exception: 4004 SIZE index out of range.

(See also LBOUND and UBOUND.)

Socket Subroutine *available only in Gold Edition; see Chapter 25 of Gold manual.*

SOUND Statement

SOUND *numex*, *numex*

The SOUND statement sounds a note with frequency (in Hertz) given by the first *numex* and duration (in seconds) given by the second *numex*. The duration may be fractional. A negative duration is converted to 0.

True BASIC continues executing statements while the sound is being generated. The note stops when its time runs out, or when a new SOUND statement with a zero duration is given. If a subsequent SOUND statement with a non-zero duration is encountered before the first note stops, True BASIC waits for the first note to stop, then starts the second note, and continues executing statements. The SOUND statement is ignored on systems that don't support sound. (See also the PLAY statement.)

SQL Subroutine *available only in Gold Edition; see Chapter 26 of Gold manual.*

SQR Function

SQR(*numex*)

`SQR(x)` returns the positive square root of `x`, where *x* must be greater than or equal to zero. For example:

`SQR(196)`	returns 14
`SQR(0)`	returns 0
`SQR(17)`	returns 4.12311, approximately
`SQR(-1)`	will cause an exception

The SQR function may be thought of as raising to the 1/2 power, as in:

```
DEF SQR(x) = x^(1/2)
```

This form suggests that a general *n*-th root function can be defined as:

```
DEF ROOT(x,n) = x^(1/n)
```

Exception: 3005 SQR of negative number.

STOP Statement

STOP

Stops execution of the program.

STR$ Function

STR$(*numex*)

Returns the number converted to a string. The number is formatted just as for the PRINT statement, except that leading and trailing spaces are omitted. For example:

`STR$(pi)`	returns "3.1415927"
`STR$(10000000000)`	returns "1.e+10"
`STR$(1e5)`	returns "100000"
`STR$(-1e5)`	returns "-100000"

`STR$(x)` converts a numeric value *x* into a string as follows:

1. If x is negative, it is made positive and the minus sign is attached later.
2. If x is an integer and contains 8 or fewer decimal digits, it is converted into those digits without a decimal point.
3. If x is a non-integer and contains 0 through 8 decimal digits before the decimal point, it is converted into an 8-digit form with the decimal point at the appropriate place, as in ".12345678" or "12345678.". Trailing zeroes, if any, are removed, as in "12.34" in place of "12.340000".
4. If x is a non-integer and contains from 1 to 4 digits after the decimal point and there are at least that number of trailing zeroes in the 8-digit representation, it is converted to a form similar to that in case 3, as in ".00123" or ".000012".
5. If x is not covered by one of the cases in 2 through 4, it is converted to scientific notation of the form "1.2345678e+3", which is read as "1.2345678 times ten to the power +3." In addition, the digit before the decimal point is not "0". Trailing zeroes, if any, are removed. The sign of the exponent is always included, and the exponent contains from one or more digits (i.e., contains no leading zeroes.)
6. If the minus sign was removed in step 1, it is now reattached, as in "-1.2345678".

STRWIDTH Function

STRWIDTH(*numex*, *strex*)

This function returns the length of a string, in pixels, with reference to the current font, font style, and font size, in a physical window. If the value of the first argument is not the ID number of a physical window that currently exists, an error occurs. Note that the standard output window (the default physical window) is always opened upon program startup (although it might not yet be visible,) and that this window has 0 as its ID.

Exceptions:	-11220	Unknown or invalid ID.
	-11221	Cannot reference a freed object ID.

SUB Structure

sub-structure::	*sub-statement*
	… *statement*
	END SUB
sub-statement::	SUB *identifier*
	SUB *identifier* (*subparmlist*)
subparmlist::	*subparm* …, *subparm*

subparm::	*simple-numvar*
	simple-strvar
	arrayparm
	#integer

The arguments in the CALL statement are matched with the *subparms*, the first argument with the first parameter, and so on, and must agree in type. Arguments that are *numvars*, *strvars*, *arrays*, or channels are passed to the corresponding *subparm* ***by reference***. That is, any changes in the value of a *subparm* also changes the corresponding argument. Arguments that are more general expressions are evaluated and stored upon entry to the subroutine, and are therefore said to be passed ***by value***.

If the *subroutine* is internal, the variables and arrays that appear in the body of the *subroutin*e, but not in the SUB statement or in a DIM, LOCAL, or DECLARE PUBLIC statement within the *subroutine*, refer to variables and arrays in the containing *program-unit*. If the *subroutine* is *external*, then such variables and arrays will be local to the *subroutine* and will have their usual default initial values (i.e., 0 or null) upon entry to the *subroutine*.

The statements of the *subroutine* are then executed in order. When the END SUB statement is reached, or when an EXIT SUB statement is executed, control passes to the statement following the CALL statement that invoked the *subroutine*.

SYSTEM Subroutine

CALL System (*numex*, *strex*, *starget*, *starget*)

The System subroutine accesses certain file management facilities provided by the operating system.

```
CALL System (op, result1$, result2$, result3$)
```

The System subroutine provides access to the following file management operations:

op	Results
0	Ask directory (ASKDIR), current directory returned in `result1$`.
1	Change directory (CHDIR) to a new directory named in `result1$`.
2	Read the current directory. The template is specified in `result1$`. The file names that match the template and all directory names are returned in `result2$`. The file statistics are returned in `result3$`, except that directory names that match the template have `"d"` as the first character of the type, while all other directories have a `"D"` as their type. This option can be used, and is used by **ExecuLib**, to “climb” a directory tree. (See below for the formats of `result2$` and `result3$`.)
3	Make a new directory (MKDIR).The new directory name is specified in `result1$`.
4	Read the current directory (READDIR). The template is specified in `result1$`. The file names are returned in `result2$`. The file statistics are returned in `result3$`. (See below for the formats of `result2$` and `result3$`.)
5	Rename a file (RENAME). The old name is specified in `result1$`, and the new name is given in `result2$`. (Here, the third argument is a *strex*.)
6	Remove a directory (RMDIR.). The name of the directory to be removed is specified in `result1$`.
7	Remove a file (DEL). The name of the file to be removed is specified in `result1$`.
8	Ask utilization of the disk. `result1$` is the space in use, `result2$` is the free space, and `result3$` is ignored. Units are in bytes.
9	Set the date. `result1$` is the new date in the format "YYYYMMDD". `result2$` and `result3$` are ignored. If the format of the new date is wrong, then an exception occurs.
10	Set the time. `result1$` is the new time in the format "HH:MM:SS". `result2$` and `result3$` ignored. If the format of the new time is wrong, then an expection occurs.

The file names resulting from `op` = 2 or `op` = 4 are contained in a single string with the names separated by the system end-of-line for text files. (On some systems, this end-of-line is ASCII character 13 followed by ASCII character 10. On other systems it is simply ASCII character 13. Currently, using `op=2` will return the file and directory names in the current directory, regardless of the template, but will not actually climb. You can use the subroutine **Exec_ClimbDir** for this purpose.)

The template is specified in a standard form across platforms. For example, "*.tru" will yield file names whose extensions is ".tru"; note that the "*" is a "wild card" that matches anything.

The file statistics resulting from `op` = 2 or `op` = 4 are contain in a single string with the records (one per file) separated by the system end-of-line. Each record has a fixed format, except for the file name, which occurs at the end of the record. (In the following scheme the underline (_) stands for a character in a field. The hyphen (-) stands for a space between fields. The pound sign (#) stands for ASCII character 10, for which a printing graphic may not exist.)

```
type     size   day mon dy     time year   filename
____-__________-___-___-__-_________-____#--________
```

The `type` is a four-character string. The first character is a 'd' for a directory, and a '-' for a file. The second character is an 'r' if read permission is available on the file or directory, and a '-' if not. The third character is a 'w' if write or modify permission is available on the file or directory, and a '-' if not. The fourth character is an 'x' if the file can be executed, and '-' otherwise; a directory cannot be executed.

The `size` is the size of the file using the system conventions. For OS2 and Windows, the size is given in bytes. For Unix systems, the size is given in blocks, which usually consist of 512 bytes.

The `day`, `mon`, `dy`, and `time` fields contain information on the date and time of the last modification. The `day` field contains the first three letters of the day name.

The `mon` field contains the first three letters of the month name.

The `dy` field contains the day number, between 1 and 31, right justified.

The `time` field contains the time in the 24-hour format HH:MM:SS.

The `year` field contains the four-digit year number.

For example:

```
    -rw-       497  Tue Jul 12 08:51:32 1994#  filename.tru
```

means that the entity whose name is "filename.tru" is a file with read and write permission available, that its size (on non-Unix systems) is 497 bytes, and that it was last modified at 8:51:32 on Tuesday, July 12, 1994.

Exceptions:	-11267	Unknown or invalid directory.
	-11268	Can't get current directory.
	-11269	Unknown option for SUB System.
	-11270	Can't get STAT info for file in directory.

See also:
Exec_AskDir, Exec_ChDir, Exec_ClimbDir, Exec_MkDir, Exec_ReadDir, Exec_Rename, Exec_RmDir, all of which are packaged in the Library file ExecLib.TRU and ExecLib.TRC and are described in Chapter 22 "Interface Library Routines." (The True BASIC statement UNSAVE provides for the direct deletion of a file.)

Sys_Event Subroutine

CALL Sys_Event (*numex*, *starget*, *numvar*, *numvar*, *numvar*)

As the user of the program manipulates the mouse, clicking it on various controls, selecting windows, etc., these activities are reported back to the program as events. In True BASIC, these events do not generate interrupts, but rather are placed on a single queue (list). Calling the built-in subroutine Sys_Event allows the program to examine the event, if any, at the front of the list.

This model, examining one at a time the events from the event queue by calling the subroutine Sys_Event, provides the simplest possible way to respond to the many things that can happen within the user interface. These events occur asynchronously and can be reported in an order different from that in which the user intended.

For example, a typical use might be

```
   CALL Sys_Event (timer, event$, window, x1, x2)
```

If `timer` is > 0, Sys_Event will return when that amount of time has elapsed, regardless of whether there is an event on the event queue. If `timer` = 0, then Sys_Event will return immediately, again regardless of whether there is an event on the event queue.

The variable `event$` will contain the event name, capitalized, unless there has been no event, in which case it will contain the null string. `Window` will contain the number of the physical window in which the event occurred. `X1` and `x2` contain related information the exact nature of which depends on the type of the event.

This subroutine is used in conjunction with the Object subroutine, and cannot be used separately. The window number is the ID of a physical window created by the Object subroutine, and bears no relation with True BASIC's logical windows accessed by the WINDOW statement.

A complete and detailed discussion of the Sys_Event subroutine can be found in Chapter 20 "Sys_Event Subroutine."

TAB Function

TAB(*rnumex*)
TAB(*rnumex*, *rnumex*)

TAB can appear only in PRINT statements. Strictly speaking, TAB is not a function, as it does not return a value. TAB can take either one or two arguments.

`TAB(c)` causes the printing cursor to "tab" over to print position (column) `c`. (The first printing position is 1, the second 2, and so on.) It can appear in PRINT statements with or without channel expressions, which can refer to either text files or logical windows.

If the printing cursor is already beyond column `c`, then the printing cursor will first go to a new line and then tab to column `c`. If the printing cursor is already exactly at `c`, no action takes place. If `c` represents a position less than the printing cursor, then subsequent printing will start on the next line at the position specified. If `c` is greater than the margin, it is replaced by MOD(`c` – 1, margin) + 1. For `TAB(c)`, if `c` is less than 1, then the effect is the same as `TAB(1)` *unless* the PRINT statement is continued in a WHEN structure, and an exception occurs.

`TAB(r,c)` causes the printing cursor to be positioned on the screen at row r and column c of the current logical window. (The columns are numbered as with `TAB(c)`. The rows are numbered from one starting at the top of the window.) `TAB(r,c)` can appear only in PRINT statements without channel expressions. `TAB(r,c)` is equivalent to `SET CURSOR r,c`.

For `TAB(r,c)`, `r` must be in the range 1 ≤ `r` ≤ number of rows, and c must be in the range 1 ≤ `c` ≤ MIN(margin of the current logical window, number of available columns in the screen).

Exceptions:	4005	TAB column less than 1. (nonfatal)
	9601	Cursor set out of bounds.

(See also the SET and ASK CURSOR statements.)

TAN Function

TAN(*numex*)

`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.

For example, if OPTION ANGLE DEGREES is in effect, then `TAN(45)` is 1; if OPTION ANGLE RADIANS is in effect, then `TAN(1)` is 1.5574077...

Exception:	1003	Overflow in numeric function.

TANH Function

TANH (*numex*)

Returns the value of the hyperbolic tangent function. For example, `TANH(1)` = .76159416...

TANH may be defined in terms of other True BASIC functions as follows:

```
DEF TANH (x) = (EXP(x) - EXP(-x))/(EXP(x) + EXP(-x))
```

Exception: 1003 Overflow in numeric function.

TBD Subroutine

CALL TBD (*numex*, *numex*, *numex*, *strex*, *strex*, *strex*, *strex*, *strex*, *numex*, *numex*, *numex*, *numvar*)

The TBD subroutine is a built-in subroutine that displays several types of modal dialog boxes. (A modal dialog box is one in which control is retained in the dialog box until the user exits it and the box is closed. That is, no other activities can occur until the dialog box closes.)

The calling sequence is:

```
CALL TBD(x, y, type, title$, msg$, btn$, name$, text$, st, dflt, timeout,
result)
```

The TBD subroutine is capable of producing four different types of dialog boxes – ***standard*** dialog boxes, ***open file*** dialog boxes, ***save file*** dialog boxes, and ***list dialog*** boxes. The value of type determines the type of dialog box that will be produced, and must be a value between 1 and 4, inclusive.

A complete and detailed discussion of the TBD subroutine can be found in Chapter 21 “TBD Subroutine.”

TBDX Subroutine

CALL TBDX *(numex, numex, numex, numex, strex, numarr, numex, strex, strex, strex, strex, strex, numex, numex, numex, numvar)*

The TBDX subroutine is like the TBD subroutine, but gives more control over location and other options. The first four parameters specify the location of the dialog box in pixels using pixel screen coordinates (the origin is the upper-left corner of the screen.) If all four values are equal to -1, True BASIC will choose a default location the same as is done for TBD. If the programmer specifies -1 for the second and third location parameters, True BASIC will select a default size for the dialog box and will place its upper-left corner of the screen at pixel location (l,t).

The use of the parameters, parm1$ and parm2(), are explained in Chapter 21. They apply only to dialog boxes of types 1 or 4 (standard or list.)

The remaining parameters are the same as for the TBD subroutine.

The calling sequence is:

```
CALL TBDX(l,r,b,t,parm1$,parm2(),type,title$,msg$,btn$,name$,text$,st,
dflt,timeout,result)
```

Like the TBD subroutine, the TBDX subroutine is capable of producing four different types of dialog boxes – standard dialog boxes, open file dialog boxes, save file dialog boxes, and list dialog boxes. The value of type determines the type of dialog box that will be produced, and must be a value between 1 and 4, inclusive.

A complete and detailed discussion of the TBDX subroutine can be found in Chapter 21 “TBD Subroutine.”

Exceptions: -11223 Attribute not used for specified object.
-11273 Not enough values for attribute list in SET/GET.

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.

The TIME function is useful in timing loops. Its resolution (that is, the “tick” interval) varies from machine to machine but is usually accurate to a tenth of a second or better.

TIME$ Function

TIME$

A no-argument function, TIME$ returns a string that contains the time as measured by the 24-hour clock. For example, at 1:45 and 2 seconds P.M., TIME$ = "13:45:02", and at midnight, TIME$ = "00:00:00". If your computer does not have a clock, then TIME$ returns "99:99:99".

TRACE Statement

TRACE ON
TRACE ON TO #*rnumex*
TRACE OFF

This statement is included only for compatibility with the ANSI Standard. Its use is not recommended.

If debugging is active in the *program-unit*, execution of the TRACE ON statement causes the result of subsequent statements to be printed.

For assignment statements, the names of the variables changed and their new values are printed. For a statement that causes a transfer to other then the following statement the line-number of the next statement is printed. If the *program-unit* does not have line-numbers, the ordinal number of the line in the file containing the *program-unit* is printed.

Once started, tracing continues until a TRACE OFF or DEBUG OFF statement is executed, or the *program-unit* is exited.

If the file reference is present in the TRACE ON statement, the trace reports are directed to that file, which must be a TEXT file open and available for output in that *program-unit*.

Execution of a TRACE OFF statement stops tracing.

If debugging is inactive for the *program-unit*, the TRACE ON and TRACE OFF statements have no effect. (See the BREAK and DEBUG statements.)

Exceptions:	7302	Can't print to input file.
	7401	Channel is not open for TRACE.
	7402	Wrong file type for TRACE.

TRIM$ Function

TRIM$(*strex*)

Returns the value of the argument but with all leading and trailing blank spaces removed. For example:

```
TRIM("     a b c      ")              returns "a b c"
```

TRIM$ may be defined in terms of other True BASIC functions as follows:

```
DEF TRIM$(a$) = LTRIM$(RTRIM$(a$))
```

TRN Array Function

TRN(*numarr*)

Returns the transpose of its argument, which must be a two-dimensional numeric array. For example:

$$\text{If } A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix}, \text{ then TRN(A)} = \begin{pmatrix} 1 & 4 \\ 2 & 5 \\ 3 & 6 \end{pmatrix}$$

TRUNCATE Function

TRUNCATE(numex, rnumex)

`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)`. For example:

```
TRUNCATE(123.4567, 3)          returns 123.456
TRUNCATE(-123.4567, 3)         returns -123.456
```

```
    TRUNCATE(123.4567, -2)            returns 100
    TRUNCATE(123.4567. 0)             returns 123
```

TRUNCATE can be defined in terms of the IP function as follows:

```
DEF TRUNCATE(x,n) = IP(x*10^n)/10^n
```

Exception: 1003 Overflow in numeric function.

(See also ROUND.)

UBOUND Function

UBOUND(arrayarg, rnumex)
UBOUND(arrayarg)

The two-argument form returns the largest value (upper bound) allowed for the subscript in the dimension specified by *rnumex* in the array named. The one-argument form returns the largest value (upper bound) for the subscript in the array, which must be one-dimensional (i.e., a vector).

For example:

```
DIM a(3), b(1970:1980, -3:3)
PRINT UBOUND(a)
PRINT UBOUND(b,1)
PRINT UBOUND(b,2)
```

produces the following output:

```
3
1980
3
```

Exception: 4009 UBOUND index out of range.

(See also SIZE and LBOUND.)

UCASE$ Function

UCASE$(*strex*)

Returns the value of *strex* 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 left unchanged. (This function may thus fail to work properly on other character sets.) For example:

```
    UCASE$("Bob Smith is 65.")        returns "BOB SMITH IS 65."
```

(See also LCASE$.)

UNPACKB Function

UNPACKB(*strex*, *rnumex*, *rnumex*)

True BASIC provides two routines, PACKB and UNPACKB, for packing and unpacking numeric integer values as bits in a string. UNPACKB is a function, while PACKB is a callable subroutine.

The value produced by `UNPACKB (s$, bstart, blen)` is given by a process that may be described as follows:

1. `Bstart` and `blen` are first rounded.
2. If `bstart` is less than 1, it is replaced by 1.
3. If `bstart` is greater than the number of bits in the string `s$` (which is always a multiple of eight) then it is replaced by the actual number of bits in `s$` plus 1.
4. If the length of the bit field, as determined by `ABS(blen)`, extends beyond the end of the string `s$`, the nonexistent bits are treated as zeros.
5. If `blen` is positive, the bit field is treated as an unsigned integer. Thus, the value returned will lie in the range 0 to `2^blen` -1, inclusive.

6. If `blen` is negative, its absolute value is used as the field length, and the bit field is treated as a two's-complement signed integer. Thus, the value returned will lie in the range -(2^(`blen`-1)) to 2^(`blen`-1) -1, inclusive.
7. If `blen` is 0, then 0 is returned.
8. `Blen` should be limited to the range -32 to +31, inclusive. Values outside this range may return different values on different machines.

For example, assume the bits of the string `s$` are 0101010110101010, then

`LEN(s$)`	returns 2
`UNPACKB(s$,1,8)`	returns 85
`UNPACKB(s$,9,8)`	returns 170
`UNPACKB(s$,1,16)`	returns 21930
`UNPACKB(s$,1,-8)`	returns 85
`UNPACKB(s$,9,-8)`	returns -85
`UNPACKB(s$,9,1)`	returns 1
`UNPACKB(s$,10,1)`	returns 0
`UNPACKB(s$,37,1)`	returns 0

(See also the PACKB subroutine.)

UNSAVE Statement

UNSAVE *strex*

Unsaves (deletes) the file named. It is a good idea to make sure that the file is not currently open, as this causes problems for some operating systems.

Exception:	9002	Trouble using disk or printer.
	9003	No such file.

USE Statement

The USE statement can occur only as a part of a WHEN structure and is required. (See the WHEN structure.)

USING$ Function

USING$*(strex, expr …, expr)*

expr::	*numex*
	strex

USING$ returns the string of characters that would be produced by a PRINT USING statement with *strex* as the format string and with the *exprs* as the numeric or string expressions to be printed. That is, each of the following statements:

```
PRINT USING f$: "The answer is ", x
PRINT USING$(f$, "The answer is", x)
```

will produce the same printed result.

If the number of expressions is less than the number of fields in the format string, then the rest of the format string, starting with the first unused field, is ignored. If the number of expressions is greater than the number of fields in the format string, then the format string is reused, as many times as needed.

USING$ produces a string that you can use in WRITE and PLOT TEXT statements as well as in PRINT statements. You can also manipulate the characters of the string to produce special formatting effects. For detailed rules and examples for USING$, see Appendix D, which explains USING$ in terms of the equivalent PRINT USING statement.

If a field is inadequate (too short, not enough "^" for an argument, no "–" for a negative number), then that field will be filled with asterisks, but no exception will occur.

Exceptions:	8201	Badly formed USING string.
	8202	No USING item for output.

VAL Function

VAL(*strex*)

Returns the numerical value given by *strex*, 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. For example:

`VAL("1e5")`	returns 100000
`VAL(" 123.5 ")`	returns 123.5
`VAL("1,234,567.89")`	will cause an exception
`VAL("Hello")`	will cause an exception

Exceptions:	1004	Overflow in VAL.
	4001	VAL string isn't a proper number.

WHEN Structure

A WHEN structure, which is used for handling runtime errors (exceptions), has two forms:

when-structure:: WHEN EXCEPTION IN
protected-part
USE
handler-part
END WHEN

WHEN EXCEPTION USE *handler-name*
protected-part
END WHEN

protected-part:: ... *statement*

handler-part:: ... *statement*

handler-name:: *identifier*

The keyword ERROR may be used in place of the keyword EXCEPTION.

When the WHEN structure is encountered, the statements in the *protected-part* are executed. If no error (exception) occurs and the USE line or END WHEN line is reached, execution continues at the line following END WHEN.

If an exception occurs, control then passes to the *handler-part* following the USE statement, or passes to the exception handler named in the WHEN EXCEPTION USE statement.

If the END WHEN statement is reached, the exception is "turned off" and control passes to the first statement after END WHEN.

An error (exception) that occurs while the statements of the *handler-part* are being executed will not be handled by that WHEN structure, but will be treated as a new error, which must be handled by a higher-level enclosing WHEN structure, or which will stop the program and print an error message.

If a runtime error occurs in a low-level subroutine, True BASIC examines the CALL statements (or DRAW statements or function call) that invoked the low-level subroutine, starting at that low-level subroutine. If any of the invoking statements is contained in the *protected-part* of a WHEN structure, then the *handler-part* of that WHEN structure, or of the named detached handler, is used.

As an example, if you have a large program with many subroutines, and if the top-level CALL statement is contained in the *protected-part* of a WHEN structure, then that WHEN structure can be used to handle all runtime errors that occur anywhere in the program. In addition, you can use a CAUSE ERROR statement to jump back cleanly to the top level of your program, where the WHEN structure will handle it.

An EXIT HANDLER statement in the *handler-part* of a WHEN structure will cause the exception to "recur," as if the WHEN structure were not present.

A CONTINUE statement in the *handler-part* of a WHEN structure will transfer to the statement following the statement being executed when the exception occurred, unless the offending statement is an essential part of loop or choice structure, when the transfer will be made to the statement following the end of the structure.

A RETRY statement in the *handler-part* of a WHEN structure will transfer to the beginning of the statement being executed when the exception occurred.

The values of EXLINE, EXLINE$, EXTEXT$, and EXTYPE are set when the exception occurs, and may be examined in the *handler-part*, or later. They retain their values until a new error occurs. (See the HANDLER structure, and the EXIT HANDLER, RETRY, and CONTINUE statements.)

WINDOW Statement

WINDOW #*rnumex*

The WINDOW statement selects which of the open logical windows will be the current input and output window. Except for WINDOW #0, which is always open, the window must have been opened in a previous OPEN statement. WINDOW #0 refers to the default logical window, which occupies the entire default physical window.

Exception:	7004	Channel isn't open.
	-11005	Channel isn't a window.

WRITE Statement

WRITE #*rnumex*: *expr* …, *expr*
WRITE #*rnumex*, *write-option* …, *write-option*: *expr* …, *expr*

write-option::	*record-setter*
	IF THERE THEN *action*
record-setter::	BEGIN
	END
	NEXT
	SAME
action::	EXIT DO
	EXIT FOR
	line-number
expr::	*numex*
	strex

There can be no more than one of each of the two types of *write-options*. The WRITE statement causes its *exprs* to be evaluated and then writes them to the file referred to by #*rnumex*.

If a *record-setter* is present, then the file-pointer is repositioned before the actual writing begins. If the *record-setter* is BEGIN, the file-pointer is positioned at the beginning of the file. If END, it is positioned at the end of the file. If SAME, it is positioned to where it was at the beginning of the previous READ, WRITE, or similar operation carried out on that channel. If NEXT, the file-pointer is not changed unless it was left in the middle of a record because an exception occurred; in this case, NEXT causes the file-pointer to move to the beginning of the next record.

After the *record-setter*, if any, has been applied, the *action* of the IF THERE clause is carried out if the file-pointer is not at the end of the file. If the *action* is an EXIT DO or EXIT FOR, the WRITE statement must be contained within a DO loop or FOR loop, respectively; if the *action* is a *line-number*, it is treated as a GOTO statement, whose rules it must follow.

For a STREAM file, the values from the WRITE statement are written to the file without regard to records. New values may be added to the end of the file after using SET #*n*: POINTER END.

For a RANDOM file, the values from the WRITE statement form a single record, which is then written to the file. New records may be added to the end of the file after using SET #*n*: POINTER END.

For a RECORD file, the values are written, one to a record, starting with the current record. New records may be added to the end of the file after using SET #*n*: POINTER END.

For a BYTE file, the values of the *exprs* are written as bytes to the file. If the *expr* is numeric, eight bytes in the IEEE floating point representation are written; that is, the eight-byte string produced by the function NUM$() is written. If the *expr* is a string, the characters of the string are written without headers. It is important to remember that, unlike RECORD files, a BYTE file does not keep track of whether particular bytes were originally part of a number or part of a string.

Exceptions:	7004	Channel isn't open.
	7302	Can't output to INPUT file
	7308	Can't PRINT or WRITE to middle of this file.
	8301	Output item bigger than RECSIZE.
	-8304	Must SET RECSIZE before WRITE.
	9001	File is read or write protected.

(In addition, see the exceptions for SET POINTER.)

Write_Image Subroutine

CALL Write_Image (desttype$, imagedata$, filename$)

The Write_Image subroutine converts a BOX KEEP string in imagedata$ from the local format into a graphics image, storing the image in the file whose name is specified. The user should specify the desired type of the image file in desttype$, which may contain one of: "PICT" (Macintosh only,) "MS BMP", and possibly "OS/2 BMP". These file types must be supplied exactly as shown, although they may be lower- or mixed case.

Exceptions:	7104	Wrong type of file.
	9002	Trouble using disk or printer.
	9003	No such file.

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. The dimensions of the array are determined in one of two ways. If a *redim* is present, then an array of those dimensions will be generated. The array being assigned to in the MAT assignment statement will then be resized (see the MAT Assignment statement) to these new dimensions. If there is no *redim*, then the dimensions of the array will match those of the array that is being assigned to in the MAT statement.

Exceptions:	6005	Illegal array bounds.

(See also CON, IDN, and NUL$.)

CHAPTER

19

Object Subroutine

The built-in subroutine **OBJECT** provides access to the creation and manipulation of windows, controls, and selected graphics objects. A single calling sequence is used:

```
CALL Object (method, id, attributes$, values$, values())
```

`Method` is a number between 0..13 and 20..26 (usually represented by a variable) and denotes the ***method*** to be applied to the object or control. The argument `id` is the identification number of the object or control. `Attributes$` is a string expression that contains one or more attributes for which values need to be set (SET method) or interrogated (GET method); if there is more than one, the items in the list are separated by vertical bars "|". Additional string information is communicated through `values$`; again, multiple items are separated by vertical bars "|". Additional numeric information is communicated through a numeric list `values()`.

[!] **Note:** The descriptions in this chapter use the public variable names to refer to methods, etc. They are defined in a module at the end of the file containing True Controls (TrueCtrl.tru).

Physical windows can be created with or without borders, title bars, close boxes, resize boxes, scroll bars, etc. All controls and graphics objects will be created to belong to a certain window (the ***current physical window.***)

The controls include:

- Push buttons
- Radio buttons and radio button groups
- Check boxes
- Horizontal and vertical scroll bars
- Edit fields
- Static text fields
- List boxes
- List buttons
- List edit buttons
- Group boxes
- Text editors
- Icons (not available on all platforms)

The graphics objects include:

- Circles
- Lines
- Rectangles
- Arcs
- Pie segments
- Arrowed lines
- Round rectangles
- Polygons

Polylines
Images (from a file or from box keep strings)

Push buttons can contain a short piece of text. Under certain conditions they can be outlined and activated by pressing the Enter or Return key.

Radio buttons are round, and can come with identifying text to their right. If established in a group of radio buttons, no more than one radio button in the group can be "on" at a time. (Radio buttons that are "on" have a dark circle in their middle.)

Check boxes are small squares, and can come with identifying text to their left. A check box is "on" if an "X" appears in it. On some platforms a check mark may appear instead of an "X".

Horizontal and vertical ***scroll bars*** are equipped with arrows at each end, a ***slider region*** or ***trough***, and a so-called ***thumb*** or ***slider***. Usually, clicking in one of the arrows causes the thumb to move one unit up or down. Clicking in the gray area of the slider region causes the thumb to move several units up or down. Clicking and dragging on the thumb can cause it to move as you direct.

Edit fields are one-line editable text regions. The **OBJECT** routine lets you store a format for each region so that you may later check the text for consistency with the format.

Static text fields display a single text string, which may occupy several lines if the field is high enough. They can not be edited.

List boxes are boxes that contain a possibly scrollable list of choices. Selections are made by clicking. Multiple selections are allowed on some platforms. Selection list boxes can be combined with other controls to build, say, a modeless File Open dialog box. (For modal dialog boxes, see the **TBD** subroutine and the section on True Dials.)

List buttons are small rectangular regions that, when selected, open up into a scrollable list. Selecting an item from the list moves that item to the button area.

List edit buttons are like list buttons including the scrollable list, but the user may edit the button area.

Group boxes are merely rectangular figures, possibly with a title. They can be used to visually organize a set of controls, such as a radio group. They do not actually "combine" elements.

Text editors are complex controls that can be used to construct a wide variety of editors. Features can include: horizontal and vertical scrolling, wrapping of text at a margin, selecting subsets of the text, adding and deleting characters, etc.

Icons are not available.

Methods

The methods include:

Method	**Numeric value**	**Function**
OBJM_CREATE	0	Create an object or control
OBJM_COPY	1	Create a copy of an object or control
OBJM_SET	2	Set one or more attributes
OBJM_GET	3	Get one or more attributes
OBJM_SHOW	4	Show (make visible) an object or control
OBJM_ERASE	5	Hide an object or control
OBJM_FREE	6	Remove an object or control and free the memory
OBJM_SELECT	7	Select a physical window to be active, or to select a control
OBJM_UPDATE	8	Redraw the contents of a physical window
OBJM_SYSINFO	9	Obtain certain system information
OBJM_PRINT	10	Print the contents of a physical window

OBJM_PAGESETUP	11	Display a page setup box
OBJM_REALIZE	12	Realize or append True BASIC's color mix table to the system color mix table.
OBJM_SCROLL	13	Scroll contents of a window

There are additional methods that apply only to text edit controls:

OBJM_TXE_SUSPEND	20	Suspend activity for the text edit control
OBJM_TXE_RESUME	21	Resume activity for the text edit control
OBJM_TXE_ADD_PAR	22	Add a paragraph of text
OBJM_TXE_DEL_PAR	23	Delete a paragraph of text
OBJM_TXE_APPEND_PAR	24	Append a paragraph of text
OBJM_TXE_VSCROLL	25	Scroll the text vertically
OBJM_TXE_HSCROLL	26	Scroll the text horizontally

Remember that the above identifiers are actually variable names that are assigned numeric values in the module CONSTANT.TRU located in the TBLIBS directory. If you wish to use these variable names, you must include CONSTANT.TRU in a **LIBRARY** statement and you must **DECLARE** each variable name. The variable names may be used in upper, lower, or mixed case. (See that file for the numeric equivalents and definitions of other variables that can be used.)

The methods can be specified by numeric value, or (recommended) by the value of the publicly-declared variables OBJM_CREATE, etc., when using either TrueCtrl.trc (True Controls) or Constant.trc. In what follows, we do not include the equivalent numeric value; see Constant.tru for these.

The Create Method

When the **OBJECT** subroutine is used to create a new object or control, the attribute and string value parameters are not used. The type of object is passed in `values(1)`. Possible types are:

OBJT_GRAPHIC	Create a graphic object
OBJT_WINDOW	Create a window
OBJT_CONTROL	Create a control
OBJT_MENU	Create a menu
OBJT_GROUP	Create a (radio button) group

For graphics objects, the type of the object must be established using the OBJM_SET method. Use `"GRAPHIC TYPE"` as the attribute, and pass the type as the value of `values(1)`. The types of graphics objects are:

GRFT_CIRCLE	Circle or ellipse
GRFT_LINE	Straight line
GRFT_RECTANGLE	Rectangle
GRFT_ARC	Arc
GRFT_PIE	Pie segment
GRFT_ALINE	Line with arrows at either end
GRFT_ROUNDRECT	Rectangle with rounded corners
GRFT_POLYGON	Closed polygon
GRFT_POLYLINE	Open polygon (ends not joined)
GRFT_IMAGE	Graphical image (from a file)

Additional attributes are established using the OBJM_SET method.

For windows, many attributes are possible. They can be established using the OBJM_SET method.

For controls, the type of control must be established. This is done using the OBJM_SET method with the attribute `"CONTROL TYPE"`. The type of control is passed as the value of `values(1)`. Possible types are:

CTLT_PUSHBUTTON	Push button
CTLT_RADIOBUTTON	Radio button
CTLT_CHECKBOX	Check box
CTLT_HSCROLL	Horizontal scroll bar
CTLT_VSCROLL	Vertical scroll bar
CTLT_EDIT	Editable text field
CTLT_TEXT	Static text field
CTLT_LBOX	List box
CTLT_LISTBUTTON	List button
CTLT_LISTEDIT	List edit button
CTLT_GROUPBOX	Group box rectangle
CTLT_TXED	Text editor
CTLT_ICON	Icon

Menus are always associated with windows. Each menu and menu item must be created separately. `Values(1)` contains the value OBJT_MENU. The type of menu is passed as the value of `values(2)`. Possible types are:

MENT_BAR	A new menu bar item
MENT_ITEM	A new menu item

You must also pass, in `values(3)`, the ID of the parent of the menu or menu item. For menus, the parent ID is the ID of the window. For menu items, the parent ID is the ID of the menu in which the item falls. For hierarchical menus, the parent ID is the ID of the menu item to which the sub-menu is associated.

There are several attributes that may be set for menus and menu items. They include the text of the menu or menu item, the so-called mnemonic or hot key equivalent, whether the menu item is to be a separator, whether the menu item is enabled or not (i.e., dimmed if disabled), whether it is checked or not, and whether it is indeed checkable at all. These attributes are discussed in a later section.

Hierarchical menus may be established by passing, in `values(3)`, the ID number of a menu item (rather than a menu header) that is to serve as the start of the hierarchical menu.

Groups can be created only for groups of radio buttons. To create the group, you must have previously created all the radio buttons in the group. You then create the group using the OBJM_CREATE method with `values(1) = OBJT_GROUP` and with `values(2)` through `values(n+4)` containing, respectively: 0, 1, n (the number of buttons), ID of button 1, ..., ID of button n. You then can add items to the group using the SET method. (See the attributes for a group, later in this chapter.)

The CREATE method *assigns* a value to the second argument, `id`, as follows:

Value of id	**Element**
0-99	windows
101-9999	controls and graphic objects
10001-14999	menus
15001-19999	group

The Copy Method

The COPY method can be used to make a copy of an existing object or control, to assign to the copy a new ID. The new ID is returned in `values(1)`. Even if the original object or control is visible, the new copy will not be shown until the programmer uses the SHOW method with it.

For example, if `pbid` is the ID number of an existing push button, then

```
CALL      Object      (OBJM_COPY,      pbid,      "",      "",      values())
LET pbidnew = values(1)
```

will generate an identical push button but with ID number `pbidnew`.

All attributes that make sense will be copied. For example, the new object or control will be in the same location as the original. The programmer will almost certainly want to use the SET method with the RECTANGLE attribute to specify a new location for the object or control, which can be done before showing it.

The Set and Get Methods

The SET and GET methods specify attributes parameters, or obtain the current state or value of these parameters.

To set an attribute, use:

```
CALL Object (OBJM_SET, id, attributes$, values$, values())
```

OBJM_SET is an integer number (see the module CONSTANT in the file TRUECTRL.TRU for details.) `Id` is the ID number for the object or control. `Attrlist$` is a list of attribute names; if there are more than one, they are separated by vertical bars "|". `Value$` is a string variable that will pass string information in either direction. `Values()` is a numeric array that will pass numeric information in either direction.

To get the current value of an attribute, use:

```
CALL Object (OBJM_GET, id, attributes$, values$, values())
```

The GET method works exactly like the SET method. If an attribute has a string value, that value is passed in the string variable `values$`, for both the SET and the GET method. If the attribute has one or more numeric values, those values are passed in the numeric array `values()`, for both set and get.

For the SET method and for the GET method you can set the values of several attributes with one call to the **OBJECT** subroutine. Just include the names of all the attributes, separated by vertical bars "|", in a single string. Then provide the values in the same order to the `values$` string and the `values()` array. As an example, suppose you wished to establish a Push Button in a certain location, provide its text, and center the text. You might use:

```
MAT Redim v(5)
LET v(1) = left
LET v(2) = right
LET v(3) = bottom
LET v(4) = top
LET text$ = "My Button"
LET v(5) = 1                ! The code for center justification
CALL Object (OBJM_SET, pbid, "RECTANGLE|TEXT|TEXT JUSTIFY", text$, v())
```

The Show and Erase Methods

All objects and controls are invisible when they are created. The SHOW method can make them visible:

```
CALL Object (OBJM_SHOW, id, "", "", values())
```

where `id` is the ID number of the object or control that is to be made visible. `Values()` is ignored.

To ***hide*** an object or control, use the ERASE method.

```
CALL Object (OBJM_ERASE, id, "", "", values())
```

This simply makes the object or control invisible; it still exists, and may be manipulated in all ways behind the scenes. `Values()` is ignored.

You should be aware of several special conventions. If you erase a window (i.e., make it invisible), all the controls within that window also become invisible. Thus, you don't have to erase the controls individually. When you later show the window, all the controls in it will also become visible. In other words, the visibility of a window overrides the visibility of objects and controls in it.

Of course, during the time that a window is invisible (erased) you can erase any given object or control in it. Then when the window is made visible again, that particular object or control will remain invisible. With menus, it is necessary only to show or erase one single menu item to show or erase the entire menu structure.

It is not possible to erase a single menu or menu item. However, if part of the menu structure is changed, you must "show" one of the menus or items to display the entire revised menu structure.

The Free Method

If you no longer need a window, control, or graphical object, you can eliminate it entirely with the FREE method. For example:

```
CALL Object (OBJM_FREE, id, "", "", values())
```

will cause the object or control to disappear and will release the ID number for possible reuse. All internal storage associated with the object or control is also released.

Using the FREE method on a window will automatically free all graphics objects and controls within that window. Freeing any one particular menu item with the negative of its ID will free the entire menu structure.

If a menu item is freed, the menu items following it will be moved up. Freeing a menu header will free it and all of its items.

Note how this terminology differs from that used in regular True BASIC statements. In this context, ***show*** means to make visible, ***erase*** means to make invisible, and ***free*** renders the object or control non-existent. Recall that the **ERASE** statement used with files in True BASIC removes part or all of the contents of the file, and the **UNSAVE** statement deletes file and its contents from the operating system.

The Select Method

The SELECT method can be applied to windows and to certain controls within windows. When applied to windows, the value of `values(1)` specifies whether the window should become merely the front-most (***active***) window, or should be the one that responds to True BASIC **INPUT** and **PRINT** statements (***target***), or both.

```
LET v(1) = 1          ! Target (responds to input and output)
LET v(1) = 2          ! Make active (move to the front)
LET v(1) = 3          ! Both
CALL Object (OBJM_SELECT, wid, "", "", v())
```

Note that for physical windows, target means the same as current. Remember also that the choice of the target (or current) physical window is under the sole control of the program, while the user may make any visible window active merely by clicking in it.

When the SELECT method is applied to controls, `values$` and `values()` are ignored. This method is meaningful for push buttons, radio buttons, check boxes, edit fields, list edit buttons, and text edit controls. When applied to an edit field or a text edit control, that control becomes active and absorbs keystrokes. When applied to a push button, that button becomes selected and will be deselected when the Enter or Return key is pressed. When applied to a radio button or check box, the result will be as if the user clicked on the button or box. Generally, the effect of applying this method is the same as if the user had clicked on the window or control. In either case, a SELECT or CONTROL SELECT event will occur.

The SELECT method is ignored if applied to other types of objects or controls.

The Update Method

If the value of `method` is OBJM_UPDATE, the **OBJECT** subroutine invokes the UPDATE method to redraw the contents of the physical window whose ID is specified as `id`. The UPDATE method is applicable only to WINDOW objects; any attempt to invoke it for an object that is not a window will result in an error.

The purpose of the UPDATE method is to redraw a specified portion of the contents of a physical window, refreshing its earlier contents. The rectangular region of the window's contents that will be updated should be specified as `values(2)`, `values(3)`, `values(4)`, and `values(5)` representing the left, right, bottom, and top coordinates of the update region, respectively.

These coordinates may be specified in either pixel coordinates or user coordinates, as determined by the value of `values(1)`. If `values(1)` equals 0, they will be interpreted as pixel coordinates. If `values(1)` equals 1, they will be interpreted in the user coordinate system of the logical window that is currently in effect within the specified physical window.

In general, the UPDATE method will not be necessary for immune windows, since immune windows are updated automatically by True BASIC. However, the UPDATE method can be extremely useful when working with WINDOW objects that have not been defined as immune.

The Sysinfo Method

The SYSINFO method is used to obtain (get) certain True BASIC system information, and to obtain (get) or change (set) certain operating system parameters.

The attributes (value of the variable `attributes$`) that are possible with this method are given below. All numeric values are in pixels, and are returned in the `values()` numeric vector. String values are returned in the `values$` string variable. The attribute names can be in upper, lower, or mixed case, but exactly one space must separate the words.

Attributes with OB JM_SYSINFO Method

Attribute$	Value returned
DISPLAY SIZE	The size of the entire physical screen is returned in `values(1) .. values(4)`. `Values(1)` contains the leftmost pixel, which is 0. `Values(2)` contains the rightmost pixel. `Values(3)` contains the bottommost pixel. `Values(4)` contains the topmost pixel, which is 0. The number of pixels available for use will be `values(2)-values(1)+1` horizontally and `values(4)-values(3)+1` vertically.
MACHINE	The platform; one of MAC, WIN32, OS/2, or UNIX.
NATIVE WID	(Not implemented; for Unix only.)
PIPE IN	(Not implemented; for Unix only.)
PIPE OUT	(Not implemented; for Unix only.)
BLOCKING	(Not implemented; for Unix only.)
PIPE ID	(Not implemented; for Unix only.)
BOX KEEP ID	(Not implemented; for Unix only.)
NO MENUS	(Not implemented; for Unix only.)
STATIC TEXT HEIGHT	The height of a static text field is returned in `values(1)`
BUTTON HEIGHT	The height of a push button is returned in `values(1)`
EDIT TEXT HEIGHT	The height of an edit text field is returned in `values(1)`
CHECK BOX HEIGHT	The height (and width) of a check box is returned in `values(1)`
HORZ SBAR HEIGHT	The height of the horizontal scroll bar is returned in `values(1)`
RADIOBUTTON HEIGHT	The height of a radio button field is returned in `values(1)`
VERT SBAR WIDTH	The width of a vertical scroll bar is returned in `values(1)`
ENV	See below.
DEFAULT BACKGROUND COLOR	`values(1) .. values(3)` are the r-, g-, and b- values, respectively, of the background color
VERSION	The XVT version the current implementation is based upon

KEY CODES	See below.
MENU HEIGHT	The height of the menu bar is returned in `values(1)`.
APPLICATION NAME	`Value$` will contain the complete application name, including the pathname. This is ignored by True BASIC; it is only for the convenience of the programmer.
FONTS AVAILABLE	The names of the available fonts are returned in `value$`, separated by vertical bars.
LANGUAGE	"US English"or "Japanese" is returned in value$
GRAB CURSOR	See below.
TITLE BAR HEIGHT	The height of the title bar, in pixels, is returned in `values(1)`.
BORDER WIDTH	The width of a full border, in pixels, is returned in `values(1)`.
BORDER HEIGHT	The height of a full border, in pixels, is returned in `values(1)`.
DOUBLE BORDER WIDTH	The width of a double border, in pixels, is returned in `values(1)`.
DOUBLE BORDER HEIGHT	The height of a double border, in pixels, is returned in `values(1)`.
RESIZE BORDER WIDTH	The width of a full border with resize box is returned in `values(1)`.
RESIZE BORDER HEIGHT	The height of a full border with resize box is returned in `values(1)`.
PREFFOLDER	(Macintosh only) The system-defined preferences folder is returned in `value$`.
TMP	(Macintosh only) The system-defined temporary folder is returned in `value$`.

W You can both set and get the values of the following attribute, but on the Windows platform only:

KEY CODES

To set the key code conventions, set `values(1)=1` and `values(2)` to 0 (for cross-platform compatible codes; the default) or to 1 (for True BASIC for DOS compatible codes.) To get the key code conventions, set `values(1)=0`. Upon return, `values(2)=0` if cross-platform codes are in effect, and `values(2)=1` if DOS 4.0 compatible codes are in effect. Note that the cross-platform-compatible codes may not correspond to the codes on any particular platform. Some typical values are:

F1 .. F15	331 .. 345
up arrow	301
down arrow	302
right arrow	303
left arrow	304
page up	305
page down	306
home	309
end	310

These may not be available on all platforms. Also, some platforms may offer additional key codes. You may have to write a simple program using `GET KEY key` to find out.

The attribute, ENV, is used to obtain or change certain operating system parameters. For example, to get the PATH parameter, you might use:

```
LET parameter$ = "PATH"
LET values(1) = 0          ! Get
```

```
   CALL Object (OBJM_SYSINFO, 0, "ENV", parameter$, values())
   PRINT parameter$
```

You might see something like "C:\;C:\WINDOWS;C:\TBSILVER".

Note that the fourth argument must be a string variable as it is used as both an input and an output parameter. If there is an error, the null string will be returned.

To set the PATH parameter, you might use something like this:

```
   LET parameter$ = "PATH= .... "
   LET values(1) = 1           ! Set
   CALL Object (OBJM_SYSINFO, 0, "ENV", parameter$, values())
```

It may not be possible to set environment variables using this method on all systems.

M The Macintosh operating system does not use environment variables. However, you can obtain the path names of the system-defined preferences folder and the system-defined temporary folder as follows:

```
   LET v$ = "PREFOLDER"
   LET v(1) = 0
   CALL Object (OBJM_SYSINFO, 0, "ENV". v$, v)
```

Upon return, v$ = "PREFOLDER|<path-name of the preference folder" For the temporary folder, use:

```
   LET v$ = "TMP
   LET v(1) = 0
   CALL Object (OBJM_SYSINFO, 0, "ENV". v$, v)
```

Consult your operating system manual for more information about the environment variables.

The Print Method

If the value of `method` is OBJM_PRINT, the **OBJECT** subroutine invokes the PRINT method to redraw on a printer the contents of the physical window specified by `id`. The PRINT method applies only to WINDOW objects; any attempt to invoke it for an object which is not a window will result in error -11240.

The purpose of the PRINT method is to redraw the graphical contents of a physical window to a printer.

When invoked, the PRINT method will produce a standard printing dialog box allowing the user to configure the printing task. If the user chooses to print the window by accepting the dialog box, the method completes the task. If the user cancels the dialog box, the method is aborted and no output is sent to the printer.

Note that the PRINT method draws only the graphics and text contained within the specified physical window. Any controls in that window will not appear on the printed results.

The Page Setup Method

The PAGE SETUP method merely displays a typical page setup box, allowing the user to vary the parameters of the PRINT method. The page setup parameters also control the properties of printing to text files. Thus, the page setup associated with print to text files is given by the currently-targeted window.

[!] Note: There is a print record for each window and each may be different. It is highly recommended that you give the user some way of confirming the page setup before printing.

The Realize Method

On most modern platforms there is a system color mix palette in addition to the local color mix table that True BASIC maintains. When you specify a particular color mix in True BASIC and then attempt to draw in that color, the underlying system may "map" your color mix values to an entry in the system's color mix table that is "closest" but not exactly equal to your color. Or the underlying system may attempt to approximate the color you want by mixing several of its own colors in a process called *dithering*.

The realize method helps get around this problem. When invoked, this method adds the entries in the True BASIC color mix table that are now already in the system's color mix table to the system color mix table. Note: it will add only those that are not already there. Thus, when you attempt to draw in a certain color, there will be an exact match of that color in the system's color mix table.

For newly added colors to be properly displayed after invoking this method, it is necessary to Refresh the contents of the window. This method will automatically refresh.

The Scroll Method

The SCROLL method allows scrolling part or all of the contents of the physical window specified by `id`. The details are provided in the `values()` vector, as follows:

`values(1)`	0 for pixel coordinates, 1 for user coordinates
`values(2)`	left edge of the rectangle to scroll
`values(3)`	right edge of the rectangle to scroll
`values(4)`	bottom edge of the rectangle to scroll
`values(5)`	top edge of the rectangle to scroll
`values(6)`	amount to scroll horizontally; positive to the right, negative to the left
`values(7)`	amount to scroll vertically; positive up, negative down

The TXE Methods Calling Sequence

The next seven methods apply only to the text-edit control. The calling sequence is

```
CALL Object (method, txid, "", values$, values())
```

The TXE Suspend Method

If `method` is OBJM_TXE_SUSPEND, the text edit control is suspended; that is, changes to the underlying text are not reflected on the screen. `Value` and `values()` are ignored.

The TXE Resume Method

If `method` is OBJM_TXE_RESUME, the text edit control resumes; that is, changes are reflected on the screen as they occur. Changes that occurred during the time of suspension are also reflected. `Value` and `values()` are ignored.

The TXE Add Paragraph Method

If `method` is OBJM_TXE_ADD_PAR, the paragraph that appears in `values$` is inserted into the text of the text edit control *before* the paragraph whose paragraph number appears in `values(1)`.

The TXE Delete Paragraph Method

If `method` is OBJM_TXE_DEL_PAR, the paragraph whose paragraph number appears in `values(1)` is deleted.

As an example, suppose you wish to replace paragraph 17 with a new paragraph. You might use:

```
LET values(1) = 17
CALL Object (OBJM_TXE_DEL_PAR, txid, "", "", values())
CALL Object (OBJM_TXE_ADD_PAR, txid, "", newtext$, values())
```

The TXE Append Paragraph Method

If `method` is OBJM_TXE_APPEND_PAR, the paragraph in `values$` is inserted into the text of the text edit control at the end of the paragraph whose paragraph number appears in `values(1)`.

The TXE VScroll Method

If `method` is OBJM_TXE_VSCROLL, the text of a text edit control is scrolled vertically by the number of lines specified in `values(1)`. If `values(1)` is positive, the text is scrolled down; that is, a portion nearer the end will be displayed. If `values(1)` is negative, the text is scrolled up.

The TXE HScroll Method

If `method` is OBJM_TXE_HSCROLL, the text of a text edit control is scrolled horizontally by the number of pixels specified in `values(1)`. If `values(1)` is positive, the text is scrolled to the right; that is, a portion nearer the right margin will be displayed. If `values(1)` is negative, the text is scrolled to the left.

Attributes

Generally speaking, each type of object or control has one or more properties that can be set using the OBJM_SET method. In addition, the current state of the attributes can be obtained using the OBJM_GET method.

```
   CALL Object (OBJM_SET, id, attributes$, values$, values())
```

or

```
   CALL Object (OBJM_GET, id, attributes$, values$, values())
```

The sections below list the attributes for each type of object or control, together with any limitations. The large number of window attributes are organized here according to general purpose: structure, features, contents, scroll bars, and auxiliary. The name of the attribute itself is supplied as the third parameter in the call to the **OBJECT** subroutine. String-valued arguments, if any, are passed as the fourth parameter. Numeric-valued arguments, if any, are passed as values in the numeric array that is passed as the fifth parameter.

If the argument if string-valued, then the values that might be present in the numeric array are ignored, and vice versa. (There are a small number of attributes that require both a string-valued and numeric-valued arguments.)

Window Structure Attributes

RECTANGLE — The four coordinates of the rectangle are passed in `values()`.

`values(1)`	left edge
`values(2)`	right edge
`values(3)`	bottom edge
`values(4)`	top edge

These values are in pixels and in full screen coordinates. Recall that the left edge of the screen is 0 with values increasing to the right, and the top edge of the screen is 0 with values increasing downward.

These coordinates specify the available user “real estate” within the window, assuming that the window has a title bar and a menu bar. On Windows, if you fail to specify a menu, then you will actually have more space than these RECTANGLE values would suggest. Also on Windows, if you specify a window larger than the actual screen size, the window will be truncated and may be displayed somewhat offset vertically. On the Macintosh, you may specify anything you want for the usable region, and that is what you will get. For child windows, the RECTANGLE coordinates are relative to the client area of the parent window.

PARENT — The id is the number of the current window, which will become a child window. Values(1) should contain the id number of the parent window. This attribute must be set before the window is shown the first time.

TYPE — `Values(1)` contains the type of the window. Possible types are:

```
LET values(1) = WINT_DOC ! 1
LET values(1) = WINT_PLAIN      ! 2
LET values(1) = WINT_DOUBLE    ! 3
LET values(1) = WINT_NO_BORDER ! 7
```

Document windows are standard windows and may contain embellishments such as a title bar, close box, etc.

Plain windows have a single line border and may not contain embellishments.

Double windows have a double line border and may not contain embellishments.

Windows with no borders are just that, and they may not contain embellishments, but they may be child windows of another window.

Regular windows, or parent windows, cannot be of type 7. Child windows cannot be of type 1 or 3.

CLOSE BOX — If `values(1)=0` (default), then no close box will be included in the window. If `values(1)` = 1, then a close box will be included. (This attribute must be set before the window is shown for the first time.)

TITLE — The title bar will contain the string in `values$` as the title of the window.

RESIZE BOX — If `values(1)=0` (default,) then no resize box will be included in the window. If `values(1)=1`, then a resize box will be included. (This attribute must be set before the window is shown for the first time.)

Window Feature Attributes

IMMUNE — If `values(1)` = 1 (default,) then the window will be made an immune window. If `values(1)` = 0, the window will not be immune.

If a window is immune, and it is partially or completely covered by another window, you do not have to regenerate its contents when the window becomes completely visible again.

If a window is RESIZEd, then you may wish to regenerate its contents, whether the window is immune or not. The previous contents will be shown in their previous size: if the window is made smaller, they will be clipped; if the window is made larger, they will no longer fill it. If you wish to have your graphics expand or contract to the new size of the window, you will have to re-draw them.

If a window is merely moved to another location on the screen, nothing needs to be done, for either an immune or non-immune window.

VISIBLE — This attribute is used for two purposes with windows. If `values(1)` = 0 with the SET method before the window is shown for the first time, there will be a one-step delay in showing the window. That is, two invocations of the SHOW method will be needed to actually show the window. This feature may be needed for setting certain properties that can be set only after the window is "shown". Or, you may wish to generate all the contents of the window before showing it the first time. Then later the window will be actually made visible with a second invocation of the SHOW method.

If `values(1)=1` (default), with the SET method, then the window will be made visible when it is first shown.

Once the window has been shown for the first time, the SET method cannot be used for this attribute. Instead, use ERASE and SHOW to make it invisible or visible.

When this attribute is used with the GET method, `values(1)` = 1 if the window is showing, and = 0 if it is not.

ICONIZABLE

If `values(1)=1`, then this window is iconizable; that is, it can be reduced to a small icon on the screen. If `values(1)=0` (default), it cannot. This attribute must be set before the window is shown.

This attribute may have no effect on some systems.

FOCUS ORDER

The focus order of controls, such as edit fields, determines the order in which they are selected by, for example, pressing the Return key. Generally speaking, the focus order is the order in which the controls were created.

Used with the GET method, `values(1)` is the ID of a window. Upon return, `values(1)` is the number of controls, and `values(2)` through `values(n+1)` are the IDs of the individual controls. All controls are included, regardless of sensitivity, visibility, or activeness.

Used with the SET method, the focus order can be modified. `Values(1)` must be the ID of the item to be moved to a new position, and `values(2)` is that new position in the focus order list. (The list positions start with 0.) The IDs between the new position and the former position of the item being moved are moved up in the list. Thus, if the focus order list looks like this:

```
101   102   103   104   105   106   107
```

and you use

```
LET v(1) = 104
LET v(2) = 1
CALL Object (OBJM_SET, window, "FOCUS ORDER", "", v)
```

the resulting focus order list will now look like this:

```
101   104   102   103   105   106   107
```

If the ID is not in the focus order list in the first place, an error occurs.

MOUSE MOVE

This attribute controls whether the MOUSE MOVE event is returned by the subroutine **Sys_Event** for the window. If `values(1)` = 0 (default,) this event is not returned; if `values(1)` = 1, this event is generated each time the system detects that the mouse is in a different position than previously.

NO HIDE

This attribute controls the automatic hiding of a window when the close box is clicked and event handling is turned on. If values(1) = 1, the window will not be automaticall hidden; if values(1) = 0 (default) it will be hidden. In either case, Sys_Event will return the HIDE event.

Window Content Attributes (Pen, Brush, Color, Font)

WIDTH

`Values(1)` contains the width in pixels of the pen that is used for line drawings. The default pen width is one pixel. If the width is more than one pixel, the approximate center of the line lies on the path or line given.

PEN COLOR

`Values(1)` contains the pen's color number, which is an index into the color map table. The color map table can be changed using the True BASIC **SET COLOR MIX** statement. The default color is -1 (black.)

PEN STYLE

`Values(1)` contains the style for the pen. Possible values are:

```
LET values(1) = PENS_SOLID        ! Solid
LET values(1) = PENS_DOT          ! Dotted
LET values(1) = PENS_DASH         ! Dashed
```

These styles apply only if the WIDTH of the pen is one pixel. Otherwise, the pen will be solid. The default style is "SOLID".

PEN PATTERN `Values(1)` contains the pattern to use for the pen. Possible values are:

```
LET values(1) = PBP_SOLID         ! Solid
LET values(1) = PBP_HOLLOW        ! Hollow
LET values(1) = PBP_RUBBER        ! Rubber
```

Rubber is the pattern used when a rectangle is stretched with the mouse; it often appears as a dashed line in motion. The default pattern is “SOLID”.

BRUSH COLOR `Values(1)` contains the brush’s color number, which is an index into the color map table. The color map table can be changed using the True BASIC **SET COLOR MIX** statement. The default color is -1 (black.)

BRUSH PATTERN `Values(1)` contains the brush pattern. Possible values are:

```
LET values(1) = PBP_SOLID      ! Solid (default)
LET values(1) = PBP_HOLLOW     ! No visible pattern
LET values(1) = PBP_HORZ       ! Horizontal lines
LET values(1) = PBP_VERT       ! Vertical lines
LET values(1) = PBP_FDIAG      ! Forward diagonal lines
LET values(1) = PBP_BDIAG      ! Backward diagonal lines
LET values(1) = PBP_CROSS      ! Crossed lines, a la
                                 checkerboard
LET values(1) = PBP_DIAGCROSS ! Crossed lines, diagonal
```

BACKGROUND COLOR `Values(1)` contains the background color number, which is an index into the color map table. The color map table can be changed using the True BASIC **SET COLOR MIX** statement. The default background color is -2 (white.)

DRAWMODE `Values(1)` contains the logical drawing mode for the window. Possible modes are:

```
LET values(1) = DM_COPY        ! Ignore what is there (default)
LET values(1) = DM_OR    ! Bit-by-bit logical OR with what is there
LET values(1) = DM_XOR   ! Bit-by-bit logical XOR with what is there
LET values(1) = DM_CLEAR       ! Clear what is there to color 0
LET values(1) = DM_NOT_COPY    ! Logical negation of COPY
LET values(1) = DM_NOT_OR      ! Logical negation of OR
LET values(1) = DM_NOT_XOR     ! Logical negation of XOR
LET values(1) = DM_NOT_CLEAR   ! Logical negation of CLEAR
```

The drawing mode determines how the window’s pen and brush interact with the background, including what has already been drawn. As an example, suppose there are four bit planes (i.e., there are sixteen entries in the color map table),the background is color 6 (binary 0110), and the pen is color 10 (binary 1010). Then the above drawing modes would give, for each pixel covered by the pen:

DM_COPY	Color 10 (binary 1010)
DM_OR	Color 14 (binary 1110)
DM_XOR	Color 12 (binary 1100)
DM_CLEAR	Color 0 (binary 0000)
DM_NOT_COPY	Color 5 (binary 0101)
DM_NOT_OR	Color 1 (binary 0001)
DM_NOT_XOR	Color 3 (binary 0011)
DM_NOT_CLEAR	Color 15 (binary 1111)

SOLID MIX If `values(1)=1`, then colors will be mapped to the nearest solid color. If `values(1)=0` (default), then they will be not so mapped.

When you use the **SET COLOR MIX** statement, the system will normally attempt to create a color that exactly matches the mix you requested. This may result in a dithered color, where not every pixel is the same color. Using the SOLID MIX attribute will mean that the system, instead of creating the exact color, will find the nearest color that is solid, where every pixel is the same color.

This attribute is effective only for the Windows operating system environments.

PLOT TEXT OPAQUE Executing the SET method with `values(1)=1` will cause all PLOT TEXT statements in the specified window to be opaque (like PRINT statements). Executing it with `values(1)=0`, will restore it to its default non-opaque state. Executing the GET method will return 1 in `values(1)` if the attribute is set, and 0 if it is not set.

FONT NAME `Value$` contains the name of the font to be used, or currently in use. The available fonts will vary with the operating system. However, certain common fonts can be found on all systems; they are:

HELVETICA	FIXED	TIMES	SYSTEM

The FIXED font is a mono-spaced font; it may be Courier, Monaco, or similar font. The SYSTEM font is Helvetica on some systems. The default font is FIXED.

FONT STYLE `Values$` contains the style of the font to be used, or currently in use. The available styles are:

PLAIN	BOLD	ITALIC	BOLD ITALIC

On many systems, the name of the font and its style may be combined into the font name. For example, you could specify “Times Roman Italic” as the font name instead of having to specify “Times Roman” as the font name and “Italic” as the font style. The default style is PLAIN.

FONT SIZE `Values(1)` contains the size of the font, in points, to be used or currently in use. The default font size is ten points.

FONT METRICS Usable only with the GET method, `values(1)` through `values(6)` return the following properties on the font in use; the units are in pixels (*not* points):

`values(1)`	leading
`values(2)`	ascent
`values(3)`	descent
`values(4)`	horizontal size
`values(5)`	vertical size
`values(6)`	bearing

The ***leading*** is the space between lines in the font. The ***ascent*** is the distance in pixels of the top of the highest character above the base line. The ***descent*** is the distance in pixels of the lowest descender before the base line. The ***horizontal size*** is the size of the “M” for proportional fonts. The ***vertical size*** is the total vertical space in pixels needed to display a line of text, and is the sum of the *leading*, the *ascent*, and the *descent*.. The ***bearing*** is always 1.

These values are related to the font size specified by the FONT SIZE attribute, but the actual appearance depends on the system, the pixels-per-inch of the monitor, etc.

CURSOR `Values$` contains the name of the cursor to be used, or currently in use. Possible names are:

ARROW	IBEAM	CROSS
PLUS	WAIT	USER

Cursor type USER is not currently implemented; an attempt to set the cursor to “USER” is ignored.

Window Scroll Bar Attributes

VSCROLL	If `values(1)=1`, there is an attached vertical scroll bar; if 0, there is not.
HSCROLL	If `values(1)=1`, there is an attached horizontal scroll bar; if 0, there is not.
POSITION VERTICAL	`Values(1)` contains the position of the slider (thumb) in the vertical scroll bar attached to the window. For the OBJM_GET method, the current position is returned in `values(1)`. For the OBJM_SET method, the new position is provided in `values(1)`. The default value is 0.
START RANGE VERTICAL	`Values(1)` contains the start of the range of values that the vertical scroll bar represents. If the slider (thumb) is in its topmost position, the POSITION VERTICAL will be this value. The default value is 0. The value must be a non-negative integer.
END RANGE VERTICAL	`Values(1)` contains the end of the range of values that the vertical scroll bar represents. If the slider (thumb) is in its bottom-most position, the POSITION VERTICAL will be this value minus the PROPORTION VERTICAL. The default value is 100. The value must be a non-negative integer.
PROPORTION VERTICAL	`Values(1)` contains the proportional size of the slider for an attached vertical scroll bar. The proportion is defined in terms of the START RANGE VERTICAL and END RANGE VERTICAL attributes. Proportional sliders are not available on all systems. The default value is 1. The value must be a non-negative integer.
POSITION HORIZONTAL	`Values(1)` contains the position of the slider (thumb) in the horizontal scroll bar attached to the window. For the OBJM_GET method, the current position is returned in `values(1)`. For the OBJM_SET method, the new position is provided in `values(1)`. The default value is 0. The value must be a non-negative integer.
START RANGE HORIZONTAL	`Values(1)` contains the start of the range of values that the horizontal scroll bar represents. If the slider (thumb) is in its leftmost position, the POSITION HORIZONTAL will be this value. The default value is 0. The value must be a non-negative integer.
END RANGE HORIZONTAL	`Values(1)` contains the end of the range of values that the horizontal scroll bar represents. If the slider (thumb) is in its rightmost position, the POSITION HORIZONTAL will be this value minus the PROPORTION HORIZONTAL. The default value is 100. The value must be a non-negative integer.
PROPORTION HORIZONTAL	`Values(1)` contains the proportional size of the slider for an attached horizontal scroll bar. The proportion is defined in terms of the START RANGE HORIZONTAL and END RANGE HORIZONTAL attributes. Proportional sliders are not available on all systems. The default value is 1. The value must be a non-negative integer.

Window Auxiliary Attributes

NAME	The name to use is passed in `values$`. This attribute is used only to store the name of an object or control, and is for user convenience only. It is not used by True BASIC in any way.
SINGLE VERTICAL	`Values(1)` contains the value to be used as the vertical increment when the up or down arrow is clicked in the vertical scroll bar attached to the window. This attribute is not used by TRUE BASIC; it stores a value for programmer convenience only. The default value is 1.

PAGE VERTICAL	`Values(1)` contains the value to be used as the vertical increment when the user clicks in the grayed area of the vertical scroll bar attached to the window. This attribute is not used by TRUE BASIC; it stores a value for programmer convenience only. The default value is 10.
SINGLE HORIZONTAL	`Values(1)` contains the value to be used as the horizontal increment when the left or right arrow is clicked in the horizontal scroll bar attached to the window. This attribute is not used by TRUE BASIC; it stores a value for programmer convenience only. The default value is 1.
PAGE HORIZONTAL	`Values(1)` contains the value to be used as the horizontal increment when one clicks in the grayed area of the horizontal scroll bar attached to the window. This attribute is not used by TRUE BASIC; it stores a value for programmer convenience only. The default value is 10.
TEXTEDIT	`Values(1)` contains the ID of an attached text edit control, if there is one. If there isn't, `values(1) =`-1. This attribute is provided as a convenient storage spot for the programmer; True BASIC does not use this number.

Menu Attributes

The following menu attributes may be set (using the OBJM_SET method) or found (using the OBJM_GET method). The menu or menu item ID is passed as the second parameter in the call to the **OBJECT** routine. The attribute name is passed in the third parameter of the call to the **OBJECT** routine. It may be in lower or upper or mixed case.

Several of the attributes (TEXT, MKEY, SEPARATOR, CHECKABLE) will not take effect until the menu is later reshown. The attributes ENABLED and CHECKED will have their effects displayed immediately.

TEXT	The text is passed as a string in `values$`. `Values()` is ignored.
MKEY	The ASCII value of the hot-key or mnemonic is passed in `values(1)`. `Value$` is ignored. In some systems, such as Windows and OS/2, the ASCII code must be one of the letters in the text of the menu item. This attribute is not allowed for menu headers on the Macintosh.
SEPARATOR	If `values(1)=1`, then that menu item will be displayed as a separator. If `values(1)=0`, then the text of the menu item will be shown. `Values$` is ignored. This attribute is not allowed for a menu header.
ENABLED	If `values(1)=1` then that particular menu item will be enabled, i.e., allowed to be selected. If `values(1)=0`, that menu item will be disabled and not selectable. In addition, it will appear dimmed. If you disable a menu header, the entire menu below it will be disabled. `Value$` is ignored
CHECKABLE	If `values(1)=1` (default), then that particular menu item can be checked. If `values(1)=0`, it cannot be checked. (If the menu item was previously checked and the checkable attribute is set to 0, the check mark will be removed.) `Value$` is ignored. There are several restrictions. For example, you cannot check a separator.
CHECKED	If `values(1)=1`, then that particular menu item will be checked. If `values(1)=0` (default), it will not be checked; if there was a check mark, it will be removed. `Value$` is ignored.

Graphics Attributes — Overview

The graphics objects described here are, in a sense, another form of True BASIC output such as **BOX LINES** or printing. Thus, controls (such as push buttons) "float" above these objects on most systems, just as they float above ordinary printed output. If the output scrolls, which it will if there are many **PRINT** statements, these graphics objects will also scroll.

The graphics type, location, content, and name attributes apply to all types of graphics objects. The several attributes are grouped below according to general purpose.

Graphics Type Attributes

GRAPHIC TYPE `Values(1)` is the type of the graphic object. Possible types are:

GRFT_CIRCLE	Circle or ellipse
GRFT_LINE	Straight line
GRFT_RECTANGLE	Rectangle
GRFT_ARC	Arc
GRFT_PIE	Pie segment
GRFT_ALINE	Line with arrows at either end
GRFT_ROUNDRECT	Rectangle with rounded corners
GRFT_POLYGON	Closed polygon
GRFT_POLYLINE	Open polygon (ends not joined)
GRFT_IMAGE	Graphical image (from a file or box keep string)

This attribute should be set when the object is created, and before it is shown for the first time.

VISIBLE For the GET method, `values(1)` = 1 if the object is visible, and = 0 if the object is invisible. (Visibility may be overridden by the visibility of the containing window.)

Attempting to use the SET method with this attribute will cause an error.

Graphics Location Attributes

RECTANGLE The four coordinates of the rectangle are passed in `values()`.

`values(1)`	left edge
`values(2)`	right edge
`values(3)`	bottom edge
`values(4)`	top edge

These values may be either in pixels or in user coordinates, depending on the value of the UNITS attribute. In either case, the coordinates refer to the interior of the containing window and not to the full screen.

If units are in pixels, the left edge of the interior of the physical window is 0 with values increasing to the right, and the top edge of the interior of the physical window is 0 with values increasing downward.

If the units are user coordinates, then the current user coordinates will be used. Note that when a new physical window is created, the default user coordinates for the interior portion of the physical window are 0, 1, 0, 1. That is, the left edge is 0, the right edge is 1, the bottom edge is 0, and the top edge is 1.

UNITS If `values(1)=0` (default), the RECTANGLE attribute values are interpreted as pixels; if = 1, then they are interpreted in terms of user coordinates.

RELATIVE If `values(1)=1`, then the sides of the graphics object will be positioned relative to the sides of the window. For example, if the window is enlarged, the graphics object will also be enlarged in proportion.

If `values(1)` = 0 (default), then no such proportional sizing or positioning will be done.

LEFT RELATIVE	If `values(1)=1`, then the left side of the graphics object will be positioned relative to the left side of the window, but the size of the graphics object will not be changed, unless another side is also positioned to be relative. If `values(1)=0` (default), then the left side will not be so positioned when the size of the window is changed.
RIGHT RELATIVE	If `values(1)=1`, then the right side of the graphics object will be positioned relative to the right side of the window, but the size of the graphics object will not be changed, unless another side is also positioned to be relative. If `values(1)=0` (default), then the right side will not be so positioned when the size of the window is changed.
BOTTOM RELATIVE	If `values(1)=1`, then the bottom edge of the graphics object will be positioned relative to the bottom edge of the window, but the size of the graphics object will not be changed, unless another side is also positioned to be relative. If `values(1)=0` (default), then the bottom edge will not be so positioned when the size of the window is changed.
TOP RELATIVE	If `values(1)=1`, then the top edge of the graphics object will be positioned relative to the top edge of the window, but the size of the graphics object will not be changed, unless another side is also positioned to be relative. If `values(1)=0` (default), then the top edge will not be so positioned when the size of the window is changed.

Graphics Contents Attributes (Pen, Brush, Color)

WIDTH	`Values(1)` contains the width of the pen, in points. The default width is one pixel. See the description of the WIDTH attribute for windows for further details.
PEN COLOR	`Values(1)` is the color of the pen, referring to the color mix table. See the description of the PEN COLOR attribute for windows for further details. The default value is -1 (black.)
PEN STYLE	`Values$` is the name of the pen style. See the description of the PEN STYLE attribute for windows for further details. The default style is “SOLID”.
PEN PATTERN	`Values$` is the name of the pen pattern. See the description of the PEN PATTERN attribute for windows for further details. The default pattern is “SOLID”.
BRUSH COLOR	`Values$` is the name of the brush color. See the description of the BRUSH COLOR attribute for windows for further details. The default color is -2 (white).
BRUSH PATTERN	`Values$` is the name of the brush pattern. See the description of the BRUSH PATTERN attribute for windows for further details. The default brush pattern is “SOLID”.
BACKGROUND COLOR	`Values(1)` is the color number of the background color. See the description of the BACKGROUND COLOR attribute for windows for further details. The default background color is -2 (white.)
DRAWMODE	`Values(1)` contains the number of the drawing mode to be used. See the description of the DRAWMODE attribute for windows for further details.

Graphics Name Attributes

NAME	The name to use is passed in `values$`. This attribute is used only to store the name of an object or control, and is for user convenience only. It is not used by True BASIC.

Graphics Circle, Line, and Rectangle Attributes

There are no circle, line, or rectangle attributes beyond those common to all graphics objects. However, specifying the RECTANGLE attribute for a line is interpreted as follows:

Starting point of line: (`xleft`, `ybottom`)
Ending point of line: (`xright`, `ytop`)

It will often be the case that `xleft` will be greater than `xright`, and the same for the `y`-values. As an example, suppose you want to draw a line from (300,200) to (100,400). You would use:

```
MAT Redim v(4)
LET v(1) = 300
LET v(2) = 100
LET v(3) = 200
LET v(4) = 400
CALL Object (OBJM_SET, rid, "RECTANGLE", "", v())
```

Graphics Arrowed Line Attributes

See the section above on line attributes for an explanation on how setting the RECTANGLE attribute determines the starting and ending points of the arrowed line.

START ARROW	`Values(1)` = 1 if the starting end of the line should have (has) an arrow head; = 0 (default) if it should (does) not.
END ARROW	`Values(1)` = 1 if the ending end of the line should have (has) an arrow head; = 0 (default) if it should (does) not.

Graphics Round Rectangle Attributes

OVAL WIDTH	`Values(1)` is the width, in the units specified at creation, of the oval (ellipse) defining the roundedness.
OVAL HEIGHT	`Values(1)` is the height, in the units specified at creation, of the oval (ellipse) defining the roundedness.

Graphics Arc and Pie Attributes

Arc and pie segments are defined as follows:

The circumference of the arc or pie segment falls on the ellipse inscribed in the defining rectangle. The start of the arc is defined by the intersection of the ellipse circumference with a line drawn from the center of the rectangle to the point defined by the START X and START Y attributes. The end of the arc is similarly defined but using the STOP X and STOP Y attributes.

For arcs, only that portion of the ellipse circumference between the start line and the stop line is drawn. For pie segments, two lines are drawn from the ends of the arc to the center of the rectangle.

START X	`Values(1)` is the x-starting value.
START Y	`Values(1)` is the y-starting value.
STOP X	`Values(1)` is the x-stopping value.
STOP Y	`Values(1)` is the y-stopping value.

These four attributes may be combined into one invocation of the **OBJECT** subroutine as follows:

```
LET attribute$ = "START X|START Y|STOP X|STOP Y"
MAT Redim values(4)
LET values(1) = startx
LET values(2) = starty
LET values(3) = stopx
LET values(4) = stopy
CALL Object (OBJM_SET, gid, attribute$, "", values())
```

Graphics Polygon and Polyline Attributes

POINTS — `Values()` contains the points that define the polygon or polyline. `Values(1)` contains the number of points. `Values(2)` contains the x-coordinate of the first point. `Values(3)` contains the y-coordinate of the first point. And so on.

To draw an isosceles triangle with the polygon object, you would use:

```
MAT Redim v(7)
LET v(1) = 3
LET v(2) = 100
LET v(3) = 400
LET v(4) = 300
LET v(5) = 400
LET v(6) = 200
LET v(7) = 100
CALL Object (OBJM_SET, gid, "POINTS", "", v())
```

Graphics Image Attributes

FILENAME — When this attribute is set, the graphical image will be loaded from the named file into the graphics object, and the width and height information set. `Values()` are not used. The image may be a BMP file. On the Macintosh, it may also be a PICT file. On X (Unix), it may also be an xbm or pbm file. The graphical image will be forced to fit exactly in the specified RECTANGLE coordinates. If you require no distortion, find out the IMAGE WIDTH and IMAGE HEIGHT (see below) and adjust the rectangle before showing the image.

When used with OBJM_GET, the filename used will be returned in `values$`.

FILETYPE — This attribute specifies the type of the graphical image. It may be one of "JPEG", "MS BMP", "OS/2 BMP", "PICT" (Macintosh only), and possibly "PCX". The type must be specified after creating the graphical image but before specifying the filename containing the image. If a null string is supplied as the file type, the system will attempt to determine the type from the contents of the file.

IMAGE — This attribute used with the OBJM_SET method converts a BOX KEEP string in `value$` into an image and displays it on the screen, and in the "graphics layer." When used with the OBJM_GET method, it converts an image on the screen into the local BOX KEEP format, storing it in the string variable specified by `value$`.

IMAGE WIDTH — When used with the OBJM_GET method, the image width in pixels will be returned in `values(1)`. This attribute cannot be used with the OBJM_SET method.

If the RECTANGLE attribute does not have the same width, the image will be shrunk or expanded to fit into the rectangle.

IMAGE HEIGHT — When used with the OBJM_GET method, the image height in pixels will be returned in `values(1)`. This attribute cannot be used with the OBJM_SET method.

If the RECTANGLE attribute does not have the same height, the image will be shrunk or expanded to fit into the rectangle.

FORCE PALETTE — When used with the OBJM_SET method, the value of `values(1)` determines which of two palettes to use. If 0 (default), the current palette will be used. If 1, the image will force its palette to become the current palette.

General Control Attributes

The following attributes apply to all types of controls.

CONTROL TYPE

The type of a control is normally set just after the control is created using the CREATE method. And it must be set before the control is first shown. Its value, which is passed in the `values()` array, must be one of:

CTLT_PUSHBUTTON	Push button
CTLT_RADIOBUTTON	Radio button
CTLT_CHECKBOX	Check box
CTLT_HSCROLL	Horizontal scroll bar
CTLT_VSCROLL	Vertical scroll bar
CTLT_EDIT	Editable text field
CTLT_TEXT	Static text field
CTLT_LBOX	List box
CTLT_LISTBUTTON	List button
CTLT_LISTEDIT	List edit button
CTLT_GROUPBOX	Group box rectangle
CTLT_TXED	Text editor
CTLT_ICON	Icon

SENSITIVE

The value of this attribute is passed in the `values()` array; 0 means insensitive and 1 (default) means sensitive. (A control is sensitive if clicking on it causes an event that reflects the click.)

VISIBLE

Available only with the GET method, the value of this attribute is passed in the `values()` array. If `values(1)=1`, the control is visible (subject to the visibility of the containing window); otherwise, the control is not visible.

An attempt to set this attribute will cause an error.

General Control Location Attributes

RECTANGLE

This attribute requires, or returns, four values in the `values()` array. The values are left edge, right edge, bottom edge, and top edge. These values will be in pixels, with the (0,0) point being in the upper left corner of the physical window, unless the UNITS attribute has been set. In this case, and only for the OBJM_SET method, the rectangle positions are in user coordinates for current logical window. (User coordinates are specified by the True BASIC statement **SET WINDOW**. If none have yet been specified, the defaults are 0, 1, 0, 1, with (0,0) in the lower-left corner of the window.)

UNITS

If `values(1)=0` (default), the RECTANGLE attribute values are interpreted as pixels; if = 1, then they are interpreted in terms of user coordinates of the current logical window. This attribute applies only to the OBJM_SET method.

RELATIVE

If `values(1)=1`, then the sides of the control will be positioned relative to the sides of the window. For example, if the window is enlarged, the graphics object will also be enlarged in proportion.

If `values(1)=0` (default), then no such proportional sizing or positioning will be done.

LEFT RELATIVE

If `values(1)=1`, then the left side of the control will be positioned relative to the left side of the window, but the size of the graphics object will not be changed, unless another side is also positioned to be relative.

If `values(1)=0` (default), then the left side will not be so positioned when the size of the window is changed.

RIGHT RELATIVE	If `values(1)=1`, then the right side of the control will be positioned relative to the right side of the window, but the size of the graphics object will not be changed, unless another side is also positioned to be relative. If `values(1)=0` (default), then the right side will not be so positioned when the size of the window is changed.
BOTTOM RELATIVE	If `values(1)=1`, then the bottom edge of the control will be positioned relative to the bottom edge of the window, but the size of the graphics object will not be changed, unless another side is also positioned to be relative. If `values(1)=0` (default), then the bottom edge will not be so positioned when the size of the window is changed.
TOP RELATIVE	If `values(1)=1`, then the top edge of the control will be positioned relative to the top edge of the window, but the size of the graphics object will not be changed, unless another side is also positioned to be relative. If `values(1)=0` (default), then the top edge will not be so positioned when the size of the window is changed.

General Control Name Attribute

NAME	The value of this attribute is passed in `values$`. This attribute is for user convenience only. It is not used by True BASIC or the system in any way.

Push Button Attributes

TEXT	`Value$` is the text (to be used) for the push button. The text appears inside the button, and may be truncated if too long.
TEXT JUSTIFY	If `values(1)` = 0, left justification is in effect (to be used). If `values(1)` = 1 (default), center justification is used. If = 2, right justification is used. (This may have no effect on some systems.)
DEFAULT	If values(1) = 1, the button will appear with a special outline, as if it were “active”. If values(1) = 0, the button will appear normal. This attribute has an effect only on the appearance of the button. Any use is up to the programmer.

Radio Group Attributes

Groups can be created only for groups of radio buttons. To create the group, you must have previously created all the radio buttons in the group. You then create the group using the OBJM_CREATE method with `values(1)` = OBJT_GROUP and with `values(2)` through `values(n+4)` containing, respectively: 0, 1, n (the number of buttons), ID of button 1, ..., ID of button n. You then can add items to the group using the SET method.

[!] Note: The Radio Button *groups* are NOT controls; general attributes such as RECTANGLE, do not apply. radio buttons, however, are controls, and the general attributes apply.

STATE	With the SET method, if `values(1)=0` (default), then the radio button whose ID is given is turned off. If `values(1)=1`, then that button is turned on, and the others in the group are turned off. With the GET method, `values(1)=1` if the radio button is on, and 0 otherwise.
ON	Available only for the GET method, `values(1)` gives the ID of the radio button that is on, if any, or is 0 otherwise. `Values(2)` gives the ordinal number of the button that is on, starting the count at 0. For example, if the IDs of three radio buttons are 110, 111, and 112,

	and button 111 is on, then `values(1)=111` and `values(2)=1`. Note: the `id` (second parameter) must be the ID of the radio group as a whole.
TEXT	`Value$` is the text (to be used) for the radio button. The text appears to one side of the button.
TEXT JUSTIFY	The justification that is in effect (get) or to be used (set) is: If `values(1)` = 0 (default,) left justification is in effect (to be used). If `values(1)` = 1, center justification is used. If = 2, right justification is used. The justification is relative to the available space specified by the RECTANGLE attribute for the individual radio buttons.
""	When used with the SET method, will add new radio buttons to the group. `Values(1)` = 1 (add), `values(2)` = number of new buttons, `values(3)...` values (n+2) = id numbers of the buttons to be added.
"" or GROUP	When used with the GET method, will return the id numbers of the buttons in the group. `Values(1)` = 0, `values(2)... values (n+1)` = the id numbers.

Check Box Attributes

STATE	With the SET method, if `values(1)=0` (default), then the check box whose ID is given is unchecked. If `values(1)=1`, then that check box is checked. With the GET method, returns 1 in `values(1)` if the check box is checked, and 0 otherwise.
TEXT	`Value$` is the text (to be used) for the check box. The text appears to one side of the check box.
TEXT JUSTIFY	The justification that is in effect (get) or to be used (set) is: If `values(1)` = 0 (default), left justification is in effect (to be used). If `values(1)` = 1, center justification is used. If `val-ues(1)` = 2, right justification is used. The justification is relative to the available space specified by the RECTANGLE attribute for the check box.

Scroll Bar Property Attributes

These attributes are for scroll bars that are *not* attached to windows.

POSITION	For the GET method, `values(1)` contains the current position of the slider (thumb) in the scroll bar. For the SET method, `values(1)` contains the new desired position of the slider in the scroll bar. The value must be a non-negative integer. The position is determined relative to the current values of the START RANGE, END RANGE, and PROPORTION attributes. In particular, when the slider is at the top of its range, the POSITION value will be that of the START RANGE value. When the slide is at the bottom of its range, the POSITION value will be that of the END RANGE value minus the PROPORTION value. The default value is 0. The value must be a non-negative integer.
START RANGE	`Values(1)` is the value to be used for the start of the slider range. The start is the top for vertical scroll bars, and the left for horizontal scroll bars. The default value is 0. The value must be a non-negative integer.
END RANGE	`Values(1)` is the value to be used for the end of the slider range. But note that the slider can move no farther than the END RANGE value reduced by the PROPORTION value. The default value is 100. The value must be a non-negative integer.
PROPORTION	`Values(1)` is the proportional size of the slider computed with reference to the START RANGE and END RANGE values. For example, if the PROPORTION value is the same as the difference between the END RANGE and the START RANGE, the slider will fill the entire slide bar area. Proportional sliders are not available on all systems. The default value is 1. The value must be a non-negative integer.

Scroll Bar Auxiliary Attributes

These attributes are for scroll bars that are *not* attached to windows.

TEXTEDIT	`Values(1)` contains the ID of the text edit control associated with the scroll bar. This is provided as a convenient storage spot for the programmer; True BASIC does not use this number.
SINGLE INCREMENT	`Values(1)` is the increment (default is 1) to be used for a click in an up or down arrow in the scroll bar. This is provided as a convenient storage spot for the user; True BASIC does not use this value.
PAGE INCREMENT	`Values(1)` is the increment (default is 10) to be used for a click in the trough in the scroll bar. This is provided as a convenient storage spot for the user; True BASIC does not use this value.

Edit Field Attributes

TEXT	`Value$` contains the text to be used in the edit field. For the GET method, it contains the text currently in the field.
EXIT CHAR	`Values(1)` is the ASCII code for the character to be registered as an exit character, in addition to the Escape (27) and Horizontal Tab (9) characters. For example, the return-key character (13) is often used to advance to the next edit field in the form.
	Supplying the negative of the ASCII code with the SET method will unregister that character as the exit character.
	Note: when an edit field is deselected by means of an exit character, the particular character code will be returned as the `x1` parameter, along with the CONTROL DESELECTED event, in the call to the **Sys_Event** subroutine.
FORMAT	`Values$` contains the format of the text to be used in the edit field. True BASIC does not use this value; it is provided for the convenience of the programmer. When a particular edit field has been deselected, the programmer may wish to check the text against the desired format, but this does not happen automatically.

Static Text Field Attributes

TEXT	`Values$` contains the text to be used in the static text field. If the field is tall enough, the text will be wrapped. If the text is still too long to fit into the field, the text is truncated on the right. If you wish to have the text break at a certain point, simply insert `chr$(13)` into the text string at the desired break point.
TEXT JUSTIFY	The justification that is in effect (get) or to be used (set) is: If `values(1)=0` (default), left justification is in effect (to be used). If `values(1)=1`, center justification is used. If `values(1)=2`, right justification is used. The justification is relative to the available space specified by the RECTANGLE attribute for the static text field.

List Box Attributes

SELECTION MODE	`Values(1)` contains the selection mode to be used. 0 (default) means to allow single selections only; 1 means to permit multiple selections; 2 means to allow the list to be viewed but not selected, i.e., read-only. The selection mode must be set before the list box is shown the first time. (Multiple selection mode may not be available on all systems.)
LIST	`Values$` contains the items to appear in the selection list; the items are separated by vertical bars "`|`".

SELECTION	When used with the GET method, `values(1)` contains the number of items selected, and `values(2) .. values(values(1)+1)` contain the ordinal numbers of the selected items, starting the counting with 0. When used with the SET method, `values(1)` contains 1 (or any non-zero) to highlight a list item and 0 to unhighlight a list item; `values(2)` contains the item to be highlighted or unhighlighted. Items are numbered `0...n-1`, where `n` is the number of items.

List Button Attributes

LIST	`Values$` contains the items to appear in the list button; the items are separated by vertical bars “|”. Note that the first item in the list also appears as the button name.
SELECTION	When used with the GET method, `Values(1)` contains the number of items selected, and v(2) contains the ordinal number of the selected item, starting the counting with 0. (Since the multiple selection mode is not available for list buttons, the selection list will never contain more than one element.) When used with the SET method, `v(1)` contains the ordinal number of the item initially shown, starting the counting with `0`.

List Edit Button Attributes

TEXT	When used with the SET method, `values$` contains the text to appear in the edit field of the list edit button. When used with the GET method, `values$` contains the current (latest,) possibly edited, value in the edit field. The programmer can interrogate this value at any time.
LIST	`Values$` contains the items to appear in the scrollable list beneath the editable button; the items are separated by vertical bars “|”. Note that selecting an item from the list moves that item into the editable field, where it can be further edited.
FORMAT	`Values$` contains the format to be applied to the edit field. This is ignored by True BASIC and is provided only as a convenience to the programmer.

Group Box Attributes

TITLE	`Values$` gives the title of the group box. Note that the group box is a rectangle that visually groups whatever it contains, such as a radio group. It does not actually combine several controls into a single group.

Text Editor Attributes — Overview

Text editor attributes include the following: TEXT, SELECTION, INSERTION, BORDER, ACTIVE, WRAP, MARGIN, NUM LINES, NUM PARS, ORIGIN, TRAP CHAR, CHAR LIMIT, KEY EVENTS, MOUSE EVENTS, FORE COLOR, BACK COLOR, BORDER COLOR, FONT NAME, FONT STYLE, FONT SIZE, FONT METRICS, READONLY, NUM CHARS, LINES IN PAR, LINE MAX WIDTH, VSCROLL, and HSCROLL.

The text editor attributes are grouped below according to general purpose: properties, contents, manipulation, and auxiliary.

Text Editor Property Attributes

BORDER	`Values(1)=1` if there is (to be) a border around the text edit control, and `=0` (default) if there is not (to be).
ACTIVE	`Values(1)=1` if the text edit control is (to be) active; that is, can process keystrokes and events. `Values(1)=0` if the text edit control is inactive. If a control is sensitive, it can be either active or inactive; if it is not sensitive, it cannot be active.

WRAP	If `values(1)=1`, the text is to be wrapped at the margin, which is in pixels. The wrapping point of the text need not take place at the right edge of the text edit control. If `values(1)=0` (default), the text will not be wrapped, and the value of the margin attribute will be ignored.
MARGIN	`Values(1)=` the size of the margin, in pixels, to be used with the text edit control. If there is no WRAPing, the margin is ignored. The default value is 0 for both cases.
READONLY	`Values(1)=1` if the text edit control is (to be) read only; otherwise, `values(1)=0` (default).
CHAR LIMIT	`Values(1)` contains the maximum number of characters allowed in one paragraph. The programmer can use this attribute to limit the amount of text a user can enter. The default value is 65535 characters.

Text Editor Contents Attributes

FORE COLOR — `Values(1)` gives the color to be used for the foreground (text) in the text edit control. The default foreground color is black.

BACK COLOR — `Values(1)` gives the color to be used for the background in the text edit control. The default background color is white.

BORDER COLOR — `Values(1)` gives the color to be used for the border of the text edit control. The default border color is black.

FONT NAME — `Values$` contains the name of the font to be used, or currently in use. The available fonts will vary with the operating system. However, certain common fonts can be found on all systems; they are:

HELVETICA   FIXED   TIMES   SYSTEM

The FIXED font is a mono-spaced font similar to the Courier or Monaco type faces. The SYSTEM font is Helvetica on some systems. The default font is HELVETICA.

FONT STYLE — `Values$` contains the style of the font to be used, or currently in use. The available styles are:

PLAIN   BOLD   ITALIC   BOLD ITALIC

On many systems, the name of the font and its style may be combined into the font name. For example, one might specify “Times Roman Italic” as the font name instead of having to specify “Times Roman” as the font name and “Italic” as the font style. The default style is PLAIN.

FONT SIZE — `Values(1)` contains the size of the font, in points, to be used or currently in use. The default font size is ten (points).

FONT METRICS — Usable only with the GET method, values(1) through values(6) return the following properties on the font in use; the units are in pixels (*not* points):

`values(1)`	leading
`values(2)`	ascent
`values(3)`	descent
`values(4)`	horizontal size
`values(5)`	vertical size
`values(6)`	bearing

The ***leading*** is the space between lines in the font. The ***ascent*** is the distance in pixels of the top of the highest character above the base line. The ***descent*** is the distance in pixels of the lowest descender before the base line. The ***horizontal size*** is the size of the “M” for proportional fonts. The ***vertical size*** is the total vertical space in pixels needed to display a line of text, and is the sum of the *leading*, the *ascent*, and the *descent*.. The ***bearing*** is always 1.

These values are related to the font size specified by the FONT SIZE attribute, but the actual appearance depends on the system, the pixels-per-inch of the monitor, etc.

Text Editor Manipulation Attributes

TEXT	The entire text is passed via the `values$` parameter. It is assumed to consist of lines of text delimited by the platform-specific end-of-line (EOL) sequence. The lines of the text are treated as paragraphs in the text edit control. (If the text edit control wraps the lines, there may be several “lines” visible per paragraph. In other words, lines in the text edit control do not necessarily correspond to lines in the text file. In fact, what are called ***lines*** in the text file are called ***paragraphs*** in the text edit control. See also the WRAP attribute.)
SELECTION	The parameters of selected text (selected text usually is shown in reversed color) are passed as six elements in the `values()` array. They are, in order, the paragraph, line, and character of the starting point of the selection, and then the paragraph, line, and character of the ending point of the selection. If nothing has been selected and the GET method is used, the starting values will equal the ending values.
INSERTION	The parameters of the text insertion cursor are passed as three elements in the `values()` array. They are, in order, the paragraph, line, and character position of the cursor. The cursor is in front of the character specified; remember that the numbering of paragraphs, lines, and characters all start with 0. Thus, character position 0 for the cursor means in front of the first character on the line. And a character position equal to the number of characters in the line means in back of the last character on the line.
NUM LINES	`Values(1)` gives the total number of lines in the text of the text edit control. This is available only for the GET method.
NUM PARS	`Values(1)` gives the total number of paragraphs in the text of the text edit control. This is available only for the GET method. Note that the number of lines is the same as the number of paragraphs for unWRAPped text.
NUM CHARS	`Values(1)` gives the number of characters in the text of the text edit control. This is available only for the GET method.
LINES IN PAR	This is available only for the GET method. `Values(1)` contains the paragraph number. Upon return, `values(2)` contains the number of lines in the paragraph. Note: paragraph numbering starts with 0.
LINE	This is available only for the GET method. `Values(1)` contains the number of the paragraph, and `values(2)` contains the number of the line within the paragraph. Upon return, `values$` contains the text of the line. Note that paragraph and line numbering both begin with 0.
MAX WIDTH	This is available only for the GET method. `Values(1)` is the length in pixels of the longest line if WRAP is set to off, or the margin itself if WRAP is set to on. Note: if the text edit control is not visible, this value will be some large number.
ORIGIN	This attribute returns the position within the text of the upper left corner of the text edit control. There are four values:

`Values(1)`	The starting paragraph number
`Values(2)`	The starting line number within that paragraph
`Values(3)`	The absolute line number
`Values(4)`	The number of pixels the text has been shifted to the left

TRAP CHAR	`Values(1)` contains the number of key code/stop code pairs. `Values(2)` through `values(2n+1)` contain the key code/stop code pairs. The key codes are the ASCII codes for the key; for example, the RETURN key has the key code 13. The stop codes are defined as follows:

Stop Code	**Result**
1	The key code is returned as a KEYPRESS event. The text edit control *is* suspended. The key is *not* absorbed by the text edit control.
2	The key code is returned as a KEYPRESS event. The text edit control is *not* suspended. The key *is* absorbed by the text edit control.
3	Exactly like stop code 1, but will be treated as an ordinary character *unless* there is selected text.
< 0	The particular key code is *unregistered*.

All other stop codes are ignored.

As examples, if you wish to use the escape character as a way of exiting from the text edit control, give it a stop code of 1. If you wish to readjust the scroll bars whenever the user presses the return key, give it a stop code of 2. If you wish to indent selected text when the user enters a “>”, give it a stop code of 3.

KEY EVENTS	If `values(1)=0` (default), all key events, except those specified as TRAP CHARs, will be absorbed by the text edit control, and will not be returned by the **Sys_Event** subroutine. If `values(1)=1`, then all key events will be returned by the **Sys_Event** subroutine, as well as being acted on by the text edit control.
MOUSE EVENTS	If `values(1)=0` (default), all mouse events within the text edit control will be absorbed by the text edit control, and not returned by the **Sys_Event** subroutine. If `values(1)=1`, mouse events will be acted on by the text edit control and also returned by the **Sys_Event** subroutine.

Text Editor Auxiliary Attributes

VSCROLL	`Values(1)` = the ID number of an attached vertical scroll bar, if any. True BASIC does not use this value; it is provided only for the convenience of the programmer.
HSCROLL	`Values(1)` = the ID number of an attached horizontal scroll bar, if any. True BASIC does not use this value; it is provided only for the convenience of the programmer.

Exceptions

These exceptions apply in general.

Unknown or invalid object ID. (-11220)
Cannot reference a freed object ID. (-11221)
Attribute not used for specified object. (-11223)
Unknown or invalid group method. (-11224)
Unknown or invalid attribute in SET/GET. (-11225)
Unknown or invalid font name. (-11226)
Unknown in invalid font style. (-11227)
Font size must be greater than zero. (-11228)
Can’t set FONT METRICS. (-11249)
Object ID out of range. (-11236)
Unknown window method. (-11237)

Unknown object method. (-11238)
Unable to SHOW window. (-11239)
Unknown or invalid object type specification in CREATE. (-11240)
Too many EXIT CHARS for Edit Field. (-11241)
Can’t set ACTIVE until object is visible. (-11242)
Color must be >= 0. (-11251)
Unknown or invalid menu item type specification. (-11254)
Can’t check a menu separator. (-11255)
Menu separators are not checkable. (-11256)
Unknown or invalid control object type. (-11257)
Unknown or invalid graphic object type. (-11258)
Unknown or invalid window object type. (-11259)
Unknown or invalid group object type. (-11260)
Can’t check a menubar item. (-11261)
Can’t make menubar item a separator. (-11262)
Menu parent incorrect for menu type. (-11263)
Can’t SELECT an unSHOWn window. (-11264)
Unknown or invalid brush pattern. (-11265)
Unknown or invalid pen pattern. (-11266)
RECTANGLE minimum = maximum. (-11271)
No Help File opened. (-11272)
Not enough values for attribute list in SET/GET. (-11273)

These exceptions apply only to the Text Edit control.

TextEdit method passed to non-TextEdit object. (-11229)
Can’t SUSPEND TextEdit object when not visible. (-11230)
Can’t RESUME TextEdit object when not visible. (-11231)
Error adding paragraph. (-11232)
Paragraph number is too large. (-11233)
Error deleting paragraph. (-11234)
Error appending paragraph. (-11235)
Can’t set NUM LINES. (-11243)
Can’t set NUM PARS. (-11244)
Can’t set NUM CHARS. (-11245)
Can’t set LINES IN PAR. (-11246)
Can’t set MAX WIDTH. (-11248)
Too many trap chars for TextEdit. (-11250)
Paragraph out of range for GET LINE. (-11252)
Line out of range for GET LINE. (-11253)

CHAPTER

20

Sys_Event Subroutine

As the user of the program manipulates the mouse, clicking on various controls, selecting windows, etc., these activities are reported back to the program as events. In True BASIC, these events do not generate interrupts, but rather are placed on a single queue (list). Calling the built-in subroutine **SYS_EVENT** allows the program to examine the event, if any, at the front of the list.

(We strongly recommend that you use True Controls. Its subroutine, TC_Event, calls Sys_Event and then performs several operations on scroll bars, check boxes, radio buttons, etc. Call Sys_Event directly only for special purposes not covered by True Controls.)

This model, examining one at a time the events from the event queue by calling the subroutine **SYS_EVENT**, provides the simplest possible way to respond to the many things that can happen within the user interface. These events occur asynchronously and can be reported in an order different from that in which the programmer intended.

The standard way to build a program might be illustrated as follows:

```
DO
  CALL Sys_Event ...
  SELECT CASE event$
  CASE "event 1"
    CALL Response1
  CASE "event 2"
    CALL Response2
  ...
  CASE "event n"
    CALL Responsen
  CASE else
    ! Must be an error
  END SELECT
LOOP
```

Of course, the event names from **SYS_EVENT** are more suggestive, and additional information is also provided. The rest of this chapter lists the different events that can occur for each of the windows, menus, and controls.

[!] Note: Graphics objects in themselves cannot be the cause of any events

The Sys_Event Subroutine

The calling sequence for the **SYS_EVENT** subroutine is

```
CALL Sys_Event (timer, event$, window, x1, x2)
```

The value of `timer` specifies the amount of time to wait for an event if the event queue is empty. The subroutine will return immediately if `timer` = 0. In either case, if there is no event on the event queue, then `event$` will be the null string.

The name of the event itself is returned in `event$` and is capitalized. Event names are listed later according to the type of object or control that generated them.

The ID number of the physical window in which the event occurred is returned in the third argument `window`. This is the physical window ID number, and has no relation to any True BASIC logical windows that might be in use.

The last two parameters, `x1`, and `x2`, provide additional numeric information in most cases. If either, or both, is not used, it is set to 0.

The rest of this chapter lists the events, by type of object or control that can generate them.

Events for Windows

KEYPRESS	The user has pressed a key. The window ID will be that of the active window, the one designated to receive input. `X1` is the ASCII number of the character generated. `X2` contains the shift key codes; if `x2` = 1, the shift key was held down; if 2, the control key was held down; if 0, neither key was held down; if 3, both keys were held down.

The following thirteen events are mouse events. They all have the same definition for `x1` and `x2`.

x1	x-coordinate mouse position
x2	y-coordinate mouse position

SINGLE	A single click of the left button, or of the only button, on the mouse has occurred. Note that a mouse click that selects a window will return a SELECT event. But a second click in the window, once the window has been selected, will return this event or one like it.
DOUBLE	A double mouse click has occurred with the left or only mouse button. Note that this event will always be preceded by a SINGLE event. That is, if a second click has occurred within a small time increment, this second click will generate a DOUBLE event.
EXTEND	A mouse click with the shift key held down has occurred. This is used most commonly to select multiple items from a list.
SINGLE RIGHT DOUBLE RIGHT EXTEND RIGHT	These events are just like the three above, but apply to the right mouse button.
SINGLE MIDDLE DOUBLE MIDDLE EXTEND MIDDLE	These events are like those above, but apply to the middle mouse button.
MOUSE UP MOUSE UP RIGHT MOUSE UP MIDDLE	The mouse button has been released. The first event applies to the left button, the next two to the right and middle buttons, if any.
MOUSE MOVE	The mouse has moved since the last time Sys_Event was called. This event will not be returned unless the MOUSE MOVE attribute has been turned on.
SIZE	The `window` (see the third argument) has been resized. Note that just moving the window without resizing it will generate no event. `x1` and `x2` are not used. When a window has been resized, it may be necessary to regenerate its contents.

REFRESH	The window, which was formerly partially or completely hidden or covered by other windows, is now fully visible. If the window is not IMMUNE, then its contents will have to be regenerated. `X1` and `x2` are not used.
	If the window is IMMUNE, then its contents will not have to be regenerated, but the REFRESH event will still occur.
SELECT	The window has been selected (made active) or deselected. x1 = 1 if the window was selected, and = 0 if the window was deselected. (This says nothing about the window being a target for input and output.) The window selected will also have focus.
HIDE	The window has been hidden or closed. This can occur by clicking in the close box. `x1` and `x2` are not used.
	The following twelve events can occur if there are scroll bars attached to the window. `x1` and `x2` are not used.
UP	The up arrow in an attached vertical scroll bar has been clicked.
DOWN	The down arrow in an attached vertical scroll bar has been clicked.
LEFT	The left arrow in an attached horizontal scroll bar has been clicked.
RIGHT	The right arrow in an attached horizontal scroll bar has been clicked.
PAGEUP	The trough above the slider in an attached vertical scroll bar has been clicked.
PAGEDOWN	The trough below the slider in an attached vertical scroll bar has been clicked.
PAGELEFT	The trough to the left of the slider in an attached horizontal scroll bar has been clicked.
PAGERIGHT	The trough to the right of the slider in an attached horizontal scroll bar has been clicked.
VSCROLL	The slider in an attached vertical scroll bar is being moved. (By how much and in what direction can be determined using the OBJM_GET method.)
HSCROLL	The slider in an attached horizontal scroll bar is being moved. (By how much and in what direction can be determined using the OBJM_GET method.)
END VSCROLL	The end of a slider movement in an attached vertical scroll bar has occurred. This will happen when the user releases the mouse button.
END HSCROLL	The end of a slider movement in an attached horizontal scroll bar has occurred. This will happen when the user releases the mouse button.
MENU	The user has selected a menu item. `X2` contains the ID number of the menu item selected.

Events for Graphics Objects

Graphics objects cannot generate events.

Events for Controls

Many controls can generate the same event. For completeness, all such events are listed for the particular type of control.

Events for Push Buttons

CONTROL SELECT	The user has clicked the mouse on a push button. `X2` contains the ID number of the push button.
	Normally, this event will not be returned by the **SYS_EVENT** subroutine until the matching CONTROL DESELECTED event has occurred.

CONTROL DESELECTED	The user has released the mouse button whilst still pointing to a push button. `X2` contains the ID number of the push button.

Events for Radio Buttons

CONTROL SELECT	The user has clicked the mouse on a radio button. `X2` contains the ID number of the radio button.
	Normally, this event will not be returned by the **SYS_EVENT** subroutine until the matching CONTROL DESELECTED event has occurred.
CONTROL DESELECTED	The user has released the mouse button whilst still pointing to a radio button. `X2` contains the ID number of the radio button.
	Which radio button is on can also be determined by using the OBJM_GET method.

Events for Check Boxes

CONTROL SELECT	The user has clicked the mouse on a check box. `X2` contains the ID number of the check box.
	Normally, this event will not be returned by the **SYS_EVENT** subroutine until the matching CONTROL DESELECTED event has occurred.
CONTROL DESELECTED	The user has released the mouse button whilst still pointing to a check box. `X2` contains the ID number of the check box.
	The current state of the check box can be determined by using the OBJM_GET method.

Events for Scroll Bars

The events for scroll bars that are not attached to windows follow. In each case, `x2` contains the ID number of the scroll bar control.

UP	The user has clicked in the up arrow of a vertical scroll bar.
DOWN	The user has clicked in the down arrow of a vertical scroll bar.
LEFT	The user has clicked in the left arrow of a horizontal scroll bar.
RIGHT	The user has clicked in the right arrow of a horizontal scroll bar.
PAGEUP	The user has clicked in the trough above the slider in a vertical scroll bar.
PAGEDOWN	The user has clicked in the trough below the slider in a vertical scroll bar.
PAGELEFT	The user has clicked in the trough to the left of the slider in a horizontal scroll bar.
PAGERIGHT	The user has clicked in the trough to the right of the slider in a horizontal scroll bar.
VSCROLL	The user is in the process of moving the vertical slider in a scroll bar.
HSCROLL	The user is in the process of moving the horizontal slider in a scroll bar.
END VSCROLL	The user has released the mouse button after having moved the vertical slider.
END HSCROLL	The user has released the mouse button after having moved the horizontal slider.

Events for Edit Fields

CONTROL SELECT	The user has clicked the mouse on an edit field. `X2` contains the ID number of the edit field.
CONTROL DESELECTED	The user has finished editing one edit field, and has moved the mouse to a new edit field, causing the first edit field to be deselected. Certain key strokes can also be used to advance to the next edit field, causing the former edit field to become deselected. At this point it is possible to interrogate the text the user has edited and to match the text against a desired format. `x2` contains the ID number of the edit field deselected.

If the deselection was caused by the user pressing a key specified as an EXIT CHAR, then the character number is returned in `x1`.

Events for Static Text

Static Text cannot generate events.

Events for Selection List Boxes

CONTROL SINGLE	The user has clicked on an item in the selection list. `X2` contains the ID number of the selection list box control. Which item has been selected can be determined using the OBJM_GET method.
CONTROL DOUBLE	The user has double-clicked on an item in the selection list. `X2` contains the ID number of the selection list box control. Which item has been selected can be determined using the OBJM_GET method. This event will always be preceded by a CONTROL SINGLE event. (Note: the CONTROL DOUBLE event often is a signal that the user has completed the examination of the selection list box; thus it is the equivalent of selecting an “Ok” button, were there one to select.)

Events for List Buttons

CONTROL SINGLE	The user has clicked on an item in the list button. `X2` contains the ID number of the list button control. Which item has been selected can be determined using the OBJM_GET method.

Events for List Edit Buttons

CONTROL SELECT	The user has clicked the mouse on a list edit button. `x2` contains the ID number of the list edit button.
CONTROL DESELECTED	The user has finished editing the list edit button, and has moved the mouse elsewhere. Certain key strokes can also be used to generate this event. At this point it is possible to interrogate the text the user has edited. `X2` contains the ID number of the edit field deselected.
KEYPRESS	The user has pressed a key that has been defined to cause a focus shift. `X1` contains the ASCII value of the key.

Events for Group Boxes

Groups Boxes cannot generate events.

Events for Text Edit Controls

TXE MOUSE	The user has clicked the mouse in a text edit control when MOUSE EVENTS had been selected as an option.
TXE VSCROLL	The user is in the process of scrolling the text vertically by holding the mouse button down while moving the mouse to the extreme top or bottom of the text edit control.
TXE HSCROLL	The user is in the process of scrolling the text horizontally by holding the mouse button down while moving the mouse to the extreme left or right of the text edit control.
TXE KEYPRESS	The user has pressed a key when KEY EVENTS has been selected as an option, or the user has pressed one of the TRAP characters. If the Shift Key is also pressed, 256 will be added to the key code. If the Control Key is also pressed, 512 will be added to the key code. If both are pressed, 768 (256 + 512) will be added to the key code.

Events for Groups (currently, only Radio Groups)

CONTROL SELECT	The user has clicked the mouse on a radio button. `X2` contains the ID number of the radio button. Normally, this event will not be returned by the **SYS_EVENT** subroutine until the matching CONTROL DESELECTED event has occurred.
CONTROL DESELECTED	The user has finished clicking on a radio button, and has moved the mouse elsewhere. `X2` contains the ID number of the button deselected. (At this point it is possible to interrogate the radio group control to learn which button is on.)

Summary of Events

All the possible events are listed below, followed by the objects or controls that can generate them.

The following nine events can be generated by mouse clicks within windows, but not clicks on most controls (which generate events like CONTROL SELECT.) Window contains the window ID number. `X1` contains the x-coordinate of the mouse position in window coordinates, and `x2` contains the y-coordinate of the mouse position.

SINGLE
DOUBLE
EXTEND
SINGLE RIGHT
DOUBLE RIGHT
EXTEND RIGHT
SINGLE MIDDLE
DOUBLE MIDDLE
EXTEND MIDDLE
MOUSE UP
MOUSE UP RIGHT
MOUSE UP MIDDLE
MOUSE MOVE

The following four events are all window-related. The window ID is returned in `window`. `x1` and `x2` are ignored.

SIZE
REFRESH
SELECT
HIDE
MENU

The following twelve events are generated by scroll bars. If the scroll bars are attached to a window, then `x1` and `x2` are ignored. If the scroll bars are not attached to a window, then `x2` contains the ID number of the scroll bar.

UP
DOWN
LEFT
RIGHT
PAGEUP
PAGEDOWN
PAGELEFT
PAGERIGHT
VSCROLL
HSCROLL
END VSCROLL
END HSCROLL

The following two events can be generated only from selection list boxes, but only the first for list buttons. In each case `window` gives the window ID, and `x2` gives the control ID.

CONTROL SINGLE
CONTROL DOUBLE

The following two events can be generated from push buttons, radio buttons, check boxes, edit fields, list edit buttons, and text edit controls. In each case `window` gives the window ID, and `x2` gives the control ID.

CONTROL SELECT
CONTROL DESELECTED

The following event can be generated from a window. `Window` gives the window ID, `x1` gives the ASCII values of the key stroke, and `x2` gives the shift key code; if `x2 = 1`, the shift key was held down; 2, the control key was held down; if 0, neither key was held down; if 3, both keys were held down.

KEYPRESS

The following four events can be generated only from a text edit control. For TXE KEYPRESS, `x1` is the key and `x2` is the control ID. For the other three, `x1` is 0 and `x2` is the ID.

TXE KEYPRESS
TXE MOUSE
TXE VSCROLL
TXE HSCROLL

CHAPTER

21

TBD or TBDX Subroutines

The **TBD** subroutine is a built-in subroutine that displays several types of modal dialog boxes. A ***modal dialog box*** is one in which control is retained in the dialog box until the user exits it and the box is closed. That is, no other activities can occur until the dialog box closes.

The calling sequence is:

```
   CALL TBD(x, y, type, title$, msg$, btn$, name$, text$, st, dflt, timeout, result)
```

The **TBD** subroutine is capable of producing four different types of dialog boxes – ***standard*** dialog boxes, ***open file*** dialog boxes, ***save file*** dialog boxes, and ***list dialog*** boxes. The value of `type` determines the type of dialog box that will be produced, and must be a value between 1 and 4, inclusive.

Since each of these four types of dialog box is significantly different from the others, each will be discussed separately in the sections below.

For dialog boxes of type 1 and type 4, the values of `x` and `y` determine the upper left corner of the dialog box in pixels in the coordinate system of the full screen. If `x` < 0, then True BASIC will choose a convenient value for the left edge that roughly centers the dialog box across the screen. If `y` < 0, then True BASIC will do the same for the top edge to center the dialog box vertically on the screen. This feature is not available for dialog boxes of type 2 or type 3.

If you want the dialog box to have a title bar, specify the actual title you wish it to have as the value of `title$`. If the value of `title$` is the null string, the dialog box will not have a title bar. Note: the title bar feature is not available for Standard Dialog Boxes (type 1) on the Macintosh.

The **TBD** subroutine also allows you to determine the `timeout` for dialog boxes of type 1 and type 4 (open file and save file dialog boxes may not be timed out). A dialog box's timeout is the length of time in seconds that the user will be given to respond to that dialog box. If the timeout period expires before the user selects one of the dialog boxes push buttons, control will return to the line immediately following the **CALL** statement; the value of `result` will be set to the number of the default button. If there is no default button, the **TBD** subroutine returns to the line following the **CALL** statement and sets the value of `result` to 0. If the value of `timeout` is 0, then the resulting dialog box will never time out; that is, it will remain on the screen until the user selects one of its push buttons.

As mentioned above, location and timeout refer to type 1 and type 4 dialog boxes only. The remaining sections discuss each type of dialog box separately.

Standard Dialog Boxes

If `type` equals 1, **TBD** produces a ***standard dialog box***. These include message and warning boxes, and single-line and multi-line input boxes. A standard dialog box may contain from one to four push buttons and from zero to ten editable text fields.

The value of `msg$` specifies a message that will appear above the edit fields (if any) and push buttons. By default, the value of `msg$` will appear on a single line (truncated to fit within the dialog box). However, you may insert line breaks into the value of `msg$` by including vertical bar characters (`|`). The **TBD** subroutine will treat each occurrence of a vertical bar as a line break. The dialog box will be resized to make room for as many lines as specified.

It is your responsibility to ensure that the resulting dialog box fits in the visible region of the screen. Note: the title bar feature is not available for Standard Dialog Boxes on the Macintosh.

A standard dialog box may contain from one to four push buttons, which will be evenly spaced along the bottom edge of the dialog box. The value of `btn$` specifies the text to appear in up to buttons. Use the vertical bar (|) to separate values for individual buttons. The buttons are of a fixed size, and the text is centered within the button. It is your responsibility to ensure that the text you have specified fits within the buttons. If the value of `btn$` is the null string, an error will be generated.

A standard dialog box may also contain one or more edit fields. The value of `name$` determines the number of edit fields included in the dialog box. `Name$` may contain up to ten items separated by the vertical bar (|). Each item, in order, will be used as the label for an edit field, and there will be as many edit fields as there are items in the value of `name$`. Each label appears to the left of the edit field with which it is associated. If a label is too long to fit in the space allotted, it will be truncated.

The value of `text$` may be used to specify the default text (if any) for each edit field. Like `name$`, `text$` may contain up to ten items separated by the vertical bar (|). Each item, in order, will be used as the default text for an edit field. If there are fewer items in `text$` than there are in `name$`, then the remaining edit fields specified by `name$` will be empty by default. If there are more items in `text$` than there are in `name$`, then the dialog box will contain an edit field for each item in the value of `text$` but the extra fields will not have labels. In other words, the number of fields displayed will be the maximum of the number of fields in `name$` and the number of fields in `text$.`

If both `name$` and `text$` are equal to the null string, the TBD subroutine produces a dialog box with no edit fields.

If a standard dialog box contains edit fields, the value of `st` determines which edit field will be active when the dialog box is displayed. The active edit field is the one that will receive anything typed by the user. If the value of `st` is 0 (or greater than the number of edit fields), none of the edit fields will be active when the dialog box is displayed.

The value of `dflt` specifies which button, if any, should be treated as the default button. The buttons are numbered sequentially beginning with 1 in the order in which they appear within the value of `btn$`. The default button will be identified in a system-dependent manner, and once the dialog box has been displayed, the user can select the default button simply by pressing the Enter (or Return) key. If the value of `dflt` is 0, the resulting dialog box will not contain a default button; the user will be required to click on a specific button with the mouse.

Once displayed, a standard dialog box remains on the screen until the user selects one of its push buttons or until its timeout period expires, whichever occurs first.

Upon returning to the **CALL** statement, **TBD** returns the number of the selected push button in `result`. If the standard dialog box contained edit fields, the value of `text$` will contain the contents of the edit fields at the time the push button was selected. The contents of each edit field will be separated by vertical bars (|) within the value of `text$`, and the number of the edit field which was active when the push button was selected will be returned as the value of `st`.

Open File Dialog Boxes

If `type` equals 2, the **TBD** subroutine produces an open file dialog box that is standard for the current operating system. The exact nature and operation of this dialog box will vary between operating systems, but in general it presents a list of files in the current directory and allows the user to choose a filename. It also allows the user to locate files in other directories.

In an open file dialog box, the value of `msg$` specifies the file name extension that can be used, on some platforms, to limit the files displayed in the dialog box's file list. On Windows and similar platforms, simply provide the extension without the period. You can use upper or lower or mixed case. Thus, supplying "TRU" or "tru" will limit the file names displayed to those having the extension .tru.

On the Macintosh, the `msg$` field is used to specify the file type, such as TEXT or TEXTTRUE. There is no way to limit the file names displayed based on the extension.

The value of `dflt` determines whether the current directory may be changed by the user's actions. If the value of `dflt` is 0, then the user may specify a file in any directory, but the current directory will remain unchanged. In this case, the **TBD** subroutine returns the full pathname of the selected file. If the value of `dflt` is 1, then if the user specifies a file in a directory other than the current directory, the directory containing that file is made the new current directory. In this case, the **TBD** subroutine returns only the file name; a pathname is not necessary since the file will be in the current directory.

For an open file dialog box, the **TBD** subroutine ignores the values of `btn$`, `name$`, `text$`, and `st`.

Once displayed, an open file dialog box remains on the screen until the user selects a file name and pushes a button.

Upon returning to the **CALL** statement, the **TBD** subroutine returns the number of the push button selected as the value of `result`. If the user selected the "Open" push button, `result` will be returned equal to 1, and the specified file name, possibly including an appropriate path, is returned as the new value of `btn$`. If the user selected the "Cancel" push button, result will be returned equal to 0 and `btn$` will be the null string.

Note that the open file dialog box does not actually open the file; it merely returns the file name selected by the user.

Save File Dialog Boxes

If `type` equals 3, the **TBD** subroutine produces a save file dialog box that is standard for the current operating system. The exact nature and operation of this dialog box will vary between operating systems, but in general it allows the user to specify a name and location for the file being saved.

In a save file dialog box, the value of `msg$` specifies the file name extension that can be used, on some platforms, to limit the files displayed in the dialog box's file list. On Windows and similar platforms, simply provide the extension without the period. You can use upper or lower or mixed case. Thus, supplying "TRU" or "tru" will limit the file names displayed to those having the extension .tru.

In a save file dialog box, the value of `btn$` specifies the default file name. This file name will be provided as a suggestion to the user. Of course, they user may edit this value to any legal filename before pushing a button.

For a save file dialog box, the **TBD** subroutine allows the user to specify a different directory in which the file should be saved, but it does not change the current directory. Thus, when the **TBD** subroutine is used to create a save file dialog box, it always returns a complete pathname for the specified file.

When used to create a save file dialog box, the **TBD** subroutine ignores the values of `name$`, `text$`, `st`, and `dflt`.

Once displayed, a save file dialog box remains on the screen until the user selects a file name and pushes one of the buttons.

Upon returning to the **CALL** statement, the **TBD** subroutine returns the number of the push button selected as the value of `result`. If the user selected the "Save" push button, `result` will be returned equal to 1, and the specified file name, including the appropriate path, is returned as the new value of `btn$`. If the user selected the "Cancel" push button, `result` will be equal to 0 and `btn$` will be the null string.

[!] Note: The save file dialog box does not actually save the file; it merely returns the file name provided by the user.

List Dialog Boxes

If `type` equals 4, the **TBD** subroutine displays a list dialog box. The list may contain any number of items. If necessary, the list box will include a vertical scroll bar to allow the user to view all of the available choices. The user may select any one of the items appearing in the list box.

The value of `msg$` specifies a message that will appear above the list box and push buttons. By default, the value of `msg$` will appear on a single line (truncated to fit within the dialog box). However, you may insert line breaks into the value of `msg$` by including vertical bar characters (`|`). Each occurrence of a vertical bar will be treated as a line break by the TBD subroutine. The dialog box will be resized to make room for as many lines as specified. It is your responsibility to ensure that the resulting dialog box fits in the visible region of the screen.

A list dialog box may contain from one to four push buttons which will be evenly spaced along the right-hand edge of the dialog box. The value of `btn$` specifies the text to appear in up to buttons. Use the vertical bar (`|`) to separate values for individual buttons. The buttons are of a fixed size, and the value is printed centered within the button. It is your responsibility to ensure that the text you have specified fits properly within the buttons. If the value of `btn$` is the null string, an error will be generated.

This list box displays one or more items, any one of which may be selected by the user. The number and contents of the items in this list is determined by the value of `name$`. The value of `name$` specifies the text to appear in list. Use the vertical bar (`|`) to separate values for individual list items. The list box is of a fixed size, and the list items are printed left-justified within the list box, one item per line. If a list item is longer than can be displayed in the allotted width of the list box, only the left-most portion will be seen. If there are more items that can be shown, a vertical scroll bar will be provided.

For a list dialog box, the value of `st` determines which list item will be selected (highlighted) when the dialog box is displayed. The active list item is the one which is highlighted. If the value of `st` is less than 0, the first list item will be highlighted when the dialog box is displayed. If the value of `st` is greater than the number of list items in the list box, none of the list items will be highlighted initially.

For a list dialog box, the value of `dflt` specifies which button, if any, should be treated as the default button. The buttons are numbered sequentially beginning with 1 in the order in which they appear within the value of `btn$`. The default button will be identified by a heavy black border, and once the dialog box has been displayed, the user can select the default button simply by pressing the Enter (or Return) key. If the value of `dflt` is 0, the resulting dialog box will not contain a default button; the user will be required to click on a specific button with the mouse.

Once displayed, a list dialog box remains on the screen until the user pushes one of its buttons or until its timeout period expires, whichever occurs first.

Upon returning from the **CALL** statement, `result` contains the number of the push button selected. The **TBD** subroutine returns the number of the selected list item in `st`.

Exceptions:	-11300	Dialog box has no buttons specified.
	-11301	Unknown or invalid dialog box specification.

The TBDX Subroutine

```
CALL TBDX (l, r, b, t, parm1$, parm2(), type, title$, msg$, btn$,
   name$, text$, st, dflt, timeout, result)
```

The TBDX subroutine is like the TBD subroutine except that the first two parameters of TBD (x, y) have been replaced by six parameters (l, r, b, t, parm1$, parm2().) All subsequent parameters remain the same in sequence and meaning. It allows placing a dialog box of almost any size in any location on the screen.

L, r, b, and t specify the location of the dialog box in pixels, measuring from the top left corner of the screen. If all four are >= 0, it is the programmer's responsibility to ensure that the dialog box is large enough to contain any elements in it. If all four are = -1, True BASIC will automatically size the dialog box and center it in the currently-targeted physical window. (If True BASIC does not have a currently-targeted window, it will be centered on the screen.) If r = -1 and b = -1, then True BASIC will automatically size the dialog box and place the top left corner at the point (l, t) in pixel screen coordinates.

Parm1$ and parm2() allow extended functionality and user control. These parameters are ignored for dialog boxes of type 2 or 3 (open file and save file.)

Parm1$ is a string of attributes or values with the usual "|" as a delimiter. Parm2() is an array of numeric values. If there are not enough values in parm2() to satisfy the attributes specified in parm1$, an exception occurs. The following parameters are defined:

If parm1$ includes "MAXLENGTH", True BASIC will take the next value from parm2() and use it as the maximum length, in pixels, of each line of message information in the dialog box. (Note: there may be many lines.) Message lines which exceed this will be truncated. (For the TBD subroutine, True BASIC decides the maximum length.) If the value for MAXLENGTH is 0, then True BASIC will decide on a maximum length and automatically wrap the message text. A "|" in the message text will still be treated as a "hard return." If MAXLENGTH is not present, then the message length will be treated as in the TBD subroutine.

If parm1$ includes "MULTIPLE" and the dialog box is of type 4 (a selection list box,) then the list box will allow multiple selections. The number of selections will be returned in parm2(), followed by the index of each selected item. If you attempt to use this attribute with a type 1 dialog box, an exception will occur.

If parm1$ includes "SELECTIONS", if "MULTIPLE" has already been defined, and if the dialog box is of type 4 (a selection list box,) True BASIC will look in parm2() for the number of selections to pre-select and the index of each selection to pre-select. If you attempt to use this attribute with a type 1 dialog box, an exception will occur.

If parm1$ includes "MESSAGE JUSTIFY", and the dialog box is of type 1, True BASIC will "justify" the message text according to the corresponding value in parm2(). A value of 0 means left justify, 1 means center, and 2 means right justify. If this attribute is used for a type 4 dialog box, an exception will occur.

If parm1$ includes "BUTTON JUSTIFY", and the dialog box is of type 1, True BASIC will justify the text in the buttons according to the corresponding value in parm2(). A value of 0 means left justify, 1 means center, and 2 means right justify. If this attribute is used for a type 4 dialog box, an exception will occur.

If parm1$ contains an undefined attribute, an exception will occur.

Exceptions:	-11273	Not enough values for attribute list in SET/GET.
	-11223	Attribute not used for specified object.

CHAPTER

22

Interface Library Routines

This chapter contains technical descriptions of the routines discussed in Chapters 12 “Files for Data and Output” and 14 “Interface Elements.” The routines are contained in three library files: TRUECTRL.TRC, TRUEDIAL.TRC, and EXECLIB.TRC. The routines in the TRUECTRL.TRC library all have names that begin with “TC_”. The routines in the TRUEDIAL.TRC library all have names that begin with “TD_”. The routines in the EXECLIB.TRC library all have names that begin with “EXEC_”. In addition, this chapter contains the descriptions of the routines provided for communications support. These routines are contained in the library file COMMLIB.TRC. The routines in the COMMLIB.TRC library all have names that begin with “Com_”. (The libraries COMLIB.TRU and COMLIB.TRC contains the same routines but with the traditional subroutine names.)

TRUECTRL.TRC contains routines for creating and manipulating objects and controls, including windows, menus, graphical objects, buttons and such, and text editors. These routines, known collectively as True Controls, provide easy access to the built-in subroutines **Object** and **Sys_Event**.

TRUEDIAL.TRC contains routines for creating and using modal dialog boxes. (Modal dialog boxes require a response before the program can continue.) Variations include dialog boxes for: issuing warnings, receiving input, file opening and file saving, and displaying a selection list. These routines, known collectively as True Dials, provide simple access to the built-in subroutine **TBD**.

EXECLIB.TRC contains routines for manipulating directories. The routines provide simple access to the built-in subroutine **System**.

COMMLIB.TRC contains routines supporting communication through the serial ports. The routines provide simple access to the built-in subroutines **ComOpen** and **ComLib**.

The source code for all four libraries is included. This lets you examine the detailed use of the built-in subroutines **Object**, **Sys_Event**, **TBD**, **System, ComOpen,** and **ComLib**, and it provides a starting point if you wish to write your own subroutines.

The rest of this chapter is organized by library. With each library, the routines are grouped according to function, and not alphabetically. (See the Index of True BASIC Constructs for an alphabetical list of all True BASIC statements, functions, subroutines, whether built-in or in a library.)

True Controls

The True Controls subroutines can be grouped into several categories including general routines that apply to all objects and controls as well as other routines that apply only to specific objects or controls.

Almost all subroutines that create objects or controls have the same calling sequence:

```
   CALL TC_XXX_Create (id, type$, xl, xr, yb, yt)
```

If the creation is successful, the newly assigned ID number is returned in `id`. The argument `type$` varies according to the object or control – it is either a string or a string array.

The four coordinates `xl` (x-left), `xr` (x-right), `yb` (y-bottom), and `yt` (y-top) specify (in almost all cases) the rectangular region in which the object or control is defined. Either pixel coordinates or window coordinates can be used. For pixel coordinates, the useful (client) area of the window will be placed with respect to the full screen. Graphics objects and controls will be placed with respect to the containing physical window.

User coordinates for placing windows can be used only in conjunction with the True Controls routines TC_Win_Create and TC_Win_ChildCreate. Here, the user coordinates behave like SCREEN coordinates in True BASIC; that is, 0 refers to the left and bottom edges, and 1 to the right and top edges, of the full screen. In addition, True Controls adjusts the client area so that all the embellishments (title bars, borders, etc.) are visible. User coordinates for placing graphical objects or controls refer to the window coordinates for the *current* logical window.

Pixel coordinates always have the (0,0) location at the top-left corner. The x value increases to the right. The y value increases *down* the window or screen. (This is the reverse of the usual True BASIC coordinate system in which the y value increases as you move *up*.)

The **SET WINDOW** statement establishes the user coordinates as follows:

```
   SET WINDOW xleft, xright, ybottom, ytop
```

(Note that you may establish a coordinate system in which increasing y-values move down.) In the absence of a **SET WINDOW** statement, the default coordinates are (0, 1, 0, 1).

Location of windows, graphical objects, other controls is in user coordinates, by default. If you prefer to use pixel coordinates, you can call the subroutine **TC_SetUnitsToPixels**.

Numerous errors can arise if windows, graphical objects, and other controls are improperly specified or located. These errors are of two types. Errors identified by the system subroutine **Object** are outlined at the end of Chapter 19 “Object Subroutine” and are not repeated here; in general they deal with such conditions as specifying an ID number that does not exist, or an invalid type or option. Other errors are detected and generated within the True Controls subroutines; these are included in this chapter.

General Purpose Subroutines

TC_Init

This routine “initializes” the event handler, allowing **TC_Event** to operate properly. There are no arguments to this subroutine.

```
   CALL TC_Init
```

Exceptions:	800	Can’t call TC_Init during module startup.

TC_Cleanup

This routine must be called after completing all tasks that require **TC_Event** and before the program terminates. Failure to do so may leave the operating system in an inconsistent state. There are no arguments to this subroutine.

```
   CALL TC_Cleanup
```

Exceptions:	800	Can’t call TC_Cleanup during module startup.

TC_Event

All activity generated through the use of windows and the various controls is reported to the program in the form of events. This subroutine returns the next event from the event queue.

```
   CALL TC_Event (timer, event$, window, x1, x2)
```

`Timer` specifies the amount of time, in seconds, to wait for an event to occur if there is none on the event queue. If there is an event, or if `timer` = 0, the return from this subroutine will be immediate, except in the case of certain scrolling events arising from a text edit control.

`Event$` contains the name of the event, which can be the null string. A complete list of events is given in Chapter 20 “Sys_Event Subroutine.” This current chapter describes the events appropriate for each type of object or control along with the convenience routines for that object or control.

`Window` is a numeric variable that contains the ID number of the window in which the event occurred. The event can be a window event, or it can be an event associated with a control located in that window.

`X1` and `x2` provide additional information. In general, `x2` will be the ID number of the control causing the event. In the case of a KEYPRESS event, `x1` will be the ASCII code of the key.

In addition to merely returning the event produced by **Sys_Event, TC_Event** carries out certain routine tasks associated with the controls that have been created and the menus. The events and actions are as follows. Note that graphical objects, static text controls, and group box controls do not generate events.

MENU	**TC_Event** returns the menu number in x1 and the item number in x2. In addition, if you have created a text edit control and notified True Controls of the menu equivalents for Cut, Copy, and Paste, **TC_Event** carries out these operations on the text edit control.
DOWN, UP, LEFT, RIGHT, PAGEDOWN, PAGEUP, PAGELEFT, PAGERIGHT, VSCROLL, HSCROLL, END VSCROLL, END HSCROLL	**TC_Event** carries out the appropriate scroll bar operation. If the scroll bar is associated with a text edit control, **TC_Event** moves the text accordingly.
CONTROL DESELECTED	If an edit field has been deselected by means of a trap character (i.e., carriage return,) **TC_Event** advances the focus to the next available edit field. If a check box is being deselected, **TC_Event** changes the state of the check box. If a radio button is being deselected, **TC_Event** change the state of the radio button. In addition, TC_Event returns the id number of the radio button group, not the id number of the button itself.
SIZE	If a text edit control has been “attached” to the window, **TC_Event** also resizes the text edit control so that it continues to fill the window, and adjusts the scroll bar parameters accordingly.
TXE KEYPRESS	If the character is an EOL, **TC_Event** updates the vertical scroll bar, if any.
TXE HSCROLL TXE VSCROLL	**TC_Event** initiates bookkeeping to keep track of text scrolling so it can properly update the scroll bars, if any, at the end of the text scrolling. While the text is being scrolled, i.e., while TXE HSCROLL or TXE VSCROLL keeps occurring, TC_Event keeps looping, so that these events are never actually returned by **TC_Event**.

TC_Set

This general purpose subroutine lets you specify or “set” the value(s) of any attribute(s) defined for the object or control whose ID is given.

```
CALL TC_Set (id, attributes$, value$, values())
```

The second argument must be a string expression that contains the attribute names, separated by vertical bars (|). The third and fourth arguments contain the string values and the numeric values required by the attributes, in the same order as the attribute names themselves. Multiple string values are separated by vertical bars. Multiple numeric values appear in consecutive entries in the list `values()`. The lowest subscript of the list must be 1, and the list must be dimensioned large enough to contain all the numeric attribute values expected.

This calling sequence is similar to that described in Chapter 19 “Object Subroutine” for use with the OBJM_SET method. If `id` does not refer to an existing object or control, or if an attribute name is invalid for that type of object or control, or if it is not possible to set the value of an attribute, an error occurs. See Chapter 19 for details.

TC_Get

This general purpose subroutine lets you obtain or "get" the value(s) for any attribute(s) defined for the object or control whose ID is given.

```
CALL TC_Get, (id, attributes$, value$, values())
```

The second argument must be a string expression that contains the attributes names, separated by vertical bars (|). The third and fourth arguments contain the current values, string or numeric, of the attributes, in the same order as the attribute names themselves. Multiple string values are separated by vertical bars. Multiple numeric values appear as consecutive values in the numeric list. The lowest subscript of the list is 1, and the list `values()` is redimensioned to the exact size needed.

This calling sequence is similar to that described in Chapter 19 "Object Subroutine" for use with the OBJM_GET method. If `id` does not refer to an existing object or control, or if an attribute name is invalid for that type of object or control, an error occurs. See Chapter 19 for details.

TC_GetSysInfo

This subroutine provides a simpler interface to the OBJM_SYSINFO method with the built-in **Object** subroutine.

```
CALL TC_GetSysInfo (attribute$, value$, values())
```

For any valid attribute, the attribute values, string or numeric, will be returned. For details on valid attribute names, see Chapter 19 "Object Subroutine."

TC_Erase

This routine hides (erases or makes invisible) any True Controls object or control. It should be called with the ID number of the object or control to be erased. The statement:

```
CALL TC_Erase (id)
```

will erase the object or control whose ID is specified.

Remember that erasing (making invisible) a window also makes all graphics objects and controls contained within it not visible. All contained objects and controls that were visible will again become visible when the window is made visible again.

TC_Show

This routine reveals (shows or makes visible) any window, graphics object, or control. It should be called with the ID number of the object or control to be shown, as follows:

```
CALL TC_Show (id)
```

Remember that the window must be visible for the graphics objects and controls contained within it to be visible.

TC_Show_Default

```
CALL TC_Show_Default (flag)
```

Calling this routine with `flag` = 0 specifies that subsequently created graphics objects and controls will not automatically be shown upon creation. Calling this routine with `flag` = 1 specifies that they will be shown or displayed. Remember that the visibility of a window overrides the visibility of any control. Therefore, showing a control will not make it visible unless the containing window is also shown. Remember also that the value of the show-default flag does not affect windows; windows must be specifically shown using:

```
CALL TC_Show (wid)
```

TC_Select

```
CALL TC_Select (id)
```

This subroutine allows selecting selectable controls: push buttons, radio buttons, check boxes, edit fields, list edit buttons, and text edit controls. When applied to an edit field or a text edit control, that control becomes active and absorbs keystrokes. When applied to a push button, that button is selected and will be deselected when the Enter or Return key is pressed. When applied to a radio button or a check box, the result will be as if the user clicked on the button or box. A SELECT or CONTROL SELECT event will be generated as well. This operation may have no effect if the containing window hasn't been shown for the first time. (This routine is an interface to the SELECT method; see Chapter 19 "Object Subroutine.")

This routine cannot be used for windows. Instead use **TC_Win_Switch**, **TC_Win_Active**, or **TC_Win_Target**.

TC_Sensitize

```
CALL TC_Sensitize (id, flag)
```

If `flag` = 0, the control will be desensitized; that is, it will not respond to mouse clicks, etc. If `flag` = 1, the control will be made sensitive. If `id` does not refer to a control that can be made sensitive, no action will occur and no error will result.

TC_Free

```
CALL TC_Free (id)
```

This subroutine is the opposite of a "create" subroutine; it deletes the object or control, and frees all internal memory formerly associated with it. The ID number becomes invalid, although it may reappear in a subsequent "create" operation.

Freeing a window will free all menus and controls associated with it.

This routine should not be used to free an entire menu structure; there is a special subroutine for that purpose (see **TC_Menu_Free**, later.)

TC_SetUnitsToPixels
TC_SetUnitsToUsers

Calling one of these routines tells True Controls what units to use in placing windows on the screen, and controls and graphics objects within windows. The units remain in effect until changed by a call to the other routine.

When True Controls creates a new physical window it also opens a logical window within it whose default user coordinates are (0, 1, 0, 1).

TC_PixToUser
TC_UserToPix

Controls and graphics objects can be placed within windows in either the pixel coordinates of the physical window or user coordinates specified by the programmer. All True BASIC graphical output (**PLOT**, **BOX LINES**, etc.) are placed in user coordinates. These two routines transform one set of coordinates into the other.

```
CALL TC_PixToUser (px, py, wx, wy)
```

will convert the pixel point `(px,py)` into the user-coordinates point `(wx,wy)`.

```
CALL TC_UserToPix (wx, wy, px, py)
```

will convert the user point `(wx,wy)` into the pixel-coordinates point `(px,py)`.

Warning: when using either of these routines, it is imperative that control be in the correct logical windows. If the logical window is the default logical window within a physical window, this can be assured by using

```
CALL TC_Win_Target (mywin)
```

If the logical window is one that you created using the OPEN SCREEN statement, this can be assured by using

```
WINDOW #n
```

TC_SetRect

Normally, objects and controls are placed on the screen or in the physical window upon creation. But the program can move them around dynamically by re-specifying their location coordinates. If `cid` is the ID of a control, then

```
CALL TC_SetRect (cid, newxl, newxr, newyb, newyt)
```

will move that control to a new location in the window. If `cid` is the ID of a window, then calling this routine will move the window to a new location on the screen.

Warning: this routine makes no assumptions about the UNITS being used to locate a window or control. If the object is a window, the four values will be taken to be pixel coordinates of the full screen. If the object is a control or graphics object, the four values will be taken to be in whatever the current UNITS of the object or control. Furthermore, to use user coordinates, you must make sure you are in the correct logical window; otherwise, True BASIC will take the window coordinates of the current logical window to figure the new location of the object or control.

TC_GetRect

```
CALL TC_GetRect (cid, xl, xr, yb, yt)
```

This routine finds the current location, in pixels, of a window, graphics object, or control on the screen or within the containing window. As an example, if you want to move a control 10 pixels up and to the right, and if the control was placed originally using pixel coordinates, you might use:

```
CALL TC_GetRect (cid, xl, xr, yb, yt)
CALL TC_SetRect (cid, xl+10, xr+10, yb-10, yt-10)
```

Remember that in pixel coordinates smaller y values are closer to the top of the screen.

If the control or graphics object was placed originally using user coordinates, the procedure is more complicated. Suppose you want to move a control 0.1 user units up and to the right. You first must make sure you are in the correct logical window. This you must convert 0.1 into a certain number of pixels. Finally, you can now move the control. The following code illustrates these steps:

```
CALL TC_Win_Target (mywin)          ! Only one of this statement
WINDOW #n                           ! and this statement is needed.
CALL TC_UserToPix (0, 0, x1, y1)
CALL TC_UserToPix (.1, .1, x2, y2)
LET xdelta = x2 - x1
LET ydelta = y2 - y1
CALL TC_GetRect (cid, xl, xr, yb, yt)
CALL TC_SetRect (cid, xl+xdelta, xr+xdelta, yb+ydelta, yt+ydelta)
```

Note: since pixels coordinates increase from top to bottom, `ydelta` will probably be negative.

Two other routines, TC_SetRectUsers and TC_SetRectPixels, can be used to bypass the UNITS setting of the object or control. (Windows have no such setting; their location internally is always in pixels.)

Warning: as suggested in the description of TC_SetRect, use of the TC_GetRect subroutine is for expert use only; strange results can occur if misused.

TC_SetRectUsers
TC_SetRectPixels

```
CALL TC_SetRectUsers (cid, newxl, newxr, newyb, newyt)
CALL TC_SetRectPixels (cid, newxl, newxr, newyb, newyt)
```

These two routines are similar to TC_SetRect, except that they bypass the current value of the UNITS setting. That is, if user units were in effect upon the creation of the graphical object or control, then True BASIC will assume you mean user coordinates if you use TC_SetRect. If you need to use pixel coordinates for some special purpose, you can use TC_SetRectPixels. This routine first sets the UNITS to pixels coordinates, changes the location, and then sets the UNITS back to their original setting.

TC_SetRectUsers behaves similarly.

If you use TC_SetRectUsers to locate a new position for a window, the routine will convert from screen coordinates to pixel coordinates, and then relocate the window. However, no attempt will be made to make sure the window embellishments are entirely visible.

Warning: use of these two routines is for special purposes only. Misuse can result in strange behavior.

TC_SetText

```
CALL TC_SetText (id, text$)
```

This subroutine can be used with any control that allows setting the text – edit fields, static text fields, push buttons, check boxes, individual radio buttons, and text edit controls. (Edit fields and text edit controls also have similar routines that do the same thing. If you use this routine for a text edit control, the scroll bars will not be adjusted.)

TC_GetText

```
CALL TC_GetText (id, text$)
```

This subroutine can be used with any control that allows retrieving the text – edit fields, static text fields, push buttons, check boxes, and text edit controls. (Edit fields and text edit controls also have similar routines that do the same thing.)

TC_SetTextJustify

```
CALL TC_SetTextJustify (cid, justify$)
```

This subroutine alters the text justification for any control that permits it. The permissible values of `justify$` are: `"LEFT"`, `"CENTER"`, and `"RIGHT"`. Case doesn't matter. To have an effect, this routine must be called before the control is shown the first time. Furthermore, setting the justification may work only for static text boxes; certain operating systems might also allow justification for push buttons, check boxes, and rodir buttons.

Exception:	801	Invalid text justify option: ooooo

TC_SetList

```
CALL TC_SetList (cid, text$())
```

This routine can be used to set (or re-set) the list of a list button, a list edit button, or a list box. The subscript lower bound of `text$()` must be 0 or 1. If used to set the text of a list edit button, the 0-th entry, if any, will be used to set the contents of the list edit button itself.

TC_FontsAvailable

This subroutine returns the names of the fonts available on the current computer system.

```
CALL TC_FontsAvailable (fonts$())
```

will return the names, in the string list `fonts$()`, of the fonts currently available on the system.

TC_GetScreenSize

```
CALL TC_GetScreenSize (left, right, bottom, top)
```

This subroutines gives the size of the full screen in pixels. Specifically, `left` is the leftmost pixel, which is always numbered 0. `Right` is the rightmost pixel, which is always positive. `Bottom` is the bottom-most pixel, which is always positive. And `top` is the topmost pixel, which is always 0.

The number of horizontal pixels is `right - left + 1`, or simply `right+1`, since `left = 0`. The number of vertical pixels is `bottom - top + 1`, or simply `bottom+1`, since `top = 0`.

TC_Env_Set

This subroutine can be used only on Unix machines! On Unix machines:

```
   CALL TC_Env_Set (attribute$, value$)
```

will set the environmental attribute named to the value specified.

If used with non-Unix systems, an error will occur.

Window Subroutines

TC_Win_Create

```
   CALL TC_Win_Create (wid, options$, xl, xr, yb, yt)
```

creates a physical window. The rectangular coordinates do not define the outer dimensions of the window, but rather the interior portion of the window available to the user; this interior portion is known as the *client area*.. If you are using pixel coordinates (see TC_SetUnitsToPixels,) the actual placement of the window may depend on the operating system. Suppose you specify a rectangle that forces some portion of the window, either the client area or the embellishments, to be off the screen. On Windows, the size of the client area may be diminished so that the total size of the window, including embellishments, is no larger than the viewing screen, although the window may be offset. On the Macintosh, the client area will not be diminished, and could be partly or entirely off the screen.

If you are using True BASIC screen coordinates (see TC_SetUnitsToUsers,) True Controls will adjust the size and position of the client area, if necessary, so that it and all the embellishments (title bars, borders, etc.) will be visible on the screen. The screen coordinates are interpreted relative to the full screen.

Window number 0 is always created by the system and placed in a central location on the screen. It will be shown (i.e., made visible) upon a call to **TC_Show** with argument 0, or upon the first True BASIC output statement (**PRINT** or **PLOT** for example, but not **CLEAR**.)

The `option$` string may be a string variable or a string expression. This string will contain a sequence of words that will specify certain aspects of the window. The words may be in uppercase, lowercase, or mixed case, and they may be separated by spaces or vertical bars (|). The options are as follows:

`TITLE`	A title bar will be created.
`SIZE`	A resize box will be created.
`CLOSE`	A close box will be created.
`SHOW`	The window will be shown upon its creation
`HSCROLL`	A horizontal scroll bar will be attached.
`VSCROLL`	A vertical scroll bar will be attached.
`BORDER FULL`	The window will have a full border, and can have a title bar, close box, and resize box (default).
`BORDER SINGLE`	The window will have a single-line border, and cannot have a title bar, etc. An error may occur on some systems.
`BORDER DOUBLE`	The window will have a double-line border, and cannot have a title bar, etc. An error may occur on some systems.
`IMMUNE`	The window will be an immune window (default).
`NONIMMUNE`	The window will not be an immune window.
`ICONIZABLE`	The window can be made into an icon on those platforms that allow it.

Also, `"BORDER FULL"` windows may automatically have a title bar on some systems.

Windows are not shown as they are created unless the “SHOW” option is included. You can always show a window with:

```
   CALL TC_Show (wid)
```

Exceptions:	802	Can't have BORDER NONE for a regular window.
	802	Can't specify two or more border types.
	802	Can't have features in non-document windows.
	803	Can't create a child window with this routine.

TC_Win_ChildCreate

```
CALL TC_Win_ChildCreate (wid, options$, pwid, xl, xr, yb, yt)
```

creates a physical window that is a child window to the parent window identified by `pwid`. The properties of child windows may vary across systems. For example, on the Macintosh, a child window will always be in front of (i.e., active) relative to its parent.

The placement of child windows is always relative to the client area of the parent window. The child window will be clipped at the edges of the parent window.

The rectangular coordinates do not define the outer dimensions of the window, but rather the interior portion of the window available to the user; this interior portion is known as the *client area*.. If you are using pixel coordinates (see TC_SetUnitsToPixels,) the actual placement of the window may depend on the operating system. Suppose you specify a rectangle that forces some portion of the window, either the client area or the embellishments, to be outside the parent window. On Windows, the size of the client area may be diminished so that the total size of the window, including the border, is no larger than the viewing screen, although the window may be offset. On the Macintosh, the client area will not be diminished, and could be partly or entirely off the screen.

If you are using True BASIC screen coordinates (see TC_SetUnitsToUsers,) True Controls will adjust the size and position of the client area, if necessary, so that it and its border, if any, will be visible in the parent window. The screen coordinates will be interpreted relative to the client area of the parent window.

The `option$` string may be a string variable or a string expression. This string will contain a sequence of words that will specify certain aspects of the child window. The words may be in uppercase, lowercase, or mixed case, and they may be separated by spaces or vertical bars (`|`). The options are as follows:

`BORDER SINGLE`	The child window will have a single-line border.
`BORDER NONE`	The child window will have no border.
`SHOW`	The window will be shown upon its creation, provided the parent window is visible.
`IMMUNE`	The child window will be an immune window (default).
`NONIMMUNE`	The child window will not be an immune window.

Windows are not shown as they are created unless the "SHOW" option is included. You can always show the window with:

```
CALL TC_Show (wid)
```

Exceptions:	802	Can't specify two or more border types.
	804	Must specify type of border.
	804	Child window must have BORDER SINGLE or BORDER NONE.
	804	Can't have embellishments on child windows.

TC_Win_Target

This subroutine directs output to a specific physical window. The statement:

```
CALL TC_Win_Target (wid)
```

will direct subsequent output to physical window whose ID is given. Calling this subroutine will not affect either the window's visibility or, if visible, its placement (front and back) on the screen.

To direct output to a specific logical window within the physical window, the call to **TC_Win_Target** should be followed by a True BASIC **WINDOW** statement; otherwise, output will be directed to the default logical window that fills the physical window. If the desired logical window was created using an OPEN SCREEN statement, then it is necessary only to use a True BASIC WINDOW statement; the call to TC_Win_Target may be omitted.

Exception:	805	Can't make this window the target: nnn

TC_Win_Active

```
CALL TC_Win_Active (wid)
```

This subroutine can be used to move a visible window to the front position on the screen. Calling it has no effect on where the output is directed.

Exception:	806	Can't make this window active: nnn

TC_Win_Switch

```
CALL TC_Win_Switch (wid)
```

This subroutine will both direct output to the physical window specified and, if the window is visible, move it to the front of the screen; it combines the actions of **TC_Win_Target** and **TC_Win_Active**.

Exception:	807	Can't switch to this window: nnn

TC_Win_SwitchCurrent

```
CALL TC_Win_SwitchCurrent
```

This subroutine switches to the logical window that fills the currently-targeted physical window. You cannot use a WINDOW statement because this logical window's number is hidden. If you know the window ID of the currently-targeted window, you can also use:

```
CALL TC_Win_Switch (wid)
```

TC_Win_NoHide

```
CALL TC_Win_NoHide (wid, flag)
```

Normally, when the user clicks on the close box of a window and TC_Init has been called, True BASIC erases the window but doesn't stop running the program. Calling this routine with `flag` = 1 will prevent this automatic erasing. Calling it with `flag` = 0 will restore the default situation.

TC_Win_MouseMove

```
CALL TC_Win_MouseMove (wid, flag)
```

This subroutine is used to activate or inhibit (default) the mouse move event in a particular window. If `flag` = 0, mouse move events will be inhibited (i.e., not be returned by **TC_Event**); if `flag` = 1, mouse move events will be generated (i.e., returned by **TC_Event**).

The mouse move event is "MOUSE MOVE". It can be used to track the mouse within a given window. Alternatively, the True BASIC statement **GET MOUSE** can also be used.

TC_Win_Valid

```
TC_Win_Valid (wid)
```

Calling this subroutine will be a "no operation" if the window ID `wid` refers to an existing window. If the window ID is not valid, an error will occur.

Exception:	808	Illegal window number: nnn

TC_Win_SetTitle

If the window has previously been created with a `"BORDER FULL"` and `"TITLE"` options, this subroutine should be used to set the title.

```
CALL TC_Win_SetTitle (wid, "My New Window")
```

The title of a visible window can be changed at any time, and takes effect immediately.

TC_Win_GetTitle

This subroutine finds the current title of a window as follows:

```
CALL TC_Win_GetTitle (wid, title$)
```

TC_Win_SetCursor

This subroutine sets the type of cursor to be used with the window.

```
CALL TC_Win_SetCursor (wid, cursor$)
```

The cursor type can be `"ARROW"`, `"IBEAM"`, `"CROSS"`, `"PLUS"`, `"WAIT"`, and `"USER"`. The system will then show the I-beam when the mouse position is inside the edit field or text editor, but will show in another form when the mouse is outside. (Note: Type `"USER"` is not yet implemented.)

Exception:	809	Invalid cursor shape: ccccc

TC_Win_SetPen

```
CALL TC_Win_SetPen (wid, width, color, style$, pattern$)
```

This subroutine sets certain attributes of the pen for the window. These include: the pen width, the pen color, the pen style, and the pen pattern. Attributes will not be changed if the arguments to this subroutine are < -2 (if numeric) or the null string (if string).

The pen width is specified in pixels. The default is one pixel.

The pen color is specified by an index into the current color mix table (see Chapter 13 “Graphics”). The default color is -1 (black.)

The pen style must be one of:

`"SOLID"`	Solid line (default)
`"DOT"`	Dotted line
`"DASH"`	Dashed line

The pen pattern must be one of:

`"SOLID"`	Solid (default)
`"HOLLOW"`	The line traced by the pen is invisible
`"RUBBER"`	Small dashed lines that move on some systems.

There are several restrictions on combinations of width, style, and pattern. If the width is one pixel, then styles `DOT` and `DASH` override the pattern, making it `SOLID`. If the width is greater than one pixel, then style `SOLID` and pattern `SOLID` override. In other words, `DOT`, `DASH`, and `RUBBER` can only happen with one-pixel pens. And the style setting overrides the pattern setting.

For example:

```
CALL TC_Win_SetPen (0, 1, 4, "DOT", "")
```

will set the pen in the standard output window to width one pixel, color 4 (usually red), and style dotted.

Exception:	810	Invalid window pen setting.

TC_Win_SetBrush

```
CALL TC_Win_SetBrush (wid, backcolor, color, pattern$)
```

This subroutine controls the color and pattern for the brush, as well as the background color for both the pen and the brush.

The background color refers to the current color mix table. If `backcolor` is < -2, the current background color will not be changed. The default background color is -2 (white.)

The brush color, like the pen color, refers to the color mix table. If `color` is < -2, the current brush color will not be changed. The default color is -1 (black.)

The brush pattern must be one of:

`"SOLID"`	(default)
`"HOLLOW"`	
`"HORIZ"`	
`"VERT"`	
`"FDIAG"`	
`"BDIAG"`	
`"CROSS"`	
`"DIAGCROSS"`	

For example,

```
CALL TC_Win_SetBrush (0, 3, -1, "diagcross")
```

will change the background color for the standard output window to color number 3, and will set the brush to a diagonal crosshatch pattern. Note: the background color will not take effect immediately, but only after a **CLEAR** or similar operation.

Exception: 811 Invalid window brush setting.

TC_Win_RealizePalette

```
CALL TC_Win_RealizePalette (wid)
```

Calling this subroutine causes True BASIC to add to the system color mix table those entries in the True BASIC color mix table that are not already there. This may eliminate "dithering" on some systems. (Dithering occurs when there isn't an exact match in the system color mix table corresponding to the color desired. The system may then attempt to construct the desired color by mixing two or more of the existing colors in its color mix table. This process is called *dithering*.)

TC_Win_SetDrawmode

```
CALL TC_Win_SetDrawmode (wid, mode$)
```

The string values allowed for `mode$` are as follows; note that where there is a space in the string, there must be exactly one space:

`"COPY"`	ignore current contents, draw over anything that is there (default)
`"OR"`	perform bitwise OR between bit plans of each currently displayed pixel and the new pixel which is to overlay it
`"XOR"`	perform bitwise XOR between bit plans of each currently displayed pixel and the new pixel which is to overlay it
`"CLEAR"`	clear the screen to the extent covered by the item being drawn
`"NOT COPY"`	the bitwise negation of COPY
`"NOT OR"`	the bitwise negation of OR
`"NOT XOR"`	the bitwise negation of XOR
`"NOT CLEAR"`	the bitwise negation of CLEAR

The drawing mode determines how the window's pen and brush interact with the background, including what has already been drawn. As an example, assume there are four bit planes (i.e., there are sixteen entries in the color map table), the background is color 6 (binary 0110), and the pen is color 10 (binary 1010). Then the above drawing modes would give the following results for each pixel covered by the pen:

`"COPY"`	Color 10 (binary 1010)
`"OR"`	Color 14 (binary 1110)
`"XOR"`	Color 12 (binary 1100)
`"CLEAR"`	Color 0 (binary 0000)
`"NOT COPY"`	Color 5 (binary 0101)
`"NOT OR"`	Color 1 (binary 0001)
`"NOT XOR"`	Color 3 (binary 0011)
`"NOT CLEAR"`	Color 15 (binary 1111)

Exception:	812	Invalid window drawmode setting.

Something similar may operate for a larger number of bit planes, and will sometimes yield strange-looking results. But the modes `"COPY"` and `"CLEAR"` should work as expected. And `"XOR"` should have the property that, if you draw the object twice, you will get back the original background.

TC_Win_SetFont

```
CALL TC_Win_SetFont (window, fontname$, fontsize, fontstyle$)
```

This subroutine sets certain attributes of the font used in **PRINT** statements. The `fontname$` must be a legal font name; the names "Helvetica", "Fixed", "Times", and "System" will always be legal.

The `fontsize` is given in points, which are approximately 1/72 of an inch. (How big the font is on the screen will depend on the characteristics of the monitor.)

The `fontstyle$` may be one of "Plain", "Bold", "Italic", or "Bold Italic".

The font name and font style may be given in uppercase, lowercase, or mixed case.

If `fontsize` is a negative number, the font size will not be changed. If `fontname$` or `fontstyle$` is the null string, then it will not be changed.

For example:

```
CALL TC_Win_SetFont (0, "Fixed", 10, "Bold")
```

will set the printing font for the standard output window to 10 point fixed (nonproportional) bold. The Fixed font is usually something like Courier.

Exceptions:	813	Invalid window font name: nnnnn
	814	Invalid window font size: sss
	815	Invalid window font style: sssss

If you have included attached scroll bars, the following twelve routines can be used to manipulate them. They operate the same as the corresponding routines for scroll bars that are not attached to windows.

TC_WinHSBar_SetPosition

```
CALL TC_WinHSBar_SetPosition (wid, position)
```

This subroutine sets the position of the slider (the thumb) of an attached horizontal scroll bar. The position is defined in terms of the scrollbar parameters `srange`, `erange`, and `prop`. If the `position` is <= `srange`, the slider will be moved to the left of the scroll bar. If the `position` is >= `erange - prop`, the slider will be moved to the right of the scroll bar. If the `position` is in between, the slider will be moved to the corresponding location. The default value is 0.

TC_WinVSBar_SetPosition

```
CALL TC_WinVSBar_SetPosition (wid, position)
```

This subroutine sets the position of the slider (the thumb) of an attached vertical scroll bar. The position is defined in terms of the scrollbar parameters `srange`, `erange`, and `prop`. If the `position` is <= `srange`, the slider will be moved to the top of the scroll bar. If the `position` is >= `erange-prop`, the slider will be moved to the bottom of the scroll bar. If the `position` is in between, the slider will be moved to the corresponding location. The default value is 0.

TC_WinHSBar_GetPosition

```
CALL TC_WinHSBar_GetPosition (wid, position)
```

This subroutine finds out the current location of an attached horizontal scroll bar slider. The current position must be interpreted in terms of the scrollbar parameters `srange`, `erange`, and `prop`.

TC_WinVSBar_GetPosition

```
CALL TC_WinVSBar_GetPosition (wid, position)
```

This subroutine finds out the current location of an attached vertical scroll bar slider. The current position must be interpreted in terms of the scrollbar parameters `srange`, `erange`, and `prop`.

TC_WinHSBar_SetRange

```
CALL TC_WinHSBar_SetRange (wid, srange, erange, prop)
```

This subroutine is used to set the scrollbar parameters that specify the extreme positions of the slider for an attached horizontal scroll bar, as well as the proportional size of the slider on those systems that provide for varying-sized sliders.

These parameters are arbitrary. The relation to the position (and size of the slider on certain systems) is as follows: When the slider is at the left, its position is equal to `srange`. When the slider is at the right, its position is equal to `erange - prop`. On certain systems, the size of the slider relative to the size of the slider trough (the region in which the slider moves) is given by `prop/(erange - srange)`, but it is never greater than one. The default values are 0, 100, and 1, respectively.

TC_WinVSBar_SetRange

```
CALL TC_WinVSBar_SetRange (wid, srange, erange, prop)
```

This subroutine is used to set the scrollbar parameters that specify the extreme positions of the slider for an attached vertical scroll bar, as well as the proportional size of the slider on those systems that provide for varying sized sliders.

These parameters are arbitrary. The relation to the position (and size of the slider on certain systems) is as follows: When the slider is at the top its position is equal to `srange`. When the slider is at the bottom, its position is equal to `erange - prop`. On certain systems, the size of the slider relative to the size of the slider trough (the region in which the slider moves) is given by `prop/(erange - srange)`, but never greater than one. The default values are 0, 100, and 1, respectively.

TC_WinHSBar_GetRange

```
CALL TC_WinHSBar_GetRange (wid, srange, erange, prop)
```

This subroutine finds out the current values of the scrollbar parameters for an attached horizontal scroll bar.

TC_WinVSBar_GetRange

```
CALL TC_WinVSBar_GetRange (wid, srange, erange, prop)
```

This subroutine finds out the current values of the scrollbar parameters for an attached vertical scroll bar.

TC_WinHSBar_SetIncrements

```
CALL TC_WinHSBar_SetIncrements (wid, single, page)
```

This subroutine is used to set the scrollbar increments for an attached horizontal scroll bar. The `single` increment defines how far the slider (thumb) moves when the user clicks in the left-arrow or right-arrow boxes at the ends of the scroll bar. The `page` increment defines how far the slider moves when the user clicks in the gray area either left or right of the slider. The default values are 1 and 10, respectively.

The actual movement of the slider is in terms of the parameters (see **TC_WinHSBar_SetRange** for details.)

TC_WinVSBar_SetIncrements

```
CALL TC_WinVSBar_SetIncrements (wid, single, page)
```

This subroutine is used to set the scrollbar increments for an attached vertical scroll bar. The `single` increment defines how far the slider (thumb) moves when the user clicks in the up-arrow or down-arrow boxes at the ends of the scroll bar. The `page` increment defines how far the slider moves when the user clicks in the gray area either above or below of the slider. The default values are 1 and 10, respectively.

The actual movement of the slider is in terms of the parameters (see **TC_WinVSBar_SetRange** for details.)

TC_WinHSBar_GetIncrements

```
CALL TC_WinHSBar_GetIncrements (wid, single, page)
```

This subroutine finds out the current values of the increments for an attached horizontal scroll bar.

TC_WinVSBar_GetIncrements

```
CALL TC_WinVSBar_GetIncrements (wid, single, page)
```

This subroutine finds out the current values of the increments for an attached vertical scroll bar.

TC_Win_PageSetup

```
CALL TC_Win_PageSetup (wid)
```

This subroutine causes a page setup dialog box to be displayed. The properties specified then apply to all printing that takes place from that window, or printing to a file that occurs while that window is the target window.

TC_Win_Print

```
CALL TC_Win_Print (wid)
```

This subroutine prints the contents of the window, including graphics, to the current printer.

Note: printed text can be sent directly to the printer by first OPENing the printer, as in

```
OPEN #1: printer
```

and then using the PRINT statement, as in

```
PRINT #1: ...
```

The text will be printed in the font type, size, style, and color of the currently targeted physical window.

TC_Win_Update

```
CALL TC_Win_Update (wid, left, right, bottom, top)
```

This subroutine will update the contents of the window specified, or a sub-portion thereof. Generally, you will not need to use this process if you are using immune windows.

Menu Subroutines

TC_Menu_Set

```
CALL TC_Menu_Set (wid, menu$(,))
```

This subroutine sets, or resets, the menus for the given window. The rows of the `menu$` matrix correspond to the menus. They must be numbered starting with 0 or 1. (At present, the 0-th row is ignored.) The columns of the `menu$` matrix correspond to the menu items. They must be numbered starting with 0.

The entries in the menu matrix are the words that are displayed in the menu header and in the individual menus. The 0-th column gives the menu header names. The remaining columns give the menu item names.

When the event handler **TC_Event** returns an event of type MENU, the menu and item numbers are also returned; they refer to the original `menu$(,)` matrix.

The number of columns in `menu$(,)` must be large enough to contain the longest menu. For shorter menus, the remaining entries should consist of the null string.

The words that form the menu header and the menu items are displayed exactly as provided by the `menu$(,)` matrix. (The words may be changed at any time by calling the subroutine **TC_Menu_SetText**.) They may contain any letters or other characters, except the at-sign "@", which has special meanings.

If a menu item consists of "@" alone, a dashed-line separator will appear in that position. If the final two characters of a menu item consist of "@" followed by a letter, that letter will be used as a hot-key alternative for activating the menu. (The conventions differ on different platforms. For Windows and OS/2, the letter following the "@" must be contained within the menu item itself. When the menu is displayed, that letter will be underlined.)

Hierarchical menus can also be constructed. First, remember that all menus, hierarchical or not, occupy a single row in the `menu$(,)` matrix. The hierarchical part is indicated as follows: The last two characters of the parent menu item consist of the sequence "@@". And the menu header (the entry in the 0-th column of `menu$(,)`) must start with a single "@" but then contain the same word as the parent. For example, if "Color" is to be the start of a hierarchical menu, it should appear as "Color@@". Correspondingly, at a later row in the `menu$(,)` matrix, there must be a menu header that appears as "@Color".

Note that although the menus will appear on the screen as hierarchical menus, you will treat them simply as rows and columns in the `menu$(,)` matrix.

You can specify platform-dependent menu items. The following example illustrates this use:

```
MAT READ menus$(2, 0 to 3)
DATA |MAC|File|OTHER|File@F|, Open@O, Save@S, |MAC|Quit@Q|OTHER|Exit@X|
DATA |MAC|Edit|OTHER|Edit@E|, |MAC|Cut@X|OTHER|Cut@T|, Copy@C, |MAC|Paste@V|OTHER|Paste@P|
```

Acceptable platform names are MAC, WIN32, OS/2, and UNIX. WIN32 refers to all Windows platforms: Windows 3.1 with Win32s, Windows 95, and Windows NT.

Exceptions:	820	Lower bound for menu$ rows must be 0 or 1.
	821	Lower bound for menu$ columns must be zero.
	822	Menu$ array must have elements.
	823	Don't recognize menu item: iiiii
	824	Can't find parent menu: ppppp

TC_Menu_AddItem

```
CALL TC_Menu_AddItem (wid, menu, text$)
```

This subroutine adds a menu item at the end of an existing menu. The new menu item may be a separator, or may have a hot-key equivalent. But such an item cannot be the start of a new hierarchical menu.

Note that the added menu item does not appear in your original `menu$(,)` matrix. You must therefore be careful when using this subroutine. Furthermore, if you wish to change a menu item in the middle of a menu, it will be much safer to delete the entire menu structure, make the changes in the `menu$(,)` matrix, and then call **TC_Menu_Set** afresh.

Exceptions:	825	Menu or item number must not be negative.
	826	To add a menu item, menu must already exist.
	827	Added menu item cannot be hierarchical.

TC_Menu_DelItem

```
CALL TC_Menu_DelItem (wid, menu, item)
```

This subroutine deletes a specific menu item from a menu. However, **TC_Event** will report menu events in terms of the row and column of the original `menu$(,)` matrix. So, you must be quite careful when using this subroutine. It would be much safer to delete the entire menu structure, make the changes in the `menu$(,)` matrix, and then call **TC_Menu_Set** afresh.

TC_Menu_AddMenu

```
CALL TC_Menu_AddMenu (wid, menu$())
```

This subroutine is used to add an entirely new menu onto the end of the current menu structure. The new menu is given in the list `menu$()`, which must have a 0-th entry to contain the menu header. This subroutine will normally be used to add a special menu, one that will later be deleted.

TC_Menu_DelMenu

```
CALL TC_Menu_DelMenu (wid)
```

This subroutine will delete the last menu of the menu structure. Used in conjunction with **TC_Menu_AddMenu**, this subroutine will delete a special menu previously added.

TC_Menu_SetText

```
CALL TC_Menu_SetText (wid, menu, item, newtext$)
```

This subroutine changes the word or text that appears in the given menu position. The new word cannot contain a "@"; in other words, separators, hot keys, and hierarchical menus cannot be specified with this subroutine.

Exception: 828 Invalid window, id, menu, item combination.

TC_Menu_GetText

```
CALL TC_Menu_GetText (wid, menu, item, text$)
```

This subroutine returns the word or text that appears in the given menu position.

Exception: 828 Invalid window, id, menu, item combination.

TC_Menu_SetCheck

```
CALL TC_Menu_SetCheck (wid, menu, item, flag)
```

This subroutine sets, or unsets, the check mark that can appear just to the left of the menu item word. If `flag` = 1, the check mark will be added; if `flag` = 0, it will be removed.

It is not permitted to "check" a menu header or a separator. Moreover, the menu item must be "checkable" to permit adding a check; that is, there must be a character space to the left of the menu item as it is displayed. (Menu items are checkable by default. To make them non-checkable, perhaps to save space, use the **Object** subroutine directly; see Chapter 19.)

Exception: 828 Invalid window, id, menu, item combination.

TC_Menu_GetCheck

```
CALL TC_Menu_GetCheck (wid, menu, item, flag)
```

This subroutine is used to determine the check mark status of a menu item. If the item is checked, the value of `flag` will be 1; if not checked, the value of `flag` will be 0.

Exception:	828	Invalid window, id, menu, item combination.

TC_Menu_SetEnable

```
CALL TC_Menu_SetEnable (wid, menu, item, flag)
```

This subroutine is used to "gray" or "ungray" a menu item. If `flag` = 0, the menu item will be "grayed" or disabled; that is, it will appear dimmed. Such an item cannot be selected. If `flag` = 1, the menu item will be enabled and will appear normal. If you disable a menu header (item 0,) the entire menu will be disabled. Menu separators cannot be disabled.

Exception:	828	Invalid window, id, menu, item combination.

TC_Menu_GetEnable

```
CALL TC_Menu_GetEnable (wid, menu, item, flag)
```

This subroutine is used to determine the state – "gray" or "ungray" – of a menu item. If `flag` = 0, the menu item is now "grayed" or disabled; that is, it will appear dimmed. Such an item cannot be selected. If `flag` = 1, the menu item is now enabled and appears normal.

Exception:	828	Invalid window, id, menu, item combination.

TC_Menu_Free

```
CALL TC_Menu_Free (wid)
```

This subroutine should be used to free the entire menu structure associated with the window. For example, you may wish to replace the menu structure with an entirely different one.

It is not necessary to call this subroutine if you intend to "free" the window itself; freeing a window automatically frees all entities associated with it.

Check Box Subroutines

TC_Checkbox_Create

```
   CALL TC_Checkbox_Create (cid, text$, xl, xr, yb, yt)
```

This subroutine creates a check box, which is a small square box that can either be empty or can contain an “X”. The text provided will appear just to the right of the check box and will be left-justified.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If either `yb` or `yt` < 0, the default height of a checkbox will be used. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window. If `yb` or `yt` = -99999, the default height of a check box will be used.

The check box will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.
If text$ ends with `"|LEFT"`, `"CENTER"`, or `"|RIGHT"`, and the operating system permits, the checkbox text will be justified accordingly.

TC_Checkbox_Set

```
   CALL TC_Checkbox_Set (cid, state)
```

If `state` = 1, an “X” will appear in the check box. If `state` = 0, the check box will be cleared.

TC_Checkbox_Get

```
   CALL TC_Checkbox_Get (cid, state)
```

This subroutine finds the state of a check box. If the check box is checked, `state` =1; if the check box is not checked, `state` = 0.

You can change the text of a check box using

```
   CALL TC_SetText (cid, newtext$)
```

Edit Field Subroutines

TC_Edit_Create

```
CALL TC_Edit_Create (cid, initialtext$, xl, xr, yb, yt)
```

This subroutine creates a one-line editable field that allows the usual editing operations — left and right cursor controls, mouse clicks, keystrokes, delete and/or backspace — according to the conventions of the operating system.

The initial text of the edit field can also be set at this time.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If either `yb` or `yt` < 0, the default height of an edit field will be used. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window. If `yb` or `yt` = -99999, the default height of an edit field will be used.

The edit field will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.

TC_Edit_SetText

```
CALL TC_Edit_SetText (cid, newtext$)
```

This subroutine sets or changes the text that appears in the edit field.

You can also change the text of an edit field with

```
CALL TC_SetText (cid, newtext$)
```

TC_Edit_GetText

```
CALL TC_Edit_GetText (cid, text$)
```

This subroutine finds out the text in an edit field. This can be done at any time. However, the user may still be making modifications to the text. It is a good idea to require the user to select another control, or to select some push button, to determine that the user has, in fact, completed his or her editing.

TC_Edit_SetFormat

```
CALL TC_Edit_SetFormat (cid, format$)
```

This subroutine sets a format for the edit field against which the text can be compared.

Allowable formats are:

null string	No checking is done
`ALPHA`	Only letters of the alphabet and spaces are allowed
`ALPHANUM`	Only letters, digits, and spaces are allowed
`NUMBER`	Only numbers are allowed (True BASIC numeric constants)
`INTEGER`	Only digits are allowed
`RANGE`	Only integers in the range specified are allowed (see later)
`FRANGE`	Only numbers in the range specified are allowed (see later)
`ZIP`	Only five-digit or nine-digit zip codes are allowed
`PHONE`	Only phone numbers are allowed (several formats permitted)
`SS`	Only Social Security numbers are allowed
`DATE`	Only dates are allowed (several formats permitted)
`LENGTH`	Only sequences of a certain length are allowed (see later)
`FORMAT`	Only sequences that match, character by character, the format string provided are allowed
`LIST`	Only entries that match one of the items in a list are allowed

The `NUMBER` and `INTEGER` checks are made by attempting to use the True BASIC **VAL** function. In the case of `INTEGER`, the entry is first checked to see if it contains anything but digits.

To use the `RANGE` format, you must supply the smallest integer and the largest integer that will be allowed. These integers follow the word `RANGE` with but one space between. For example:

```
"RANGE 20 35"
```

will allow the user to enter any integer number in the range 20 to 35, inclusive.

Similarly, with `FRANGE` you supply the smallest and largest general numbers that will be allowed, separated by one space. For example,

```
"FRANGE -2.7 12.3"
```

will allow any general number between -2.7 and 12.3, inclusive.

The `ZIP` format will accept any sequence of five digits, or any sequence of five digits followed by a hyphen and followed by another four digits.

The `PHONE` format will accept phone numbers in three formats: 999-9999, 999-999-9999, or 999/999-9999.

The `SS` format will accept only sequences in the format 999-99-9999.

The `DATE` format will accept strings in any of these format: YYYYMMDD, YYYY_MMM_DD, YYYY/MMM/DD, DD MMM YYYY, and MMM DD, YYYY. (MM stands for the month number, while MMM stands for the three-letter abbreviation of the month name.)

The `LENGTH` format may require the user to supply exactly the correct number of characters, or a number of characters in some range. For example:

```
LENGTH 15
LENGTH 12 24
```

The first case requires the user to supply exactly 15 characters, while the second requires a number of characters between 12 and 24, inclusive.

The `FORMAT` format allows you to specify the type of each character supplied by the user. The character types are:

A	The character must be a letter or space
9	The character must be a digit
X	The character must be a letter or a digit or a space
?	The character can be anything
all others	The character must match exactly

For example, if you wish the user to enter a dollar value of the form “$1,234.56”, you might specify as the format “$9,999.99”.

Finally, for the `LIST` format, the user’s entry is required to match one of a sequence of words, without regard to case. The words may be separated by spaces or commas. For example, if you require the user to enter M or F, you might use

```
"LIST M F"
```

TC_Edit_CheckField

```
CALL TC_Edit_CheckField (cid, errormess$)
```

This routine checks the contents of the edit field specified by `cid` against the format (see TC_Edit_SetFormat). If the contents are consistent with the format, `errormess$` is the null string; otherwise, `errormess$` contains the reason for the inconsistency.

Graphical Subroutines

TC_Graph_Create

```
CALL TC_Graph_Create (gid, type$, xl, xr, yb, yt)
```

This subroutine creates a graphics object of the type specified. The types available are:

`CIRCLE`	A circle or ellipse
`RECTANGLE`	A rectangle
`ROUNDRECT`	A rectangle with rounded corners
`LINE`	A straight line segment
`ALINE`	A line segment with arrow heads
`ARC`	An arc of a circle or ellipse
`PIE`	A pie section of a circle or ellipse
`POLYLINE`	A set of joined line segments, but not necessarily closed
`POLYGON`	A set of line segments forming a closed region
`IMAGE`	An image (bmp, pict, etc.) from a file.

The graphical object will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window.

For the `CIRCLE`, `RECTANGLE`, and `ROUNDRECT`, the coordinates define the bounding rectangle. The roundness of the corners of the rounded rectangle is set separately. For the `LINE` and `ALINE`, the coordinates are interpreted as follows: starting point (`xl`, `yb`), ending point (`xr`, `yt`). The presence or absence of arrowheads in the `ALINE` is set separately.

For a `LINE` or `ALINE`, the four coordinates define the start and end of the line or arrowed line as follows:

xl = starting x
yb = starting y
xr = ending x
yt = ending y

Thus, to draw a line from (1,2) to (3,4), you would use:

```
CALL TC_Graph_Create (gid, "LINE", 1, 3, 2, 4)
```

`ARC` and `PIE` start with the circle or ellipse defined by the coordinates. `ARC` consists of a portion of the circumference of the circle or ellipse. `PIE` consists of an arc plus straight lines from the ends of the arc to the center of the circle or ellipse. The extent of the `ARC` or `PIE` is specified separately.

`POLYLINE` consists of a set of (x,y) points joined with straight line segments. `POLYGON` is a polyline with the first and last point joined. For these objects, the location coordinates are ignored, except for applying **TC_Graph_Scale**.

`IMAGE` prepares the way for importing an image (such as a BMP or PICT) from a file. The filename from which the image is to be imported, or the box keep string, is specified separately.

Graphics objects float above the True BASIC output canvas as do controls such as push buttons. But, if the window is cleared with a **CLEAR** statement, visible graphics objects are made invisible. They must individually be reshown using a call to **TC_Show**.

Exception: 840 Invalid graphics type: ttttt

TC_Graph_SetPoly

```
CALL TC_Graph_SetPoly (gid, pts(,))
```

This subroutine provides the (x,y) points that define the polyline. The two-dimensional matrix is assumed to have exactly two columns, and to have exactly the number of rows as there are point-pairs. (It is not necessary that the lower bounds be 1, or anything else.)

The new polyline or polygon is shown as soon as the points, or the new points, are specified.

Exception:	841	POINTS must be in pairs for vertices.

TC_Graph_SetArc

```
TC_Graph_SetArc (gid, starta, stopa)
```

This subroutine defines the starting and ending points of an arc, or those points of the arc that is a part of a pie segment. The ends of the `ARC` are defined as follows: the arc is a section of the circumference of the largest circle or ellipse contained in the rectangular region specified in the call to **TC_Graph_Create**. The starting angle `starta` defines a radius starting at the center of the circle or ellipse. The intersection of this radius with the circumference defines the starting point of the arc. The ending angle `stopa` similarly defines a radius, the intersection of which with the circumference defines the ending point of the arc. The angles are measured in degrees starting with the positive x-axis, and increasing counterclockwise. The arc is drawn *counterclockwise* from the starting point to the ending point, regardless of the coordinate system.

For the `PIE` type, the straight lines between the center of the circle or ellipse and the starting and ending points are made visible .

TC_Graph_SetRoundRect

```
CALL TC_Graph_SetRoundRect (gid, owidth, oheight)
```

This subroutine sets the degree of roundness for the corners of a rounded rectangle. It works like this: Imagine an ellipse with half-length equal to `owidth`, and half-height equal to `oheight`. Cut this ellipse into four quadrant parts. These four parts become the four corners of the rounded rectangle.

If the units for placing the object are in pixels, then these values should also be in pixels. If the units are in user coordinates, then these values should also be given in user coordinates.

TC_Graph_SetAline

```
CALL TC_Graph_SetAline (gid, start, end)
```

This subroutine adds or removes arrowheads from either end of an `ALINE`. If `start` = 1, an arrowhead will be placed at the beginning of the line; if `start` = 0, such an arrowhead will be removed. If `end` = 1, an arrowhead will be placed at the end of the line; if `end` = 0, such an arrowhead will be removed.

TC_Graph_SetImage

```
CALL TC_Graph_SetImage (gid, filename$, adjustflag)
```

This subroutine causes an image to be imported from the file specified. The value of adjustflag specifies how the image will be displayed in terms of the rectangle specified in the call to **TC_Graph_Create**.

This subroutine merely calls TC_Graph_SetImageFromFile with a null filetype, and is retained for compatibility only. A description of adjustflag is given in the description of TC_Graph_SetImageFromFile.

TC_Graph_SetImageFromFile

```
CALL TC_Graph_SetImageFromFile (gid, filename$, filetype$, adjustflag)
```

This subroutine causes an image to be imported from the file specified. If you know the file type, you should supply it. Allowable file types are: `JPEG`, `PICT` (Macintosh only), `MS BMP`, `OS/2 BMP`, and possibly `PCX`. Spaces are important but case (upper or lower) is not.

Adjustflag	Action
1	The image will be centered at the center of the rectangle specified at the creation of the graphics object. The size of the image will not be changed. Other than its center, the actual size of the rectangle will be ignored. Part or all of the image may be off the screen.
0	The image will be forced into the rectangle specified at the creation of the graphics object. The image will be expanded or contracted as needed.
-1	The image will be centered in the window, and will retain its original size. The rectangle specified at creation time will be ignored. If the image is larger than the window, portions of the image may not show.

In cases +1 and -1, the defining rectangle for the graphics object will be changed.

TC_Graph_SetImageFromBox

```
CALL TC_Graph_SetImageFromBox (gid, boxstring$)
```

This subroutine causes an image to be constructed from a box keep string, which must be in the local format.

If you need to adjust the size of the image, you will have to save the box string into an image file using CALL Write_Image, and then use CALL TC_Graph_SetImageFromFile to display the image, possibly with adjustment of its size.

See also the subroutines Read_Image and Write_Image, described in Chapter 18. Read_Image reads a bit-mapped image (i.e., BMP) from a file and converts it into a box keep string. Write_Image converts a box keep string into a bit-mapped image and writes it to a file.

TC_Graph_GetImageToBox

```
CALL TC_Graph_GetImageToBox (gid, boxstring$)
```

This subroutine takes an image being displayed on the screen and converts it into a box keep string in the local format.

See also the subroutines Read_Image and Write_Image, described in Chapter 18. Read_Image reads a bit-mapped image (i.e., BMP) from a file and converts it into a box keep string. Write_Image converts a box keep string into a bit-mapped image and writes it to a file.

TC_Graph_Shift

```
TC_Graph_Shift (gid, deltax, deltay)
```

This subroutine allows you to move or shift a graphics object by the amount `deltax` in the x-direction and `deltay` in the y-direction. The actual movement on the screen depends on the orientation of the coordinate system. For example, if you used pixel coordinates when creating the object and you specify a positive value for `deltay`, the object will be moved downward.

TC_Graph_Scale

```
TC_Graph_Scale (gid, scalex, scaley)
```

This subroutine allows you to change the size of a graphics object. The object will be stretched in the x-direction is `scalex` is greater than 1, or shrunk if `scalex` is less than 1; similarly for the y-direction. The stretching or shrinking is done relative to the center of the graphics object, and not in relation to the origin of the coordinate system (as with the SHIFT transformation with the **DRAW** statement.)

Certain graphics objects may be distorted if expanded beyond the edges of the window.

TC_Graph_SetPen

```
CALL TC_Graph_SetPen (gid, width, color, style$, pattern$)
```

This subroutine can be used to set certain attributes of the pen that draws the particular graphics object. These include: the pen width, the pen color, the pen style, and the pen pattern. The width will not be changed if it is negative; the color will not be changed if it is < -2. The style or pattern will not be changed if the null string is provided. Each graphics entity can have its own pen characteristics.

The pen width is specified in pixels. The default width is one pixel.

The pen color is specified by an index into the current color mix table. The default color is -1 (black.)

The pen style must be one of:

`SOLID`	Solid line (default)
`DOT`	Dotted line
`DASH`	Dashed line

The pen pattern must be one of:

`SOLID`	Solid (default)
`HOLLOW`	The line drawn is invisible
`RUBBER`	Small dashed lines that move on some systems.

There are several restrictions on combinations of width, style, and pattern. If the width is one pixel, then styles `DOT` and `DASH` override the pattern, taking it to be `SOLID`. If the width is greater than one pixel, then style `SOLID` and pattern `SOLID` override. In other words, `DOT`, `DASH`, and `RUBBER` can only happen with one-pixel pens. And the style setting overrides the pattern setting. If the width is more than one pixel, the approximate center of the curve or line will follow the path specified.

For example

```
CALL TC_Graph_SetPen (gid, 10, 4, "", "")
```

will set the pen for graphics object gid to width ten pixels, color 4 (usually red), and with no change to the style or pattern.

(The conventions are identical with those of **TC_Win_SetPen**.)

Exceptions:	842	Invalid pen style: sssss
	844	Invalid pen pattern: ppppp

TC_Graph_SetBrush

```
CALL TC_Graph_SetBrush (gid, backcolor, color, pattern$)
```

This subroutine controls the color and pattern for the brush that is used to fill the interior of several of the graphics objects, as well as the background color for both the pen and the brush.

The brush color, like the pen color, refers to the color mix table. If `color` is < -2, the current brush color will not be changed. The default value is -1 (black.)

The brush pattern must be one of:

`SOLID`	(default)
`HOLLOW`	
`HORIZ`	
`VERT`	
`FDIAG`	
`BDIAG`	
`CROSS`	
`DIAGCROSS`	

The background color also refers to the current color mix table. If `backcolor` is < -2, the current background color will not be changed. The default background color is -2 (white.)

For example,

```
CALL TC_Graph_SetBrush (gid, -1, 3, "cross")
```

will not change the background color for the graphics object, will set the brush color to 3, and will set the brush to a crosshatch pattern.

(These conventions are identical to those of **TC_Win_SetBrush**.)

Exception: 845 Invalid brush pattern: ppppp

TC_Graph_SetDrawmode

```
CALL TC_Graph_SetDrawmode (gid, mode$)
```

The string values allowed for `mode$` are as follows; note that where there is a space in the string, there must be exactly one space:

`"COPY"`	ignore current contents, draw over anything that is there (default)
`"OR"`	perform bitwise OR between bit plans of each currently displayed pixel and the new pixel which is to overlay it
`"XOR"`	perform bitwise XOR between bit plans of each currently displayed pixel and the new pixel which is to overlay it
`"CLEAR"`	clear the screen to the extent covered by the item being drawn
`"NOT COPY"`	the bitwise negation of COPY
`"NOT OR"`	the bitwise negation of OR
`"NOT XOR"`	the bitwise negation of XOR
`"NOT CLEAR"`	the bitwise negation of CLEAR

The drawing mode determines how the graphical object's pen and brush interact with the background, including what has already been drawn. As an example, assume there are four bit planes (i.e., there are sixteen entries in the color map table), the background is color 6 (binary 0110), and the pen is color 10 (binary 1010). Then the above drawing modes would give the following results for each pixel covered by the pen:

`"COPY"`	Color 10 (binary 1010)
`"OR"`	Color 14 (binary 1110)
`"XOR"`	Color 12 (binary 1100)
`"CLEAR"`	Color 0 (binary 0000)
`"NOT COPY"`	Color 5 (binary 0101)
`"NOT OR"`	Color 1 (binary 0001)
`"NOT XOR"`	Color 3 (binary 0011)
`"NOT CLEAR"`	Color 15 (binary 1111)

Exception: 843 Invalid draw mode: mmmmm

Something similar may operate for a larger number of bit planes, and will sometimes yield strange-looking results. But the modes `"COPY"` and `"CLEAR"` should work as expected. And `"XOR"` should have the property that, if you draw the object twice, you will get back the original background.

Group Box Subroutines

TC_Groupbox_Create

```
CALL TC_Groupbox_Create (gbid, title$, xl, xr, yb, yt)
```

This subroutine draws a rectangular box, with title.

The four coordinates refer to the inside of the currently targeted (for output) physical window or, if in user coordinates, the current logical window.

The group box will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.

The group box is opaque, and should be created or shown before generating the output or showing the controls contained within it.

List Button Subroutines

TC_ListBtn_Create

```
CALL TC_ListBtn_Create (lbid, text$(), xl, xr, yb, yt)
```

This subroutine creates a list button. A list button appears as an ordinary push button but, when selected, reveals a scrollable list of items that can be selected.

The list `text$()` must have a subscript lower bound <= 1. The text of the button is the same as item one in the list.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If either `yb` or `yt` < 0, the default height of a list button will be used. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window. If `yb` or `yt` = -99999, the default height of a list button will be used.

The list button will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.

You can change the contents of a list button using

```
CALL TC_SetList (lbid, text$())
```

TC_ListBtn_Get

```
CALL TC_ListBtn_Get (lbid, selection)
```

This subroutine returns the number, starting with one, of the item currently showing in the list button. Note that the user may still be in the process of manipulating the list. It is a good idea to require the user to select another control, or to select some push button, to determine that the user has, in fact, made a selection from the list button. At this point, the list button could be erased.

TC_ListBtn_Set

```
CALL TC_ListBtn_Set (lbid, selection)
```

This subroutine sets the item, starting with one, initially highlighted in a list button.

List Edit Button Subroutines

TC_ListEdit_Create

```
   CALL TC_ListEdit_Create (leid, text$(), xl, xr, yb, yt)
```

This subroutine creates a list edit button. A list edit button appears as an ordinary list button but with two exceptions. First, it can be edited, like an edit field. Second, a scrollable list of choices appears when it is selected. Selecting one of the choices moves that choice to the editable field.

The items are given in the `text$()` list. In addition, the initial text of the list edit button is taken from `text$(0)`. Thus, the smallest subscript of `text$()` must be <=0.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If either `yb` or `yt` < 0, the default height of a list edit button will be used. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window. If `yb` or `yt` = -99999, the default height of a list edit button will be used.

The list edit button will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.

You can change the contents of a list edit button with

```
   CALL TC_SetList (leid, text$())
```

If there is a 0-th element, it will be used to set the text of the button itself.

Exception:	855	List Edit list subscript must start with 0.

TC_ListEdit_Get

```
   CALL TC_ListEdit_Get (leid, text$)
```

This subroutine finds out the text left by the user in the list edit button. Note that the user may still be in the process of editing the text. It is a good idea to require the user to select another control, or to select some push button, to determine that the user has, in fact, made a selection from the list button. At this point, the list button could be erased.

TC_ListEdit_Set

```
   CALL TC_ListEdit_Set (leid, text$)
```

This subroutine sets the text shown in a list edit field. The text may be changed at any time with this subroutine.

List Box Subroutines

TC_ListBox_Create

```
CALL TC_ListBox_Create (lbid, mode$, xl, xr, yb, yt)
```

This subroutine will present a selection list box. This list box is similar to others that appear in, for example, file open dialog boxes, but consists only of the selection list part. The list itself is scrollable, and must be set by **TC_SetList**.

The selection mode is given by `mode$`. Its possible values are:

`SINGLE`	Only single selections permitted (default)
`MULTIPLE`	Multiple selections permitted
`READONLY`	No selections permitted

You can set or change the list of a selection list box using:

```
CALL TC_SetList (lbid, text$())
```

Note that the calling sequence **TC_ListBox_Create** differs from the calling sequences for **TC_ListBtn_Create** and **TC_ListEdit_Create**. Here, a separate call to **TC_SetList** is required to set the list.

TC_ListBox_Set

```
CALL TC_ListBox_Set (lbid, selection)
```

This subroutine allows you to preselect a particular entry in the list. The value of selection must be in the range from 1 to the number of entries in the list.

Note: this subroutine is not to be confused with **TC_SetList**, which is used to specify the entries in the list box.

TC_ListBox_Clear

```
CALL TC_ListBox_Clear (lbid, selection)
```

This subroutine allows you to “unselect” an entry in the list that might have been selected previously using **TC_ListBox_Set**. The value of selection must be in the range of 1 to the number of entries in the list.

TC_ListBox_Get

```
CALL TC_ListBox_Get (lbid, selection())
```

This subroutine retrieves the selection(s) of the user. Since it is possible in certain systems to make multiple selections, this subroutine returns a numeric vector `selection()` whose entries are those selections. The number of selections is given by the upper bound of `selection()`. If only one selection is allowed, then `selection(1)` = the item selected. If no selection has yet been made then the selection list will have *no* entries.

Push Button Subroutines

TC_PushBtn_Create

```
CALL TC_PushBtn_Create (pbid, text$, xl, xr, yb, yt)
```

This routine creates a push button with the specified text at the location and of the size specified.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If either `yb` or `yt` < 0, the default height of a push button will be used. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window. If `yb` or `yt` = -99999, the default height of a push button will be used.

The push button will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.

If text$ ends with `"|LEFT"`, `"|CENTER"`, or `"|RIGHT"`, and the operating system permits, the push button text will be justified accordingly.

Push buttons created in this way cannot be outlined; that is, they cannot be activated by pressing the Return or Enter key. Selecting a push button with the mouse and releasing it causes the subroutine **TC_Event** to produce an event pair: CONTROL SELECT, CONTROL DESELECTED.

You can change the text of a push button at any time using

```
CALL TC_SetText (pbid, text$)
```

Radio Button Group Subroutines

TC_RadioGroup_Create

```
CALL TC_RadioGroup_Create (rgid, text$(), xl, xr, yb, yt)
```

This routine creates a group of radio buttons at the location specified. The number of buttons is determined from the size of the string list `text$()`, which also supplies the text that appears just to the right of each button. The smallest subscript of `text$()` must be one.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If either `yb` or `yt` < 0, the default height of a radio button will be used to determine the height of the group. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window. If `yb` or `yt` = -99999, the default height of a radio button will be used to determine the height of the group.

The radio group will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.

If text$(1) ends with `"|LEFT"`, `"|CENTER"`, or `"|RIGHT"`, and the operating system permits, the push button text will be justified accordingly.

A radio group has the property that no more than one button may be “on” at a time. Selecting any button to be “on” turns off any other button that happens to be on. Initially, no button is on.

Selecting a radio button with the mouse and releasing it causes the subroutine **TC_Event** to produce an event pair: CONTROL SELECT, CONTROL DESELECTED. At this point you can call the subroutine **TC_RadioGroup_On** to determine which button is now on.

Exceptions:	865	Radio button group list must start with 1.
	866	Radio button group must have at least one button.

TC_RadioGroup_SetText

You can change the text of a radio button at any time using

```
CALL TC_RadioGroup_SetText (rgid, button, newtext$)
```

TC_RadioGroup_On

```
CALL TC_RadioGroup_On (rgid, button)
```

This subroutine reports which radio button is on. The numbering refers to the original `text$()` list used to create the group. `Rgid` is the ID of the group as a whole. If no button happens to be on, button will be 0.

TC_RadioGroup_Set

```
CALL TC_RadioGroup_Set (rgid, button)
```

This subroutine turns on the button specified. If any other one is on, it will be turned off.

Scroll Bar Subroutines

TC_SBar_Create

```
CALL TC_SBar_Create (sbid, type$, xl, xr, yb, yt)
```

This subroutine creates a scroll bar of the type specified and at the location specified. The type must be either `"HSCROLL"` or `"VSCROLL"`, but the case doesn't matter.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If either xl or xr < 0, or if either `yb` or `yt` < 0, the default width or height of a scroll will be used. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window. If either xl or xr = -99999, or if either `yb` or `yt` = -99999, the default width or height of a scroll bar will be used.

The scroll bar will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.

The subroutine **TC_Event** processes all scroll bar actions, using the parameter and increment values specified by **TC_SBar_SetRange** and **TC_Sbar_SetIncrements**. The possible scroll bar events that can be returned by **TC_Event** are: UP, DOWN, PAGEUP, PAGEDOWN, END VSCROLL, END HSCROLL, LEFT, RIGHT, PAGELEFT, and PAGERIGHT, but it must be remembered that TC_Event has already taken care of adjusting the scroll bars. The manner in which these events interact with the visual scroll bar, and the values of the parameters and increments, is discussed later.

Note that scroll bars may be created and associated with windows, or created with text edit controls. Like ordinary scroll bars, they are handled automatically, and the events are reported as noted above. TC_Event also adjusts the text of a text edit control to correspond to the movement of the associated scroll bars.

Exception:	870	Scroll bar type must be VSCROLL or HSCROLL.

TC_SBar_SetPosition

```
CALL TC_SBar_SetPosition (sbid, position)
```

This subroutine sets the position of the slider (the thumb). The position is defined in terms of the scrollbar parameters `srange`, `erange`, and `prop`. If the `position` is <= `srange`, the slider will be moved to the top (left) of the scroll bar. If the `position` is >= `erange - prop`, the slider will be moved to the bottom (right) of the scroll bar. If the `position` is in between, the slider will be moved to the corresponding location. The default value is 0.

TC_SBar_GetPosition

```
CALL TC_SBar_GetPosition (sbid, position)
```

This subroutine finds out the current location of the scrollbar slider. The current position must be interpreted in terms of the scrollbar parameters `srange`, `erange`, and `prop`.

TC_SBar_SetRange

```
CALL TC_SBar_SetRange (sbid, srange, erange, prop)
```

This subroutine is used to set the parameters that specify the extreme positions of the slider, as well as the proportional size of the slider on those systems that provide for varying sized sliders.

These parameters are arbitrary. The relation to the position (and size of the slider on certain systems) is as follows: When the slider is at the top (left), its position is equal to `srange`. When the slider is at the bottom (right), its position is equal to `erange - prop`. On certain systems, the size of the slider relative to the size of the slider trough (the region in which the slider moves) is given by `prop/(erange - srange)`, but never greater than one. The default values are 0, 100, and 1, respectively.

For scroll bars associated with text edit controls, the subroutine **TC_Event** takes care of setting the parameters to match the length and width of the actual text.

TC_SBar_GetRange

```
CALL TC_SBar_GetRange (sbid, srange, erange, prop)
```

This subroutine finds out the current values of the scroll bar parameters.

TC_Sbar_SetIncrements

```
CALL TC_Sbar_SetIncrements (sbid, single, page)
```

This subroutine is used to set the scrollbar increments. The `single` increment defines how far the slider (thumb) moves when the user clicks in the up-arrow or down-arrow boxes (or left-arrow or right-arrow boxes) at the ends of the scroll bar. The `page` increment defines how far the slider moves when the user clicks in the gray area either above or below (left or right) of the slider. The default values are 1 and 10, respectively.

The actual movement of the slider is in terms of the parameters (see **TC_SBar_SetRange** for details.)

For scroll bars defined in connection with text edit controls, the subroutine **TC_Event** takes care of setting the increments.

It should be noted that the increments of a scroll bar are not used by the True BASIC subroutine **Object**. They are simply convenient storage locations for the programmer. However, the subroutine **TC_Event** *does* use these values to define scroll bar slider movements.

TC_SBar_GetIncrements

```
CALL TC_SBar_GetIncrements (sbid, single, page)
```

This subroutine finds out the current values of the increments.

Static Text Subroutines

TC_SText_Create

```
CALL TC_SText_Create (stid, text$, xl, xr, yb, yt)
```

This subroutine creates a static text field with initial text as given by `text$`.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If either `yb` or `yt` < 0, the default height of a static text box will be used. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window. If `yb` or `yt` = -99999, the default height of a static text box will be used.

The font is the "System" font. Its type, style, and size may be changed but not by True Controls.

If the text is longer than can be contained in the field, it will be truncated on the right.

Static text fields cannot generate any events.

You can change the text of a static text field at any time using

```
CALL TC_SetText (stid, text$)
```

If text$ ends with `"|LEFT"`, `"|CENTER"`, or `"|RIGHT"`, and the operating system permits, the push button text will be justified accordingly.

Text Editor Subroutines

TC_Txed_Create

```
CALL TC_Txed_Create (txid, options$, xl, xr, yb, yt)
```

This subroutine creates a text-editor control with the specified options and location.

If the units are pixels (TC_SetUnitsToPixels,) the four coordinates refer to the containing physical window. If the units are users (TC_SetUnitsToUsers,) the four coordinates refer to the window coordinates of the current logical window.

The text-edit control will not be shown if the show default is set to 0 or the physical window is not showing, but will be shown if the show default is 1 and the containing window is visible.

The `options$` string contains the options to be used. The particular options may be separated by spaces or vertical bars, and may be given in upper, lower, or mixed case. The options are:

`ATTACHED`	The text edit control will fill the physical window, and will be resized if the window is resized. Scroll bars, if any, must be associated with the window. See TC_Win_Create.
`READONLY`	The text can be read but not modified.
`WRAP`	The lines of the text are "wrapped" or "folded" at the margin of the text edit control.
`MARGIN ddd`	The margin is set to `ddd` pixels. (There must be one space between the word "MARGIN" and the digits.) This defines the width of the text, and the point at which the lines of the text are wrapped or folded. If the text is not wrapped, the margin is ignored. If ATTACHED, the margin is determined automatically.
`BORDER`	There will be a border around the text edit control. If the text edit control is ATTACHED to a physical window, an additional border will be superfluous.
`KEY EVENTS`	Keystroke events will be processed by the text edit and returned as TXE KEYPRESS events by **TC_Event**. If this option is not included, keystroke events will not be turned by **TC_Event**, except those generated by trap characters. By default, a text edit control created with this subroutine will return the event TXE KEYPRESS when the return or enter key is pressed; the value of `x1` returned by **TC_Event** will be the

	ASCII code of the key. Additional so-called trap characters may be set with the subroutine **TC_Txed_SetTrapChar**.
`MOUSE EVENTS`	Mouse events that occur within the text edit control will be acted on the by control and returned by **TC_Event** as TXE MOUSE events.
`VSCROLL`	A vertical scroll bar will be attached to the text edit control. This attribute is ignored if the attribute ATTACHED is present.
`HSCROLL`	A horizontal scroll bar will be attached to the text edit control. This attribute is ignored if the attribute ATTACHED is present.

The text portion of a text edit control must be supplied by a subsequent call to **TC_Txed_ReadTextFromFile** or **TC_Txed_ReadTextFromArray**.

TC_Txed_ReadTextFromFile
TC_Txed_WriteTextToFile
TC_Txed_ReadTextFromArray
TC_Txed_WriteTextToArray

These first two routines can be used to read and write the text edit text from and to a file, which is assumed to be in a "flat file" format with the system end-of-line character at the end of each paragraph. The calling sequences are"

```
CALL TC_Txed_ReadTextFromFile (txid, filename$)
CALL TC_Txed_WriteTextToFile (txid, filename$)
```

The latter routine will overwrite the previous contents of the file.

The last two routines can be used to read and write the text edit text from and to a string array. Their calling sequences are:

```
CALL TC_Txed_ReadTextFromArray (txid, list$())
CALL TC_Txed_WriteTextToArray (txid, list$())
```

In the second routine, the string list will be dimensioned to the exact length needed.

TC_Txed_SetText

```
CALL TC_Txed_SetText (txid, text$)
```

This routine supplies the entire text of a text edit control. The text is a flat file in the same format as it might be stored on disk.

You can append additional text lines to the text by using **TC_Txed_Append**.

TC_Txed_GetText

```
CALL TC_Txed_GetText (txid, text$)
```

This subroutine retrieves the text from a text edit control, including all user modifications up to that point. The text is a flat file in the same format as it might be stored on disk.

TC_Txed_Append

```
CALL TC_Txed_Append (txid, text$, revealflag)
```

This subroutine will append the text supplied to the end of the text in the text edit control. The text must consist of a single paragraph without an end-of-line. (With unwrapped text, a paragraph is the same as a line.)

If revealflag = 0, the text will not be scrolled; thus, the new line may be out of view. If revealflag = 1, the cursor will be set to the end of the last paragraph, thus forcing the new line to be visible.

TC_Txed_SetFont

```
CALL TC_Txed_SetFont (txid, fontname$, fontsize, fontstyle$)
```

This subroutine changes the characteristics of the font used in the text edit control. If the `fontname$` or the `fontstyle$` is the null string, that characteristic will not be changed. If the `fontsize` is <= 0, the font size will not be changed. Case doesn't matter in the specification of the font name or font style.

The font name must be a legal font name. The available fonts will differ from system to system, but will always include: "Helvetica", "Fixed", "Times", and "System". The Fixed font is a fixed width font that might be a Courier or similar font. The System font varies depending on the operating system. The default font is Helvetica. You may learn about other available fonts by calling TC_FontsAvailable.

The font size is given in points; one point is approximately 1/72 of an inch. The font size is not necessarily related to the size of the font as displayed on the screen, or to any particular number of pixels. The default size is 10 points. If the System font is being used, it may not be possible change its size.

The font style must be one of "Plain", "Bold", "Italic", or "Bold Italic". The default style is Plain. If the System font is being used, not all styles may be available.

TC_Txed_SetColor

```
CALL TC_Txed_SetColor (txid, forecolor, backcolor, bordercolor)
```

This subroutine allows changing the colors associated with a text edit control. The color numbers refer to entries in the color mix table. The default colors for forecolor and border color are black (-1), and the default color for backcolor is white (-2.)

TC_Txed_SetTrapChar

```
TC_Txed_SetTrapChar (txid, char, action)
```

This subroutine sets additional so-called "trap" characters. Such characters are returned by **TC_Event** as TXE KEYPRESS events even if KEY EVENTS had not been chosen as an option in **TC_Txed_Create**.

`Char` is the number of the key to be trapped, and `action` is what is to happen when that key is pressed by the user. Possible actions are:

1	The key code is returned as a TXE KEYPRESS event. The text edit control *is* suspended. The key is *not* absorbed by the text edit control.
2	The key code is returned as a TXE KEYPRESS event. The text edit control is *not* suspended. The key *is* absorbed by the text edit control.
3	Exactly like stop code 1, but will be treated as an ordinary character *unless* there is selected text.
< 0	The particular key code is *unregistered*.

By default, the Return or Enter key is established as a type 2 trap character.

TC_Txed_Resume

```
CALL TC_Txed_Resume (txid)
```

This routine should be used to "resume" the activity of a text edit control that has been suspended. A text edit control can be suspended if the user presses a "trap character" of type 1 or type 3.

TC_Txed_Suspend

```
CALL TC_Txed_Suspend (txid)
```

This routine can be used to "suspend" the activity of a text edit control. One use may be to reduce flicker caused by appending more than one line to the text in the control. A suspended text edit control can be made active by calling TC_Txed_Resume.

TC_Txed_Cut

```
CALL TC_Txed_Cut (txid)
```

This subroutine “cuts” or removes text that has been selected, and places the text onto the system clipboard, erasing the prior contents of the clipboard. If no text has been selected, no action takes place, and the prior contents of the system clipboard remain.

TC_Txed_Copy

```
CALL TC_Txed_Copy (txid)
```

This subroutine “copies” text that has been selected, placing it onto the system clipboard and erasing the prior contents of the clipboard. If no text has been selected, no action takes place, and the prior contents of the system clipboard remain.

TC_Txed_Paste

```
CALL TC_Txed_Paste (txid)
```

This subroutine “pastes” the current contents of the system clipboard into the text of the text edit control. If there is selected text, the clipboard contents replace that text, and the selected text disappears. If no text has been selected, the clipboard contents are inserted at the current cursor location of the text edit text. The contents of the clipboard remain intact.

TC_Txed_SetCutCopyPaste

```
CALL TC_Txed_SetCutCopyPaste (txid, xmenu, xitem, cmenu, citem, pmenu, pitem)
```

If the text edit control is “attached” to the containing window, and the window is equipped with menus that include the Cut, Copy, and Paste possibilities, calling this subroutine allows **TC_Event** to carry out those particular functions directly. You specify the menu and item numbers of the three menu selections.

If you establish these menu associations, you should not also call **TC_Txed_Cut**, **TC_Txed_Copy**, or **TC_Txed_Paste** as **TC_Event** will carry out these functions.

TC_Txed_Find

```
CALL TC_Txed_Find (txid, case, word, key$, p, l1, c1, l2, c2, found)
```

This subroutine “finds” certain text. The search string is provided in `key$`. If `case` = 1, the search will be case-sensitive; that is, case (upper or lower) will be taken into account. If `case` = 0, case will be ignored in the search. If `word` = 1, only whole word matches be accepted. If `word` = 0, finding the search string within another word will be allowed. If the search is successful, the returned value of `found` will be 1; if not successful, it will be 0.

At the call, the values of `p, l1`, and `c1` specify the paragraph, line, and character in the line for the commencement of the search. At the return, if `found` = 1, these parameters give the paragraph, line, and character with the line of the first character of the match; `l2` and `c2` give the line and character of the last character of the match, which must be contained in a single paragraph. If `found` = 0, these parameters are not changed.

Note: paragraph, line, and character numbering start with 0, not 1. Paragraphs are text strings that end with an EOL when stored in a file. If the text is not wrapped, paragraphs and lines are the same, whereas if the text is wrapped, a paragraph may contain several lines. Finally, the lines will depend on the current margin; thus, if a text edit control that uses wrapped text is resized, then the lines will change, while the paragraphs will not.

TC_Txed_SetSelection

```
CALL TC_Txed_SetSelection (txid, p1, l1, c1, p2, l2, c2)
```

This subroutine “selects” text. The selected text will start at paragraph `p1`, line `l1`, character `c1`, and extend to paragraph `p2`, line `l2`, character `c2`, inclusive. This can be done following a successful “find” operation to show the user the location of the found text.

Remember that paragraph, line, and character counting start at 0, not 1.

TC_Txed_GetSelection

```
CALL TC_Txed_GetSelection (txid, p1, l1, c1, p2, l2, c2)
```

This subroutine allows you to identify text that may have been selected by the user. The selected text starts as paragraph p1, line l1, character c1, and ends at paragraph p2, line l2, character c2, inclusive. If no text has been selected, then p2 = p1, l2 = l1, and c2 = c1, where p1, l1, c1 is the current location of the cursor (i.e., the cursor is immediately in front of characters c1.)

Remember that paragraph, line, and character counting start at 0, not 1.

TC_Txed_SetCursor

```
TC_Txed_SetCursor (txid, p, l, c)
```

This subroutine allows you to set the cursor to any desired position. The cursor will be set to just in front of character *c*, in line *l* of paragraph *p*. This routine merely calls TC_Txed_SetSelection with the starting and ending paragraph, line, and character the same. That is, it “deselects” any text that may have been selected.

Remember that paragraph, line, and character counting start at 0, not 1.

TC_Txed_GetCursor

```
TC_Txed_GetCursor (txid, p, l, c)
```

This subroutine allows you to locate the current position of the cursor. The cursor will be just in front of character *c* in line *l*, paragraph *p*. This routine merely calls TC_Txed_GetSelection and returns the *starting* position of any selected text. The presence of selected text or the position of the cursor is not changed.

Remember that paragraph, line, and character counting start at 0, not 1.

TC_Txed_SetMargin

```
TC_Txed_SetMargin (txid, margin)
```

This subroutine can be used to set the margin for a text edit control. The margin must be specified in pixels. If margin <0, the margin will be set to the actual width of the test edit control.

Setting the margin has effect only if the WRAP option is in effect. The margin is set automatically for text edit controls ATTACHED to windows.

True Dials

True Dials is a library of subroutines that give easy access to the True BASIC built-in subroutine **TBD**. (See Chapter 21 "TBD Subroutine" for details on it.) These subroutines are contained in a library file TRUEDIAL.TRC.

These routines are all higher level calls to the built-in **TBD** subroutine. For a better understanding of how they work, see the source code of this library: TRUEDIAL.TRU.

The dialog boxes displayed by **TBD** are ***modal*** dialog boxes. That is, no other activity (such as menu selection) is permitted until the user has complete the use of the dialog box or it has timed out.

The subroutines have several features in common. Two of the subroutines (TC_Message and TC_InputM) allow you to specify a title, except on the Macintosh platform where titles are not available. This will appear at the top of the dialog box. Except for file dialog boxes, a message can be specified. If the message string `message$` contains vertical bars "`|`", they will be taken as line breaks. Messages of up to ten lines can be displayed.

The text of the buttons to show in the lower portion of the dialog box are specified in a single string, separated by vertical bars. For example, to display three buttons with text "Yes", "No", and "Cancel", use the button string `"Yes|No|Cancel"`. From one to four buttons are required.

The number of the button the user selected is returned in `result`. Which of the buttons is initially outlined can be specified by the value of `default`.

Finally, a timeout limit can be set. If the user does not select one of the buttons, or does not press the Return key to activate the outlined button, before the time specified elapses, the dialog box will close with `result` = 0 if none of the buttons is outlined, or the number of the outlined button. The length of the timeout is expressed in seconds. If the `timeout` is zero, *no* timeout will occur.

Several errors can arise if the following subroutines are improperly specified or located. They are described at the end of Chapter 21 "TBD Subroutine" and are not repeated here. No additional errors are generated by misuse of the following subroutines.

TD_SetLocation

```
CALL TD_SetLocation (xloc, yloc)
```

This Dialog boxes (except for **TB_GetFile** and **TD_SaveFile**) are ordinarily centered in the currently-targeted physical window. By calling this routine, you can position the upper-left corner of a dialog box at the pixel screen coordinates (xloc, yloc). To return to the default positioning, use -1 for xloc and yloc.

If you require control over the size of your dialog boxes, in addition to the location of the upper-left corner, use the TBDX subroutine directly.

TD_Warn

```
CALL TD_Warn (message$, button$, default, result)
```

This subroutine displays a (warning) message. To query the user about whether to continue or cancel an operation, you might use

```
CALL TD_Warn ("What is your pleasure?", "Continue|Cancel", 1, result)
```

When the dialog box is displayed initially, the "Continue" button, button number 1, will be outlined. The value of `result` tells which button the user actually selected.

TD_Message

```
CALL TD_Message (title$, message$, button$, default, result)
```

This subroutine displays a dialog box similar to **TD_Warn** except that a title can be included. The operation is the same as with **TD_Warn**. For example:

```
LET title$ = "Report from the Boss"
LET message$ = "When are you going back to work?"
LET button$ = "Now|Later|Never"
CALL TD_Message (title$, message$, button$, 2, result)
```

The above code displays a titled dialog box with a one-line message, along with three buttons from which the user can choose. The second button "Later" will be outlined initially. (The title is not available on the Macintosh.)

TD_YN

```
CALL TD_YN (message$, default, result)
```

This subroutine displays a message with two buttons: "Yes" and "No". In all other respects it is like **TD_Warn**. In fact, the calling sequence above displays a dialog box identical to that displayed by:

```
CALL TD_Warn (message$, "Yes|No", default, result)
```

TD_YNC

```
CALL TD_YNC (message$, default, result)
```

This subroutine is a slight extension of **TD_YN** in that there are three buttons: "Yes", "No", and "Cancel". The calling sequence above displays a dialog box identical to that displayed by:

```
CALL TD_Warn (message$, "Yes|No|Cancel", default, result)
```

TD_LineInput

```
CALL TD_LineInput (message$, text$)
```

This dialog box can be used to input a single line of text. The user can edit this text using the usual methods: selecting the position of the cursor using the mouse, typing characters, and using the left and right cursor keys. For example:

```
CALL TD_LineInput ("Enter your name", intext)
```

displays a dialog box with a one-line message, and a boxed editable field. It will have a single button "Ok". Upon return, what the user typed is returned in the string variable `intext$`.

TD_Input

```
CALL TD_Input (message$, buttons$, text$, default, result)
```

This dialog box can be used to input a single line of text. The user can edit this text using the usual methods: selecting the position of the cursor using the mouse, typing characters, and using the left and right cursor keys. For example:

```
CALL TD_Input ("Enter your name", "Done|Cancel", intext$, 1, result)
```

displays a dialog box with a one-line message, with a boxed editable field, and with two buttons. Upon return, what the user typed is returned in the string variable `intext$`.

TD_InputM

```
CALL TC_InputM (title$, message$, buttons$, labels$(), text$(), highlight,
   default, result)
```

This subroutine displays an input box with multiple editable lines. This dialog box can have a title bar and a title much like a window. (The title is not available on the Macintosh.) In addition, the editable lines can have labels to the left of each line. For example:

```
LET title$ = "Data Entry Box"
LET message$ = "Enter your name."
LET button$ = "Ok|Cancel"
DIM labels$(3), text$(3)
```

```
MAT READ First name, Middle name, Last name
CALL TD_InputM (title$, message$, button$, labels$(), text$(), 1, 2, result)
```

displays a titled dialog box with the title bar displaying “Data Entry Box”, with an inside message “Enter your name.”, and with three editable fields. These boxes will be labeled on the left. There will be two buttons, “Ok” and “Cancel”. The first editable field will be highlighted, and the second (“Cancel”) will be outlined.

The number of editable text fields displayed is the larger of the sizes of the labels array and the initial text array, except that no more than ten fields will be displayed. Upon return, the text array will have a size consistent with the number of fields.

TD_GetFile

```
CALL TD_GetFile (extension$, filename$, changedir)
```

This subroutine displays a typical File Open dialog box. The user may change directories while looking for the file; if `changedir` = 0, the current directory will not be changed, and the full path name of the file will be returned in `filename$`. If `changedir` = 1, the current directory will be changed, and only the local name of the file will be returned.

On Windows and similar platforms, if `extension$` is a valid file extension(such as “tru”,) then only matching files will be displayed; if `extension$` is the null string, all files will be displayed. On the Macintosh, `extension$` can be a file type, such as TEXT or TEXTTRUE; there is no way to limit the file names based on a possible extension to the file names.

Two buttons are displayed, “Ok” and “Cancel”. If “Ok” is clicked and no selection has been made, the dialog box stays on the screen. If “Cancel” is clicked, a null string is returned in `filename$` whether or not any name has been selected.

This dialog box cannot be timed out.

Note: this subroutine does NOT actually open the file; that is up to the programmer.

For example, on Windows:

```
CALL TD_GetFile ("tru", filename$, 1)
```

presents only file names having the extension `".tru"`, but will allow the user to change the current directory. Only the local name of the file will be returned in `filename$`.

TD_SaveFile

```
CALL TD_SaveFile (extension$, filename$)
```

This subroutine displays a typical Save File dialog box.

On Windows and similar platforms, if `extension$` is a valid file extension(such as “tru”,) then only matching files will be displayed; if `extension$` is the null string, all files will be displayed. On the Macintosh, `extension$` can be a file type, such as TEXT or TEXTTRUE; there is no way to limit the file names based on a possible extension to the file names.

The initial value of `filename$` will appear as the suggested file name. Upon return, `filename$` will contain the full pathname of the file name selected by the user. The current directory is not changed.

Two buttons are displayed, “Ok” and “Cancel”. If “Ok” is clicked but the file name box is empty, the dialog box stays on the screen. If “Cancel” is clicked, a null string is returned in `filename$` whether or not any name appears in the file name box.

This dialog box cannot be timed out.

Note: this subroutine does NOT actually save the file; that is up to the programmer.

For example, on Windows:

```
LET filename$ = "MyFile.tru"
```

```
CALL TD_SaveFile ("tru", filename$)
```

will present a Save File dialog box displaying only file names with the extension `".tru"`, and with `"MyFile.tru"` as the suggested name of the file to be saved. Upon return, `filename$` will contain the full path name of the name actually selected by the user.

TD_List

```
CALL TD_List (message$, button$, list$(), choice, default, result)
```

This subroutine presents a selection list box. The `message$` appears near the top of the box. The list of names is provided in a string list `list$()`. The particular name to be highlighted initially is specified in `choice`. If `choice` is < 1, then the first item is highlighted; if `choice` is > the number of items, then none of the items is highlighted. There may be up to four buttons specified in `button$` with text for each button separated by vertical bars (|). `Default` specifies which one, if any, is to be highlighted initially. Upon return, `choice` contains the number of the selected entry, while `result` contains the number of the selected button.

If a timeout occurs before the use has selected a button, the number of the highlighted button is returned; this may or may not be the same as the default button. The highlighted item number will be returned in `choice`; if none is highlighted, 0 is returned.

TD_SetTimeout

```
CALL TD_SetTimeout (timeout)
```

This routine sets the timeout, in seconds, for all subsequent uses of the True Dials subroutines except for TD_GetFile and TD_SaveFile. If the argument is 0, no timeout will occur. For example:

```
CALL TD_SetTimeout (10)
```

sets the timeout to be 10 seconds, while:

```
CALL TD_SetTimeout (0)
```

will prevent any timeout from happening.

TD_GetTimeout

```
CALL TD_GetTimeout (timeout)
```

This subroutine finds out what the current timeout value is.

TD_SetDelimiter

```
CALL TD_SetDelimiter (demin$)
```

This subroutine changes the delimiter used internally to break up lines for TD_Warn, TD_Message, TD_Input, TD_InputM, etc. (i.e., dialog boxes of type 1.) The default value of the delimiter is the vertical bar `"|"`. If your messages must include the vertical bar, of if you expect the user response to TD_Input or TD_InputM to include vertical bars, you can use this routine to change the delimiter to a neutral value.

TD_AskDelimiter

```
CALL TD_AskDelimiter (delim$)
```

This subroutine returns the current delimiter (default is the vertical bar `"|"`) used internally to break up lines for TD_Warn, TD_Message, TD_Input, TD_InputM, etc.

ExecLib

ExecLib is a library of subroutines that provide access to directory information. The subroutines are in the library file EXECLIB.TRC. The names of the subroutines in this library all begin with “EXEC_”.

These routines are all higher level calls to the built-in **System** subroutine. For a better understanding of how they work, see the source code of this library in EXECLIB.TRU.

Several errors can arise if the following routines are improperly used. These errors are of two types. Errors identified by the system subroutine **System** are outlined where that subroutine is described in Chapter 18. Several other errors are detected by the ExecLib subroutines; these are included below.

Exec_AskDir

```
CALL Exec_AskDir (dirname$)
```

This subroutine returns the full path name of the current directory.

Exec_ChDir

```
CALL Exec_ChDir (newdir$)
```

This subroutine changes from the current directory to the new directory specified. You may specify the new directory relatively in terms of the current directory, or absolutely by providing the full pathname of the new directory. If the new directory is either invalid or does not exist, then an error will occur.

Exception:	895	Bad new directory in Exec_ChDir: nnnnn

Exec_DiskSpace

```
CALL Exec_DiskSpace (used,free)
```

This subroutine returns the amount of disk space in use, and the amount still free. Units are in bytes.

Exec_MkDir

```
CALL Exec_MkDir (newdir$)
```

This subroutine creates a new directory. You may specify either the full path name of the new directory, or give it relatively in terms of the current directory. If the directory you specify already exists, or the name is invalid, an error will occur.

Exception:	896	Bad new directory in Exec_MkDir: nnnnn

Exec_RmDir

```
CALL Exec_RmDir (baddir$)
```

This subroutine removes (deletes) the directory named. On some operating systems, this cannot be done until the directory is emptied of contents. If the directory does not exist, or is invalid, an error occurs.

Exception:	897	Can’t remove directory in Exec_RmDir: ddddd

Exec_ReadDir

```
CALL Exec_ReadDir (template$, name$(), size(), dlm$(), tlm$(), type$(), vname$)
```

This subroutine returns a list of names of the files and directories in the current directory. The names in the string list `name$()` will be the short names, not the full path names. The sizes of the files are given in bytes. The dates-last-modified and times-last-modified are given in the format of the True BASIC **DATE$** and **TIME$** functions. For example, the date might be “19950201” and the time might be “14:22:07” if it is 2:22 and 7 seconds PM on February 1, 1995.

The template is specified in a standard form across platforms. For example, "*.tru" will yield file names whose extensions are ".tru"; note that the "*" is a "wild card" that matches anything.

The type consists of a four-character string of the form "drwx". In each position, either the letter or a hyphen "-" will appear. If the "d" is present, the file is actually a directory; if a hyphen appears in that position, it is a true file. If the "r" is present, reading the file is allowed; if a hyphen appears in that position, the file cannot be read. If the "w" is present, the file can be written to or appended to; if a hyphen appears in that position, the file cannot be written. Finally, if an "x" appears, the file can be executed (i.e., run as a free-standing program.) If a hyphen appears in the last position, the file cannot be directly executed. If the "x" appears but the file does not contain a free-standing program, an error of some sort will occur.

`Vname$` will simply contain the name of the current directory.

Exec_ClimbDir

```
CALL Exec_ClimbDir (dir$, template$, name$(), size(), dlm$(), tlm$(), type$())
```

This subroutine is similar to Exec_ReadDir with these exceptions:

First, a list of file and directory names contained in the directory specified in the first argument, and all subdirectories, will be given.

Second, the full path names of all files and directories are given.

Exec_Rename

```
CALL Exec_Rename (oldname$, newname$)
```

This subroutine is the only one that deals with files rather than directories. It allows you to rename a file without having to copy the file. If the old file does not exist or is in an invalid format, or if the new file already exists or is in an invalid format, an error will occur.

Exception:	897	Bad old or new name in Exec_Rename: ooooo, nnnnn

Exec_SetDate

```
CALL Exec_SetDate (new_date$)
```

This subroutine can be used to set the computer's current date. The format is "YYYYMMDD". An exception occurs for an invalid format.

Exec_SetTime

```
CALL Exec_SetTime (new_time$)
```

This subroutine can be used to set the computer's current time. The format is "HH:MM:SS". An exception occurs for an invalid format.

CommLib

CommLib is a library of subroutines that provide access to the serial ports. They allow you to interface True BASIC to anything you can hook up to the serial port — printers, modems, lab instruments, or bulletin boards. The subroutines are contained in the file COMMLIB.TRC. The source code for this library file is in COMMLIB.TRU. (For past users of True BASIC, the files COMLIB.TRC and its source code companion COMLIB.TRU contain the same suboutines but with the traditional names.)

Getting Started

This library supports asynchronous, RS-232 communications using the asynchronous communications adaptor or equivalent. Both input and output are buffered and interrupt-driven, and the routines can support one or two ports at speeds up to 38,400 baud, and sometimes higher. There is optional flow control for input and output, and full control of the modem signals, parity, byte length, and stop bits.

Buffering

If the speed of the communications line is 1200 baud, the number of bytes you can get across it in a second is 1/10th of that, or 120 characters. If the other side is sending at full speed, you'll receive a character every 1/120th of a second, ready or not. That's probably enough time to save the character in a string, but not too much more — if your program ever blinks, if it takes a half-second out to read the disk, you'll just *lose* 60 characters.

That's where buffering can help you. This library sets up a separate process (an "interrupt handler") to watch the communication line no matter what your program is doing. If a character comes in, it saves it (in a *buffer*) until the next time the program's ready to read it. It'll save up to 20480 bytes, so at 1200 baud, your program could "blink" for more than 10 seconds, instead of only 1/120th sec.

Even at 19,200 baud, there's enough room for a full second's worth of data. The *output* buffer is smaller — 10240 bytes on the Macintosh, system-dependent on Windows and OS/2 — but it's still usually enough to keep the line running at full speed.

A Short Example

Here's a simple program that will make your PC act like a dumb terminal:

```
library "Commlib.trc"
call Com_Open (#1, 1, 14400, "") ! Open comm line at 14400 baud
call Com_SendCR ("ATD 6436300")  ! Dial up the computer

do
   call Com_Receive (s$)         ! get any input from computer
   call Output (s$)              ! routine to print it on screen
   if key input then             ! get anything the user's typed
      get key k
      select case k
      case 323                   ! F9 = end session
         stop
      case 315                   ! F1 = break
         call Com_SendBreak
      case else                  ! else send to the computer
         call Com_Send (Chr$(k))
      end select
   end if
loop

SUB Output (s$)                  ! Handle CR & LF characters
```

```
    do                           ! first strip all CRs
       let i = Pos(s$,Chr$(13))  ! find first CR
       if i = 0 then exit do     ! none = all done
       let s$[i:i] = ""          ! remove the first
    loop

    do                           ! now end line on line-feed
       let i = Pos(s$,Chr$(10))  ! find next line-feed
       if i = 0 then exit do
       print s$[1:i-1]           ! print each separate line
       let s$ = s$[i+1:maxnum]   ! remove that line
    loop

    print s$;                    ! print partial line

 end sub

 end
```

How the program works

The program uses routines to access the communications line. All of them are from the library COMMLIB.TRC which is the compiled version of the communications library. Here's what they do:

```
 CALL Com_Open (#1, 1, 14400, "")
```

Starts up the communication routines. You must call it first. #1 is a dummy file, that'll be associated with the line. *Don't* try to do normal file I/O with it — things like READ or PRINT. The only thing you can do with it is close it, which shuts down the buffering. The "1" is which communication port to use (in case you have two). "14400" is the baud rate, and the last argument is a string of options.

```
 CALL Com_SendCR (""ATD 6436300"")
```

This sends the string argument out over the communication line followed by a carriage-return. In this case, the string is just a dial-up command for an auto-dial modem. Com_SendCR is a special case of Com_Send.

```
 CALL Com_Receive (s$)
```

Com_Receive sets its argument to a string containing all characters that have come in from the other end since the last time you called Com_Receive. If nothing new has come in, it will return the null string. That will usually be the case, in this program.

When things are idle, the program will sit looping, alternately checking "key input" and calling Com_Receive. In the meantime, the program will harmlessly print the null string (without new-line) repeatedly on the screen.

```
 CALL Com_Send (Chr$(k))
```

This sends the key the user typed to the "other end" (the modem, or whatever the line is hooked up to). The string argument can be as long as you want.

```
 CALL Com_SendBreak
```

A *break* isn't a character, and can't be sent with the normal Com_Send routine. Com_SendBreak transmits the same signal over the line as the break key on most terminals.

The rest of the program (the subroutine Output) is there to treat carriage-returns and line-feeds more like a terminal does.

There's a complete technical description of all the routines at the end of this section. The routines let you get at almost every feature of the communications hardware, mostly through the option string for Com_open. There are quite a lot of bells and whistles. If you really need to, for example, you can support the 75 baud 5-bit code that AP and UPI wire services use.

Options and Controls

There are lots of controls and options available. If you're very knowledgeable about communications, you may want to skip to the technical descriptions of the library routines in the back of this manual. If you're not already an expert, it can be difficult to know what to pay attention to and what to ignore.

Here's a rough road map: if you're using the communication line to transfer binary files, you should know about parity and number of data bits. If you're doing file transfers *from* the PC, you should read about XOFF / XON flow control. You probably won't need to pay attention to modem control and status signals unless you already know about them.

Parity, Data Bits, and Stop Bits

These can be set when you open the line, or changed in the middle of things with *Com_control*. Virtually all communication uses 8 bit characters, with one start and one stop bit (10 bits total, which is why the baud rate is 10 times the number of characters per second). Having two stop bits will slow things down slightly, but otherwise won't make a difference. Only old 10 cps teletypes require 2 stop bits.

To get a total of 8 bits in each character, you should either have 8 data bits and no parity ("D8 P–" in the options), or 7 data bits and a parity bit. In general, it's better to use 7 bits and "even" or "space" parity *unless* you're doing binary file transfers, in which case you should use 8 bits and no parity. If you are receiving normal text, you should avoid 8-bit no-parity mode. Otherwise, the parity bits may turn

innocent letters into strange hieroglyphics (the characters above 128 in the IBM character set). Note also that *Receive_line* won't recognize a carriage-return with the parity bit set if you've requested 8 data bits.

XOFF / XON Flow Control

Flow control is a way for one end to make the other stop sending temporarily if it's getting behind. The side that's getting behind sends an XOFF character (control-S), and the other side stops sending. When it's caught up again, the one that sent the XOFF sends an XON (control-Q), and the other side resumes sending. This is a pretty common convention, but not universal.

The SXOFF option will make the communication routines automatically send an XOFF if its input buffer is nearly full (within 256 bytes). It's on by default. The RXOFF option makes output stop whenever an XOFF is received. It's off by default. If you're sending files *from* the PC, you probably want RXOFF on, so you won't send

too much too fast. If you're receiving binary files, you probably want RXOFF off, so that you can receive the XOFF character just like any other. These options can be set when you call *Com_open*.

Modem Control and Status

The control signals (RTS and DTR) can be set by Com_control or Com_open, and you can check the status lines (CTS, DSR, RI and RLSD) with Com_status. If you want to, you can find out whether your modem is really hooked up or whether it has detected a carrier signal. You almost never need to pay attention to these. Often the wires aren't even connected. CTS and RTS are for half-duplex lines, which aren't often seen nowadays.

Hardware Requirements

To use the communication library, you'll need a "serial port." You can use either serial port, or both at the same time, with this library.

Summary of COMMLIB

The following routines can all be found in the library COMMLIB.TRC on your communications disk. The routines are described in more detail in the following pages. First, the subroutines:

`Com_Open (#1, port, speed, opt$)`	Opens communication line.
`Com_Switch (p)`	Switches to port *p*.
`Com_Control (opt$)`	Resets options and modem signals.
`Com_Send (s$)`	Sends *s$* (starts sending).
`Com_SendLine (line$)`	Sends *line$* followed by CR/LF.
`Com_SendCR (line$)`	Sends *line$* followed by CR.
`Com_SendBreak`	Sends a break.
`Com_Receive (buf$)`	Gets all bytes received since last call.
`Com_ReceiveLine (line$)`	Gets the next line, up to a CR.
`Com_WaitLine (wtime, f, l$)`	Like Com_ReceiveLine, with timeout.
`Com_WaitPrompt (p$, wtime, f, s$)`	Waits for specified prompt, with timeout.

There are also two numeric functions that give the current status of the communication line:

`Com_Buf (type)`	Returns buffer space, in bytes, according to *type:* 0 — bytes waiting to be sent 1 — free space in output buffer 2 — bytes in input buffer 3 — free space in input buffer.
`Com_Status (type$)`	Returns line status; *type$* may be DSR, CTS, RLSD, RI, ERR, RXOFF, SXOFF, or CR.

Com_Open

```
CALL  Com_Open (#1, port, speed, options$)
```

Before you use any other communication library routine, you have to use Com_open to tell the system what port you're using, what speed to run, and a host of other details, like whether to set Data Terminal Ready, that you usually needn't worry about.

The file number you pass isn't used by the other routines. The only time you'll need it again is when you close the file. You simply use the CLOSE statement to close the communications port. Remember that local files (files used inside external subroutines or functions that weren't passed as parameters) are automatically closed when that routine returns.

You *can't* use any normal True BASIC file operations (READ, PRINT, etc.) on that file number, except for the CLOSE statement.

Port is the number of the communications port, and can be from 1 to n, where n is the number of available ports. Note that 1 can stand for what DOS calls COM1 and so on.

Speed is the number of bits per second to send across the line, also known as the *baud rate*. The machine on the other side must also be using the same speed. Usually the speed should be 14400 or 28800 if you're using a modem. If the wire from the PC is plugged directly into another piece of equipment, you may be able to use an even higher rate.

If you pass a zero, the baud rate won't be changed.

Options$ is a string containing additional parameters, separated by spaces. See *Com_control* for a full explanation of the parameters. The default setting will work well in most cases. If you just use the null string, the line will be set up for 7-bit characters, even parity, and one stop bit. Data Terminal Ready and Request To Send will both be set, and the receiver will be programmed to send an XOFF character when its buffer is nearly full but not to respond to one from the other end. This corresponds to an option string of "D7 PE S1 DTR RTS SXOFF RXOFF–."

Exceptions:	9003	No such file.
	7003	Channel is already open.
	7001	Channel number must be 1 to 1000.
	960	Unknown communication option.

CLOSE

```
CLOSE #1
```

Instead of calling a subroutine to close the communications port, simply use the True BASIC CLOSE statement. The port will be closed automatically when your program stops, or when the subroutine owning the file returns. The modem control signals (DTR & RTS) aren't changed, so closing the port doesn't force disconnection; you should be careful to disconnect cleanly.

Com_Switch

```
CALL Com_Switch (port)
```

If you are using more than one port, Com_switch will switch between them. All the operations (other than open and close) will apply to the last port number used with Com_switch or Com_open. The port number you use as the argument should be 1, 2, etc. It is *not* the file number you passed to Com_open, but rather the port number — `Com_open`'s second argument.

Some things to keep in mind when using both ports:

You can't use the same file number for both ports. You don't need the file number for any purpose other than closing the port at the end, but the file must stay open while you're using it. If you call Com_open from a subroutine, the port will be closed when you leave the subroutine unless the channel number was a parameter.

There is a 20480 byte input buffer for each port, and Com_buf(3) gives back the same number for both. There are also separate output buffers, however, each with a system-dependent size.

Example:

```
! Copy port 1 to port 2, changing speed

call Com_Open (#5,2,1200,"d8 rxoff sxoff")
call Com_Open (#6,1,300,"d8 rxoff sxoff")

do until key input
     call Com_Receive (x$)
     call Com_Switch (2)
     call Com_Send (x$)
     call Com_Switch (1)
loop
```

Exceptions:	7004	Channel isn't open.

Com_Control

```
SUB Com_Control (options$)
```

Once a port has been opened, you can change the options by calling Com_control. These options have exactly the same meaning as they do for Com_open. The string *options$* can contain any number of these separated by spaces, in either upper or lower case.

DTR	Turn on Data Terminal Ready.
DTR–	Turn off Data Terminal Ready.
RTS	Turn on Request To Send.
RTS–	Turn off Request To Send.
RXOFF	Turn on input flow control.
RXOFF–	Turn off input flow control.
SXOFF	Turn on output flow control.

SXOFF–	Turn off output flow control.
D*n*	Use *n* data bits per character (5, 6, 7, or 8).
S1	Send one stop bit after each character.
S2	Send two stop bits after each character.
PE	Use even parity.
PO	Use odd parity.
PS	Use space parity (parity bit always zero). (Macintosh only)
PM	Use mark parity (parity bit always one). (PC only)
P–	Don't send any parity bit (used with D8).

Almost all data transmission uses 8 bits followed by a stop bit. Sometimes the eighth bit is used for parity, other times as another data bit. The only reasonable combinations of parity and number of data bits are "D8 P–," "D7 PE," or "D7 PS" (except for odd applications like news wire services).

The parity bit is both generated *and* checked by the same method. If it's wrong, an error will be recorded, so the next time you look at Com_status *("ERR")*, you'll see that there was a parity error. In most cases, you should just ignore parity errors, since whatever's sending you characters may not be using the same algorithm (some use even, some use space). Almost no-one checks parity.

Exceptions:	7004	Channel isn't open.
	960	Unknown communication option: XXX

Com_Send

```
CALL Com_Send (s$)
```

Com_Send simply sends the string *s$* over the communications line. It actually just puts the string into an output buffer and *starts* sending. The transmission goes on in background, while your program continues to run. The output buffer is 10240 bytes long, typically, and Com_Send won't return immediately if your program gets farther ahead than that — it will have to wait for some of the data to actually be sent over the communication line.

If you need to know when the output is finished, see Com_buf.

Com_Send is the workhorse sending subroutine. Subroutines that send lines terminated with a CR-LF or just a CR are Com_SendLine and Com_SendCR.

(Use `CALL Send (s$)` if you are using COMLIB.TRC.)

Com_SendLine

```
CALL Com_SendLine (line$)
```

Com_SendLine is just like Com_Send except that it sends a carriage-return and a line-feed after sending *line$*. It's useful for line printers. If you want the line to be terminated with a different character sequence, just change the code in CommLib.tru and recompile. Or, just use the Com_Send subroutine and overtly send the line termination character sequence you need.

(Use `CALL Send_Line (line$)` if you are using COMLIB.TRC.)

Com_SendCR

```
CALL Com_SendCR (line$)
```

Com_SendCR is just like Com_Send, except that it sends a carriage-return (but no line-feed) after sending *line$*. It's useful for impersonating a person at a terminal, sending command lines to a remote computer system for example. If you want the line to be terminated with a different character sequence, just change the code in CommLib.tru and recompile. Or, just use the Com_Send subroutine and overtly send the line termination character sequence you need.

(Use `CALL Send_CR (line$)` if you are using COMLIB.TRC.)

Com_SendBreak

```
CALL Com_SendBreak
```

A *break* is not a character, and so can't simply be put into a string and given to the Com_Send subroutine. Com_SendBreak waits for all other output to stop, then sends a break (which means holding the line in the zero, or spacing, state for 200 ms.), then returns. This allows you to simulate the effect of the break key on most terminals.

If your program needs to recognize breaks received from the communications line, take a look at Com_status *("ERR")*. That will tell you if a break has been received.

Exceptions: 7004 Channel isn't open.

(Use `CALL Send_Break` if you are using COMLIB.TRC.)

Com_Receive

```
CALL Com_Receive (buf$)
```

Com_Receive sets *buf$* to whatever data has been received since the last time it was called. It never waits. If nothing has yet come in, *buf$* will be set to the null string.

Com_Receive is the workhorse receiving subroutine. A subroutines that look for CR in the received string is Com_ReceiveLine.

(Use `CALL Receive (buf$)` if you are using COMLIB.TRC.)

Example:

```
call Com_SendLine (command$)     ! request another block
let block$ = ""
do   ! should get 2k bytes
   call Com_Receive (x$)
   let block$ = block$ & x$      ! accumulate bytes
loop while len(block$)#<2048     ! until we get all 2K
```

Com_ReceiveLine

```
CALL Com_ReceiveLine (line$)
```

Com_ReceiveLine gets the next line of input, up to a carriage-return character. The carriage-return is removed, as is a line-feed (if present). It may have to wait, unlike Com_Receive, until the line is complete.

This subroutine is not recommended for very high data rates. Instead, you should try to use Com_Receive. It should generally not be used with the "D8" option (see Com_control), since it looks for a carriage-return character without a parity bit. If the carriage-return was sent with even parity, and received with the "D8" option, it won't be recognized.

To be exact, an *initial* line-feed character (if one is present) is removed from the line, but the routine won't wait to see if a line-feed *follows* the carriage-return. This will be important only if you mix calls to Com_Receive and Com_ReceiveLine.

If there's nothing coming in on the communication line, Com_ReceiveLine will just keep waiting forever. See Com_WaitLine if your program needs to time out and take corrective action in such cases.

If you prefer to scan for a terminating character other than a CR, simply make the change in the source code in CommLib.tru and recompile.

Exceptions: 7004 Channel isn't open.

Parity errors and the like don't cause exceptions. They will just be recorded, and can be checked for with *Com_status ("ERR")*.

(Use `CALL Receive_Line (line$)` if you are using COMLIB.TRC.)

Com_Buf

```
DEF COM_BUF (type)
```

Com_buf returns the amount of buffer space (in bytes) currently free or currently used, for either the send or receive buffers, depending on the parameter *type*.

Type	Result
0	Number of bytes waiting to be sent (when this number goes to zero, the transmitter is idle)
1	Number of bytes free in output buffer (number of bytes you can send without waiting)
2	Number of bytes waiting in input buffer (if you called *Receive*, this is how long the string would be)
3	Number of bytes free in input buffer

Example:

```
declare def Com_Buf
do
loop until Com_Buf(0) = 0 ! wait til all data is out
call Com_Control ("DTR-") ! then hang up the phone
```

Exceptions:	7004	Channel isn't open.

Com_Status

```
DEF Com_Status (type$)
```

The function Com_Status provides for a grab-bag of rarely-used information. It can tell you are the modem status lines, various kinds of transmission errors, and whether either input or output are currently pausing because of an XOFF (control-s) character. You use *type$* to say what you're checking for. Usually the result is either 0 or 1 ("ERR" is the only exception). If the information is not available, or if type$ is misspelled, result will be -1.*Type$* may be in upper or lower case.

Type	Result
DSR	1 if Data Set Ready is on.
RLSD	1 if Receive Line Signal Detected (Carrier).
DCD	Same as RLSD ("Data Carrier Detected").
CTS	1 if Clear To Send.
RI	1 if the phone is Ringing.
RXOFF	1 if output stopped because we received an XOFF.
SXOFF	1 if we sent an XOFF because the input buffer was full.
CR	1 if there's a Carriage-Return in the input buffer (so *Receive_line* won't have to wait).
ERR	Returns the highest-numbered error since the last call, or zero if none.

Error types returned by Com_status ("ERR")

Type	Result
0	No error since the last call.
1	Parity error (usually this should be ignored, most computers are pretty lax about parity).
2	Framing error (the stop bit was a zero, not a 1).
3	A break was received (a null character will mark the spot in the input stream).
4	Chip overrun (another character came in before the computer could respond to the last one).
5	Input buffer overrun (too many characters came in since the last time you called *Receive*).

Example:

```
print "Connecting...
call Com_open (#1,1,1200,"")     ! Open up the modem
let start_time = Time     ! Wait up to 60 sec for carrier
do
   if Time > start_time + 60 then cause error 1, "Modem Timeout."
loop until Com_status("RLSD") = 1
print "Connected."
```

Exceptions:	7004	Channel isn't open.
	961	Unknown communication status type: YYY

Com_WaitLine

```
CALL Com_WaitLine (wtime, tflag, line$)
```

Com_WaitLine waits for the next line of input (up to a carriage-return), but will time-out after *wtime* seconds. If a carriage-return is received within the specified time, Com_WaitLine will set *line$* to the line received (without carriage-return or line-feed), then set *tflag* to zero, and return. But if more than *wtime* seconds go by before a carriage-return is received, Com_WaitLine will return with *tflag* set to 1 and *line$* to the partial line received so far.

Com_WaitLine is useful if you want your program to retry when the thing you're communicating with doesn't respond. It's very much like Com_ReceiveLine, except that it won't wait forever like Com_ReceiveLine will. In fact, Call Com_ReceiveLine (l$) is equivalent to Call Com_WaitLine (maxnum, xxx, l$).

(Use `CALL WaitLine (wtime, tflag, line$)` if you are using COMLIB.TRC.)

Com_WaitPrompt

```
CALL Com_WaitPrompt (prompt$, wtime, tflag, buf$)
```

Com_WaitPrompt waits for the specified string, *prompt$,* to be received. It will return as soon as that string is received, or when *wtime* seconds have gone by. The time-out flag *tflag* will be set to zero if the prompt string *was* received, otherwise it will be one. In either case, *buf$* will contain everything Com_WaitPrompt receives from the port up to the time it returns.

If you want Com_WaitPrompt to wait indefinitely, without time-out, pass MAXNUM for *wtime*.

(Use `CALL Wait_Prompt (prompt$, wtime, tflag, buf$)` if you are using COMLIB.TRC.)

Example:

```
! Subroutine to sign on to time-sharing system

sub Signon (#1)
   call Open_com (#1,1,9600,"")     ! open the port
   call Com_WaitPrompt ("login:",10,t,s$)  ! wait for prompt
   if t = 0 then
      call Com_SendCR ("vicki")     ! send username
      call Com_WaitLine (1,t,s$)    ! first line is echo
      call Com_WaitLine (5,t,s$)    ! next is message of the day
   end if
   if t = 0 then
      print s$                      ! print message of the day
   else
      print "System isn't responding.
   end if
end sub
```

Exceptions:	7004	Channel isn't open.

A Note About Speed

The communication library is capable of supporting speeds up to 38400 baud fairly easily. But it's quite easy to write programs that can't keep up. Obviously the more processing you do for each character (or each line) of input, the lower the line speed you'll be able to handle. That may not matter for output — there's usually no penalty, except time, for sending fewer bytes per second than the line could support. It may not matter for input either, if you use input flow control (the SXOFF option). But even so, you'll probably want to get the best performance you can from the communication line.

Here are a few hints for high-speed communication. The most important one is *avoid character-by-character processing*. The subroutines in the communication library work most efficiently with large blocks, rather than individual characters. When you receive input from the communication line, it comes in a string, containing whatever was received since the last time you checked. The higher your loop overhead, the more data you'll get each time you call Com_Receive. But if you avoid looking at every byte of the string, your loop will be more efficient for longer strings. A natural equilibrium will be reached that depends on how fast the bytes are coming in.

More hints for high data rates:

- Don't process *either* input or output one byte at a time.
- Don't use Com_ReceiveLine — use Com_Receive instead. Com_ReceiveLine doesn't allow the kind of equilibrium mentioned above. It's also less efficient.
- If you must look at every byte of a string, use Unpackb, rather than taking single-character substrings. It's convenient to use a loop with a step size of 8 when using Unpackb.
- Use flow control (the SXOFF and RXOFF options) if possible. This at least means speed mismatches won't result in lost data. Sometimes it's just not possible to process input at the full line speed.
- If you're using files, use byte files. Use a fairly large record size for reading or writing them. Byte files are faster than text or record files, but their chief advantage is to allow you to process the data in large batches.

Low Level Subroutines

All the subroutines and functions previously described depend on two low-level builtin subroutines. The are: ComOpen and ComLib.

ComOpen Subroutine

```
CALL ComOpen (method, #1, port, speed, options$)
```

The ComOpen subroutine provides for opening a communications port. The methods are:

Method 0	Open the communications port. P1 is the port number. P2 is the intended data speed. Options$ are the same as desribed for Com_Open The port number must be from 1 to whatever the number of ports available on your platform. P2 must be as described for Com_Open.
Method 8	Close the communications port. This may also be done with the CLOSE #1 statement.

ComLib Subroutine

```
CALL ComLib (method, p1, p2, ps$)
```

The ComLib subroutine provides access to the communications ports. The methods are numbered from 0 to 8. The description of each method is as follows:

Method 0: Open

This method opens a communications port. The ports are numbered from 1 to n, where n is the number of communications ports available. On Windows and similar machines port 1 is the same as COM1, port 2 is the same as COM2, and so on. P2 is the desired speed. Most implementations can handle any speed up to a system-dependent maximum. Ps$ provides the options, which are the same as for Com_Open.

Opening a communications port in this way is not recommended. Using ComOpen or Com_Open is preferred as True BASIC automatically closes all channels on program termination.

Method 1: Switch

This method allow you to switch to another port, which, of course, must have been previously opened. P1 specifies the new port. P2 and ps$ are ignored.

Method 2: Control

This method allows you to send control options to a communications port. Typical options are:

DTR	Turn on Data Terminal Ready.
DTR–	Turn off Data Terminal Ready.
RTS	Turn on Request To Send.
RTS–	Turn off Request To Send.
RXOFF	Turn on input flow control.
RXOFF–	Turn off input flow control.
SXOFF	Turn on output flow control.
SXOFF–	Turn off output flow control.
D*n*	Use *n* data bits per character (5, 6, 7, or 8).
S1	Send one stop bit after each character.
S2	Send two stop bits after each character.
PE	Use even parity.
PO	Use odd parity.
PS	Use space parity (parity bit always zero).
PM	Use mark parity (parity bit always one).
P–	Don't send any parity bit (used with D8).

Method 3: Send

This method is the workhorse method for sending a character string to a communications port. Ps$ contains the string of characters to be sent. P1 contains the number of bytes sent. P2 is ignored.

Method 4: Receive

This method is the workhorse method for receiving characters from a communications port. If p1 = 0, then all bytes present are returned in ps$. Otherwise, p1 gives the number of bytes returned; more may still be out there. P2 is ignored.

Method 5: Status

This method allows you to determine the status of any of several conditions. Use ps$ to specify the condition you wish to check. Its status is returned in p1. If the status is unavailable, or if it is spelled wrong, p1 will be -1. P2 is ignored. Typical conditions are:

DSR	1 if Data Set Ready is on.
RLSD	1 if Receive Line Signal Detected (Carrier).
DCD	Same as RLSD ("Data Carrier Detected").
CTS	1 if Clear To Send.
RI	1 if the phone is Ringing.
RXOFF	1 if output stopped because we received an XOFF.
SXOFF	1 if we sent an XOFF because the input buffer was full.
CR	1 if there's a Carriage-Return in the input buffer (so Com_ReceiveLine won't have to wait).

ERR	Returns the highest-numbered error since the last call, or zero if none. The error numbers are:

0	No error since the last call.
1	Parity error (usually this should be ignored, most computers are pretty lax about parity).
2	Framing error (the stop bit was a zero, not a 1).
3	A break was received (a null character will mark the spot in the input stream).
4	Chip overrun (another character came in before the computer could respond to the last one).
5	Input buffer overrun (too many characters came in since the last time you called Com_Receive).

Method 6: Scan

This method allows you to scan the characters that have been received so far for the presence of a certain character or character string. Place the search key in ps$. P1 contains the first location in the characters received so far that matches the search key; p1 is 0-based, that is, p1 = 0 refers to the first character. If no match was found, p1 = -1.

Method 7: Break

This method sends a break. The other arguments are ignored.

Method 8: Close

This method can be used to close a communications port. P1 is the port to be closed. P2 and ps$ are ignored. If you use ComOpen or Com_Open to open a communications port (recommended,) then you can close the port using a CLOSE #1 or similar statement.

CHAPTER

23

Additional Library Procedures

In addition to the functions and subroutines built in to True BASIC, there are several small libraries that extend the functions subroutines available. These are all stored in the subdirectory TBLIB that is installed in the main True BASIC directory.

To use any of these libraries of procedures, you must include a **LIBRARY** statement in your program of the form:

```
   LIBRARY "c:\TBV5\TBLIBS\SORTLIB.TRC"
```

using the appropriate path name and file names for each library containing functions or subroutines you wish to use. Although source code may also be provided for some libraries, your programs will run much faster if you use the compiled version of any libraries.

As with all functions stored in external procedures, you must also name any library functions in **DECLARE DEF** statements before you can use them.

This chapter describes library procedures, listed alphabetically within four categories — math, strings, sorting, and graphics:

Mathematical Tools

MATHLIB.TRC	hyperbolic and arc functions
HEXLIB.TRC	bit, octal, and hexadecimal manipulation routines

String Tools

STRLIB.TRC	string creation, conversion, formatting, editing

Sorting and Searching Tools

SORTLIB.TRC	sorting, searching, and reversing items on arrays

Graphics Tools

BGLIB.TRC	for pie charts, bar charts, and histograms
SGLIB.TRC	for plotting data and function values
SGFUNC.TRC	for plotting values of functions that you define

Math Libraries

The built-in trigonometric and hyperbolic functions include: SIN, COS, TAN, CSC, SEC, COT, ATN, ASIN, ACOS, SINH, COSH, and TANH. The math libraries contain the following additional functions:

Math Libraries

Library	**Functions**
MATHLIB.TRC	Additional hyperbolic and arc functions (radian measure): ACOT, ACSC, ASEC, COTH, CSCH, SECH, ACOSH, ACOTH, ACSCH, ASECH, ASINH and ATANH.
HEXLIB.TRC	Several bit, octal, and hexadecimal manipulation routines: AND (bit-by-bit), BIN$, CONVERT, HEX$, HEXW$, OR (bit-by-bit), and XOR (bit-by-bit)

In addition, TBLIBS contains short files that you can copy and paste or " include" near the beginning of your program. These files contain a **LIBRARY** statement and a **DECLARE DEF** statement to specify the names of the functions in the library, and are named MATHDECL.TRU and HEXDECL.TRU.

Each of the math library functions are described below; the functions are listed alphabetically.

ACOSH Function

Library: MATHLIB.TRC

Syntax: ACOSH (*numex*)

Usage: `LET a = ACOSH (n)`

Summary: Returns the value of the hyperbolic arccosine of its argument `n`.

Details: The **ACOSH** function returns the value of the hyperbolic arccosine of its argument. Since neither the argument to nor the result of a hyperbolic function is an angle (and since the function definition is stored in a library), the results of the **ACOSH** function are unaffected by the current setting of the **OPTION ANGLE** statement.

The absolute value of `n` must be greater than or equal to 1.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Acosh
PRINT Acosh(1)
END
```

produces the following output:

```
0
```

Exceptions: 1003 Overflow in numeric function.
-3000 Argument not in range.

See also: **COSH**, **SINH**, **TANH**, **COTH**, **SECH**, **CSCH**, **ASINH**, **ATANH**, **ACOTH**, **ASECH**, and **ACSCH**. The first three are built-in.

Note: The **ACOSH** function may be defined in terms of other True BASIC constructs as:

```
DEF Acosh(x)
    IF Abs(x) < 1 then
       CAUSE ERROR -3000, "Argument not in range."
    ELSE
       LET Acosh = Log(x + Sqr(x*x-1))
    END IF
END DEF
```

ACOT Function

Library: MATHLIB.TRC

Syntax: ACOT(*numex*)

Usage: `LET y = ACOT(n)`

Summary: Returns the value of the arccotangent of its argument n.

Details: The **ACOT** functions returns the values of the arccotangent in radians of its argument.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Acot
PRINT Acot(1)
END
```

produces the output

```
 .78539816
```

Exceptions: 1003 Overflow in numeric function.

See also: **ATN**, **ASIN**, **ACOS**, **ASEC**, and **ACSC**. The first three are built-in.

Note: The **ACOT** function may be defined in terms of other True BASIC constructs as:

```
DEF Acot(x) = PI/2 - Atn(x)
```

ACOTH Function

Library: MATHLIB.TRC

Syntax: ACOTH (*numex*)

Usage: `LET a = ACOTH (n)`

Summary: Returns the value of the hyperbolic arccotangent of its argument `n`.

Details: The **ACOTH** function returns the value of the hyperbolic arccotangent of its argument. The absolute value of `n` must be greater than or equal to 1.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Acoth
PRINT Acoth(1)
END
```

produces the following output:

```
0
```

Exceptions: 1003 Overflow in numeric function.
-3000 Argument not in range.

See also: **COSH**, **SINH**, **TANH**, **COTH**, **SECH**, **CSCH**, **ASINH**, **ATANH**, **ACOSH**, **ASECH**, and **ACSCH**. The first three are built-in.

Note: The **ACOTH** function may be defined in terms of other True BASIC constructs as:

```
DEF Acoth(x)
    IF Abs(x) <= 1 then
       CAUSE ERROR -3000, "Argument not in range."
    ELSE
       LET Acoth = Log( (x+1)/(x-1) ) / 2
    END IF
END DEF
```

ACSC Function

Library: MATHLIB.TRC

Syntax: ACSC(*numex*)

Usage: `LET y = ACSC(n)`

Summary: Returns the value of the arccosecant of its argument n.

Details: The **ACSC** functions returns the values of the arccosecant in radians of its argument.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Acsc
PRINT Acsc(1)
GET KEY key
END
```

produces the output

```
 1.57079632
```

Exceptions:
1003 Overflow in numeric function.
-3000 Argument not in range.

See also: **ATN**, **ASIN**, **ACOS**, **ASEC**, and **ACOT**. The first three are built-in.

Note: The **ACOT** function may be defined in terms of other True BASIC constructs as:

```
DEF Acsc(x)
    IF Abs(x) < 1 then CAUSE ERROR -3000
    LET Acsc = Asin(1/x)
END DEF
```

ACSCH Function

Library: MATHLIB.TRC

Syntax: ACSCH (*numex*)

Usage: `LET a = ACSCH (n)`

Summary: Returns the value of the hyperbolic arccosecant of its argument n.

Details: The **ACSCH** function returns the value of the hyperbolic arccosecant of its argument.

The absolute value of n must not be 0, or an exception occurs.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Acsch
PRINT Acsch(1)
END
```

produces the following output:

```
 .88137359
```

Exceptions:
1003 Overflow in numeric function.
–3000 Argument not in range.

See also: **COSH**, **SINH**, **TANH**, **COTH**, **SECH**, **CSCH**, **ACOSH**, **ASINH**, **ATANH**, **ACOTH**, and **ASECH**. The first three are built-in.

Note: The **ACSCH** function may be defined in terms of other True BASIC constructs as:

```
DEF Asech(x)
    IF x = 0 then
       CAUSE ERROR -3000, "Argument not in range."
    ELSE
       LET Acsch = Log((1+sgn(x)*Sqr(x*x+1)) / x)
    END IF
END DEF
```

AND Function

Library: HEXLIB.TRC

Syntax: AND (*numex*, *numex*)

Usage: `LET n = AND (a, b)`

Summary: Returns the result of a bit-by-bit logical AND of the values of `a` and `b`.

Details: The **AND** function returns the result of a bit-by-bit logical AND of the values of `a` and `b`. That is, it compares each bit in the value of `a` with the corresponding bit in the value of `b` and sets the corresponding bit in the resulting value to 1 if both bits being compared are set to 1. Otherwise, that bit in the resulting value is set to 0.

Note that if the values of `a` and `b` are not integers, the **AND** function uses the greatest integer values which are less than their actual values.

Example: The following program:

```
LIBRARY "HEXLIB.TRC"
DECLARE DEF And

PRINT And(0, 0)
PRINT And(1, 0)
PRINT And(1, 1)
PRINT And(5, 6)
PRINT And(-5, 6)
PRINT And(5.8, 6.9)
PRINT And(255, 127)

END
```

produces the following output:

```
 0
 0
 1
 4
 2
 4
 127
```

Exceptions: None

See also: **OR**, **XOR**

ASEC Function

Library: MATHLIB.TRC

Syntax: ASEC(*numex*)

Usage: `LET y = ASEC(n)`

Summary: Returns the value of the arcsecant of its argument n.

Details: The **ASEC** functions returns the values of the arcsecant in radians of its argument.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Asec
PRINT Asec(1)
END
```

produces the output

```
 0
```

Exceptions:

1003	Overflow in numeric function.
-3000	Argument not in range.

See also: **ATN**, **ASIN**, **ACOS**, **ACSC**, and **ACOT**. The first three are built-in.

Note: The **ASEC** function may be defined in terms of other True BASIC constructs as:

```
DEF Asec(x)
    IF Abs(x) < 1 then CAUSE ERROR -3000
    LET Asec = Acos(1/x)
END DEF
```

ASECH Function

Library: MATHLIB.TRC

Syntax: ASECH (*numex*)

Usage: `LET a = ASECH (n)`

Summary: Returns the value of the hyperbolic arcsecant of its argument `n`.

Details: The **ASECH** function returns the value of the hyperbolic arcsecant of its argument.

The absolute value of `n` must be less than or equal to 1, or an exception occurs.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Asech
PRINT Asech(1)
END
```

produces the following output:

```
0
```

Exceptions:

1003	Overflow in numeric function.
–3000	Argument not in range.

See also: **COSH**, **SINH**, **TANH**, **COTH**, **SECH**, **CSCH**, **ACOSH**, **ASINH**, **ATANH**, **ACOTH**, and **ACSCH**. The first three are built-in.

Note: The **ASECH** function may be defined in terms of other True BASIC constructs as:

```
DEF Asech(x)
    IF Abs(x) > 1 then
       CAUSE ERROR -3000, "Argument not in range."
    ELSE
       LET Asech = Log((1+Sqr(1-x*x)) / x)
    END IF
END DEF
```

ASINH Function

Library: MATHLIB.TRC

Syntax: ASINH (*numex*)

Usage: `LET a = ASINH (n)`

Summary: Returns the value of the hyperbolic arcsine of its argument `n`.

Details: The **ASINH** function returns the value of the hyperbolic arcsine of its argument.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Asinh
PRINT Asinh(1)
END
```

produces the following output:

```
 .88137359
```

Exceptions: 1003 Overflow in numeric function.

See also: **COSH**, **SINH**, **TANH**, **COTH**, **SECH**, **CSCH**, **ACOSH**, **ATANH**, **ACOTH**, **ASECH**, and **ACSCH**. The first three are built-in.

Note: The **ASINH** function may be defined in terms of other True BASIC constructs as:

```
DEF Asinh(x) = Log(x + Sqr(x*x+1))
```

ATANH Function

Library: MATHLIB.TRC

Syntax: ATANH (*numex*)

Usage: `LET a = ATANH (n)`

Summary: Returns the value of the hyperbolic arctangent of its argument `n`.

Details: The **ATANH** function returns the value of the hyperbolic arctangent of its argument.

The absolute value of `n` must be less than 1, or an exception occurs.

Example: The following program:

```
LIBRARY "FMATHLIB.TRC"
DECLARE DEF Atanh
PRINT Atanh(.5)
END
```

produces the following output:

```
 .54930614
```

Exceptions:

1003	Overflow in numeric function.
–3000	Argument not in range.

See also: **COSH**, **SINH**, **TANH**, **COTH**, **SECH**, **CSCH**, **ACOSH**, **ASINH**, **ACOTH**, **ASECH**, and **ACSCH**. The first three are built-in.

Note: The **ATANH** function may be defined in terms of other True BASIC constructs as:

```
DEF Atanh(x)
    IF Abs(x) >= 1 then
       CAUSE ERROR -3000, "Argument not in range."
    ELSE
       LET Atanh = Log((1+x)/(1-x)) / 2
    END IF
END DEF
```

BIN$ Function

Library: HEXLIB.TRC

Syntax: BIN$ (*numex*)

Usage: `LET binary$ = BIN$ (n)`

Summary: Returns a string containing the signed binary representation of the value of its argument `n`.

Details: The **BIN$** function returns a string containing a binary representation of the value of `n`. The resulting string contains only the number of digits necessary to represent the value of `n`; leading zeroes are not added.

Note that the resulting binary value inherits the sign of the value of `n`. That is, the results of the **BIN$** function will be the unsigned binary representation of the absolute value of `n`, with the sign of `n` appearing as the first character if `n` is negative.

Example: The following program:

```
LIBRARY "HEXLIB.TRC"
DECLARE DEF Bin$

PRINT Bin$(0)
PRINT Bin$(1)
PRINT Bin$(-1)
PRINT Bin$(255)
PRINT Bin$(1037)

END
```

produces the following output:

```
0
1
-1
11111111
10000001101
```

Exceptions: None

See also: **HEX$**, **HEXW$**, **OCT$**, **CONVERT**

CONVERT Function

Library: HEXLIB.TRC

Syntax: CONVERT (*strex*)

Usage: `LET decimal = CONVERT (number$)`

Summary: Converts the hexadecimal, octal, binary, or decimal value represented by `number$` into its decimal numeric equivalent.

Details: The **CONVERT** function converts a hexadecimal, octal, binary, or decimal value represented as a string into the equivalent decimal numeric value.

The value of `number$` may be specified in a number of common formats.

If the value of `number$` begins or ends with the character H, h, X, x, or $, it is interpreted as representing a hexadecimal value.

If the value of `number$` begins or ends with the character O, o, Q, or q, it is interpreted as representing an octal value.

If the value of `number$` begins or ends with the character B or b, it is interpreted as representing a binary value.

If the value of `number$` begins with an ampersand (&), the second character is considered. If the second character is an H or an h, the value of `number$` is interpreted as representing a hexadecimal value. However, if the second character is a digit, the value of `number$` is interpreted a representing an octal value.

If the value of `number$` begins with a zero, the second character is considered. If the second character is a digit, the value of `number$` is interpreted a representing an octal value, unless the value ends with one of the characters mentioned above. If the second character is an X or an x, the value of `number$` is interpreted as representing a hexadecimal value.

If none of the codes mentioned above is present in the value of `number$`, it will be interpreted as representing a decimal value, and the **CONVERT** function will behave in a manner identical to the **VAL** function.

If the value is signed, the sign (+ or –) must be the first character. (The sign is stripped prior to the application of the above rules.)

If the value of `number$` does not obey these rules or represents a non-numeric value, an exception will occur.

Example: The following program:

```
LIBRARY "HEXLIB.TRC"
DECLARE DEF Convert

PRINT Convert("2CH"), Convert("2ch")
PRINT Convert("17Q")
PRINT Convert("00110011B")
PRINT Convert("$3B")
PRINT Convert("&73")
END
```

produces the following output:

```
 44             44
 15
 51
 59
 59
```

Exceptions: 4001 Val string isn't a proper number.

See also: **VAL**, **BIN$**, **HEX$**, **HEXW$**, **OCT$**

COTH Function

Library: MATHLIB.TRC

Syntax: COTH (*numex*)

Usage: `LET a = COTH (n)`

Summary: Returns the value of the hyperbolic cotangent of its argument `n`.

Details: The **COTH** function returns the value of the hyperbolic cotangent of its argument.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Coth
PRINT Coth(1)
END
```

produces the following output:

```
1.3130353
```

Exceptions: 1003 Overflow in numeric function.

See also: **COSH**, **SINH**, **TANH**, **SECH**, **CSCH**, **ACOSH**, **ASINH**, **ATANH**, **ACOTH**, **ASECH**, and **ACSCH**. The first three are built-in.

Note: The **COTH** function may be defined in terms of other True BASIC constructs as:

```
DEF Coth(x) = 1 / ((Exp(x) - Exp(-x))/(Exp(x) + Exp(-x)))
```

CSCH Function

Library: MATHLIB.TRC

Syntax: CSCH (*numex*)

Usage: `LET a = CSCH (n)`

Summary: Returns the value of the hyperbolic cosecant of its argument `n`.

Details: The **CSCH** function returns the value of the hyperbolic cosecant of its argument.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Csch
PRINT Csch(1)
END
```

produces the following output:

```
.85091813
```

Exceptions: 1003 Overflow in numeric function.
Division by zero.

See also: **COSH**, **SINH**, **TANH**, **COTH**, **SECH**, **ACOSH**, **ASINH**, **ATANH**, **ACOTH**, **ASECH**, and **ACSCH**. The first three are built-in.

Note: The **CSCH** function may be defined in terms of other True BASIC constructs as:

```
DEF Sech(x) = 1 / (Exp(x) - Exp(-x))
```

HEX$ Function

Library: HEXLIB.TRC

Syntax: HEX$ (*numex*)

Usage: `LET hexadecimal$ = HEX$ (n)`

Summary: Returns a string containing the signed hexadecimal representation of the value of its argument `n`.

Details: The **HEX$** function returns a string containing a hexadecimal representation of the value of `n`. The resulting string contains only the number of digits necessary to represent the value of `n`; leading zeroes are not added.

Note that the resulting hexadecimal value inherits the sign of the value of `n`. That is, the results of the **HEX$** function will be the unsigned hexadecimal representation of the absolute value of `n`, with the sign of `n` appearing as the first character if `n` is negative.

Example: The following program:

```
LIBRARY "HEXLIB.TRC"
DECLARE DEF Hex$

PRINT Hex$(0)
PRINT Hex$(1)
PRINT Hex$(-1)
PRINT Hex$(255)
PRINT Hex$(1037)

END
```

produces the following output:

```
0
1
-1
FF
40D
```

Exceptions: None

See also: **HEXW$**, **BIN$**, **OCT$**, **CONVERT**

HEXW$ Function

Library: HEXLIB.TRC

Syntax: HEXW$ (*numex*)

Usage: `LET hexadecimal$ = HEXW$ (n)`

Summary: Returns a four-character string containing the unsigned hexadecimal representation of the value of its argument `n`.

Details: The **HEXW$** function returns a string containing the unsigned hexadecimal representation of the value of `n`. The resulting string will always contain exactly four characters; leading zeroes will be added if necessary.

Negative values of `n` are treated as two's complement. Thus, the resulting value will be an unsigned value.

Example: The following program:

```
LIBRARY "HEXLIB.TRC"
DECLARE DEF Hexw$

PRINT Hexw$(0)
PRINT Hexw$(1)
PRINT Hexw$(-1)
PRINT Hexw$(255)
PRINT Hexw$(1037)

END
```

produces the following output:

```
0000
0001
FFFF
00FF
040D
```

Exceptions: None

See also: **HEX$**, **BIN$**, **OCT$**, **CONVERT**

OCT$ Function

Library: HEXLIB.TRC

Syntax: OCT$ (*numex*)

Usage: `LET octal$ = OCT$ (n)`

Summary: Returns a string containing the signed octal representation of the value of its argument `n`.

Details: The **OCT$** function returns a string containing a signed octal representation of the value of `n`. The resulting string contains only the number of digits necessary to represent the value of `n`; leading zeroes are not added.

Note that the resulting octal value inherits the sign of the value of `n`. That is, the results of the **OCT$** function will be the unsigned octal representation of the absolute value of `n`, with the sign of `n` appearing as the first character if `n` is negative.

Example: The following program:

```
LIBRARY "HEXLIB.TRC"
DECLARE DEF Oct$

PRINT Oct$(0)
PRINT Oct$(1)
PRINT Oct$(-1)
PRINT Oct$(255)
PRINT Oct$(1037)

END
```

produces the following output:

```
0
1
-1
377
2015
```

Exceptions: None

See also: **HEX$**, **HEXW$**, **BIN$**, **CONVERT**

OR Function

Library: HEXLIB.TRC

Syntax: OR (*numex*, *numex*)

Usage: `LET n = OR (a, b)`

Summary: Returns the result of a bit-by-bit logical OR of the values of `a` and `b`.

Details: The **OR** function returns the result of a bit-by-bit logical OR of the values of `a` and `b`. That is, it compares each bit in the value of `a` with the corresponding bit in the value of `b` and sets the corresponding bit in the resulting value to 0 if both bits being compared are set to 0. Otherwise, that bit in the resulting value is set to 1.

Example: The following program:

```
LIBRARY "HEXLIB.TRC"
DECLARE DEF Or

PRINT Or(0, 0)
PRINT Or(1, 0)
PRINT Or(1, 1)
PRINT Or(5, 6)
PRINT Or(-5, 6)
PRINT Or(5.8, 6.9)
PRINT Or(255, 127)

END
```

produces the following output:

```
 0
 1
 1
 7
-1
 8.7
 255
```

Exceptions: None

See also: **AND**, **XOR**

SECH Function

Library: MATHLIB.TRC

Syntax: SECH (*numex*)

Usage: `LET a = SECH (n)`

Summary: Returns the value of the hyperbolic secant of its argument `n`.

Details: The **SECH** function returns the value of the hyperbolic secant of its argument.

Example: The following program:

```
LIBRARY "MATHLIB.TRC"
DECLARE DEF Sech
PRINT Sech(1)
END
```

produces the following output:

```
.64805427
```

Exceptions: 1003 Overflow in numeric function.

See also: **COSH**, **SINH**, **TANH**, **COTH**, **CSCH**, **ACOSH**, **ASINH**, **ATANH**, **ACOTH**, **ASECH**, and **ACSCH**. The first three are built-in.

Note: The **SECH** function may be defined in terms of other True BASIC constructs as:

```
DEF Sech(x) = 1 / (Exp(x) + Exp(-x))
```

XOR Function

Library: HEXLIB.TRC

Syntax: XOR (*numex*, *numex*)

Usage: `LET n = XOR (a, b)`

Summary: Returns the result of a bit-by-bit logical XOR of the values of `a` and `b`.

Details: The **XOR** function returns the result of a bit-by-bit logical XOR (“exclusive OR”) of the values of `a` and `b`. That is, it compares each bit in the value of `a` with the corresponding bit in the value of `b` and sets the corresponding bit in the resulting value to 0 if both bits being compared are equal. Otherwise, that bit in the resulting value is set to 1.

Example: The following program:

```
LIBRARY "HEXLIB.TRC"
DECLARE DEF Xor

PRINT Xor(0, 0)
PRINT Xor(1, 0)
PRINT Xor(1, 1)
PRINT Xor(5, 6)
PRINT Xor(-5, 6)
PRINT Xor(5.8, 6.9)
PRINT Xor(255, 127)

END
```

produces the following output:

```
 0
 1
 0
 3
-3
 4.7
 128
```

Exceptions: None

See also: **AND**, **OR**

String Handling Libraries

In addition to the strings functions included in True BASIC, additional tools are provided in the library STRLIB.TRC. All of the string functions and subroutines are described below in alphabetical order.

The operations provided by these functions and subroutines include:

Return sets of string constants	
	ALPHANUM$, CONTROL$, DIGITS$, LETTERS$, LOWER$, PUNCT$, RNDSTR$, UPPER$
Return simple parts of strings	
	LEFT$, MID$, RIGHT$
Return date and time strings:	
	NICEDATE$, NICETIME$, NOW$, SHORTDATE$, TODAY, WEEKDAY, WEEKDAY$
Convert between numeric and other representations of numbers:	
	DOLLARS$, DOLLARVAL, ENGLISHNUM$, EVAL, LVAL, ROMAN$, SUPERVAL
Parse a string	BREAKUP, EXPLODE, EXPLODEN, NEXTWORD
Trim extra spaces	INTRIM$, LTRIM$, RTRIM$, TRIM$, JUSTIFY$, NOSPACE$
Format text	CENTER$, FILLARRAY, FILLFROM, HEADER$, JUSTIFY$, JUSTIFYARRAY, JUSTIFYFROM, LJUST$, RJUST$
Reverse a string	REVERSE$
Find shared or non-shared characters in two strings	
	CHARDIFF$, CHARINT$, CHARS$, CHARUNION$, UNIQ$
Remove or replace characters from strings	
	DELCHAR$, DELMIX$, DELSTR$, KEEPCHAR$, MAPCHAR$, NPLUGCHAR$, NREPCHAR$, PLUGCHAR$, PLUGMIX$, PLUGSTR$, REPCHAR$, REPMIX$, REPSTR$

ALPHANUM$ Function

Library: STRLIB.TRC

Syntax: ALPHANUM$

Usage: `LET set$ = ALPHANUM$`

Summary: Returns the set of alphabetic and digit characters.

Details: The **ALPHANUM$** function returns a string containing the set of characters representing the letters of the alphabet, both uppercase and lowercase, as well as the digits, arranged in ascending order according to their ASCII codes. (For a table of the ASCII codes and their corresponding characters, see Appendix A.)

That is, it returns the equivalent of the string constant:

`"0123456789ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"`

Example: None.

Exceptions: None

See also: **DIGITS$**, **LETTERS$**, **UPPER$**, **LOWER$**, **PUNCT$**, **CONTROL$**

BREAKUP Subroutine

Library: STRLIB.TRC

Syntax: CALL BREAKUP (*strex*, *strex*, *strex*)

Usage: `CALL BREAKUP (phrase$, word$, delims$)`

Summary: Returns the next word, as delineated by a delimiter character, from the specified phrase.

Details: The **BREAKUP** subroutine returns the next "word" from the specified "phrase."

A phrase is defined as any series of characters, and a word is defined as any series of characters except those defined as delimiters.

When invoking the **BREAKUP** subroutine, you pass the phrase as `phrase$` and the characters to be defined as delimiters as `delims$`. The subroutine then examines the phrase, looking for the first delimiter character, and returns the characters up to but not including this delimiter as the value of `word$`. It also updates the value of `phrase$` to eliminate the returned word and the delimiter.

Note that the **BREAKUP** subroutine strips any leading and trailing spaces off of the return values of `word$` and `phrase$`.

For example, assume you have passed `delim$` with a value of "*#;" and `phrase$` with a value of "***abc***def**" to the **BREAKUP** subroutine. The subroutine would return `word$` with a value of the null string and `phrase$` with a new value of "**abc***def**" since there is no non-delimiter value between the beginning of the phrase and the first delimiter. The process of breaking a phrase into words is often referred to as ***parsing*** a phrase.

If the value of `phrase$` does not contain any delimiters, the **BREAKUP** subroutine will return the value of `phrase$` in `word$` and return `phrase$` equal to the null string.

Note that the **BREAKUP** subroutine is closely related to, but subtly different from, the **NEXTWORD** subroutine. The **BREAKUP** subroutine treats each individual delimiter character as a delimiter in its own right, while the **NEXTWORD** subroutine treats a series of contiguous delimiter characters as a single delimiter. For some applications you will want to use the **BREAKUP** subroutine, for others you will want to use the **NEXTWORD** subroutine.

Example: The following program:

```
LIBRARY "StrLib.trc"
LET s$ = "Now is the time for all good men"
DO
   CALL BreakUp (s$, word$, " ")
   IF word$ = "" then EXIT DO
   PRINT word$,
LOOP
END
```

produces the following output:

```
all          good         men
Now          is           the          time         for
```

Exceptions: None

See also: **NEXTWORD**, **EXPLODE**, **EXPLODEN**

CENTER$ Function

Library: STRLIB.TRC

Syntax: CENTER$ (*strex*, *numex*, *strex*)

Usage: `LET a$ = CENTER$ (text$, width, back$)`

Summary: Returns a string of the specified length containing the value of `text$` center-justified.

Details: The **CENTER$** function takes the value of `text$` and adds characters alternately to the beginning and end of it as necessary to create a string containing `width` characters with the value of `text$` centered within it. The characters added will be determined by repeating the sequence of characters specified by the value of `back$`.

Note that if the value of `text$` cannot be perfectly centered within the result, the extra space will appear to the left of the value of `text$` within the resulting string.

If the length of `text$` is greater than the value specified by `width`, the **CENTER$** function returns the value of `text$` truncated to `width` characters. If the value of `width` is less than or equal to 0, the **CENTER$** function returns the null string.

The background pattern added to the beginning and end of the value of `text$` will be formed in such a way that all strings formed with the same value of `back$` will have identical background patterns, regardless of the value of `text$`. If the value of `back$` is the null string or a space, the background pattern will consist solely of blanks, or spaces.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Center$
LET s$ = "Hello, out there"
FOR w = 20 to 25
   LET t$ = Center$ (s$, w, "*")
PRINT t$
NEXT w
END
```

produces the following output:

```
**Hello, out there**
**Hello, out there***
***Hello, out there***
***Hello, out there****
****Hello, out there****
****Hello, out there*****
```

Exceptions: None

See also: **LJUST$**, **RJUST$**, **JUSTIFY$**, **HEADER$**. **FILLARRAY**, **FILLFROM**, **JUSTIFYARRAY**, **JUSTIFYFROM**

CHARDIFF$ Function

Library: STRLIB.TRC

Syntax: CHARDIFF$ (*strex*, *strex*)

Usage: `LET difference$ = CHARDIFF$ (a$, b$)`

Summary: Returns the set of characters contained within the value of `a$` and not within the value of `b$`.

Details: The **CHARDIFF$** function returns a string containing the difference of the sets of characters represented by the values of its arguments `a$` and `b$`.

That is, the **CHARDIFF$** function returns a string which contains one of each character which appears in the value of its argument `a$` but not within the value of its argument `b$`. The characters will be organized within the resulting string in ascending order by their ASCII codes. Thus, uppercase letters will be listed before all lowercase letters.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF CharDiff$
LET a$ = "Hello, out there"
LET b$ = "aeiouy"
PRINT "*"; CharDiff$ (a$, b$); "*"
END
```

produces the following output:

```
* ,Hhlrt*
```

Exceptions: None

See also: **CHARUNION$**, **CHARINT$**, **CHARS$**, **UNIQ$**

CHARINT$ Function

Library: STRLIB.TRC

Syntax: CHARINT$ (*strex*, *strex*)

Usage: `LET intersection$ = CHARINT$ (a$, b$)`

Summary: Returns the set of characters contained within the values of both `a$` and `b$`.

Details: The **CHARINT$** function returns a string containing the intersection of the sets of characters represented by the values of its arguments `a$` and `b$`.

That is, the **CHARINT$** function returns a string which contains one of each character which appears in the values of both its arguments. The characters will be organized within the resulting string in ascending order by their ASCII codes. Thus, uppercase letters will be listed before all lowercase letters.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF CharInt$
LET a$ = "Hello, out there"
LET b$ = "aeiouy"
PRINT "*"; CharInt$ (a$, b$); "*"
END
```

produces the following output:

```
*eou*
```

Exceptions: None

See also: **CHARUNION$**, **CHARDIFF$**, **CHARS$**, **UNIQ$**

CHARS$ Function

Library: STRLIB.TRC

Syntax: CHARS$ (*numex*, *numex*)

Usage: `LET set$ = CHARS$ (start, end)`

Summary: Returns the set of characters whose ASCII codes range from the value of `start` to the value of `end`.

Details: The **CHARS$** function returns a string containing the set of characters whose ASCII codes range in value from `start` to `end`.

If the value of `start` is less than or equal to the value of `end`, then the characters within the resulting string will be arranged in ascending order by ASCII codes. If the value of `start` is greater than the value of `end`, the resulting string will be arranged in descending order.

If either `start` or `end` has a value less than 0, a value of 0 will be used instead. Likewise, if either has a value greater than 255, a value of 255 will be used instead.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Chars$
PRINT Chars$ (33, 47)
END
```

produces the following output:

```
!"#$%&'()*+,-./
```

Exceptions: None

See also: **CHARINT$**, **CHARDIFF$**, **CHARUNION$**, **UNIQ$**

CHARUNION$ Function

Library: STRLIB.TRC

Syntax: CHARUNION$ (*strex*, *strex*)

Usage: `LET union$ = CHARUNION$ (a$, b$)`

Summary: Returns the set of characters contained within the values of either `a$` or `b$`.

Details: The **CHARUNION$** function returns a string containing the union of the sets of characters represented by the values of its arguments `a$` and `b$`.

That is, the **CHARUNION$** function returns a string which contains one of each character which appears in the value of either of its arguments. The characters will be organized within the resulting string in ascending order by their ASCII codes. Thus, uppercase letters will be listed before all lowercase letters.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF CharUnion$
LET a$ = "Now is the time"
LET b$ = "for all good men"
PRINT CharUnion$ (a$, b$)
END
```

produces the following output:

```
 Nadefghilmnorstw
```

(Note: there is a space before the 'N'.)

Exceptions: None

See also: **CHARINT$**, **CHARDIFF$**, **CHARS$**, **UNIQ$**

CONTROL$ Function

Library: STRLIB.TRC

Syntax: CONTROL$

Usage: `LET set$ = CONTROL$`

Summary: Returns the set of control characters.

Details: The **CONTROL$** function returns a string containing the set of control characters, arranged in ascending order according to their ASCII codes. (For a table of the ASCII codes and their corresponding characters, see Appendix A.)

That is, it returns a string composed of those characters with ASCII codes between 0 and 31, inclusive.

Example: None.

Exceptions: None

See also: **DIGITS$**, **LETTERS$**, **ALPHANUM$**, **UPPER$**, **LOWER$**, **PUNCT$**

DELCHAR$ Function

Library: STRLIB.TRC

Syntax: DELCHAR$ (*strex*, *strex*)

Usage: `LET a$ = DELCHAR$ (text$, oldchars$)`

Summary: Returns the value of `text$` with all characters appearing in `oldchars$` removed.

Details: The **DELCHAR$** function removes from `text$` all characters which are members of the character set represented by the value of `oldchars$`. That is, it returns the value of `text$` after having deleted any occurrences of characters appearing in the value of `oldchars$`.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF CharDiff$, Control$, DelChar$

OPEN #1: NAME "InFile", ORG BYTE
OPEN #2: NAME "OutFile", ORG BYTE, CREATE NEW
ERASE #2

LET crlf$ = Chr$(13) & Chr$(10)
LET bad$ = CharDiff$(Control$, crlf$)

DO WHILE MORE #1
   READ #1, BYTES 10000: block$
   WRITE #2: DelChar$(block$, bad$)
LOOP

END
```

removes all control characters except carriage return and line feed from the file named INFILE, and stores the results in a file named OUTFILE.

Exceptions: None

See also: **KEEPCHAR$**, **REPCHAR$**, **NREPCHAR$**, **MAPCHAR$**, **PLUGCHAR$**, **NPLUGCHAR$**

DELMIX$ Function

Library: STRLIB.TRC

Syntax: DELMIX$ (*strex*, *strex*)

Usage: `LET a$ = DELMIX$ (text$, old$)`

Summary: Returns the value of `text$` with all occurrences of the value of `old$`, in any mix of upper- and lowercase, removed.

Details: The **DELMIX$** function removes from `text$` all occurrences of the value of `old$`, without regard to case. That is, it returns the value of `text$` after having deleted any occurrences of the value of `old$` in any mix of uppercase and lowercase letters.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF DelMix$
LET a$ = "Now is the time for all good men"
LET b$ = "e"
PRINT DelMix$ (a$, b$)
END
```

produces the following output:

```
Now is th tim for all good mn
```

Exceptions: None

See also: **DELSTR$**, **DELCHAR$**, **REPMIX$**, **PLUGMIX$**

DELSTR$ Function

Library: STRLIB.TRC

Syntax: DELSTR$ (*strex*, *strex*)

Usage: `LET a$ = DELSTR$ (text$, old$)`

Summary: Returns the value of `text$` with all occurrences of the value of `old$` removed.

Details: The **DELSTR$** function removes from `text$` all occurrences of the value of `old$`. That is, it returns the value of `text$` after having deleted any occurrences of the value of `old$`.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF DelStr$

DO
   LINE INPUT PROMPT "Enter a line (Return to quit): ": line$
   IF line$ = "" then EXIT DO
   PRINT DelStr$(line$, ",")
LOOP

END
```

deletes all commas from lines input by the user.

Exceptions: None

See also: **DELMIX$**, **DELCHAR$**, **REPSTR$**, **PLUGSTR$**

DIGITS$ Function

Library: STRLIB.TRC

Syntax: DIGITS$

Usage: `LET set$ = DIGITS$`

Summary: Returns the set of digit characters.

Details: The **DIGITS$** function returns a string containing the set of characters representing the digits arranged in ascending order according to their ASCII codes. (For a table of the ASCII codes and their corresponding characters, see Appendix A.)

That is, it returns the equivalent of the string constant:

```
"0123456789"
```

Example: None.

Exceptions: None

See also: **LETTERS$**, **UPPER$**, **LOWER$**, **ALPHANUM$**, **PUNCT$**, **CONTROL$**

DOLLARS$ Function

Library: STRLIB.TRC

Syntax: DOLLARS$ (*numex*)

Usage: `LET formatted$ = DOLLARS$ (number)`

Summary: Returns the string representation of its numeric argument as a dollar amount.

Details: The **DOLLARS$** function returns the string representation of `number` as a nicely formatted dollar amount.

The resulting string will begin with a dollar sign, have a comma between each set of three digits to the left of the decimal point, and end with two digits to the right of the decimal point. The resulting string will not contain any extraneous spaces or zeros (except those zeros which may be necessary to provide two digits to the right of the decimal point).

The **DOLLARS$** function is a convenient method for creating a very specific formatting of numeric values. The **USING$** function, however, provides much greater flexibility in how numeric values are formatted.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Dollars$
PRINT Dollars$ (1234.5678)
END
```

produces output similar to the following:

```
$1,234.57
```

Exceptions: None

See also: **USING$** (built-in function)

DOLLARVAL Function

Library: STRLIB.TRC

Syntax: DOLLARVAL (*strex*)

Usage: `LET number = DOLLARVAL (string$)`

Summary: Returns the numeric value represented by the contents of its string argument, allowing dollar signs, commas, asterisks, and embedded spaces.

Details: The **DOLLARVAL** function is a more flexible form of the **VAL** function. Like the **VAL** function it returns the numeric value of contents of its string argument `string$`, but it ignores any dollar signs, commas, asterisks, and spaces that may be embedded within the string.

Once the embedded dollar signs, commas, asterisks, and spaces have been removed, the string value of `string$` must represent a valid numeric constant in a form suitable for use with an **INPUT** or **READ** statement. Note that this value may represent a valid numeric constant expressed in exponential (or scientific) notation.

If the value of `string$` does not represent a valid numeric constant once the embedded dollar signs, commas, asterisks, and spaces have been removed, the **DOLLARVAL** function will generate an error.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF DollarVal
PRINT DollarVal ("$1,234.57")
END
```

produces the following output:

```
 1234.57
```

Exceptions:

1004	Overflow in VAL.
4001	VAL string isn’t a proper number.

See also: **VAL**, **LVAL**, **EVAL**, **SUPERVAL**

ENGLISHNUM$ Function

Library: STRLIB.TRC

Syntax: ENGLISHNUM$ (*numex*)

Usage: `LET english$ = ENGLISHNUM$ (number)`

Summary: Returns the string representation of its numeric argument in English.

Details: The **ENGLISHNUM$** function returns the string representation of the proper English name of `number`. For example, if passed a value of 117, the **ENGLISHNUM$** function would return the string value:

```
One Hundred and Seventeen
```

As you can see, the English name of the numeric value will be represented in mixed upper- and lowercase letters.

The English representation of a negative number will begin with the word “negative,” and all numbers are rounded to five digits after the decimal point. Thus, if passed a value of –3.1415926, the **ENGLISHNUM$** function will return the string value:

```
negative Three point One Four One Five Nine
```

The **ENGLISHNUM$** function uses the American, rather than British, names for large numbers. Thus, passing a value of 1000000 results in a return value of:

```
One Million
```

and passing a value of 1000000000 results in a return value of:

```
One Billion
```

Note that the results of the **ENGLISHNUM$** function can be wrong for very large numbers (usually those greater than 9,000,000,000) because of the inaccuracy introduced by floating point round off errors.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF EnglishNum$

FOR i = 1 to 10
    FOR j = 1 to 10
        PRINT EnglishNum$(i); " times "; EnglishNum$(j);
        PRINT " is "; EnglishNum$(i*j)
    NEXT j
NEXT i

GET KEY k
END
```

produces a multiplication table in English (that is, without using digits).

Exceptions: None

See also: **ROMAN$**

EVAL Function

Library: STRLIB.TRC

Syntax: EVAL (*strex*)

Usage: `LET result = EVAL (expression$)`

Summary: Returns the value of the constant-based expression represented by the contents of its string argument.

Details: The **EVAL** function evaluates the numeric expression represented by the value of `expression$` and returns the resulting value.

The value of `expression$` must represent a numeric expression which is valid under the rules of True BASIC. This expression may contain numeric constants, but not variables. (For the evaluation of expressions containing variables, see the **SUPERVAL** subroutine.)

The value of `expression$` may incorporate any of the following operators:

Operators Available to EVAL

```
+     -     *     /     ^     (     )
```

In addition, the value of `expression$` may incorporate any of the following numeric functions:

Functions Available to EVAL

SIN	COS	TAN	ATN	SQR
SINH	COSH	TANH	ASIN	ACOS
SEC	CSC	COT	MAXNUM	EPS
PI	SGN	ABS	RAD	DEG
LOG	LOG2	LOG10	EXP	RND
INT	ROUND	IP	FP	CEIL
	DATE		TIME	

Note that numeric functions requiring two arguments, including the two-argument form of the **ROUND** function, are not available for use in the value of `expression$`.

Also note that the **EVAL** function is not as fast as the **VAL** function, but it is a great deal more flexible.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF Eval

PRINT Eval("1.23e-3 + 2*(Sin(Pi/2)+Rnd)")

INPUT PROMPT "Enter an expression: ": expr$
PRINT Eval(expr$)

GET KEY k
END
```

produces output similar to the following:

```
 3.0982454
Enter an expression: 25 + 12 / 16 * Sin(Pi/3)
 25.649519
```

Exceptions:

1	Eval string isn't a proper expression.
2	Unknown function or variable in Eval.
3002	Negative number to non-integral power.
3003	Zero to negative power.
3004	LOG of number <= 0.
3005	SQR of negative number.
4001	VAL string isn't a proper number.

See also: **SUPERVAL**, **VAL**, **LVAL**, **DOLLARVAL**

EXPLODE Subroutine

Library: STRLIB.TRC

Syntax: CALL EXPLODE (*strex*, *strarrarg*, *strex*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL EXPLODE (phrase$, words$(), delims$)`

Summary: Parses the contents of the specified phrase into an array of words, as delineated by the specified delimiters.

Details: The **EXPLODE** subroutine breaks the specified "phrase" into the "words" which comprise it and returns the resulting words in the form of a string array.

A phrase is defined as any series of characters, and a word is defined as any series of characters except those defined as delimiters.

When invoking the **EXPLODE** subroutine, you pass the phrase as phrase$ and the characters to be defined as delimiters as delims$. The subroutine then examines the phrase, looking for the first set of one or more non-delimiter characters which are set off from the rest of the phrase by delimiters, and assigns this set of non-delimiter characters, less any leading or trailing delimiters, to the next available element of the `words$` array. It then repeats the process, beginning the search after the last assigned word and its surrounding delimiters, until all of the words in the phrase have been assigned to the `words$` array.

The process of breaking a phrase into its component words is called ***parsing*** the phrase.

The **EXPLODE** subroutine does not alter the value of `phrase$`. It will, however, adjust the upper bound of the `words$` array to ensure that the returned array has exactly one element for each word in the phrase.

To parse a phrase composed of numeric constants into a numeric array, use the **EXPLODEN** subroutine instead.

Example: The following program:

```
LIBRARY "STRLIB.TRC", "SortLib"
DIM words$(1)
```

```
LET punct$ = " !""#$%&'()*+,-./:;<=>?@[\]^)_`{|}~"

INPUT PROMPT "Enter name of file: ": fname$
OPEN #1: NAME fname$, ORG BYTE
ASK #1: FILESIZE fsize
READ #1, BYTES fsize: file$

LET delims$ = punct$ & Chr$(13) & Chr$(10)

CALL Explode(file$, words$(), delims$)
CALL SortS(words$())
MAT PRINT words$

GET KEY k
END
```

prints a list of the words in a specified file arranged alphabetically.

Exceptions: None

See also: **EXPLODEN**, **NEXTWORD**, **BREAKUP**

EXPLODEN Subroutine

Library: STRLIB.TRC

Syntax: CALL EXPLODEN (*strex*, *numarrarg*, *strex*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL EXPLODEN (phrase$, numbers(), delims$)`

Summary: Parses the contents of the specified phrase into an array of numbers, as delineated by the specified delimiters.

Details: The **EXPLODEN** subroutine breaks the specified "phrase" into the numeric "words" which comprise it and returns the resulting words in the form of a numeric array.

A phrase is defined as any series of characters, and a word is defined as any series of characters except those defined as delimiters. However, if a word does not define a valid numeric constant, the **EXPLODEN** subroutine will generate an error.

When invoking the **EXPLODEN** subroutine, you pass the phrase as phrase$ and the characters to be defined as delimiters as delims$. The subroutine then examines the phrase, looking for the first set of one or more non-delimiter characters which are set off from the rest of the phrase by delimiters, and assigns the numeric value of this set of non-delimiter characters to the next available element of the `numbers` array. It then repeats the process, beginning the search after the last assigned numeric value, until all of the numeric values in the phrase have been assigned to the `numbers` array.

The process of breaking a phrase into its component parts is called ***parsing*** the phrase.

The **EXPLODEN** subroutine does not alter the value of `phrase$`. It will, however, adjust the upper bound of the `numbers` array to ensure that the returned array has exactly one element for each numeric value in the phrase.

To parse a phrase composed of non-numeric or mixed numeric values, use the **EXPLODE** subroutine instead.

Example: The following program:

```
LIBRARY "StrLib.trc"
DIM numbers(0)
LET phrase$ = "123,456,789.777"
CALL Exploden (phrase$, numbers(), ",.")
MAT PRINT numbers;
END
```

produces the following output:

```
 123  456  789  777
```

Exceptions: 4001 VAL string isn't a proper number.

See also: **EXPLODE**, **NEXTWORD**, **BREAKUP**

FILLARRAY Subroutine

Library: STRLIB.TRC

Syntax: CALL FILLARRAY (*strarr*, *strarrarg*, *numex*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL FILLARRAY (from$(), to$(), width)`

Summary: "Fills" the contents of the `to$` array to the specified `width` with the contents of the `from$` array.

Details: The **FILLARRAY** subroutine processes the contents of the `from$` array to create the "filled" array `to$`. The width to which the resulting `to$` array will be filled is determined by the value of `width`.

A common operation in text processing, ***filling*** is the process of organizing blocks of text so that each line contains as many words as will fit within a specified margin. A blank line or a line which begins with one or more spaces acts a "break," which means that the following line will not be absorbed into the previous line.

If a single word is encountered which has a length greater than `width`, it will be placed on a line by itself; no error will be generated.

It is possible to pass the same array as both `from$` and `to$`, eliminating the need for a temporary array.

The **FILLARRAY** subroutine is useful for wrapping text to fit within a window with a known margin.

Example: The following program:

```
LIBRARY "StrLib.trc"
DIM from$(16), to$(0)
MAT READ from$
DATA Now, is, the, time, for, all, good, men
DATA to, come, to, the, aid, of, their, party

CALL FillArray (from$(), to$(), 20)
FOR i = 1 to ubound(to$)
   PRINT to$(i)
NEXT i
END
```

produces the following output:

```
Now is the time for
all good men to come
to the aid of their
party
```

Exceptions: None

See also: **FILLFROM**, **JUSTIFYARRAY**, **JUSTIFYFROM**, **LJUST$**, **RJUST$**, **CENTER$**, **JUSTIFY$**. **HEADER$**

FILLFROM Subroutine

Library: STRLIB.TRC

Syntax: CALL FILLFROM (#*rnumex*, *numex*, *strvar*, *strvar*)

Usage: `CALL FILLFROM (#1, width, result$, work$)`

Summary: Returns a single "filled" line from the specified file.

Details: The **FILLFROM** subroutine retrieves a "filled" line from the text file associated with the specified channel number and returns it as `result$`. The width to which the resulting `to$` array will be filled is determined by the value of `width`.

A common operation in text processing, ***filling*** is the process of organizing blocks of text so that each line contains as many words as will fit within a specified margin. A blank line or a line which begins with one or more spaces acts a "break," which means that the following line will not be absorbed into the previous line.

The **FILLFROM** subroutine treats the entire contents of the text file associated with the specified channel number as the pool of text which needs to be filled. To process an entire file, continue invoking the **FILLFROM** subroutine until the end of the file is reached.

The value of `work$` is used internally by the **FILLFROM** subroutine. The first time you invoke the **FILLFROM** subroutine for a given file, pass a `work$` argument with a null string value. Then continue passing the same `work$` argument, without changing its contents, to each invocation of the **FILLFROM** subroutine which you intend to read from the same file. Failure to do so could result in the loss of data.

If a single word is encountered which has a length greater than `width`, it will be returned as a line by itself; no error will be generated.

Example: The following program:

```
LIBRARY "STRLIB.TRC"

OPEN #1: NAME "TextFile"
OPEN #2: PRINTER

DO WHILE MORE #1
   CALL FillFrom(#1, 65, line$, work$)
   PRINT #2: line$
LOOP

END
```

would produce a printed listing of the filled contents of the file TEXTFILE.

Exceptions: None

See also: **FILLARRAY**, **JUSTIFYARRAY**, **JUSTIFYFROM**, **LJUST$**, **RJUST$**, **CENTER$**, **JUSTIFY$**. **HEADER$**

HEADER$ Function

Library: STRLIB.TRC

Syntax: HEADER$ (*strex*, *strex*, *strex*, *numex*, *strex*)

Usage: `LET a$ = HEADER$ (left$, center$, right$, width, back$)`

Summary: Creates a "header" of the specified width which contains the specified text items.

Details: The **HEADER$** function takes the values of `left$`, `center$`, and `right$` and returns a string of `width` characters which contains the values of these strings. The value of `left$` will be left-justified within the resulting string, the value of `center$` will be centered within the resulting string, and the value of `right$` will be right-justified within the resulting string. The extra characters between them will be comprised of the background pattern specified by the value of `back$`.

The string resulting from an invocation of the **HEADER$** function is commonly used as a header or footer for formatted text.

Any or all of the values of `left$`, `center$`, and `right$` may be the null string to eliminate text in the appropriate position within the header.

If the value of `back$` is the null string or a space, the extra characters between the items in the header will be spaces.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Header$
LET left$ = "Contents"
LET center$ = "Page 1"
LET right$ = date$
PRINT Header$(left$, center$, right$, 50, " ")
END
```

produces the following output:

```
Contents             Page 1            19980403
```

Exceptions: None

See also: **LJUST$**, **RJUST$**, **CENTER$**, **JUSTIFY$**. **FILLARRAY**, **FILLFROM**, **JUSTIFYARRAY**, **JUSTIFYFROM**

INTRIM$ Function

Library: STRLIB.TRC

Syntax: INTRIM$ (*strex*)

Usage: `LET a$ = INTRIM$ (string$)`

Summary: Returns the value of its argument `string$` with all series of spaces within it reduced to a single space.

Details: The **INTRIM$** function returns the value of its argument with all series of spaces within it mapped to a single space. That is, it processes the value of `string$` looking for any occurrence of two or more contiguous spaces. Upon finding such an occurrence, the **INTRIM$** function replaces the series of spaces with a single space.

The **INTRIM$** function is useful for cleaning up the results of the **USING$** function, which can create strings with several extra spaces. The **INTRIM$** function may also be used to undo the results of the **JUSTIFY$** function.

To completely remove all of the spaces from a string, use the **NOSPACE$** function.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF InTrim$

INPUT PROMPT "Enter two numbers: ": a, b
PRINT InTrim$(Using$("#,### plus #,### equals ##,###", a, b, a+b))

END
```

produces output similar to the following:

```
Enter two numbers: 13, 1200
 13 plus 1,200 equals 1,213
```

Exceptions: None

See also: **LTRIM$**, **RTRIM$**, **TRIM$**, **JUSTIFY$**, **NOSPACE$**

JUSTIFY$ Function

Library: STRLIB.TRC

Syntax: JUSTIFY$ (*strex*, *numex*)

Usage: `LET a$ = JUSTIFY$ (text$, width)`

Summary: Returns a string of the specified length containing the value of `text$` justified.

Details: The **JUSTIFY$** function takes the value of `text$` and adds spaces as necessary to create a string containing `width` characters with the "words" in the value of `text$` evenly spaced within it. The extra spaces will be added to the value of `text$` where spaces already exist in order to "fill out" the value of `text$` to the specified number of characters.

By filling out a series of lines, you can produce text with even left and right margins.

The **JUSTIFY$** function never adds spaces to the beginning of the value of `text$`, even if it begins with one or more spaces. This allows the **JUSTIFY$** function to operate properly when used with lines of indented text.

If the value of `width` is less than the length of `text$`, the **JUSTIFY$** function generates an error. Likewise, if the value of `text$` contains no spaces, it cannot be filled, and the **JUSTIFY$** function generates an error. If the value of `text$` is the null string, then the **JUSTIFY$** function returns the null string.

The **JUSTIFY$** function does not work for lines containing more than 100 words. If you pass a value of `text$` which contains more than 100 words, the **JUSTIFY$** function will generate an error.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Justify$
LET text$ = "Now is the time for all good men"
FOR w = 32 to 40 step 2
   PRINT Justify$(text$, w)
NEXT w
END
```

produces the following output:

```
Now is the time for all good men
Now is the time  for all good  men
Now  is  the time for  all  good men
Now  is  the  time  for  all good  men
Now  is  the  time   for  all  good  men
```

Exceptions:

1	More than 100 words in line.
2	Can't justify line.

See also: **LJUST$**, **RJUST$**, **CENTER$**, **HEADER$**. **FILLARRAY**, **FILLFROM**, **JUSTIFYARRAY**, **JUSTIFYFROM**

JUSTIFYARRAY Subroutine

Library: STRLIB.TRC

Syntax: CALL JUSTIFYARRAY (*strarr*, *strarrarg*, *numex*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL JUSTIFYARRAY (from$(), to$(), width)`

Summary: "Fills" and justifies the contents of the `to$` array to the specified `width` with the contents of the `from$` array.

Details: The **JUSTIFYARRAY** subroutine processes the contents of the `from$` array to create the "filled" and justified array `to$`. The width to which the resulting `to$` array will be filled and justified is determined by the value of `width`.

A common operation in text processing, ***filling*** is the process of organizing blocks of text so that each line contains as many words as will fit within a specified margin. A blank line or a line which begins with one or more spaces acts a "break," which means that the following line will not be absorbed into the previous line, and the previous line will not be justified.

Another common operation in text processing, ***justification*** is the process of adding spaces within a line so that the last character on the line falls evenly at the right margin. Spaces are always added where spaces already exist within the line.

The **JUSTIFYARRAY** subroutine never adds spaces to the beginning of a line, even if it begins with one or more spaces. This allows the **JUSTIFYARRAY** subroutine to operate properly when it encounters indented lines.

If it encounters a line which contains no spaces, the **JUSTIFYARRAY** subroutine generates an error, since such a line cannot be justified.

The **JUSTIFYARRAY** subroutine also cannot justify lines containing more than 100 words. If it encounters a line which contains more than 100 words, the **JUSTIFYARRAY** subroutine will generate an error.

It is possible to pass the same array as both `from$` and `to$`, eliminating the need for a temporary array.

Example: The following program:

```
LIBRARY "StrLib.trc"
DIM from$(4), to$(4)
MAT READ from$
DATA Now is the time
DATA for all good men
DATA to come to the aid
DATA of their party.

CALL JustifyArray (from$(), to$(), 25)
FOR i = 1 to ubound(to$)
    PRINT to$(i)
NEXT i
END
```

produces the following output:

```
Now is  the time  for all
good men to  come to  the
aid of their party.
```

Exceptions:

1	More than 100 words in line.
2	Can't justify line.

See also: **JUSTIFYFROM**, **FILLARRAY**, **FILLFROM**, **LJUST$**, **RJUST$**, **CENTER$**, **JUSTIFY$**. **HEADER$**

JUSTIFYFROM Subroutine

Library: STRLIB.TRC

Syntax: CALL JUSTIFYFROM (#*rnumex*, *numex*, *strvar*, *strvar*)

Usage: `CALL JUSTIFYFROM (#1, width, result$, work$)`

Summary: Returns a single "filled" and justified line from the specified file.

Details: The **JUSTIFYFROM** subroutine retrieves a "filled" and justified line from the text file associated with the specified channel number and returns it as `result$`. The contents `result$` will be filled and justified according to the value of `width`.

A common operation in text processing, ***filling*** is the process of organizing blocks of text so that each line contains as many words as will fit within a specified margin. A blank line or a line which begins with one or more spaces acts a "break," which means that the following line will not be absorbed into the previous line.

Another common operation in text processing, ***justification*** is the process of adding spaces within a line so that the last character on the line falls evenly at the right margin. Spaces are always added where spaces already exist within the line.

The **JUSTIFYFROM** subroutine never adds spaces to the beginning of a line, even if it begins with one or more spaces. This allows the **JUSTIFYFROM** subroutine to operate properly when it encounters indented lines.

If it encounters a line which contains no spaces, the **JUSTIFYFROM** subroutine generates an error, since such a line cannot be justified.

The **JUSTIFYFROM** subroutine also cannot justify lines containing more than 100 words. If it encounters a line which contains more than 100 words, the **JUSTIFYFROM** subroutine will generate an error.

The **JUSTIFYFROM** subroutine treats the entire contents of the text file associated with the specified channel number as the pool of text which needs to be filled and justified. To process an entire file, continue invoking the **JUSTIFYFROM** subroutine until the end of the file is reached.

The value of `work$` is used internally by the **JUSTIFYFROM** subroutine. The first time you invoke the **JUSTIFYFROM** subroutine for a given file, pass a `work$` argument with a null string value. Then continue passing the same `work$` argument, without changing its contents, to each invocation of the **JUSTIFYFROM** subroutine which you intend to read from the same file. Failure to do so could result in the loss of data.

Example: The following program:

```
LIBRARY "StrLib.trc"
OPEN #1: name "TextFile"
OPEN #2: Printer
DO while more #1
   CALL JustifyFrom (#1, 65, line$, work$)
   PRINT #2
LOOP
END
```

would produce a printed list of the justified contents of the text file TEXTFILE.

Exceptions:	1	More than 100 words in line.
	2	Can't justify line.

See also: **JUSTIFYARRAY**, **FILLARRAY**, **FILLFROM**, **LJUST$**, **RJUST$**, **CENTER$**, **JUSTIFY$**. **HEADER$**

KEEPCHAR$ Function

Library: STRLIB.TRC

Syntax: KEEPCHAR$ (*strex*, *strex*)

Usage: `LET a$ = KEEPCHAR$ (text$, oldchars$)`

Summary: Returns the value of `text$` with all characters which do not appear in `oldchars$` removed.

Details: The **KEEPCHAR$** function removes from `text$` all characters which are not members of the character set represented by the value of `oldchars$`. That is, it returns the value of `text$` after having deleted any occurrences of characters which do not appear in the value of `oldchars$`.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF KeepChar$, Punct$

OPEN #1: NAME "InFile", ORG BYTE

LET pun$ = Punct$
```

```
DO WHILE MORE #1
   READ #1, BYTES 10000: block$
   LET sum = sum + Len(KeepChar$(block$, pun$))
LOOP
PRINT sum

END
```

counts the punctuation in the file named INFILE.

Exceptions: None

See also: **DELCHAR$**, **REPCHAR$**, **NREPCHAR$**, **MAPCHAR$**, **PLUGCHAR$**, **NPLUGCHAR$**

LEFT$ Function

Library: STRLIB.TRC

Syntax: LEFT$ (*strex*, *numex*)

Usage: `LET a$ = LEFT$ (string$, chars)`

Summary: Returns the leftmost `chars` characters of `string$`.

Details: The **LEFT$** function returns a specific number of characters from the beginning of `string$`. The number of characters to be returned is given by the value of `chars`.

If the value of `chars` is less than or equal to 0, the **LEFT$** function returns the null string. If the value of `chars` is greater than the number of characters in `string$`, the **LEFT$** function returns the value of `string$`.

Note that the **LEFT$** function can be useful when converting programs written in other forms of BASIC. However, you will likely find that the substring expressions discussed in Chapter 17 provide greater speed and flexibility.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Left$
LET a$ = "Now is the time"
FOR chars = 5 to 8
    PRINT Left$(a$, chars)
NEXT chars
END
```

produces the following output:

```
Now i
Now is
Now is
Now is t
```

Exceptions: None

See also: **RIGHT$**, **MID$**

LETTERS$ Function

Library: STRLIB.TRC

Syntax: LETTERS$

Usage: `LET set$ = LETTERS$`

Summary: Returns the set of alphabetic characters.

Details: The **LETTERS$** function returns a string containing the set of characters representing the letters of the alphabet, both uppercase and lowercase, arranged in ascending order according to their ASCII codes. (For a table of the ASCII codes and their corresponding characters, see Appendix A.)

That is, it returns the equivalent of the string constant:

`"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz"`

Example: None.

Exceptions: None

See also: **DIGITS$**, **UPPER$**, **LOWER$**, **ALPHANUM$**, **PUNCT$**, **CONTROL$**

LJUST$ Function

Library: STRLIB.TRC

Syntax: LJUST$ (*strex*, *numex*, *strex*)

Usage: `LET a$ = LJUST$ (text$, width, back$)`

Summary: Returns a string of the specified length containing the value of `text$` left-justified.

Details: The **LJUST$** function takes the value of `text$` and adds characters to the end of it as necessary to create a string containing `width` characters with the value of `text$` left-justified within it. The characters added will be determined by repeating the sequence of characters specified by the value of `back$`.

If the length of `text$` is greater than the value specified by `width`, the **LJUST$** function returns the value of `text$` truncated to `width` characters. If the value of `width` is less than or equal to 0, the **LJUST$** function returns the null string.

The background pattern added to the end of the value of `text$` will be formed in such a way that all strings formed with the same value of `back$` will have identical background patterns, regardless of the value of `text$`. If the value of `back$` is the null string or a space, the background pattern will consist solely of blanks, or spaces.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF LJust$
LET a$ = "...Hello"
FOR chars = 10 to 12
    PRINT LJust$(a$, chars, "*")
NEXT chars
END
```

produces the following output:

```
...Hello**
...Hello***
...Hello****
```

Exceptions: None

See also: **RJUST$**, **CENTER$**, **JUSTIFY$**, **HEADER$**. **FILLARRAY**, **FILLFROM**, **JUSTIFYARRAY**, **JUSTIFYFROM**

LOWER$ Function

Library: STRLIB.TRC

Syntax: LOWER$

Usage: `LET set$ = LOWER$`

Summary: Returns the set of lowercase alphabetic characters.

Details: The **LOWER$** function returns a string containing the set of characters representing the letters of the alphabet, in lowercase only, arranged in ascending order according to their ASCII codes. (For a table of the ASCII codes and their corresponding characters, see Appendix A.)

That is, it returns the equivalent of the string constant:

`"abcdefghijklmnopqrstuvwxyz"`

Example: None.

Exceptions: None

See also: **UPPER$**, **LETTERS$**, **DIGITS$**, **ALPHANUM$**, **PUNCT$**, **CONTROL$**

LVAL Function

Library: STRLIB.TRC

Syntax: LVAL (*strex*)

Usage: `LET number = LVAL (string$)`

Summary: Returns the numeric value represented by the contents of its string argument, ignoring any extraneous characters on the end of the string.

Details: The **LVAL** function is a "leftmost" form of the **VAL** function. Like the **VAL** function it returns the numeric value of contents of its string argument `string$`, but it ignores any extraneous characters at the end of the string.

The string value of `string$` must begin with a valid numeric constant in a form suitable for use with an **INPUT** or **READ** statement; however, it may contain any number of invalid characters following this value. The numeric portion of the value may not contain embedded spaces. Nor may this value contain commas, a dollar sign, or more than one decimal point. Note, however, that this value may represent a valid numeric constant expressed in exponential (or scientific) notation.

If the value of `string$` does not begin with a valid numeric constant, the **LVAL** function returns a value of 0.

The **LVAL** function is useful when converting programs written in other forms of BASIC, which may interpret their VAL functions in this manner.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF LVal
PRINT LVal(" 123.45 dollars")
END
```

produces the following output:

```
 123.45
```

Exceptions: None

See also: **VAL**, **DOLLARVAL**, **SUPERVAL**, **EVAL**

MAPCHAR$ Function

Library: STRLIB.TRC

Syntax: MAPCHAR$ (*strex*, *strex*, *strex*)

Usage: `LET a$ = MAPCHAR$ (text$, oldchars$, newchars$)`

Summary: Returns the value of `text$` with all characters in `oldchars$` mapped to the associated characters in `newchars$`.

Details: The **MAPCHAR$** function maps one set of characters onto another. It returns the value of `text$` after having replaced any occurrences of characters appearing in the value of `oldchars$` with their corresponding characters appearing in the value of `newchars$`.

The correspondence of the two character sets represented by `oldchars$` and `newchars$` is based solely upon position within the set. That is, any occurrence of the first character in `oldchars$` will be replaced by the first character in `newchars$`, and so on.

For this reason, the values of `oldchars$` and `newchars$` must contain the same number of characters; otherwise, the **MAPCHAR$** function generates an error.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF Punct$, MapChar$

OPEN #1: NAME "InFile"

LET pun$ = Punct$
LET sp$ = Repeat$(" ", Len(pun$))

DO WHILE MORE #1
   LINE INPUT #1: line$
   PRINT MapChar$(line$, pun$, sp$)
LOOP

END
```

lists the contents of a text file named INFILE, mapping all occurrences of punctuation to spaces.

Exceptions: 1 MapChar$ sets have different lengths.

See also: **REPCHAR$**, **NREPCHAR$**, **DELCHAR$**, **KEEPCHAR$**, **PLUGCHAR$**, **NPLUGCHAR$**

MID$ Function

Library: STRLIB.TRC

Syntax: MID$ (*strex*, *numex*, *numex*)

Usage: `LET a$ = MID$ (string$, start, chars)`

Summary: Returns `chars` characters of `string$` beginning at character position `start`.

Details: The **MID$** function returns a specific number of characters from a specified position within `string$`. The number of characters to be returned is given by the value of `chars`. And the character position at which this series of characters should begin is given by the value of `start`.

If the value of `start` is less than 1, 1 will be used instead. If the value of `start` is greater than the number of characters in `string$` or the value of `chars` is less than or equal to 0, the **MID$** function returns the null string. If their are fewer than `chars` characters following the position indicated by `start`, only the existing characters will be returned.

Note that the **MID$** function can be useful when converting programs written in other forms of BASIC. However, you will likely find that the substring expressions discussed in Chapter 17 provide greater speed and flexibility.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Mid$
LET a$ = "abcdefghijklmno"
FOR start = 5 to 8
    PRINT Mid$(a$, start, start)
NEXT start
END
```

produces the following output:

```
efghi
fghijk
ghijklm
hijklmno
```

Exceptions: None

See also: **RIGHT$**, **LEFT$**

NEXTWORD Subroutine

Library: STRLIB.TRC

Syntax: CALL NEXTWORD (*strex*, *strex*, *strex*)

Usage: `CALL NEXTWORD (phrase$, word$, delims$)`

Summary: Returns the next word, as delineated by any series of delimiter characters, from the specified phrase.

Details: The **NEXTWORD** subroutine returns the next "word" from the specified "phrase."

A phrase is defined as any series of characters, and a word is defined as any series of characters except those defined as delimiters.

When invoking the **NEXTWORD** subroutine, you pass the phrase as `phrase$` and the characters to be defined as delimiters as `delims$`. The subroutine then examines the phrase, looking for the first set of one or more non-delimiter characters which are set off from the rest of the phrase by delimiters, and returns this set of non-delimiter characters after having stripped off any leading or trailing delimiters. It also updates the value of `phrase$` to eliminate the returned word and its adjacent delimiters.

For example, assume you have passed `delim$` with a value of "*#;" and `phrase$` with a value of "***abc***def**" to the **NEXTWORD** subroutine. The subroutine would return `word$` with a value of "abc" and `phrase$` with a new value of "def**". This process is often referred to as ***parsing*** a phrase.

If the value of `phrase$` is the null string or contains nothing but delimiters, the **NEXTWORD** subroutine will return both `word$` and `phrase$` with values of the null string.

Note that the **NEXTWORD** subroutine is closely related to, but subtly different from, the **BREAKUP** subroutine. The **NEXTWORD** subroutine treats a series of contiguous delimiter characters as a single delimiter, while the **BREAKUP** subroutine treats each individual delimiter character as a delimiter in its own right. For some applications you will want to use the **NEXTWORD** subroutine, for others you will want to use the **BREAKUP** subroutine.

Example: The following program:

```
LIBRARY "StrLib.trc"
LET s$ = "Now  is  the  time for all good men"
DO
   CALL NextWord (s$, word$, " ")
   IF word$ = "" then EXIT DO
   PRINT word$,
LOOP
END
```

produces the following output:

```
Now          is           the          time         for
all          good         men
```

Exceptions: None

See also: **BREAKUP**, **EXPLODE**, **EXPLODEN**

NICEDATE$ Function

Library: STRLIB.TRC

Syntax: NICEDATE$ (*strex*)

Usage: `LET a$ = NICEDATE$ (adate$)`

Summary: Returns the date represented by the value of `adate$` in string form consisting of the month name, the day of the month, and the year.

Details: The **NICEDATE$** function takes as its argument a date in the format produced by the **DATE$** function and returns that date in expanded form. This expanded form consists of the full name of the month, the day of the month, and the full year.

The value of `adate$` must represent a date in the same format produced by the **DATE$** function. That is, the value of `adate$` must be a string in the form "YYYYMMDD", where YYYY represents the year, MM the month, and DD the day. If `adate$` does not represent such a value, then the **NICEDATE$** function generates an error.

To return the current date in the same format, use the **TODAY$** function. To return a date in an abbreviated format, use the **SHORTDATE$** function.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF NiceDate$
PRINT NiceDate$("19971015")
END
```

produces the following output:

```
October 15, 1997
```

Exceptions: 1 Bad date given to NiceDate$: 00000000

See also: **DATE$**, **DATE**, **SHORTDATE$**, **WEEKDAY$**, **WEEKDAY**, **TODAY$**, **TIME**, **TIME$**, **NICETIME$**, **NOW$**

NICETIME$ Function

Library: STRLIB.TRC

Syntax: NICETIME$ (*strex*)

Usage: `LET a$ = NICETIME$ (atime$)`

Summary: Returns the time represented by the value of `atime$` in string form consisting of the hour and minute and an indication of a.m. or p.m.

Details: The **NICETIME$** function takes as its argument a time measured by the 24-hour clock and returns that time as measured by the 12-hour clock in string form. The returned time will be in the form "HH:MM", where HH is the hour and MM is the minute, and the string " a.m." or " p.m." will be appended as appropriate.

The value of `atime$` must represent a time measured by the 24-hour clock in the same format produced by the **TIME$** function. That is, the value of `atime$` must be a string in the form "HH:MM:SS", where HH represents the hour, MM the minute, and SS the second. If `atime$` does not represent such a value, then the **NICETIME$** function generates an error.

To return the current time in the same format, use the **NOW$** function.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF NiceTime$
PRINT NiceTime$("16:50:58")
END
```

produces output similar to the following:

```
4:50 p.m.
```

Exceptions: 1 Bad time given to NiceTime$: 99:99:99

See also: **TIME$**, **TIME**, **NOW$**, **DATE**, **DATE$**, **NICEDATE$**, **SHORTDATE$**, **WEEKDAY$**, **WEEKDAY**, **TODAY$**

NOSPACE$ Function

Library: STRLIB.TRC

Syntax: NOSPACE$ (*strex*)

Usage: `LET a$ = NOSPACE$ (string$)`

Summary: Returns the value of its argument `string$` with all spaces removed.

Details: The **NOSPACE$** function returns the value of its argument with all spaces removed.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF NoSpace$
LET s$ = "Now is the time"
PRINT s$
PRINT NoSpace$(s$)
END
```

produces output similar to the following:

```
Now is the time
Nowisthetime
```

Exceptions: None

See also: **LTRIM$**, **RTRIM$**, **TRIM$**, **JUSTIFY$**, **INTRIM$**

NOW$ Function

Library: STRLIB.TRC

Syntax: NOW$

Usage: `LET a$ = NOW$`

Summary: Returns the current time in string form consisting of the hour and minute and an indication of a.m. or p.m.

Details: The **NOW$** function returns the current time as measured by the 12-hour clock in string form. The time will be in the form “HH:MM”, where HH is the hour and MM is the minute, and the string “ a.m.” or “ p.m.” will be appended as appropriate.

If the current computer hardware is not able to report the time, then the **NOW$** function generates an error.

The **NOW$** function obtains its results from your operating system, which in turn obtains its results from your computer’s internal clock. If you find that the **NOW$** function is returning erroneous results, you most likely have to reset your computer’s internal clock.

To return an arbitrary time in the same format, use the **NICETIME$** function.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Now$
PRINT Now$
END
```

produces output similar to the following:

```
1:06 p.m.
```

Exceptions: 1 Bad time given to NiceTime$: 99:99:99

See also: **TIME$**, **TIME**, **NICETIME$**, **DATE**, **DATE$**, **NICEDATE$**, **SHORTDATE$**, **WEEKDAY$**, **WEEKDAY**, **TODAY$**

NPLUGCHAR$ Function

Library: STRLIB.TRC

Syntax: NPLUGCHAR$ (*strex*, *strex*, *strex*)

Usage: `LET a$ = NPLUGCHAR$ (text$, chars$, template$)`

Summary: Returns the value of `text$` with all characters which do not appear in `chars$` replaced by the specified template.

Details: The **NPLUGCHAR$** function replaces all characters in `text$` which are not also members of the character set represented by the value of `chars$`. Each occurrence within `text$` of a character that is not part of `chars$` is replaced by the value of `template$`.

The **NPLUGCHAR$** function differs from the **NREPCHAR$** function in that the value of `template$` is treated as a template. This means that each occurrence of the character combination "#1" within the value of template will be replaced with the value of the character which will be replaced by the template.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF AlphaNum$, NPlugChar$
LET text$ = "Now is the time"
PRINT NPlugChar$(text$, AlphaNum$, "|")
END
```

produces the following output:

```
Now|is|the|time
```

Exceptions: None

See also: **PLUGCHAR$**, **DELCHAR$**, **KEEPCHAR$**, **REPCHAR$**, **NREPCHAR$**, **MAPCHAR$**

NREPCHAR$ Function

Library: STRLIB.TRC

Syntax: NREPCHAR$ (*strex*, *strex*, *strex*)

Usage: `LET a$ = NREPCHAR$ (text$, oldchars$, new$)`

Summary: Returns the value of `text$` with all characters not appearing in `oldchars$` replaced by the value of `new$`.

Details: The **NREPCHAR$** function maps all characters which are not members of a character set to a single string. It returns the value of `text$` after having replaced any occurrences of characters not appearing in the value of `oldchars$` with the value of `new$`.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF Digits$, NRepChar$

OPEN #1: NAME "InFile"

LET chars$ = Digits$

DO WHILE MORE #1
   LINE INPUT #1: line$
   PRINT NRepChar$(line$, chars$, " ")
LOOP

GET KEY k
END
```

lists the contents of a text file named INFILE, replacing all characters which are not digits with spaces.

Exceptions: None

See also: **REPCHAR$**, **MAPCHAR$**, **DELCHAR$**, **KEEPCHAR$**, **PLUGCHAR$**, **NPLUGCHAR$**

PLUGCHAR$ Function

Library: STRLIB.TRC

Syntax: PLUGCHAR$ (*strex*, *strex*, *strex*)

Usage: `LET a$ = PLUGCHAR$ (text$, chars$, template$)`

Summary: Returns the value of `text$` with all characters which do appear in `chars$` replaced by the specified template.

Details: The **PLUGCHAR$** function replaces all characters in `text$` which are members of the character set represented by the value of `chars$`. Each occurrence of a character from `chars$` within `text$` is replaced by the value of `template$`.

The **PLUGCHAR$** function differs from the **REPCHAR$** function in that the value of `template$` is treated as a template. This means that each occurrence of the character combination "#1" within the value of `template$` will be replaced with the value of the character which will be replaced by the template.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF Control$, PlugChar$

OPEN #1: NAME "InFile"
OPEN #2: NAME "OutFile", CREATE NEW


LET ctl$ = Control$

DO WHILE MORE #1
   LINE INPUT #1: line$
   PRINT #2: PlugChar$(line$, ctl$, "/#1"
LOOP

END
```

places a slash (\) before each control character in the file named INFILE and stores the results in the file named OLDFILE.

Exceptions: None

See also: **NPLUGCHAR$**, **DELCHAR$**, **KEEPCHAR$**, **REPCHAR$**, **NREPCHAR$**, **MAPCHAR$**

PLUGMIX$ Function

Library: STRLIB.TRC

Syntax: PLUGMIX$ (*strex*, *strex*, *strex*)

Usage: `LET a$ = PLUGMIX$ (text$, old$, template$)`

Summary: Returns the value of `text$` with occurrences of the value of `old$`, in any mix of upper- and lowercase, replaced by the specified template.

Details: The **PLUGMIX$** function replaces all occurrences of the value of `old$`, in any mix of uppercase and lowercase, within `text$`. Each occurrence of the value of `old$` within `text$` is replaced by the value of `template$`.

The **PLUGMIX$** function differs from the **REPMIX$** function in that the value of `template$` is treated as a template. This means that each occurrence of the character combination "#1" within the value of `template$` will be replaced with the value of `old$` as it was found in `text$`.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF PlugMix$
LET text$ = "Now is tHe time"
PRINT PlugMix$(text$, "the", "THE")
END
```

produces the following output:

```
Now is THE time
```

Exceptions: None

See also: **PLUGSTR$**, **PLUGCHAR$**, **REPMIX$**, **DELMIX$**

PLUGSTR$ Function

Library: STRLIB.TRC

Syntax: PLUGSTR$ (*strex*, *strex*, *strex*)

Usage: `LET a$ = PLUGSTR$ (text$, old$, template$)`

Summary: Returns the value of `text$` with occurrences of the value of `old$` replaced by the specified template.

Details: The **PLUGSTR$** function replaces all occurrences of the value of `old$` within `text$`. Each occurrence of the value of `old$` within `text$` is replaced by the value of `template$`.

The **PLUGSTR$** function differs from the **REPSTR$** function in that the value of `template$` is treated as a template. This means that each occurrence of the character combination “#1” within the value of `template$` will be replaced with the value of `old$`.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF PlugMix$
PRINT PlugMix$("x/x", "x", "(#1+#1)")
END
```

produces the following output:

```
(x+x)/(x+x)
```

Exceptions: None

See also: **PLUGMIX$**, **PLUGCHAR$**, **REPSTR$**, **DELSTR$**

PUNCT$ Function

Library: STRLIB.TRC

Syntax: PUNCT$

Usage: `LET set$ = PUNCT$`

Summary: Returns the set of punctuation characters.

Details: The **PUNCT$** function returns a string containing the set of characters representing punctuation, arranged in ascending order according to their ASCII codes. (For a table of the ASCII codes and their corresponding characters, see Appendix A.)

That is, it returns the equivalent of the string constant:

```
" !""#$%&'()*+,-./:;<=>?@[\]^_`{|}~"
```

where the pair of consecutive quotation marks results in the inclusion of a single quotation mark.

Note that this function returns only those punctuation characters represented in the standard ASCII character set. Depending upon the current operating environment there may be additional punctuation characters available which are not represented within the results of the **PUNCT$** function.

Example: None.

Exceptions: None

See also: **UPPER$**, **LOWER$**, **LETTERS$**, **DIGITS$**, **ALPHANUM$**, **CONTROL$**

REPCHAR$ Function

Library: STRLIB.TRC

Syntax: REPCHAR$ (*strex*, *strex*, *strex*)

Usage: `LET a$ = REPCHAR$ (text$, oldchars$, new$)`

Summary: Returns the value of `text$` with all characters in `oldchars$` replaced by the value of `new$`.

Details: The **REPCHAR$** function maps the members of a character set to a single string. It returns the value of `text$` after having replaced any occurrences of characters appearing in the value of `oldchars$` with the value of `new$`.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF Punct$, RepChar$

OPEN #1: NAME "InFile"

LET chars$ = Punct$

DO WHILE MORE #1
   LINE INPUT #1: line$
   PRINT RepChar$(line$, chars$, "<PUNCT>")
LOOP

GET KEY k
END
```

lists the contents of a text file named INFILE, replacing all occurrences of punctuation with the phrase "<PUNCT>".

Exceptions: None

See also: **MAPCHAR$**, **NREPCHAR$**, **DELCHAR$**, **KEEPCHAR$**, **PLUGCHAR$**, **NPLUGCHAR$**, **REPSTR$**, **REPMIX$**

REPMIX$ Function

Library: STRLIB.TRC

Syntax: REPMIX$ (*strex*, *strex*, *strex*)

Usage: `LET a$ = REPMIX$ (text$, old$, new$)`

Summary: Returns the value of `text$` with all occurrences of the value of `old$`, in any mix of upper- and lowercase letters, replaced by the value of `new$`.

Details: The **REPMIX$** function replaces occurrences of one substring with another, ignoring case. It returns the value of `text$` after having replaced any occurrences of the value of `old$`, in any mix of upper- and lowercase letters, with the value of `new$`.

If the value of `old$` does not appear within the value of `text$`, the **REPMIX$** function returns the value of `text$` untouched.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF RepMix$

OPEN #1: NAME "InFile"
OPEN #2: NAME "OutFile", CREATE NEW
SET #2: MARGIN MAXNUM

DO WHILE MORE #1
   LINE INPUT #1: line$
   PRINT #2: RepMix$(line$, "print", "PRINT")
LOOP

END
```

copies the contents of the file named INFILE to the newly created file named OUTFILE, changing all occurrences of the word "print," regardless of case, into uppercase.

Exceptions: None

See also: **REPSTR$**, **REPCHAR$**, **DELSTR$**, **PLUGSTR$**

REPSTR$ Function

Library: STRLIB.TRC

Syntax: REPSTR$ (*strex*, *strex*, *strex*)

Usage: `LET a$ = REPSTR$ (text$, old$, new$)`

Summary: Returns the value of `text$` with all occurrences of the value of `old$` replaced by the value of `new$`.

Details: The **REPSTR$** function replaces occurrences of one substring with another. It returns the value of `text$` after having replaced any occurrences of the value of `old$` with the value of `new$`.

If the value of `old$` does not appear within the value of `text$`, the **REPSTR$** function returns the value of `text$` untouched.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF RepStr$

OPEN #1: NAME "InFile", ORG BYTE
OPEN #2: NAME "OutFile", CREATE NEW, ORG BYTE

LET cr$ = Chr$(13)
LET crlf$ = cr$ & Chr$(10)

DO WHILE MORE #1
   READ #1, BYTES 10000: block$
   WRITE #2: RepStr$(line$, cr$, crlf$)
LOOP

END
```

copies the contents of the file named INFILE to the newly created file named OUTFILE, changing all occurrences of a carriage return to a carriage return followed by a line feed.

Exceptions: None

See also: **REPMIX$**, **REPCHAR$**, **DELSTR$**, **PLUGSTR$**

REVERSE$ Function

Library: STRLIB.TRC

Syntax: REVERSE$ (*strex*)

Usage: `LET a$ = REVERSE$ (string$)`

Summary: Returns the value of its argument `string$` with all of its characters in reverse order.

Details: The **REVERSE$** function accepts a string argument, reverses the order of the characters which it contains, and returns the result.

If the value of its argument is the null string, the **REVERSE$** function will return the null string.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Reverse$
DIM text$(3)
MAT READ text$
DATA madam, im, adam
FOR i = 1 to 3
    IF text$(i) = Reverse$(text$(i)) then
       PRINT text$(i); " is a palindrome"
    ELSE
```

```
        PRINT text$(i); " is not a palindrome"
    END IF
NEXT i
END
```

produces the following output:

```
madam is a palindrome
im is not a palindrome
adam is not a palindrome
```

Exceptions: None

RIGHT$ Function

Library: STRLIB.TRC

Syntax: RIGHT$ (*strex*, *numex*)

Usage: `LET a$ = RIGHT$ (string$, chars)`

Summary: Returns the rightmost `chars` characters of `string$`.

Details: The **RIGHT$** function returns a specific number of characters from the end of `string$`. The number of characters to be returned is given by the value of `chars`.

If the value of `chars` is less than or equal to 0, the **RIGHT$** function returns the null string. If the value of `chars` is greater than the number of characters in `string$`, the **RIGHT$** function returns the value of `string$`.

Note that the **RIGHT$** function can be useful when converting programs written in other forms of BASIC. However, you will likely find that the substring expressions discussed in Chapter 17 provide greater speed and flexibility.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Right$
LET a$ = "Now is the time"
FOR chars = 5 to 8
    PRINT Right$(a$, chars)
NEXT chars
END
```

produces output similar to the following:

```
 time
e time
he time
the time
```

Exceptions: None

See also: **LEFT$**, **MID$**

RJUST$ Function

Library: STRLIB.TRC

Syntax: RJUST$ (*strex*, *numex*, *strex*)

Usage: `LET a$ = RJUST$ (text$, width, back$)`

Summary: Returns a string of the specified length containing the value of `text$` right-justified.

Details: The **RJUST$** function takes the value of `text$` and adds characters to the beginning of it as necessary to create a string containing `width` characters with the value of `text$` right-justified within it. The characters added will be determined by repeating the sequence of characters specified by the value of `back$`.

If the length of `text$` is greater than the value specified by `width`, the **RJUST$** function returns the value of `text$` truncated to `width` characters. If the value of `width` is less than or equal to 0, the **RJUST$** function returns the null string.

The background pattern added to the beginning of the value of `text$` will be formed in such a way that all strings formed with the same value of `back$` will have identical background patterns, regardless of the value of `text$`. If the value of `back$` is the null string or a space, the background pattern will consist solely of blanks, or spaces.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF RJust$
LET a$ = "...Hello"
FOR chars = 10 to 12
    PRINT RJust$(a$, chars, "*")
NEXT chars
END
```

produces the following output:

```
**...Hello
***...Hello
****...Hello
```

Exceptions: None

See also: **LJUST$**, **CENTER$**, **JUSTIFY$**, **HEADER$**. **FILLARRAY**, **FILLFROM**, **JUSTIFYARRAY**, **JUSTIFYFROM**

RNDSTR$ Function

Library: STRLIB.TRC

Syntax: RNDSTR$ (*strex*, *numex*)

Usage: `LET a$ = RNDSTR$ (chars$, length)`

Summary: Returns a string of the specified `length` composed of characters randomly chosen from the characters comprising the value of `chars$`.

Details: The **RNDSTR$** function returns a randomly generated string. The length of the string will be determined by the value of `length`, and the characters in the string will be drawn randomly from the characters comprising the value of `chars$`.

If a single character appears more than once in the value of `chars$`, the probability of that character appearing in the resulting string will be increased appropriately.

The **RNDSTR$** function is useful for creating names for temporary files. It is also useful for creating strings to test string handling algorithms.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF RndStr$

LET first$ = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
LET rest$ = first$ & "0123456789"

DO
   LET fname$ = RndStr$(first$,1) & RndStr$(rest$,7)
   WHEN ERROR IN
      OPEN #1: NAME fname$, CREATE NEW
      EXIT DO
   USE
      ! Do nothing
   END WHEN
LOOP
PRINT "File created: "; fname$

GET KEY k
END
```

uses the **RNDSTR$** function to create a temporary file name and then uses that name to create the file. Note that the name begins with a letter and then contains seven characters which may be letters or digits.

Exceptions: None

ROMAN$ Function

Library: STRLIB.TRC

Syntax: ROMAN$ (*numex*)

Usage: `LET romannum$ = ROMAN$ (number)`

Summary: Returns the string representation of its numeric argument in Roman numerals.

Details: The **ROMAN$** function returns the string representation of the value of `number` as Roman numerals. For example, if passed a value of 117, the **ROMAN$** function would return the string value:

```
CXVII
```

As you can see, the Roman numerals will be represented entirely in uppercase letters.

The Roman representation of a negative number will begin with a minus sign.

Since Roman numerals may only represent whole numbers, the **ROMAN$** function will generate an error if you pass it a non-integer value.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DECLARE DEF Roman$

LET year$ = Roman$(1900+Int(Date/1000))

PRINT "Copyright (c) "; year$; " by True BASIC, Inc."

END
```

produces the following output:

```
Copyright (c) MCMXCIV by True BASIC, Inc."
```

Exceptions: None

See also: **ENGLISHNUM$**

SHORTDATE$ Function

Library: STRLIB.TRC

Syntax: SHORTDATE$ (*strex*)

Usage: `LET a$ = SHORTDATE$ (adate$)`

Summary: Returns the date represented by the value of `adate$` in string form consisting of the day of the month, the month name, and the year in abbreviated format.

Details: The **SHORTDATE$** function takes as its argument a date in the format produced by the **DATE$** function and returns that date in abbreviated, but legible, form. This abbreviated form consists of the day of the month, the three-character abbreviation of the month name, and the last two digits of the year.

The value of `adate$` must represent a date in the same format produced by the **DATE$** function. That is, the value of `adate$` must be a string in the form “YYYYMMDD”, where YYYY represents the year, MM the month, and DD the day. If `adate$` does not represent such a value, then the **SHORTDATE$** function generates an error.

To return a date in an expanded format, use the **NICEDATE$** function.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF ShortDate$
PRINT ShortDate$("19971015")
END
```

produces the following output:

```
15 Oct 97
```

Exceptions: 1 Bad date given to ShortDate$: 00000000

See also: **DATE$**, **DATE**, **SHORTDATE$**, **WEEKDAY$**, **WEEKDAY**, **TODAY$**, **TIME**, **TIME$**, **NICETIME$**, **NOW$**

SUPERVAL Subroutine

Library: STRLIB.TRC

Syntax: CALL SUPERVAL (*strarrarg*, *strex*, *numvar*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL SUPERVAL (table$(), expression$, result)`

Summary: Evaluates the expression represented by the contents of `expression$` and returns the resulting value in `result`.

Details: The **SUPERVAL** subroutine evaluates the numeric expression represented by the value of `expression$` and returns the resulting value in `result`.

The value of `expression$` must represent a numeric expression which is valid under the rules of True BASIC. This expression may contain both numeric constants and variables.

You can use expressions like "a = 2" or "length, width, height = 1" to create variables and set their values.

Like the numeric constants, the variables used in `expression$` must follow True BASIC's rules for the formation of variable names. These variables are not connected with the variables used in your program code; they form their own variable pool which is created and used exclusively by the **SUPERVAL** subroutine.

The **SUPERVAL** subroutine uses the `table$` array to manage this pool of variables. You don't need to worry about maintaining the `table$` array; just pass any one-dimensional array as `table$`, and the **SUPERVAL** subroutine will do the rest. However, since the `table$` array is used to store the variable pool, you should be sure to pass the same `table$` array to each invocation of the **SUPERVAL** subroutine which you would like to have access to the variables in that pool.

The value of `expression$` may incorporate any of the following operators:

Operators Available to SUPERVAL

+	-	*	/	^	(	)

In addition, the value of `expression$` may incorporate any of the following numeric functions:

Functions Available to SUPERVAL

SIN	COS	TAN	ATN	SQR
SINH	COSH	TANH	ASIN	ACOS
SEC	CSC	COT	MAXNUM	EPS
PI	SGN	ABS	RAD	DEG
LOG	LOG2	LOG10	EXP	RND
INT	ROUND	IP	FP	CEIL
	DATE		TIME	

Note that numeric functions requiring two arguments, including the two-argument form of the **ROUND** function, are not available for use in the value of `expression$`.

Example: The following program:

```
LIBRARY "STRLIB.TRC"
DIM vars$(1)

CALL SuperVal(vars$(), "a = 3^2", result)
CALL SuperVal(vars$(), "b = Sqr(a)", result)
PRINT result

DO
   LINE INPUT PROMPT "Expression (0 to quit): ": expr$
   IF expr$ = "0" then EXIT DO
   CALL SuperVal(vars$(), expr$, result)
   PRINT result
LOOP

END
```

produces output similar to the following:

```
 3
Expression (0 to quit): a
 9
Expression (0 to quit): b
 3
Expression (0 to quit): a + b
 12
Expression (0 to quit): c = a/(b*2)
 1.5
Expression (0 to quit): a * Pi
 28.274334
Expression (0 to quit): 0
```

Exceptions:

1	SuperVal string isn't a proper expression.
2	Unknown function or variable in SuperVal.
3	Bad variable name in SuperVal: *name*
3002	Negative number to non-integral power.
3003	Zero to negative power.
3004	LOG of number <= 0.
3005	SQR of negative number.
4001	VAL string isn't a proper number.

See also: **EVAL**, **VAL**, **LVAL**, **DOLLARVAL**

TODAY$ Function

Library: STRLIB.TRC

Syntax: TODAY$

Usage: `LET a$ = TODAY$`

Summary: Returns the current date in string form consisting of the name of the day of the week, the month name, the day of the month, and the year.

Details: The **TODAY$** function returns the current date in string form. The format begins with the name of the weekday, followed by the full name of the month, the day of the month, and the full year.

For instance, the **TODAY$** function might return:

```
Tuesday, June 14, 1994
```

If the current computer hardware is not able to report the date, then the **TODAY$** function generates an error.

The **TODAY$** function obtains its results from your operating system, which in turn obtains its results from your computer's internal clock. If you find that the **TODAY$** function is returning erroneous results, you most likely have to reset your computer's internal clock.

Example: None.

Exceptions:

1	Bad date given to NiceDate$: 00000000
1	Bad date given to WeekDay: 00000000
1	Bad date given to WeekDay$: 00000000

See also: **DATE$**, **DATE**, **NICEDATE$**, **SHORTDATE$**, **WEEKDAY$**, **WEEKDAY**, **TIME**, **TIME$**, **NICETIME$**, **NOW$**

UNIQ$ Function

Library: STRLIB.TRC

Syntax: UNIQ$ (*strex*)

Usage: `LET a$ = UNIQ$ (text$)`

Summary: Returns the set of characters contained within the value of `text$`.

Details: The **UNIQ$** function returns a string containing the set of characters contained in the value of its argument.

That is, the **UNIQ$** function returns a string which contains one of each character which appears in the value of its argument. The characters will be organized within the resulting string in ascending order by their ASCII codes. Thus, uppercase letters will be listed before all lowercase letters.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Uniq$
LET s$ = "Now is the time for all good men"
PRINT s$
PRINT Uniq$(s$)
END
```

produces the following output:

```
Now is the time for all good men
 Nadefghilmnorstw
```

Exceptions: None

See also: **CHARUNION$**, **CHARINT$**, **CHARDIFF$**, **CHARS$**

UPPER$ Function

Library: STRLIB.TRC

Syntax: UPPER$

Usage: `LET set$ = UPPER$`

Summary: Returns the set of uppercase alphabetic characters.

Details: The **UPPER$** function returns a string containing the set of characters representing the letters of the alphabet, in uppercase only, arranged in ascending order according to their ASCII codes. (For a table of the ASCII codes and their corresponding characters, see Appendix A.)

That is, it returns the equivalent of the string constant:

```
"ABCDEFGHIJKLMNOPQRSTUVWXYZ"
```

Example: None.

Exceptions: None

See also: **LOWER$**, **LETTERS$**, **DIGITS$**, **ALPHANUM$**, **PUNCT$**, **CONTROL$**

WEEKDAY Function

Library: STRLIB.TRC

Syntax: WEEKDAY (*strex*)

Usage: `LET a = WEEKDAY (adate$)`

Summary: Returns the number of the weekday on which the specified date falls.

Details: The **WEEKDAY** function takes as its argument a date in the format produced by the **DATE$** function and returns the number of the day of the week on which that date falls.

The **WEEKDAY** function returns a number between 1 and 7, inclusive, where 1 indicates Sunday and 7 indicates Saturday.

The value of `adate$` must represent a date between the years 1901 and 2099, inclusive, in the same format produced by the **DATE$** function. That is, the value of `adate$` must be a string in the form “YYYYMMDD”, where YYYY represents the year, MM the month, and DD the day. If `adate$` does not represent such a value, then the **WEEKDAY** function generates an error.

To return the full name of the day of the week on which a particular date falls, use the **WEEKDAY$** function.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Weekday
PRINT Weekday("19971015")
END
```

produces the following output:

```
 4
```

Exceptions: 1 Bad date given to WeekDay: 00000000

See also: **DATE$**, **DATE**, **NICEDATE$**, **SHORTDATE$**, **TODAY$**, **WEEKDAY$**, **TIME**, **TIME$**, **NICETIME$**, **NOW$**

WEEKDAY$ Function

Library: STRLIB.TRC

Syntax: WEEKDAY$ (*strex*)

Usage: `LET a$ = WEEKDAY$ (adate$)`

Summary: Returns the full name of the weekday on which the specified date falls.

Details: The **WEEKDAY$** function takes as its argument a date in the format produced by the **DATE$** function and returns the full name of the day of the week on which that date falls.

The value of `adate$` must represent a date between the years 1901 and 2099, inclusive, in the same format produced by the **DATE$** function. That is, the value of `adate$` must be a string in the form “YYYYMMDD”, where YYYY represents the year, MM the month, and DD the day. If `adate$` does not represent such a value, then the **WEEKDAY$** function generates an error.

To return the number of the day of the week on which a particular date falls, use the **WEEKDAY** function.

Example: The following program:

```
LIBRARY "StrLib.trc"
DECLARE DEF Weekday$
PRINT Weekday$("19971015")
END
```

produces the following output:

```
Wedneadsy
```

Exceptions: 1 Bad date given to WeekDay$: 00000000

See also: **DATE$**, **DATE**, **NICEDATE$**, **SHORTDATE$**, **TODAY$**, **WEEKDAY**, **TIME**, **TIME$**, **NICETIME$**, **NOW$**

Sorting and Searching Libraries

The library file SORTLIB.TRC contains several sorting and searching utilities. Each sorting and searching subroutine comes in two forms, one for numbers and one for strings. The name of the subroutine ends with an "N" for numbers, and in "S" for strings.

The two subroutines **SORTN** and **SORTS** perform ordinary in-place sorts. The two subroutines **PSORTN** and **PSORTS** perform indexed (or pointer) sorts.

The two subroutines **CSORTN** and **CSORTS** perform sorting according to a relation specified by the programmer. The two subroutines **CPSORTN** and **CPSORTS** perform indexed (or pointer) sorts according to a relation defined by the programmer.

The four subroutines **CSEARCHN**, **CSEARCHS**, **SEARCHN**, and **SEARCHS** search lists (numeric or string) for a match. **SEARCHN** and **SEARCHS** use the ordinary relational operator "=". **CSEARCHN** and **CSEARCHS** perform searches according to a relation specified by the programmer.

CSORTN, CPSORTN, and CSEARCHN call a subroutine COMPAREN, which is included in SortLib.tru. It is currently coded to produce the usual ascending sort. If you require a different sorting relation, you can proceed in one of two ways. First, you can make the changes in the subroutine COMPAREN in SortLib.tru, and then recompile SortLib.tru. Second, you can include your own version of COMPAREN following the END statement in your main program; this definition takes precedence over the one in the library file.

CSORTS, CPSORTS, and CSEARCHS performing sorts and searches using special ordering relations specified by calling one of several relation-specifying subrouintes before invoking the sort. These special subroutine calls include:

Sort_Off	Sort using ASCII sorting order and entire string
Sort_ObserveCase	Treat upper- and lowercase as different (default)
Sort_IgnoreCase	Treat upper- and lowercase as equivalent
Sort_NiceNumbers_on	See the header of SortLib.tru for definitions
Sort_NiceNumbers_off	Ditto (default)
Sort_NoKeys	Sort using the entire string
Sort_OneKey	Sort on the substring field specified
Sort_TwoKeys	Sort on the two substring fields specified

CSEARCHN and CSEARCHS require the list to have been previously sorted using the same relations; i.e., use the same COMPAREN for CSEARCHN, and the same options for CSEARCHS as for CSORTS.The two subroutines **REVERSEN** and **REVERSES** simply reverse the order of the elements in the numeric or string array. That is, the first element will become the last, and so on.

CPSORTN Subroutine

Library: SORTLIB.TRC

Syntax: CALL CPSORTN (*numarrarg*, *numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL CPSORTN (values(), indices())`

Summary: Performs a pointer sort on the values stored in `values` and stores the pointers, or indices, to the elements in `indices` in the order specified by a customized comparison routine.

Details: The **CPSORTN** subroutine performs a "pointer sort" on the values stored in the numeric array `values`. Pointer sorts do not actually rearrange the values in the array which they are sorting, rather they create a second array that contains the first array's indices arranged in the order of the sorted values. The **CPSORTN** subroutine returns this array of indices as `indices`.

For a more detailed discussion of pointer sorts, see the **PSORTN** subroutine later in this chapter.

The **PSORTN** subroutine compares elements based upon the standard relational operators in order to create a list of indices that represent the values sorted into ascending order. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CPSORTN** subroutine allows you to specify a particular comparison that will be used to determine the way in which the items will be ordered.

Note that the **CPSORTN** subroutine sorts the entire `values` array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the resulting `indices` array will contain the indices of 900 zero-valued elements of `values` merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM name$(6), grade(6), indices(6)
MAT READ name$, grade
DATA Kollwitz, Hu, Starr, Ransom, White, Sankar
DATA 75,      93, 95,    68,     84,    88

CALL CPSortN(grade, indices)   ! Sort by grades
FOR i = 1 to 6
    LET j = indices(i)
    PRINT name$(j); grade(j)
NEXT i

END

SUB CompareN (a, b, compflag)
    IF a > b then
       LET compflag = -1
    ELSEIF a = b then
       LET compflag =  0
    ELSE
       LET compflag =  1
    END IF
END SUB
```

performs a pointer sort on a set of parallel arrays and uses the results to print both arrays sorted into descending order by grade. The result is the same as that of using PSORTN followed by CALL ReverseN (indicies).

Exceptions: None

See also: **CPSORTS**, **PSORTN**, **SORTN**

CPSORTS Subroutine

Library: SORTLIB.TRC

Syntax: CALL CPSORTS (*strarrarg*, *numarrarg*)

strarrarg:: *strarr*
strarr bowlegs

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL CPSORTS (values$(), indices())`

Summary: Performs a pointer sort on the values stored in `values$` and stores the pointers, or indices, to the elements in `indices` in the order specified by the programmer.

Details: The **CPSORTS** subroutine performs a “pointer sort” on the values stored in the string array `values$`. Pointer sorts do not actually rearrange the values in the array which they are sorting, rather they create a second array which contains the first array’s indices arranged in the order of the sorted values. The **CPSORTS** subroutine returns this array of indices as `indices`.

For a more detailed discussion of pointer sorts, see the **PSORTS** subroutine later in this chapter.

The **PSORTS** subroutine compares elements based upon the standard relational operators in order to create a list of indices that represent the values sorted into ascending order. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CPSORTS** subroutine allows you to specify the comparison that will be used to determine the way in which the items will be ordered.

Note that the **CPSORTS** subroutine sorts the entire `values$` array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the resulting `indices` array will contain the indices of 900 null-valued elements of `values$` merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM name$(6), grade(6), indices(6)
MAT READ name$, grade
DATA Kollwitz, Hu, Starr, Ransom, White, Sankar
DATA 75,      93, 95,    68,     84,    88

CALL Sort_IgnoreCase
CALL CPSortS(name$, indices)   ! Sort by grades
FOR i = 1 to 6
    LET j = indices(i)
    PRINT name$(j); grade(j)
NEXT i

END
```

performs a case-blind pointer sort on a set of parallel arrays and uses the results to print both arrays sorted by name.

Exceptions: None

See also: **CPSORTN**, **PSORTS**, **SORTS**

CSEARCHN Subroutine

Library: SORTLIB.TRC

Syntax: CALL CSEARCHN (*numarrarg*, *numex*, *numvar*, *numvar*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL CSEARCHN (array(), number, index, found)`

Summary: Searches `array` for the value `number` utilizing a user-defined comparison and returns `found` as a non-zero value if it is found. `Index` reports the subscript value of `number` within `array`.

Details: The **CSEARCHN** subroutine searches through the numeric array `array` for an element with the value `number` and returns the subscript of its location in `index`. This search is performed using a customized comparison subroutine defined by the programmer.

The **SEARCHN** subroutine compares elements based upon the standard relational operators in order to locate the value `number` within `array`. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CSEARCHN** subroutine requires that you have sorted the array using `CSORTN`, and that you continue to use the same `CompareN` subroutine.

It is your responsibility to ensure that the behavior of the `CompareN` subroutine is well-defined and bug-free. If your `CompareN` subroutine is not well-behaved, the search results may not be valid.

You may define `CompareN` in the main program file.

Since the **CSEARCHN** subroutine uses a binary search algorithm, the array must be sorted into ascending order (perhaps through an invocation of the **CSORTN** subroutine) before being passed to the **CSEARCHN** subroutine. In general, the **CSEARCHN** subroutine should utilize the same form of the `CompareN` subroutine used by the **CSORTN** subroutine which sorted the array.

If the value of `number` exists in `array`, the value of `found` is set to some non-zero value and the value of `index` is set equal to the subscript of the element which contains it.

If the value of `number` cannot be located in `array`, the value of `found` is set equal to zero and the value of `index` is set equal to the subscript of the element in which the value of `number` would have been stored if it had been present. In other words, the value of `index` is set to one subscript value past the location of the greatest value which is less than `number`. If `number` is greater than every element in `array`, the value of `index` will be returned equal to `array`'s upper bound plus 1.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(100)
RANDOMIZE
FOR i = 1 to 100
    LET array(i) = Int(100*Rnd) + 1
NEXT i
CALL CSortN(array)

DO
   INPUT PROMPT "Search value (0 to quit): ": number
   IF number <= 0 then EXIT DO
   CALL CSearchN(array,number,i,found)
   IF found <> 0 then
      PRINT "Found: "; array(i)
   ELSE
      PRINT "Not found."
   END IF
LOOP

END

SUB CompareN (a, b, compflag)
    IF a > b then
       LET compflag = -1
    ELSEIF a = b then
       LET compflag =  0
    ELSE
       LET compflag =  1
    END IF
END SUB
```

sorts a list of 20 random numbers between 1 and 100 into descending order and allows the user to search the results.

Exceptions: None

See also: **CSORTN**, **SEARCHN**, **CSEARCHS**, **CSORTS**

CSEARCHS Subroutine

Library: SORTLIB.TRC

Syntax: CALL CSEARCHS (*strarrarg*, *strex*, *numvar*, *numvar*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL CSEARCHS (array$(), string$, index, found)`

Summary: Searches `array$` for the value `string$` utilizing a user-specified relation and returns `found` as a non-zero value if it is found. `Index` reports the subscript value of `string$` within `array$`.

Details: The **CSEARCHS** subroutine searches through the string array `array$` for an element with the value `string$` and returns the subscript of its location in `index`. This search is performed using the relations specified by the programmer.

The **SEARCHS** subroutine compares elements based upon the standard relational operators in order to locate the value `string$` within `array$`. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CSEARCHS** subroutine allows you to specify the comparison that will be used to locate the items.

Since the **CSEARCHS** subroutine uses a binary search algorithm, the array must be sorted into ascending order (perhaps through an invocation of the **CSORTS** subroutine) before being passed to the **CSEARCHS** subroutine. In general, the **CSEARCHS** subroutine should use the same options used by the **CSORTS** subroutine which sorted the array.

If the value of `string$` exists in `array$`, the value of `found` is set to some non-zero value and the value of `index` is set equal to the subscript of the element which contains it.

If the value of `string$` cannot be located in `array$`, the value of `found` is set equal to zero and the value of `index` is set equal to the subscript of the element in which the value of `string$` would have been stored if it had been present. In other words, the value of `index` is set to one subscript value past the location of the greatest value which is less than `string$`. If `string$` is greater than every element in `array$`, the value of `index` will be returned equal to `array$`'s upper bound plus 1.

Example: The following program:

```
! Sort by last 3 letters, then search for same.
!
DIM array$(10)
MAT READ array$
DATA operculum, partlet, pettifog, grisette, douceur
DATA pollex, sannup, duende, keeshond, maccaboy

CALL Sort_OneKey (4, 6)
CALL CSortS(array$)
DO
   INPUT PROMPT "Search string (aaa to quit): ": string$
   IF string$ = "aaa" then EXIT DO
   CALL CSearchS(array$,string$,i,found)
   IF found<>0 then
      PRINT "Found: "; array$(i)
   ELSE
      PRINT "Not found."
   END IF
LOOP
END
```

sorts a list of string data by characters 4 through 6 in each element and then allows the user to search the list based on these same characters in an element.

Exceptions: None

See also: **CSORTS**, **SEARCHS**, **CSEARCHN**, **CSORTN**

CSORTN Subroutine

Library: SORTLIB.TRC

Syntax: CALL CSORTN (*numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL CSORTN (array())`

Summary: Sorts the specified numeric array using the customized comparison routine named `CompareN`.

Details: The **CSORTN** subroutine sorts the elements of the specified numeric array into the order determined by a customized comparison subroutine.

The **SORTN** subroutine compares elements based upon the <= relational operator in order to create a list sorted into ascending order. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CSORTN** subroutine allows you to define a particular comparison that will be used to determine the ordering of the items. You do so by defining an external subroutine named `CompareN` as in the following example:

The **CSORTN** subroutine performs an “in-place” sort, which means that it uses very little memory over and above that already occupied by the array itself.

Although it is of little consequence, you may also be interested to know that the sorting algorithm used by the **CSORTN** subroutine is not stable; if you require a stable sort, use the **CPSORTN** subroutine instead.

The sorting algorithm used is an optimized quick sort, which makes the **CSORTN** routine a very efficient, general-purpose sorting routine. Note, however, that since the **CSORTN** subroutine calls the `CompareN` subroutine for each comparison, it is not as fast as the **SORTN** subroutine.

Note that the **CSORTN** subroutine sorts the entire array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the array will have 900 zeroes merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(100)
RANDOMIZE
FOR i = 1 to 100
    LET array(i) = Int(100*Rnd) + 1
NEXT i
CALL CSortN(array)
MAT PRINT array

END

SUB CompareN (a, b, compflag)
    IF a > b then
       LET compflag = -1
    ELSEIF a = b then
       LET compflag =  0
    ELSE
       LET compflag =  1
    END IF
END SUB
```

generates an array of 100 random numbers, sorts it into descending order, and prints the sorted result on the screen.

Exceptions: None

See also: **CSORTS**, **CPSORTN**, **SORTN**, **REVERSEN**

CSORTS Subroutine

Library: SORTLIB.TRC

Syntax: CALL CSORTS (*strarrarg*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL CSORTS (array())`

Summary: Sorts the specified string array using the customized comparison specified by the programmer.

Details: The **CSORTS** subroutine sorts the elements of the specified string array into the order determined by a customized comparison.

The **SORTS** subroutine compares elements based upon the <= relational operator in order to create a list sorted into ascending order. While this is useful for the vast majority of circumstances, you may occasionally need to specify a different comparison.

The **CSORTS** subroutine allows you to specify the comparison that will be used to determine the ordering of the items.

The **CSORTS** subroutine performs an "in-place" sort, which means that it uses very little memory over and above that already occupied by the array itself.

Although it is of little consequence, you may also be interested to know that the sorting algorithm used by the **CSORTS** subroutine is not stable; if you require a stable sort, use the **CPSORTS** subroutine instead.

The sorting algorithm used is an optimized quick sort, which makes the **CSORTS** routine a very efficient, general-purpose sorting routine. Note, however, that since the **CSORTS** subroutine calls the `CompareS` subroutine for each comparison, it is not as fast as the **SORTS** subroutine.

Note that the **CSORTS** subroutine sorts the entire array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the array will have 900 null strings merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"
LIBRARY "CompNum.TRC"

DIM array$(100)
RANDOMIZE
FOR i = 1 to 100
    LET array$(i) = "Item " & Str$(Int(100*Rnd) + 1)
NEXT i
CALL Sort_NiceNumbers_on
CALL CSortS(array$)
MAT PRINT array$

END
```

generates an array of 100 strings containing numeric values, sorts it using the version of `CompareS` contained in the COMPNUM library file, and prints the sorted result on the screen.

Exceptions: None

See also: **CSORTN**, **CPSORTS**, **SORTS**, **REVERSES**

PSORTN Subroutine

Library: SORTLIB.TRC

Syntax: CALL PSORTN (*numarrarg*, *numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL PSORTN (values(), indices())`

Summary: Performs a pointer sort on the values stored in `values` and stores the pointers, or indices, to the elements in `indices` in sorted order.

Details: The **PSORTN** subroutine performs a "pointer sort" on the values stored in the numeric array `values`. Pointer sorts do not actually rearrange the values in the array which they are sorting, rather they create a second array which contains the first array's indices arranged in the order of the sorted values. The **PSORTN** subroutine returns this array of indices as `indices`.

For example, if `values` contained the following items:

```
10   12   23   14   -8   11   6
```

the resulting `indices` array would contain the following items:

```
5   7   1   6   2   4   3
```

but the items in `values` will still be in their original order:

```
10   12   23   14   -8   11   6
```

Notice that you can therefore print the elements of `values` in sorted order with code similar to the following:

```
FOR i = Lbound(indices) to Ubound(indices)
    PRINT values(indices(i))
NEXT i
```

Because they do not change the ordering of information in the `values` array, pointer sorts are particularly useful when working with "parallel arrays."

Note that the **PSORTN** subroutine sorts the entire `values` array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the resulting `indices` array will contain the indices of 900 zero-valued elements of `values` merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM name$(6), grade(6), indices(6)
MAT READ name$, grade
DATA Kollwitz, Hu, Starr, Ransom, White, Sankar
DATA 75,       93, 95,    68,     84,    88

CALL PSortN(grade, indices)   ! Sort by grades
FOR i = 1 to 6
    LET j = indices(i)
    PRINT name$(j); grade(j)
NEXT i

END
```

performs a pointer sort on a set of parallel arrays and uses the results to print both arrays sorted by grades.

Exceptions: None

See also: **PSORTN**, **CPSORTS**, **SORTS**

PSORTS Subroutine

Library: SORTLIB.TRC

Syntax: CALL PSORTS (*strarrarg*, *numarrarg*)

strarrarg:: *strarr*
strarr bowlegs

numarrarg:: *numarr*
numarr bowlegs

Usage: CALL PSORTS (values$(), indices())

Summary: Performs a pointer sort on the values stored in `values$` and stores the pointers, or indices, to the elements in `indices` in sorted order.

Details: The **PSORTS** subroutine performs a “pointer sort” on the values stored in the string array `values$`. Pointer sorts do not actually rearrange the values in the array which they are sorting, rather they create a second array which contains the first array’s indices arranged in the order of the sorted values. The **PSORTS** subroutine returns this array of indices as `indices`.

For example, if `values$` contained the following items:

```
bat   zoo   cat   ant   dog   pig
```

the resulting `indices` array would contain the following items:

```
4   1   3   5   6   2
```

but the items in `values$` will still be in their original order:

```
bat   zoo   cat   ant   dog   pig
```

Notice that you can therefore print the elements of `values$` in sorted order with code similar to the following:

```
FOR i = Lbound(indices) to Ubound(indices)
    PRINT values$(indices(i))
NEXT i
```

Because they do not change the ordering of information in the `values$` array, pointer sorts are particularly useful when working with “parallel arrays.”

Note that the **PSORTS** subroutine sorts the entire `values$` array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the resulting `indices` array will contain the indices of 900 null-valued elements of `values$` merged into the sorted result.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM name$(6), grade(6), indices(6)
MAT READ name$, grade
DATA Kollwitz, Hu, Starr, Ransom, White, Sankar
DATA 75,      93, 95,   68,    84,    88

CALL PSortS(grade$, indices)   ! Sort by grades
FOR i = 1 to 6
    LET j = indices(i)
    PRINT name$(j); grade(j)
NEXT i

END
```

performs a pointer sort on a set of parallel arrays and uses the results to print both arrays sorted by name.

Exceptions: None

See also: **PSORTN**, **CPSORTS**, **SORTS**

REVERSEN Subroutine

Library: SORTLIB.TRC

Syntax: CALL REVERSEN (*numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL REVERSEN (array())`

Summary: Reverses the order of the elements within `array`.

Details: The **REVERSEN** subroutine reverses the order of the elements stored within the specified numeric array. In other words, it swaps the first and last elements, the second and next-to-last, and so forth.

Although it can be used on any numeric array, the **REVERSEN** subroutine is most often used to reverse the results of the **SORTN** or **CSORTN** subroutines to produce a list sorted in descending order. It can also be used to reverse the pointer list produced by **PSORTN**, **CPSORTN**, **PSORTS** or **CPSORTS**.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(20)
FOR i = 1 to 20
    LET array(i) = Int(100*Rnd) + 1
NEXT i
CALL SortN(array)
CALL ReverseN(array)
MAT PRINT array

END
```

generates an array of random values between 1 and 100 and prints it sorted into descending order.

Exceptions: None

See also: **SORTN**, **CSORTN**, **REVERSES**

REVERSES Subroutine

Library: SORTLIB.TRC

Syntax: CALL REVERSES (*strarrarg*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL REVERSES (array$())`

Summary: Reverses the order of the elements within `array$`.

Details: The **REVERSES** subroutine reverses the order of the elements stored within the specified string array. In other words, it swaps the first and last elements, the second and next-to-last, and so forth.

Although it can be used on any string array, the **REVERSES** subroutine is most often used to reverse the results of the **SORTS** or **CSORTS** subroutines to produce a list sorted in descending order.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array$(20)
FOR i = 1 to 20
    LET array$(i) = Chr$(Int(26*Rnd) + 65)
NEXT i
CALL SortS(array$)
CALL ReverseS(array$)
MAT PRINT array$

END
```

generates an array of random uppercase letters and prints it sorted into descending order.

Exceptions: None

See also: **SORTS**, **CSORTS**, **REVERSEN**

SEARCHN Subroutine

Library: SORTLIB.TRC

Syntax: CALL SEARCHN (*numarrarg*, *numex*, *numvar*, *numvar*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL SEARCHN (array(), number, index, found)`

Summary: Searches `array` for the value `number` and returns `found` as a non-zero value if it is found. `Index` reports the subscript value of `number` within `array`.

Details: The **SEARCHN** subroutine searches through the numeric array `array` for an element with the value `number` and returns the subscript of its location in `index`.

Since the **SEARCHN** subroutine uses a binary search algorithm, the array must be sorted into ascending order (perhaps through an invocation of the **SORTN** subroutine) before being passed to the **SEARCHN** subroutine.

If the value of `number` exists in `array`, the value of `found` is set to some non-zero value and the value of `index` is set equal to the subscript of the element which contains it.

If the value of `number` cannot be located in `array`, the value of `found` is set equal to zero and the value of `index` is set equal to the subscript of the element in which the value of `number` would have been stored if it had been present. In other words, the value of `index` is set to one subscript value past the location of the greatest value which is less than `number`. If `number` is greater than every element in `array`, the value of `index` will be returned equal to `array`'s upper bound plus 1.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(20)
FOR i = 1 to 20
    LET array(i) = Int(100*Rnd) + 1
NEXT i
CALL SortN(array)

DO
   INPUT PROMPT "Enter a number 1 to 100 (0 to quit): ": number
   IF number <= 0 then EXIT DO
   CALL SearchN(array, number, index, found)
   IF found <> 0 then
      PRINT "Found at"; index
   ELSE
      PRINT "Not found"
   END IF
LOOP

END
```

generates an array of random values between 1 and 100 and allows the user to search it.

Exceptions: None

See also: **SORTN**, **SEARCHS**, **CSEARCHN**, **CSORTN**

SEARCHS Subroutine

Library: SORTLIB.TRC

Syntax: CALL SEARCHS (*strarrarg*, *strex*, *numvar*, *numvar*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL SEARCHS (array$(), string$, index, found)`

Summary: Searches `array$` for the value `string$` and returns `found` as a non-zero value if it is found. `Index` reports the subscript value of `string$` within `array`.

Details: The **SEARCHS** subroutine searches through the string array `array$` for an element with the value `string$` and returns the subscript of its location in `index`.

Since the **SEARCHS** subroutine uses a binary search algorithm, the array must be sorted into ascending order (perhaps through an invocation of the **SORTS** subroutine) before being passed to the **SEARCHS** subroutine.

If the value of `string$` exists in `array$`, the value of `found` is set to some non-zero value and the value of `index` is set equal to the subscript of the element which contains it.

If the value of `string$` cannot be located in `array$`, the value of `found` is set equal to zero and the value of `index` is set equal to the subscript of the element in which the value of `string$` would have been stored if it had been present. In other words, the value of `index` is set to one subscript value past the location of the greatest value which is less than `string$`. If `string$` is greater than every element in `array$`, the value of `index` will be returned equal to `array$`'s upper bound plus 1.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array$(20)
FOR i = 1 to 20
    LET array$(i) = Chr$(Int(26*Rnd) + 65)
NEXT i
CALL SortS(array$)

DO
   INPUT PROMPT "Enter an uppercase letter (a to quit): ": string$
   IF string$ = "a" then EXIT DO
   CALL SearchS(array$, string$, index, found)
   IF found <> 0 then
      PRINT "Found at"; index
   ELSE
      PRINT "Not found"
   END IF
LOOP

END
```

generates an array of random uppercase letters and allows the user to search it.

Exceptions: None

See also: **SORTS**, **SEARCHN**, **CSEARCHS**, **CSORTS**

SORTN Subroutine

Library: SORTLIB.TRC

Syntax: CALL SORTN (*numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL SORTN (array())`

Summary: Sorts the specified numeric array using a quick sort.

Details: The **SORTN** subroutine sorts the elements of the specified numeric array into ascending order. Thus, the array element with the lowest value will be found in the first element of `array` after the sort, and the array element with the highest value will be found in the last element of `array`.

The **SORTN** subroutine performs an "in-place" sort, which means that it uses very little memory over and above that already occupied by the array itself.

The sorting algorithm used by the **SORTN** subroutine is not stable; if you require a stable sort, use the **PSORTN** subroutine instead.

The sorting algorithm used is an optimized quick sort, which makes the **SORTN** routine a very efficient, general-purpose sorting routine.

Note that the **SORTN** subroutine sorts the entire array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the array will have 900 zeroes merged into the sorted result.

To sort an array into descending order, use the **REVERSEN** subroutine to reverse the results of the **SORTN** subroutine.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array(1000)
RANDOMIZE
FOR i = 1 to 1000
    LET array(i) = Rnd
NEXT i
CALL SortN(array)
MAT PRINT array
END
```

generates an array of 1000 random numbers, sorts it, and prints the sorted result on the screen.

Exceptions: None

See also: **SORTS**, **CSORTN**, **PSORTN**, **CPSORTN**, **REVERSEN**

SORTS Subroutine

Library: SORTLIB.TRC

Syntax: CALL SORTS (*strarrarg*)

strarrarg:: *strarr*
strarr bowlegs

Usage: `CALL SORTS (array$())`

Summary: Sorts the specified string array using a quick sort.

Details: The **SORTS** subroutine sorts the elements of the specified string array into ascending order. Thus, the array element with the lowest value will be found in the first element of `array` after the sort, and the array element with the highest value will be found in the last element of `array`.

The values of the elements will be compared as strings, which means that they are compared character by character on the basis of each character's numeric code. Thus, the string value `"Zebra"` will be considered less than the string value `"apple"`. This is particularly important when sorting strings which represent numeric constants, for the string value `"123"` will be considered less than the string value `"2"`, which can lead to unexpected results.

The **SORTS** subroutine performs an "in-place" sort, which means that it uses very little memory over and above that already occupied by the array itself.

The sorting algorithm used by the **SORTS** subroutine is not stable; if you require a stable sort, use the **PSORTS** subroutine instead.

The sorting algorithm used is an optimized quick sort, which makes the **SORTS** routine a very efficient, general-purpose sorting routine.

Note that the **SORTS** subroutine sorts the entire array. Thus, if you have only assigned values to the first 100 elements of a 1000-element array, the array will have 900 null strings merged into the sorted result.

To sort an array into descending order, use the **REVERSES** subroutine to reverse the results of the **SORTS** subroutine.

Example: The following program:

```
LIBRARY "SortLib.TRC"

DIM array$(1)
MAT INPUT array$(?)
CALL SortS(array$)
MAT PRINT array$
END
```

obtains an array of string values from the user, sorts it, and prints the sorted result on the screen.

Exceptions: None

See also: **SORTN**, **CSORTS**, **PSORTS**, **CPSORTS**, **REVERSES**

Graphics Libraries

This section describes subroutines for generating complicated graphical displays of data. The subroutines are contained in three library files:

BGLIB.TRC	for drawing pie charts, bar charts, and histograms; including the routines BARCHART, HISTOGRAM, IBEAM, MULTIBAR, MULTIHIST, PIECHART, and several ASK... and SET... routines for finding out about or setting attributes of graphs.
SGLIB.TRC	for plotting data and function values; including the routines ADDDATAGRAPH, ADDFGRAPH, ADDLSGRAPH, DATAGRAPH, FGRAPH, MANYDATAGRAPH, MANYFGRAPH, SORTSPOINTS, and many ASK... and SET... routines for finding out about or setting attributes of graphs
SGFUNC.TRC	for plotting values of functions that you define: ADDFGRAPH, FGRAPH, MANYFGRAPH

The graphics subroutines are described below, in alphabetical order.

ADDDATAGRAPH Subroutine

Library: SGLIB.TRC

Syntax: CALL ADDDATAGRAPH (*numarrarg*,, *numarrarg*, *numex*, *numex*, *strex*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL ADDDATAGRAPH (x(), y(), pstyle, lstyle, colors$)`

Summary: Draws another line graph of a set of data points over the current graph.

Details: The **ADDDATAGRAPH** subroutine draws a line graph of the set of data points whose coordinates are represented by the values of the `x` and `y` arrays over the current graph; it simply adds the new graph to the existing graph. Therefore, portions of the added data graph may lie off the graph.

The `x` array contains the points' x-coordinates, and the `y` array contains their y-coordinates. The coordinates in the two arrays are matched according to their subscripts; that is, the elements with subscripts of 1 within both arrays are interpreted as the coordinates of a single point, as are the elements with subscripts of 2, and so on. Thus, the `x` and `y` arrays must have the same upper and lower bounds, or an error will be generated.

The value of `pstyle` determines the point style that will be used to draw the data points which comprise the graph. The allowable values for `pstyle` are summarized in the following table:

Available Point Styles

Value of `pstyle`	Resulting Point
0	No point (display line only)
1	Dot
2	Plus sign
3	Asterisk
4	Circle
5	X
6	Box
7	Up triangle
8	Down triangle
9	Diamond
10	Solid Box
11	Solid up triangle
12	Solid down triangle
13	Solid diamond

The value of `lstyle` determines the line style that will be used to connect the data points which comprise the graph. The allowable values for `lstyle` are summarized in the following table:

Available Line Styles

Value of `lstyle`	**Resulting Line**
0	No line (display points only)
1	Solid line
2	Dashed line
3	Dotted line
4	Dash-dotted line

The graph is actually composed of a series of line segments connecting the data points. You can suppress the display of the data points by passing a value of 0 in `pstyle`, or you can suppress the display of the connecting line segments by passing a value of 0 in `lstyle`.

Note that the **ADDDATAGRAPH** subroutine draws and connects the points in the order in which they are stored in the `x` and `y` arrays. If your points are not stored in left to right order, you may wish to use the **SORTPOINTS** subroutine to order the points before passing them to the **ADDDATAGRAPH** subroutine.

The value of `colors$` determines the color that will be used to draw the new graph. It generally consists of a single color name (in any combination of uppercase or lowercase letters). The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

Note: the color “BACKGROUND” refers to the current background color.

The value of `colors$` may also contain a color number instead of a color name, allowing you to access any of the colors supported by the current computer system.

Note that the **ADDDATAGRAPH** subroutine assumes that a graph currently exists which has been created by an invocation of the **FGRAPH** or **DATAGRAPH** subroutine. The **ADDDATAGRAPH** subroutine simply adds the line representing the specified data points to the existing graph; it does not rescale the graph or redraw the labels or title. If you attempt to invoke the **ADDDATAGRAPH** subroutine when a suitable graph has not already been displayed, an error will be generated.

Example: The following program, SGData2.TRU, can be found in the directory TBDEMOS:

```
! SGData2   Chris' & Dave's cars' mileage.

! Both drove the same kind of car.  Notice that one car's mileage
! goes up and down depending on the season (winter is low).
! The other is less affected.  Also, notice a few erroneous
! data points!

LIBRARY "..\TBLibs\SGLib.trc"

DIM cmiles(0 to 200), cgallons(200), cmpg(200)
DIM dmiles(0 to 200), dgallons(200), dmpg(200)

CALL ReadData (cmiles, cgallons, cmpg)
CALL ReadData (dmiles, dgallons, dmpg)

CALL SetText ("Gas Mileage", "Miles Driven (Thousands)", "MPG")
CALL DataGraph (cmiles, cmpg, 0, 3, "red green yellow")
CALL AddDataGraph (dmiles, dmpg, 0, 1, "green")
```

```
GET KEY key

SUB ReadData (miles(), gallons(), mpg())

    READ miles(0)
    LET n = 0
    DO
       LET n = n + 1
       READ miles(n), gallons(n)
    LOOP until miles(n) = 0
    LET n = n - 1

    FOR i = 1 to n
        LET mpg(i) = (miles(i) - miles(i-1)) / gallons(i)
    NEXT i
    MAT redim mpg(n), miles(1:n)
    MAT miles = (1/1000) * miles

END SUB

! Chris's car

DATA 677.3                        ! first recorded

DATA 1104.9,9.5,  1567.6,9.3,  1869.7,6.7,  2179.5,6.0
DATA 2564.2,8.0,  2812.3,4.7,  3192.0,7.8,  3540.4,7.4
DATA 4044.4,10.2, 4317.5,5.8,  4747.8,8.7,  4946.2,3.7
DATA 5406.7,9.6,  5870.0,10.1, 6344.2,10.0, 6789.3,9.6
DATA 7208.1,9.1,  7624.8,9.6,  7786.6,3.2,  8244.3,9.4
DATA 8614.1,8.6,  9050.0,9.5,  9584,13.2,   9991.6,9.3
DATA 10389,9.4,   10804.4,9.1, 11216.1,10.3,11623.4,10.1
DATA 11970.4,9.54,12215.5,6.6, 12599.8,9.6, 12921.9,8.84
DATA 13238.1,7.7
DATA 13815.0,14.3,14170.0,8.8, 14531.0,8.3, 14880.9,9.0
DATA 15671,8.95, 16065,8.2,    16453,8.47,  16696,5.4
DATA 17144,8.8,  17568,9.1,    17997,8.65,  18450,9.3
DATA 18934,9.9,  19356,8.7,    19787,8.4,   20162,7.4
DATA 20572,8.25, 21025,8.8,    21345,9.0   ! did I read this right?
DATA 21713,5.0,  22043,6.6,    22514,9.2,  22968,9.6
DATA 23450,9.1,  23923,9.5,    24302,7.2,  24814,9.9
DATA 25272,9.1,  25738,9.0,    26128,7.7,  26603,8.9
DATA 26975,7.45, 27145,3.772
DATA 27523,7.36, 27834,6.4,    28266,8.4,  28652,8.3
DATA 29091,8.7,  29510,8.8,    29818,6.4,  30223,8.48
DATA 30626,8.9,  31056,8.24       ! ?
DATA 31410,8.16, 31786,8.6,    32161,8.9    ! ?
DATA 32550,9.2,  32941,9.045,  33302,9.3
DATA 0,0

! Dave's car

DATA 0                            ! full tank on delivery
DATA 272,6.35,   599,6.56,     924,7.44,   1281,7.56
DATA 1462,4.47,  1705,4.32,    2099,8.02,  2673,12.03
DATA 3090,8.76,  3537,8.6,     3991,9.28,  4419,8.73
DATA 4779,7.86,  5022,5.4,     5407,7.88,  5731,7.3
DATA 6049,7.04,  6388,7.61,    6836,8.56,  7204,7.87
DATA 7633,9.21,  8000,7.93,    8455,9.52,  8765,7.17
DATA 9188,9.2,   9578,9.21,    10111,13.7, 10551,10.13
DATA 10884,6.16, 11261,8.16,   11550,7.01, 11888,8.43
DATA 12255,6.79, 12690,8.11,   13237,10.8, 13563,6.47
DATA 14036,8.89, 14418,8.91,   14758,7.28, 15183,9.16
DATA 15757,11,   16394,12.75,  16752,7.95, 17108,6.83
DATA 17543,9.01, 17943,9.48,   18362,8.88, 18781,9.07
DATA 19179,8.83, 19361,4.63,   19600,6.07, 19898,6.57
DATA 0,0

END
```

produces a graph comparing the fuel economy of two cars.

Exceptions:

100	Graph's title is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
110	Data arrays have different bounds in DataGraph
117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also: **DATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**

ADDFGRAPH Subroutine

Library: SGFUNC.TRC, SGLIB.TRC

Syntax: CALL ADDFGRAPH (*numex*, *strex*)

Usage: `CALL ADDFGRAPH (style, color$)`

Summary: Draws another line graph of an externally defined function over the current graph.

Details: The **ADDFGRAPH** subroutine draws a line graph of the function F(x) over the current graph. The **ADDFGRAPH** subroutine does not change the scale of the current graph; it simply adds the new graph to the existing graph. Therefore, parts of the new function may be off the graph.

The function F(x) must be defined external to your main program. That is, it must be defined using a **DEF** statement or a **DEF** structure which appears after the **END** statement. The function you define must be defined over the entire domain specified. If it is not, the **ADDFGRAPH** subroutine may generate an error or draw the graph incorrectly.

Note that both the **ADDFGRAPH** subroutine and the **FGRAPH** subroutine utilize an externally defined function named F. Since a program may not contain two defined functions with the same name, it is your responsibility to ensure that the function F(x) is defined to calculate two different functions if you plan to use the **ADDFGRAPH** subroutine after calling the **FGRAPH** subroutine. (See the following example for one method of accomplishing this.)

The value of `style` determines the line style that will be used to connect the data points which comprise the graph. The allowable values for `style` are summarized in the following table:

Available Line Styles

Value of `style`	Resulting Line
0	No line (display points only)
1	Solid line
2	Dashed line
3	Dotted line
4	Dash-dotted line

The graph is actually composed of a series of short line segments. You can control the number of line segments used to display a graph with the **SETGRAIN** subroutine. Using more line segments creates a smoother graph, but takes longer to draw.

The value of `color$` determines the color that will be used to draw the new graph. It generally consists of a single color name (in any combination of uppercase or lowercase letters). The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `color$` may also contain a color number instead of a color name, allowing you to access any color supported by the current computer system.

If the value of `color$` contains more than one color, only the last color in the list will be used.

Note that the **ADDFGRAPH** subroutine assumes that a graph currently exists which has been created by an invocation of the **FGRAPH** or **DATAGRAPH** subroutine. The **ADDFGRAPH** subroutine simply adds the line representing the current function F(x) to the existing graph; it does not rescale the graph or redraw the labels or title. If you attempt to invoke the **ADDFGRAPH** subroutine when a suitable graph has not already been displayed, an error will be generated.

Example: The following program, SGFunc2.TRU, can be found in the directory TBDEMOS:

```
!  SGFunc2  Graph sine and cosine functions.

LIBRARY "..\TBLibs\SGFunc.trc", "..\TBLibs\SGLib.trc"

PUBLIC flag

CALL SetText ("Sine and Cosine Waves", "X Values", "Y Values")

CALL Fgraph (-2*pi, 2*pi, 1, "white white magenta")

LET flag = 1
CALL AddFgraph (2, "cyan")

GET KEY key

END

DEF F(x)
    DECLARE PUBLIC flag
    IF flag = 0 then LET F = Sin(x) else LET F = Cos(x)
END DEF
```

produces a graph of the functions Sin(x) and Cos(x). Notice the use of the public variable `flag` to change the behavior of the defined function being graphed.

Exceptions: 118 No canvas window yet.
11008 No such color: *color*.

See also: **SETGRAIN**, **FGRAPH**, **MANYFGRAPH**

ADDLSGRAPH Subroutine

Library: SGLIB.TRC

Syntax: CALL ADDLSGRAPH (*numarrarg*, *numarrarg*, *numex*, *strex*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL ADDLSGRAPH (x(), y(), style, color$)`

Summary: Computes and draws the least-squares linear fit for the specified points.

Details: The **ADDLSGRAPH** subroutine calculates and draws the least-squares linear fit of a set of data points.

The least-squares linear fit of a set of data points is the straight line which best fits the locations of those data points. That is, the least-squares linear fit of a set of data points is the straight line which minimizes the vertical distance between itself and each of the data points. Such a line may be used to help predict where data points might lie in areas for which data is unavailable.

The set of data points is specified as pairs of coordinates passed as the contents of the `x` and `y` arrays. The `x` array contains the points' x-coordinates, and the `y` array contains their y-coordinates. The coordinates in the two arrays are matched according to their subscripts; that is, the elements with subscripts of 1 within both arrays are interpreted as the coordinates of a single point, as are the elements with subscripts of 2, and so on. Thus, the `x` and `y` arrays must have the same upper and lower bounds, or an error will be generated.

The value of `style` determines the line style that will be used to draw the linear fit. The allowable values for `style` are summarized in the following table:

Available Line Styles

Value of `lstyle`	**Resulting Line**
0	No line (display points only)
1	Solid line
2	Dashed line
3	Dotted line
4	Dash-dotted line

The value of `color$` determines the color that will be used to draw the linear fit. It generally consists of a single color name (in any combination of uppercase or lowercase letters). The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

Note: the color "BACKGROUND" refers to the current background color.

The value of `color$` may also contain a color number instead of a color name, allowing you to access any of the colors supported by the current computer system.

Note that the **ADDLSGRAPH** subroutine assumes that a graph currently exists which has been created by an invocation of one of the various graphing subroutines. The **ADDLSGRAPH** subroutine simply adds the line representing the specified data points to the existing graph; it does not rescale the graph or redraw the labels or title. If you attempt to invoke the **ADDLSGRAPH** subroutine when a suitable graph has not already been displayed, an error will be generated.

Example: The following program, SGLSquar.TRU, can be found in the directory TBDEMOS:

```
!  SGLSquar   Add a least-squares line to data points.

!  Data taken from "The Shortwave Propagation Handbook" (2nd ed)
!  by George Jacobs and Theodore J. Cohen.  Page 111.

LIBRARY "..\TBLibs\SGLib.trc"

DIM x(120), y(120)

MAT READ x, y                      ! Data later

CALL SetYscale (70, 170)

CALL SetText ("Sunspots vs. Solar Flux", "Daily Sunspot Number",
"Daily Solar Flux")
CALL DataGraph (x, y, 6, 0, "red green yellow")
CALL AddLSgraph (x, y, 1, "red")
```

```
DATA  16,  17,   5,   4,  18,  19,  21,  24,  22,  25
DATA  28,  30,  32,  33,  31,  35,  21,  25,  26,  30
DATA  28,  31,  37,  37,  39,  38,  34,  25,  40,  41
DATA  43,  44,  42,  45,  47,  48,  50,  50,  52,  56
DATA  57,  59,  46,  42,  41,  45,  48,  52,  44,  45
DATA  49,  55,  58,  59,  53,  55,  55,  59,  57,  65
DATA  64,  61,  63,  64,  66,  65,  67,  69,  71,  76
DATA  75,  81,  80,  80,  81,  82,  87,  90,  84,  84
DATA  64,  65,  78,  78,  73,  80,  77,  74,  70,  70
DATA  61,  63,  73,  74,  73,  77,  79,  78,  79,  63
DATA  81,  94,  97,  93,  93,  86,  79,  98,  93, 116
DATA 116, 115, 116, 104, 127, 125, 130, 131, 123, 139

DATA  81,  84,  84,  88,  89,  87,  90,  89,  87,  87
DATA  85,  82,  91,  90,  87,  85,  96,  95,  95,  99
DATA  93,  94,  95,  98,  96, 103, 105, 111, 100,  94
DATA  99,  97,  97,  94,  97,  98, 100,  95,  97, 102
DATA 104, 104, 104, 105, 107, 109, 108, 108, 112, 115
DATA 115, 115, 116, 117, 120, 119, 127, 125, 133, 103
DATA 106, 110, 108, 111, 108, 107, 108, 107, 108, 105
DATA 110, 102, 107, 108, 108, 106, 110, 114, 118, 119
DATA 116, 115, 119, 118, 116, 114, 115, 114, 121, 122
DATA 126, 127, 125, 128, 131, 126, 127, 131, 130, 133
DATA 131, 129, 131, 123, 135, 138, 140, 144, 146, 148
DATA 158, 157, 156, 157, 154, 159, 159, 163, 162, 166

GET KEY key

END
```

produces a graph with a least-squares linear fit superimposed over it.

Exceptions: None

See also: **SETLS**, **ASKLS**, **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**

ASKANGLE Subroutine

Library: SGLIB.TRC

Syntax: CALL ASKANGLE (*strex*)

Usage: `CALL ASKANGLE (measure$)`

Summary: Reports the manner in which subsequent polar graphs drawn by the various data and function plotting subroutines will interpret angle measurements.

Details: The **ASKANGLE** subroutine is used to report the manner in which subsequent data and function polar plots produced by the **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**, **ADDFGRAPH**, and **MANYFGRAPH** subroutines will interpret angle measurements.

If the value of `measure$` is returned equal to “DEG” these subroutines will interpret angular coordinates for polar graphs as degrees. If the value of `measure$` is returned equal to “RAD” these subroutines will interpret angular coordinates for polar graphs as radians.

Note that the **ASKANGLE** subroutine only reports the interpretation of angular coordinates by polar graphs. Use the **ASKGRAPHTYPE** subroutine to report whether or not subsequent graphs will be drawn as polar graphs.

You can use the **SETANGLE** subroutine to control the manner in which the next data or function polar plot will interpret angular coordinates.

Example: None

Exceptions: None

See also: **SETANGLE**, **SETGRAPHTYPE**, **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**, **ADDFGRAPH**, **MANYFGRAPH**

ASKBARTYPE Subroutine

Library: BGLIB.TRC

Syntax: CALL ASKBARTYPE (*strvar*)

Usage: `CALL ASKBARTYPE (type$)`

Summary: Reports the arrangement of the bars within each group of subsequently drawn multiple bar chart or histogram.

Details: The **ASKBARTYPE** subroutine is used to report the arrangement of the bars within each group of a bar chart or histogram that will produced by a subsequent invocation of the **MULTIBAR** or **MULTIHIST** subroutine.

Both the **MULTIBAR** and **MULTIHIST** subroutines draw multiple bar-based graphs in a single frame. In such a graph, bars associated with a particular unit are grouped together.

The **ASKBARTYPE** subroutine allows you to report how the bars in each group will be arranged by returning one of the following values in `type$`:

Types of Bar Groupings

`Type$` **value**	**Description**
`"SIDE"`	Bars arranged side by side with space between them
`"STACK"`	Bars stacked one above the other
`"OVER"`	Bars arranged side by side but overlapped slightly

By default, the bar type is set to a value of `"SIDE"`. You can use the **SETBARTYPE** subroutine to change the current bar type setting.

Example: None

Exceptions: None

See also: **SETBARTYPE**, **MULTIBAR**, **MULTIHIST**

ASKGRAIN Subroutine

Library: SGLIB.TRC

Syntax: CALL ASKGRAIN (*numvar*)

Usage: `CALL ASKGRAIN (grain)`

Summary: Reports the grain with which subsequent invocations of the various function plotting subroutines will draw the line graph.

Details: The **ASKGRAIN** subroutine reports the `grain` with which subsequent invocations of the **FGRAPH**, **ADDFGRAPH**, and **MANYFGRAPH** subroutines will draw the line representing the function.

These subroutines actually graph the curve of the function which they are plotting as a series of line segments. The ***grain*** controls the number of line segments used to form each graphed curve. The higher the value of the grain, the more line segments are used and the smoother the resulting curve appears. However, higher grains also mean more work for the computer, and this means that each curve takes longer to draw.

By default, the **FGRAPH**, **ADDFGRAPH**, and **MANYFGRAPH** subroutines use a grain value of 64, which means that each line graph is composed of 64 individual line segments. This value strikes a generally acceptable balance of smoothness and speed, but this value can be changed using the **SETGRAIN** subroutine.

Example: None

Exceptions: None

See also: **SETGRAIN**, **FGRAPH**, **ADDFGRAPH**, **MANYFGRAPH**

ASKGRAPHTYPE Subroutine

Library: SGLIB.TRC

Syntax: CALL ASKGRAPHTYPE (*strvar*)

Usage: `CALL ASKGRAPHTYPE (type$)`

Summary: Reports the type of graph that will be drawn by subsequent data and function plotting subroutines.

Details: The **ASKGRAPHTYPE** subroutine is used to report the type of graph that will be produced for subsequent data and function plots produced by the **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**, **ADDFGRAPH**, and **MANYFGRAPH** subroutines.

The type of subsequent graphs is reported as the value of `type$`. The possible values of `type$` are:

Types of Graphs

`Type$` **value**	**Description**
`"XY"`	Normal graph
`"LOGX"`	Semi-logarithmic graph with x-axis logarithmically scaled
`"LOGY"`	Semi-logarithmic graph with y-axis logarithmically scaled
`"LOGXY"`	Logarithmic graph with both x- and y-axes logarithmically scaled
`"POLAR"`	Polar graph

You can use the **SETGRAPHTYPE** subroutine to control the type of graph that will be used for the next data or function plot.

Example: None

Exceptions: None

See also: **SETGRAPHTYPE**, **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**, **ADDFGRAPH**, **MANYFGRAPH**

ASKGRID Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL ASKGRID (*strvar*)

Usage: `CALL ASKGRID (style$)`

Summary: Reports the presence, direction, and type of the grid that will be used in subsequently drawn charts and graphs.

Details: The **ASKGRID** subroutine is used to report on the presence, direction, and type of the grid that will be drawn within the frame of graphs or charts produced by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH** subroutines.

The **ASKGRID** subroutine reports the presence and direction of the grid lines by returning one of the following values in `style$`:

Available Grid Directions

`Style$` **value**	**Description**
`""`	No grid lines
`"H"`	Horizontal grid lines only
`"V"`	Vertical grid lines only
`"HV"`	Both horizontal and vertical grid lines

The returned value of `style$` may also include instructions that indicate the type of grid lines that will be drawn. These instructions take the form of special characters appended to the letter (or letters) in the returned value of `style$`. If no such modifiers are present, grid lines will be drawn as solid lines. The following modifiers are possible:

Available Grid Type Modifiers

Modifier	Description
`-`	Dashed grid lines
`.`	Dotted grid lines
`-.`	Dash-dotted grid lines

For example, a value of `"H-.V"` would indicate that dash-dotted grid lines will be used in the horizontal direction and solid grid lines will be used in the vertical direction.

By default, the grid lines are turned off. You can use the **SETGRID** subroutine to change the current grid setting.

Example: None

Exceptions: None

See also: **SETGRID**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

ASKHLABEL Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL ASKHLABEL (*strvar*)

Usage: `CALL ASKHLABEL (hlabel$)`

Summary: Reports the value of the horizontal label which will be displayed for subsequently drawn charts and graphs.

Details: The **ASKHLABEL** subroutine is used to report the value of the horizontal label that will be used to label the frame of graphs or charts drawn by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, and **MANYDATAGRAPH** subroutines.

The **ASKHLABEL** subroutine returns the value of the horizontal label as `hlabel$`.

You may report the current values for the title, the horizontal label, and the vertical label simultaneously using the **ASKTEXT** subroutine. Use the **ASKVLABEL** and **ASKTITLE** subroutines to report the values of the vertical label and the title, respectively.

You may use the **SETHLABEL** subroutine to set the current value of the horizontal label.

Example: None

Exceptions: None

See also: **SETHLABEL**, **ASKTEXT**, **ASKVLABEL**, **ASKTITLE**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **PIECHART**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

ASKLAYOUT Subroutine

Library: BGLIB.TRC

Syntax: CALL ASKLAYOUT (*strvar*)

Usage: `CALL ASKLAYOUT (direction$)`

Summary: Reports the direction of the bars within subsequently drawn bar charts and histograms.

Details: The **ASKLAYOUT** subroutine is used to report the direction of the bars within each bar chart or histogram produced by a subsequent invocation of the **MULTIBAR** or **MULTIHIST** subroutine.

The **ASKLAYOUT** subroutine allows you to report the direction in which the bars will be drawn by returning one of the following values in `direction$`:

Types of Bar Layouts

`Type$` **value**	**Description**
`"HORIZONTAL"`	Bars oriented horizontally
`"VERTICAL"`	Bars oriented vertically

By default, the bar direction is set to a value of `"VERTICAL"`. You can use the **SETLAYOUT** subroutine to change the current bar layout setting.

Example: None

Exceptions: None

See also: **SETLAYOUT**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**

ASKLS Subroutine

Library: SGLIB.TRC

Syntax: CALL ASKLS (*numvar*)

Usage: `CALL ASKLS (flag)`

Summary: Reports whether least-squares linear fits will be drawn automatically for subsequent data plots.

Details: The **ASKLS** subroutine is used to report whether or not least-squares linear fits will be drawn automatically for subsequent data plots produced by the **DATAGRAPH**, **ADDDATAGRAPH**, and **MANYDATAGRAPH** subroutines.

If the **ASKLS** subroutine returns `flag` with a value of 1, subsequent calls to the **DATAGRAPH**, **ADDDATAGRAPH**, and **MANYDATAGRAPH** subroutines will automatically display the graph's least-squares linear fit. If it returns `flag` with a value of 0, they won't.

You can use the **SETLS** subroutine to control whether least-squares linear fitting is currently active or inactive.

Example: None

Exceptions: None

See also: **SETLS**, **ADDLSGRAPH**, **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**

ASKTEXT Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL ASKTEXT (*strvar*, *strvar*, *strvar*)

Usage: `CALL ASKTEXT (title$, hlabel$, vlabel$)`

Summary: Reports the values of the title, horizontal label, and vertical label that will be displayed for subsequently drawn charts and graphs.

Details: The **ASKTEXT** subroutine is used to report the values of the title, horizontal label, and vertical label that will be used to label the frame of graphs or charts drawn by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, and **MANYDATAGRAPH** subroutines. (These values also apply to the **PIECHART** subroutine, but only the value of the title is used.)

The **ASKTEXT** subroutine returns the value of the title as `title$`, the value of the horizontal label as `hlabel$`, and the value of the vertical label as `vlabel$`.

You may report the value of the title, the horizontal label, or the vertical label individually using the **ASKTITLE**, **ASKHLABEL**, or **ASKVLABEL** subroutines, respectively.

You may use the **SETTEXT** subroutine to set the current values of the title, the horizontal label, and the vertical label.

Example: None

Exceptions: None

See also: **SETTEXT**, **ASKTITLE**, **ASKHLABEL**, **ASKVLABEL**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **PIECHART**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

ASKTITLE Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL ASKTITLE (*strvar*)

Usage: `CALL ASKTITLE (title$)`

Summary: Reports the value of the title which will be displayed for subsequently drawn charts and graphs.

Details: The **ASKTITLE** subroutine is used to report the value of the title that will be used to label the frame of graphs or charts drawn by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**, and **PIECHART** subroutines.

The **ASKTITLE** subroutine returns the value of the title as `title$`.

You may report the current values for the title, the horizontal label, and the vertical label simultaneously using the **ASKTEXT** subroutine. Use the **ASKHLABEL** and **ASKVLABEL** subroutines to report the values of the horizontal label and the vertical label, respectively.

You may use the **SETTITLE** subroutine to set the current value of the title.

Example: None

Exceptions: None

See also: **SETTITLE**, **ASKTEXT**, **ASKHLABEL**, **ASKVLABEL**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **PIECHART**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

ASKVLABEL Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL ASKVLABEL (*strvar*)

Usage: `CALL ASKVLABEL (vlabel$)`

Summary: Reports the value of the vertical label which will be displayed for subsequently drawn charts and graphs.

Details: The **ASKVLABEL** subroutine is used to report the value of the vertical label that will be used to label the frame of graphs or charts drawn by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, and **MANYDATAGRAPH** subroutines.

The **ASKVLABEL** subroutine returns the value of the vertical label as `vlabel$`.

You may report the current values for the title, the horizontal label, and the vertical label simultaneously using the **ASKTEXT** subroutine. Use the **ASKHLABEL** and **ASKTITLE** subroutines to report the values of the horizontal label and the title, respectively.

You may use the **SETVLABEL** subroutine to set the current value of the vertical label.

Example: None

Exceptions: None

See also: **SETVLABEL**, **ASKTEXT**, **ASKHLABEL**, **ASKTITLE**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **PIECHART**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

BALANCEBARS Subroutine

Library:	BGLIB.TRC
Syntax:	CALL BALANCEBARS (*numarrarg*, *numarrarg*, *strarrarg, strarrarg, strex*)

strarrarg:: *strarr*
strarr bowlegs

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL BARCHART (d1(,), d2(,), units$(), legends$(), colors$)`

Summary: Draws a balanced bar chart, setting off d1(,) values on one side of the axis versus d2(,) values on the other.

Details: The **BALANCEBARS** subroutine draws a balanced bar chart in the current logical window, setting off d1(,) values on one side of the axis versus d2(,) values on the other. This is not a particularly common kind of bar chart, but is useful for comparing income versus expenses, etc.

Simply put, it draws a multi-bar chart of d1(,) on the top or right side of the axis, and the same style chart of d2(,) on the bottom or left side of the axis. Neither array may contain any negative values.

The data arrays d1 and d2 are as in the MULTIBAR subroutine, and the units$ and legends$ arrays label both sets of data.

The `units$` array must contain the same number of items as the `data` array. Each element of the `units$` array will be used as a label for the bar associated with the corresponding element of the `data` array.

The value of `colors$` determines the color scheme that will be used to draw the graph. It generally consists of at least three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels. And the third color will be used for the graph's data.

If `colors$` contains four colors, the third color will be used for drawing bars representing positive values, and the fourth color will be used for drawing bars representing negative values. If `colors$` contains more than four colors, the extra colors will not be used. If `colors$` contains fewer than four colors, the last color specified will be used to fill out the remaining colors. If the value of `colors$` is the null string, then the current foreground color is used for the entire graph.

By default, the **BALANCEBARS** subroutine draws the graph with the bars oriented vertically. The y-axis is automatically scaled to fit the data, and the bars are evenly spaced along the x-axis. The labels will appear beneath each bar.

You can change the graph's orientation so that the bars are drawn horizontally by first invoking the **SETLAYOUT** subroutine with the argument "HORIZONTAL". In this situation, the x-axis will be automatically scaled to fit the data, and the bars will be evenly spaced along the y-axis. The labels will appear to the left of each bar.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, BGBar3.TRU, can be found in the directory TBDEMOS:

```
!  BGBar3  Show a simple balanced bar chart of products,
!          with income/expense for last year and this year.

LIBRARY "..\TBLibs\BGLib.trc"

DIM income(4,2), expense(4,2), units$(4), legend$(2)

MAT READ income, expense, units$, legend$
DATA 43,34, 54,63, 33,12, 62,92   ! Incomes
DATA 39,24, 49,52, 17,13, 43,57   ! Expenses
DATA Faucets, Swings, Hoses, Flamingos     ! Units
DATA Last Year, This Year         ! Legend

CALL SetBarType ("over")
CALL SetText ("Income/Expense: Last 2 Years", "", "Thousands")
LET colors$ = "yellow yellow red green"
CALL BalanceBars (income, expense, units$, legend$, colors$)

GET KEY key

END
```

produces a bar chart representing quarterly profits.

Exceptions:

100	Graph's title is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.
109	Horizontal marks aren't wide enough—use SetHMarkLen.
111	Data and unit arrays don't match for BarChart.
117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also **SETBARTYPE**, **SETTEXT**

BARCHART Subroutine

Library: BGLIB.TRC

Syntax: CALL BARCHART (*numarrarg*, *strarrarg*, *strex*)

strarrarg::	*strarr*
	strarr bowlegs
numarrarg::	*numarr*
	numarr bowlegs

Usage: `CALL BARCHART (data(), units$(), colors$)`

Summary: Draws a simple bar chart of the specified data values, labeled with the specified units and drawn in the specified color scheme.

Details: The **BARCHART** subroutine draws a bar chart in the current logical window.

The bar chart will contain one bar for each element of the `data` array, and the height of each bar will be determined by the value of its corresponding element in the `data` array.

The `units$` array must contain the same number of items as the `data` array. Each element of the `units$` array will be used as a label for the bar associated with the corresponding element of the `data` array.

The value of `colors$` determines the color scheme that will be used to draw the graph. It generally consists of at least three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels. And the third color will be used for the graph's data.

If `colors$` contains four colors, the third color will be used for drawing bars representing positive values, and the fourth color will be used for drawing bars representing negative values. If `colors$` contains more than four colors, the extra colors will not be used. If `colors$` contains fewer than four colors, the last color specified will be used to fill out the remaining colors. If the value of `colors$` is the null string, then the current foreground color is used for the entire graph.

By default, the **BARCHART** subroutine draws the graph with the bars oriented vertically. The y-axis is automatically scaled to fit the data, and the bars are evenly spaced along the x-axis. The labels will appear beneath each bar.

You can change the graph's orientation so that the bars are drawn horizontally by first invoking the **SETLAYOUT** subroutine with the argument "HORIZONTAL". In this situation, the x-axis will be automatically scaled to fit the data, and the bars will be evenly spaced along the y-axis. The labels will appear to the left of each bar.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, BGBar1.TRU, can be found in the directory TBDEMOS:

```
! BGBar1   Draw a simple bar chart.

LIBRARY "..\TBLibs\BGLib.trc"

DIM units$(4), data(4)

MAT READ units$, data
DATA Q-1, Q-2, Q-3, Q-4
DATA 498, 322, 395, 430

CALL SetText ("Quarterly Profits", "Quarter", "Thousands")
CALL BarChart (data, units$, "white cyan magenta")

GET KEY key

END
```

produces a bar chart representing quarterly profits.

Exceptions:

100	Graph's title is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.
109	Horizontal marks aren't wide enough—use SetHMarkLen.
111	Data and unit arrays don't match for BarChart.
117	Can't handle this graph range: *low* to *high*.

11008 No such color: *color*.

See also: **SETTEXT**, **SETLAYOUT**, **MULTIBAR**, **HISTOGRAM**

DATAGRAPH Subroutine

Library: SGLIB.TRC

Syntax: CALL DATAGRAPH (*numarrarg*,, *numarrarg*, *numex*, *numex*, *strex*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL DATAGRAPH (x(), y(), pstyle, lstyle, colors$)`

Summary: Draws a simple line graph of a set of data points.

Details: The **DATAGRAPH** subroutine draws a line graph of the set of data points whose coordinates are represented by the values of the `x` and `y` arrays.

The `x` array contains the points' x-coordinates, and the `y` array contains their y-coordinates. The coordinates in the two arrays are matched according to their subscripts; that is, the elements with subscripts of 1 within both arrays are interpreted as the coordinates of a single point, as are the elements with subscripts of 2, and so on. Thus, the `x` and `y` arrays must have the same upper and lower bounds, or an error will be generated.

Both the x- and y-axes will be scaled automatically by the **DATAGRAPH** subroutine.

The value of `pstyle` determines the point style that will be used to draw the data points which comprise the graph. The allowable values for `pstyle` are summarized in the following table:

Available Point Styles

Value of `pstyle`	**Resulting Point**
0	No point (display line only)
1	Dot
2	Plus sign
3	Asterisk
4	Circle
5	X
6	Box
7	Up triangle
8	Down triangle
9	Diamond
10	Solid Box
11	Solid up triangle
12	Solid down triangle
13	Solid diamond

The value of `lstyle` determines the line style that will be used to connect the data points which comprise the graph. The allowable values for `lstyle` are summarized in the following table:

Available Line Styles

Value of `lstyle`	**Resulting Line**
0	No line (display points only)
1	Solid line
2	Dashed line
3	Dotted line
4	Dash-dotted line

The graph is actually composed of a series of line segments connecting the data points. You can suppress the display of the data points by passing a value of 0 in `pstyle`, or you can suppress the display of the connecting line segments by passing a value of 0 in `lstyle`.

Note that the **DATAGRAPH** subroutine draws and connects the points in the order in which they are stored in the `x` and `y` arrays. If your points are not stored in left to right order, you may wish to use the **SORTPOINTS** subroutine to order the points before passing them to the **DATAGRAPH** subroutine.

The value of `colors$` determines the color scheme that will be used to draw the graph. It generally consists of three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels. And the third color will be used for the graph's data.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, SGData1.TRU, can be found in the directory TBDEMOS:

```
!  SGData1   Average fuel economy for all cars in USA.  Source: EPA.

LIBRARY "..\TBLibs\SGLib.trc"

DIM year(36), mpg(36)

MAT READ year, mpg

DATA 1940, 1945, 1950, 1951, 1952, 1953, 1954, 1955
DATA 1956, 1957, 1958, 1959, 1960, 1961, 1962, 1963, 1964, 1965
DATA 1966, 1967, 1968, 1969, 1970, 1971, 1972, 1973, 1974, 1975
DATA 1976, 1977, 1978, 1979, 1980, 1981, 1982, 1983
DATA 15.29,15.03,14.95,14.99,14.67,14.70,14.58,14.53
DATA 14.36,14.40,14.30,14.30,14.28,14.38,14.37,14.26,14.25,14.07
DATA 14.00,13.93,13.79,13.63,13.57,13.57,13.49,13.10,13.43,13.53
DATA 13.72,13.94,14.06,14.29,15.15,15.54,16.25,16.70

CALL SetText ("Fuel Economy - All Cars", "", "MPG")
CALL DataGraph (year, mpg, 9, 1, "red green yellow")

GET KEY key

END
```

produces a graph of the average fuel economy of all new cars produced in each year from 1940 through 1983.

Exceptions:

100	Graph's title is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.
109	Horizontal marks aren't wide enough—use SetHMarkLen.
110	Data arrays have different bounds in DataGraph

117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also: **SETTEXT**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**

FGRAPH Subroutine

Library: SGFUNC.TRC, SGLIB.TRC

Syntax: CALL FGRAPH (*numex*, *numex*, *numex*, *strex*)

Usage: `CALL FGRAPH (startx, endx, style, colors$)`

Summary: Draws a simple line graph of an externally defined function.

Details: The **FGRAPH** subroutine draws a line graph of the function F(x) over the domain `startx` to `endx`.

The function F(x) must be defined external to your main program. That is, it must be defined using a **DEF** statement or a **DEF** structure which appears after the **END** statement. The function you define must be defined over the entire domain specified. If it is not, the **FGRAPH** subroutine may generate an error or draw the graph incorrectly.

The y-axis will be scaled automatically by the **FGRAPH** subroutine.

The value of `style` determines the line style that will be used to connect the data points which comprise the graph. The allowable values for `style` are summarized in the following table:

Available Line Styles

Value of `style`	**Resulting Line**
0	No line (display points only)
1	Solid line
2	Dashed line
3	Dotted line
4	Dash-dotted line

The graph is actually composed of a series of short line segments. You can control the number of line segments used to display a graph with the **SETGRAIN** subroutine. Using more line segments creates a smoother graph, but takes longer to draw.

The value of `colors$` determines the color scheme that will be used to draw the graph. It generally consists of three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels. And the third color will be used for the graph's data.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, SGFunc1.TRU, can be found in the directory TBDEMOS:

```
!  SGFunc1   Graph the function "Sin(x*x)".

LIBRARY "..\TBLibs\SGFunc.trc", "..\TBLibs\SGLib.trc"

CALL SetText ("Sin(x*x)", "X Values", "Y Values")
CALL Fgraph (-pi, pi, 2, "white white magenta")
```

```
GET KEY key

END

DEF F(x) = Sin(x*x)
```

produces a graph of the function Sin(x^2).

Exceptions:

100	Graph's title is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.
109	Horizontal marks aren't wide enough—use SetHMarkLen.
117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also: **SETTEXT**, **SETGRAIN**, **ADDFGRAPH**, **MANYFGRAPH**

HISTOGRAM Subroutine

Library: BGLIB.TRC

Syntax: CALL HISTOGRAM (*numarrarg*, *strex*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL HISTOGRAM (data(), colors$)`

Summary: Draws a simple histogram of the specified data values in the specified color scheme.

Details: The **HISTOGRAM** subroutine draws a simple histogram in the current logical window.

The histogram automatically "groups" similar values from the `data` array and draws one bar per group. The height of each bar reflects the number of members in the associated group.

For instance, if you use the **HISTOGRAM** subroutine to chart students' grades, it might group all those students with grades in the range 80 through 84 and draw a single bar to represent this group of students. The bars will be labeled "75>", "80>", "85>", and so forth. This means that the first bar represents the group of students whose grades are greater than or equal to 75 but less than 80. The second bar represents the group with grades greater than or equal to 80 but less than 85, and so forth.

The value of `colors$` determines the color scheme that will be used to draw the graph. It generally consists of at least three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels. And the third color will be used for the graph's data.

If `colors$` contains more than three colors, the extra colors will not be used. If `colors$` contains fewer than three colors, the last color specified will be used to fill out the remaining colors. If the value of `colors$` is the null string, then the current foreground color is used for the entire graph.

By default, the **HISTOGRAM** subroutine draws the graph with the bars oriented vertically. The y-axis is automatically scaled to fit the data, and the bars are evenly spaced along the x-axis. The labels will appear beneath each bar.

You can change the graph's orientation so that the bars are drawn horizontally by first invoking the **SETLAYOUT** subroutine with the argument "HORIZONTAL". In this situation, the x-axis will be automatically scaled to fit the data, and the bars will be evenly spaced along the y-axis. The labels will appear to the left of each bar.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, BGHisto1.TRU, can be found in the directory TBDEMOS:

```
!  BGHisto1   Draw a simple histogram.

LIBRARY "..\TBLibs\BGLib.trc"

DIM data(30)

MAT READ data
DATA 65, 70, 93, 85, 83, 68, 77, 92, 83, 85
DATA 89, 72, 75, 81, 80, 84, 73, 79, 78, 84
DATA 80, 79, 72, 91, 85, 82, 79, 76, 74, 79

CALL SetText ("Final Grades", "", "# of Students")
CALL Histogram (data, "white cyan magenta")

GET KEY key

END
```

produces a histogram of student grades.

Exceptions:

100	Graph's title is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.
109	Horizontal marks aren't wide enough—use SetHMarkLen.
111	Data and unit arrays don't match for Histogram.
117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also: **SETTEXT**, **SETLAYOUT**, **BARCHART**, **MULTIHIST**

IBEAM Subroutine

Library: BGLIB.TRC

Syntax: CALL IBEAM (*numarrarg*, *numarrarg*, *strarrarg*, *strex*)

strarrarg::	*strarr*
	strarr bowlegs
numarrarg::	*numarr*
	numarr bowlegs

Usage: `CALL IBEAM (high(), low(), units$(), colors$)`

Summary: Draws an "I-beam" chart of the specified data values, labeled with the specified units and drawn in the specified color scheme.

Details: The **IBEAM** subroutine draws an "I-beam" chart in the current logical window.

The I-beam chart displays ranges of values and will contain one I-beam for each element of the `high` array. The height and position of each I-beam will be determined by the difference between corresponding elements of the `high` and `low` arrays. For this reason, the `high` and `low` arrays must contain the same number of elements.

The `units$` array must contain the same number of items as the `high` and `low` arrays. Each element of the `units$` array will be used as a label for the I-beam associated with the corresponding elements of the `high` and `low` arrays.

The value of `colors$` determines the color scheme that will be used to draw the graph. It generally consists of at least three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels. And the third color will be used for the graph's data.

If `colors$` contains more than three colors, the extra colors will not be used. If `colors$` contains fewer than three colors, the last color specified will be used to fill out the remaining colors. If the value of `colors$` is the null string, then the current foreground color is used for the entire graph.

By default, the **IBEAM** subroutine draws the graph with the I-beams oriented vertically. The y-axis is automatically scaled to fit the data, and the I-beams are evenly spaced along the x-axis. The labels will appear beneath each I-beam.

You can change the graph's orientation so that the I-beams are drawn horizontally by first invoking the **SETLAYOUT** subroutine with the argument "HORIZONTAL". In this situation, the x-axis will be automatically scaled to fit the data, and the I-beams will be evenly spaced along the y-axis. The labels will appear to the left of each I-beam.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, BGIBeam.TRU, can be found in the directory TBDEMOS:

```
!  BGIBeam   Show I-beam chart of stock values.

LIBRARY "..\TBLibs\BGLib.trc"

DIM low(5), high(5), units$(5)

MAT READ low, high, units$
DATA 33.1, 33.2, 34.1, 34.1, 33.1
DATA 34.5, 33.9, 36.2, 34.7, 33.9
DATA Mon, Tues, Wed, Thurs, Fri

CALL SetText ("Stock Values", "Day", "Price")
CALL Ibeam (low, high, units$, "magenta white white")

GET KEY key

END
```

produces an I-beam chart representing the daily ranges of a stock's value over a one week period.

Exceptions: 100 Graph's title is too wide.

102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.
109	Horizontal marks aren't wide enough—use SetHMarkLen.
111	Data and unit arrays don't match for IBeam.
117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also: **SETTEXT**, **SETLAYOUT**

MANYDATAGRAPH Subroutine

Library: SGLIB.TRC

Syntax: CALL MANYDATAGRAPH (*numarrarg*,, *numarrarg*, *numex*, *strarrarg*, *strex*)

strarrarg:: *strarr*
strarr bowlegs

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL MANYDATAGRAPH (x(,), y(,), connect, legends$(), colors$)`

Summary: Draws multiple line graphs of a set of data points.

Details: The **MANYDATAGRAPH** subroutine draws several line graphs within a single frame. Each graph is based upon a set of data points whose coordinates are represented by the values of corresponding rows of the `x` and `y` arrays. For example, the statement:

```
DIM x(3,15), y(3,15)
```

would create the `x` and `y` matrices for a graph with three lines, each composed of fifteen data points.

Each row of the `x` matrix contains the x-coordinates for the points of a single line graph, and the corresponding row of the `y` matrix contains their y-coordinates. The coordinates in the separate rows of the two matrices are matched according to their second subscripts, or column numbers; that is, the elements with second subscripts of 1 within corresponding rows of both matrices are interpreted as the coordinates of a single point, as are the elements with second subscripts of 2, and so on. Thus, the `x` and `y` matrices must have the same upper and lower bounds in both dimensions, or an error will be generated.

Both the x- and y-axes will be scaled automatically by the **MANYDATAGRAPH** subroutine.

Each graph will use a different point style. These point styles will be drawn in order from the available point styles (with point styles 0 and 1 excepted). When the possible point styles are exhausted, they will be reused from the beginning of the list. For an ordered list of the available point styles, see the discussion of the **DATAGRAPH** subroutine.

If the value of `connect` is not equal to 0, the data points of each line graph will be connected by a line segment.

Note that the **MANYDATAGRAPH** subroutine draws and connects the points in the order in which they are stored in the `x` and `y` matrices. If your points are not stored in left to right order, you may wish to use the **SORTPOINTS2** subroutine to order the points before passing them to the **MANYDATAGRAPH** subroutine.

The **MANYDATAGRAPH** subroutine creates a legend just below the graph's title to assist the user in identifying the various lines. Each label for the legend will be taken from the corresponding element of the `legends$` array. Thus, the number of rows in the `x` and `y` arrays must be equal to the number of elements in the `legends$` array.

If you would like to omit the legend entirely, then pass a `legends$` array which contains no elements.

The value of `colors$` determines the color scheme that will be used to draw the graphs. It generally consists of at least three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels. And the remaining colors will be used for the graphs' data.

If the number of graphs exceeds the number of colors provided for the graphs' data, the **MANYDATAGRAPH** subroutine uses line styles to help distinguish the lines of the graphs. First, it draws solid lines in the colors specified. Then it switches to dashed, dotted, and finally dash-dotted lines. Thus, if you graph five functions with the **MANYFGRAPH** subroutine using the color scheme `"red yellow green blue"` you will get (in order): a solid green line, a solid blue line, a dashed green line, a dashed blue line, and a dotted green line.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, SGData3.TRU, can be found in the directory TBDEMOS:

```
!  SGData3   Display multiple sets of data points.

LIBRARY "..\TBLibs\SGLib.trc"

DIM x(5,10), y(5,10), legends$(5)

MAT READ legends$
DATA A, B, C, D, E

FOR i = 1 to 5
    FOR j = 1 to 10
        LET x(i,j) = j
        LET y(i,j) = (i*i*j) ^ 2
    NEXT j
NEXT i

CALL SetText ("Multiple Sets of Data", "Signal", "Reflection")
CALL SetGraphType ("logy")
LET colors$ = "white white magenta cyan"
CALL ManyDataGraph (x, y, 1, legends$, colors$)

GET KEY key

END
```

produces a graph several related data sets.

Exceptions:

100	Graph's title is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.

109	Horizontal marks aren't wide enough—use SetHMarkLen.
110	Data arrays have different bounds in DataGraph
117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also: **SETTEXT**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**

MANYFGRAPH Subroutine

Library: SGFUNC.TRC, SGLIB.TRC

Syntax: CALL MANYFGRAPH (*numex*, *numex*, *numex*, *strarr*, *strex*)

Usage: `CALL MANYFGRAPH (startx, endx, n, legends$(), colors$)`

Summary: Draws multiple line graphs based upon an externally defined function.

Details: The **MANYFGRAPH** subroutine draws several line graphs within a single frame. All of the functions drawn are based upon the definition of the function F(x) over the domain `startx` to `endx`. The number of graphs which are to be drawn is indicated by the value of `n`.

The function F(x) must be defined external to your main program. That is, it must be defined using a **DEF** statement or a **DEF** structure which appears after the **END** statement. The functions you define must be defined over the entire domain specified. If they are not, the **MANYFGRAPH** subroutine may generate an error or draw one or more of the graphs incorrectly.

The **MANYFGRAPH** subroutine uses the public variable `fnum` to inform your defined function F(x) which value to compute. The **MANYFGRAPH** subroutine sets the value of `fnum` to 1 when plotting the first function, 2 when plotting the second function, and so on until the number of functions specified by `n` have been plotted. Your defined function F(x) should contain a **PUBLIC** statement listing `fnum` so that the **MANYFGRAPH** subroutine can communicate with it properly. (See the following example for an illustration.)

The y-axis will be scaled automatically by the **MANYFGRAPH** subroutine.

The **MANYFGRAPH** subroutine creates a legend just below the graph's title to assist the user in identifying the various lines. Each label for the legend will be taken from the corresponding element of the `legends$` array. Thus, the value of `n` must be equal to the number of elements in the `legends$` array.

If you would like to omit the legend entirely, then pass a `legends$` array which contains no elements.

The value of `colors$` determines the color scheme that will be used to draw the graphs. It generally consists of at least three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels. And the remaining colors will be used for the graphs' data.

If the number of graphs (represented by the value of `n`) exceeds the number of colors provided for the graphs' data, the **MANYFGRAPH** subroutine uses line styles to help distinguish the lines of the graphs. First, it draws solid lines in the colors specified. Then it switches to dashed, dotted, and finally dash-dotted lines. Thus, if you graph five functions with the **MANYFGRAPH** subroutine using the color scheme `"red yellow green blue"` you will get (in order): a solid green line, a solid blue line, a dashed green line, a dashed blue line, and a dotted green line.

Each graph is actually composed of a series of short line segments. You can control the number of line segments used to display the graphs with the **SETGRAIN** subroutine. Using more line segments creates smoother graphs, but they take longer to draw.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, SGFunc3.TRU, can be found in the directory TBDEMOS:

```
!  SGFunc3   Graph many functions.

LIBRARY "..\TBLibs\SGFunc.trc", "..\TBLibs\SGLib.trc"

DIM legend$(3)

MAT READ legend$
DATA #1, #2, #3

CALL SetText ("Various Waves", "X Values", "Y Values")
LET colors$ = "white white cyan magenta white"
CALL ManyFgraph (-pi, 2*pi, 3, legend$, colors$)

GET KEY key

END

DEF F(x)
    PUBLIC fnum
    SELECT CASE fnum
    CASE 1
         LET F = Sin(x)
    CASE 2
         LET F = 1.5 * Cos(x*2)
    CASE 3
         LET F = .5 * Cos(x+pi/2)
    END SELECT
END DEF
```

produces a single graph of three different functions. Notice the use of the public variable `fnum` to define three distinct behaviors for the single function F(x).

Exceptions:

100	Graph's title is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.
109	Horizontal marks aren't wide enough—use SetHMarkLen.
112	Data and legend arrays don't match for ManyFGraph.
117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also: **SETTEXT**, **SETGRAIN**, **FGRAPH**, **ADDFGRAPH**

MULTIBAR Subroutine

Library: BGLIB.TRC

Syntax: CALL MULTIBAR (*numarrarg*, *strarrarg*, *strarrarg*, *strex*)

strarrarg::	*strarr*
	strarr bowlegs
numarrarg::	*numarr*
	numarr bowlegs

Usage: `CALL MULTIBAR (data(), units$(), legends$(), colors$)`

Summary: Draws a multi-bar chart of the specified data values, labeled with the specified units and legend and drawn in the specified color scheme.

Details: The **MULTIBAR** subroutine draws a multi-bar chart in the current logical window. In a multi-bar chart, each unit is represented by a cluster of bars. To produce simple bar charts with only one bar per unit, use the **BARCHART** subroutine.

The multi-bar chart will contain one cluster of bars for each row of the `data` array, and each cluster will contain one bar for each column of the `data` array. The height of each bar will be determined by the value of the appropriate element in the `data` array.

For example, if the `data` array contains five rows and three columns, the multi-bar chart will consist of five clusters, and each cluster will contain three bars.

The `units$` array must contain the same number of items as the first dimension of the `data` array. Each element of the `units$` array will be used as a label for the cluster of bars associated with the corresponding row of the `data` array.

The `legends$` array generally must contain the same number of items as the second dimension of the `data` array. The `legends$` array will be used to add a legend to the graph (positioned between the title and the graph itself) which will allow the user to identify the individual bars within the clusters. Each element of the `legends$` array provides the label for the corresponding column of the `data` array. To suppress the appearance of such a legend, pass a `legends$` array which contains zero elements.

The value of `colors$` determines the color scheme that will be used to draw the graph. It generally consists of at least three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels and the legend text. And the third color will be used for the graph's data.

If `colors$` contains more than three colors, the third and following colors will be used in repeating sequence for drawing the bars in each cluster. If `colors$` contains fewer than three colors, the last color specified will be used to fill out the remaining colors. If the value of `colors$` is the null string, then the current foreground color is used for the entire graph.

By default, the **MULTIBAR** subroutine draws the graph with the bars oriented vertically. The y-axis is automatically scaled to fit the data, and the clusters are evenly spaced along the x-axis. The labels stored in the `units$` array will appear beneath each cluster.

You can change the graph's orientation so that the bars are drawn horizontally by first invoking the **SETLAYOUT** subroutine with the argument "HORIZONTAL". In this situation, the x-axis will be automatically scaled to fit the data, and the clusters will be evenly spaced along the y-axis. The labels stored in the `units$` array will appear to the left of each cluster.

By default, the **MULTIBAR** subroutine draws the bars in each cluster side-by-side; however, they can also be drawn stacked or overlapped. Invoke the **SETBARTYPE** subroutine with an appropriate argument prior to invoking the **MULTIBAR** subroutine in order to determine the arrangement of the bars.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, BGBar2.TRU, can be found in the directory TBDEMOS:

```
!  BGBar2   Draw a simple multi-bar graph.

!  Last year's sales in yellow; this year's in green.

LIBRARY "..\TBLibs\BGLib.trc"

DIM data(4,2), units$(4), legend$(2)

MAT READ data, units$, legend$
DATA 103,106, 47,68, 112,115, 87,94
DATA Books, Software, Cards, Candy
DATA Last Year, This Year

CALL SetBarType ("side")
CALL SetLayout ("h")
CALL SetGrid ("v")
CALL SetText ("Sales: Last Year and Current",
"Thousands","Category")

CALL MultiBar (data, units$, legend$, "red red yellow green")

GET KEY key

END
```

produces a horizontal multi-bar chart representing a comparison of annual sales.

Exceptions:

100	Graph's title is too wide.
101	Graph's legend is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.
109	Horizontal marks aren't wide enough—use SetHMarkLen.
111	Data and unit arrays don't match for MultiBar.
112	Data and legend arrays don't match for MultiBar.
117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also: **SETTEXT**, **SETLAYOUT**, **SETBARTYPE**, **BARCHART**, **HISTOGRAM**

MULTIHIST Subroutine

Library: BGLIB.TRC

Syntax: CALL MULTIHIST (*numarrarg*, *strarrarg*, *strex*)

strarrarg::	*strarr*
	strarr bowlegs
numarrarg::	*numarr*
	numarr bowlegs

Usage: `CALL MULTIHIST (data(), legends$(), colors$)`

Summary: Draws multiple histograms of the specified data values in a single frame in the specified color scheme.

Details: The **MULTIHIST** subroutine draws multiple histograms in the current logical window. All histograms drawn by the **MULTIHIST** subroutine are overlaid in the same frame, with the bars for similar data values forming "clusters." To produce a simple histogram with only one bar per unit, use the **HISTOGRAM** subroutine.

Each histogram automatically "groups" similar values from a single row of the `data` array and draws one bar per group. Thus, each cluster will contain one bar for each row of the `data` array. The height of each bar reflects the number of members in the associated group.

For instance, if you use the **HISTOGRAM** subroutine to chart students' grades for two different classes, it might group all those students in the first class with grades in the range 80 through 84 and draw a single bar to represent this group of students. When the histogram for the second class was compiled, a bar representing the number of students in that class with grades in the range 80 through 84 would be added to the cluster containing the previous bar. The resulting clusters will be labeled "75>", "80>", "85>", and so forth. This means that the first cluster will contain one bar representing the group of students in the first class whose grades are greater than or equal to 75 but less than 80 and another bar representing students from the second class whose grades fall in the same range. The second cluster will contain bars representing the groups with grades greater than or equal to 80 but less than 85, and so forth.

The `legends$` array generally must contain the same number of items as the second dimension of the `data` array. The `legends$` array will be used to add a legend to the graph (positioned between the title and the graph itself) which will allow the user to identify the individual bars within the clusters. Each element of the `legends$` array provides a label for one of the histograms produced from the `data` array. To suppress the appearance of such a legend, pass a `legends$` array which contains zero elements.

The value of `colors$` determines the color scheme that will be used to draw the graph. It generally consists of at least three color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame, including the horizontal and vertical labels and the legend text. And the third color will be used for the graph's data.

If `colors$` contains more than three colors, the third and following colors will be used in repeating sequence for drawing the bars in each cluster. If `colors$` contains fewer than three colors, the last color specified will be used to fill out the remaining colors. If the value of `colors$` is the null string, then the current foreground color is used for the entire graph.

By default, the **MULTIHIST** subroutine draws the graph with the bars oriented vertically. The y-axis is automatically scaled to fit the data, and the clusters are evenly spaced along the x-axis. The cluster labels will appear beneath each cluster.

You can change the graph's orientation so that the bars are drawn horizontally by first invoking the **SETLAYOUT** subroutine with the argument "HORIZONTAL". In this situation, the x-axis will be automatically scaled to fit the data, and the clusters will be evenly spaced along the y-axis. The cluster labels will appear to the left of each cluster.

By default, the **MULTIHIST** subroutine draws the bars in each cluster side-by-side; however, they can also be drawn stacked or overlapped. Invoke the **SETBARTYPE** subroutine with an appropriate argument prior to invoking the **MULTIHIST** subroutine in order to determine the arrangement of the bars.

The text used for the graph's title and vertical and horizontal labels will be the values most recently set by the **SETTEXT** subroutine.

Example: The following program, BGHisto2.TRU, can be found in the directory TBDEMOS:

```
!  BGHisto2   Draw a multiple histogram.

LIBRARY "..\TBLibs\BGLib.trc"

DIM data(2,30), legend$(2)

MAT READ data, legend$
DATA 65, 70, 93, 85, 83, 68, 77, 92, 83, 85
DATA 89, 72, 75, 81, 80, 84, 73, 79, 78, 84
DATA 80, 79, 72, 91, 85, 82, 79, 76, 74, 79
DATA 75, 60, 83, 75, 73, 88, 67, 82, 73, 75
DATA 79, 62, 65, 71, 70, 74, 63, 69, 68, 74
DATA 70, 69, 62, 81, 75, 72, 69, 66, 64, 69

DATA Day, Evening

CALL SetBarType ("over")
CALL SetText ("Final Grades", "", "# of Students")
CALL MultiHist (data, legend$, "white cyan magenta cyan")

GET KEY key

END
```

produces a horizontal multi-bar chart representing a comparison of annual sales.

Exceptions:

100	Graph's title is too wide.
101	Graph's legend is too wide.
102	Graph's horizontal label is too wide.
103	Graph's vertical label is too long.
104	Need more room for graph's vertical marks.
105	Need more room for graph's horizontal marks.
106	Need greater width for graph.
107	Need greater height for graph.
108	Vertical marks aren't wide enough—use SetVMarkLen.
109	Horizontal marks aren't wide enough—use SetHMarkLen.
111	Data and unit arrays don't match for MultiHist.
112	Data and legend arrays don't match for MultiHist.
117	Can't handle this graph range: *low* to *high*.
11008	No such color: *color*.

See also: **SETTEXT**, **SETLAYOUT**, **SETBARTYPE**, **HISTOGRAM**, **BARCHART**

PIECHART Subroutine

Library: BGLIB.TRC

Syntax: CALL PIECHART (*numarrarg*, *strarrarg*, *strex*, *numex*, *numex*)

strarrarg::	*strarr*
	strarr bowlegs
numarrarg::	*numarr*
	numarr bowlegs

Usage: `CALL PIECHART (data(), units$(), colors$, wedge, percent)`

Summary: Draws a pie chart of the specified data values, labeled with the specified units and drawn in the specified color scheme.

Details: The **PIECHART** subroutine draws a pie chart in the current logical window.

A pie chart is displayed as a circle divided into wedges. The pie chart will contain one wedge for each element of the `data` array, and the proportion of the circle's area allocated to each wedge will be determined by the proportional relationship of the value of its corresponding element in the `data` array to the sum of the elements of the `data` array.

The wedge associated with the first element of the `data` array is placed at the top of the pie, and the remaining items of the `data` array are arranged in order clockwise around the remaining portion of the pie.

The `units$` array must contain the same number of items as the `data` array. Each element of the `units$` array will be used as a label for the wedge of the pie associated with the corresponding element of the `data` array. Each label will be connected to its associated wedge by a line. If an element of the `units$` array has a value of the null string, the associated wedge will have neither a label nor a connecting line.

The value of `colors$` determines the color scheme that will be used to draw the graph. It generally consists of at least four color names (in any combination of uppercase or lowercase letters) separated by spaces. The valid color names are:

RED	MAGENTA	YELLOW
GREEN	BLUE	CYAN
BROWN	WHITE	BLACK
	BACKGROUND	

The value of `colors$` may also contain color numbers instead of color names, allowing you to access any of the colors supported by the current computer system.

The first color specified by the value of `colors$` will be used for the graph's title. The second color will be used for the graph's frame. And the remaining colors will be used repeatedly for the wedges of the pie.

If the value of `wedge` fall between the lower and upper bounds of the `data` array, inclusive, the wedge of the pie associated with the element of `data` whose index is represented by the value of `wedge` will be exploded out of the pie. That is, it will be drawn slightly separated from the rest of the pie in order to draw the user's attention. If the value of `wedge` falls outside this range, no wedge will be exploded out of the pie.

If the value of `percent` is non-zero, each wedge will be labeled not only with the corresponding element of the `units$` array, but also with the percentage of the total which it represents. If the value of `percent` is 0, the wedges will be labeled only with the elements of the `units$` array. Note that the percentages are rounded before being displayed. Therefore, it is not guaranteed that they will add up to exactly 100%.

Example: The following program, BGPie.TRU, can be found in the directory TBDEMOS:

```
! BGPie   Draw a simple pie chart.

! Highlight hammers, and show percentages.

LIBRARY "..\TBLibs\BGLib.trc"

DIM data(5), units$(5)

MAT READ data, units$
DATA 120, 34, 87, 65, 21
DATA Nails, Hammers, Saws, Pliers, Awls

CALL SetTitle ("Honest Boy (tm) Product Income")
CALL PieChart (data, units$, "yellow green red", 2, 1)

GET KEY key

END
```

produces a pie chart representing income by product, highlighting hammers and displaying percentages with each label.

Exceptions:	100	Graph's title is too wide.
	106	Need greater width for graph.
	107	Need greater height for graph.
	111	Data and unit arrays don't match for PieChart.
	11008	No such color: *color*.

See also: **SETTITLE**

SETANGLE Subroutine

Library: SGLIB.TRC

Syntax: CALL SETANGLE (*strex*)

Usage: `CALL SETANGLE (measure$)`

Summary: Controls the manner in which subsequent polar graphs drawn by the various data and function plotting subroutines will interpret angle measurements.

Details: The **SETANGLE** subroutine is used to control the manner in which subsequent data and function polar plots produced by the **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**, **ADDFGRAPH**, and **MANYFGRAPH** subroutines will interpret angle measurements.

When these subroutines interpret angle measurements, they interpret them as radians by default. However, by passing a value of "DEG" as `measure$`, you can instruct them to interpret angles in degrees. Passing a value of "RAD" to the **SETANGLE** subroutine will reset the default interpretation.

Note that the **SETANGLE** subroutine only controls the interpretation of angular coordinates by polar graphs. Use the **SETGRAPHTYPE** subroutine to cause subsequent graphs to be drawn as polar graphs.

You can use the **ASKANGLE** subroutine to determine the manner in which the next data or function polar plot will interpret angular coordinates.

Example: None

Exceptions: None

See also: **ASKANGLE**, **SETGRAPHTYPE**, **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**, **ADDFGRAPH**, **MANYFGRAPH**

SETBARTYPE Subroutine

Library: BGLIB.TRC

Syntax: CALL SETBARTYPE (*strex*)

Usage: `CALL SETBARTYPE (type$)`

Summary: Controls the arrangement of the bars within each group of a multiple bar chart or histogram.

Details: The **SETBARTYPE** subroutine is used to control the arrangement of the bars within each group of a bar chart or histogram produced by a subsequent invocation of the **MULTIBAR** or **MULTIHIST** subroutine.

Both the **MULTIBAR** and **MULTIHIST** subroutines draw multiple bar-based graphs in a single frame. In such a graph, bars associated with a particular unit are grouped together.

The **SETBARTYPE** subroutine allows you to control how the bars in each group will be arranged by passing one of the following values in `type$`:

Types of Bar Groupings

`Type$` **value**	**Description**
`"SIDE"`	Bars arranged side by side with space between them
`"STACK"`	Bars stacked one above the other
`"OVER"`	Bars arranged side by side but overlapped slightly

The value of `type$` may be specified in any combination of uppercase and lowercase letters.

If the value of `type$` does not represent one of these values, an error will be generated.

By default, the bar type is set to a value of `"SIDE"`. You can use the **ASKBARTYPE** subroutine to report the current bar type setting.

Example: See the example programs in the desciptions of BALANCEBARS (BGBar3.TRU,) MULTIBAR (BGBar2.TRU,) and MULTIHIST (BGHisto2.TRU) for examples of the use of this subroutine.

Exceptions: 130 No such barchart type: *xxx*

See also: **ASKBARTYPE**, **MULTIBAR**, **MULTIHIST**

SETGRAIN Subroutine

Library: SGLIB.TRC

Syntax: CALL SETGRAIN (*numex*)

Usage: `CALL SETGRAIN (grain)`

Summary: Controls the grain with which subsequent invocations of the various function plotting subroutines will draw the line graph.

Details: The **SETGRAIN** subroutine controls the grain with which subsequent invocations of the **FGRAPH**, **ADDFGRAPH**, and **MANYFGRAPH** subroutines will draw the line representing the function.

These subroutines actually graph the curve of the function which they are plotting as a series of line segments. The ***grain*** controls the number of line segments used to form each graphed curve. The higher the value of the grain, the more line segments are used and the smoother the resulting curve appears. However, higher grains also mean more work for the computer, and this means that each curve takes longer to draw.

By default, the **FGRAPH**, **ADDFGRAPH**, and **MANYFGRAPH** subroutines use a grain value of 64, which means that each line graph is composed of 64 individual line segments. This value strikes a generally acceptable balance of smoothness and speed, but you can change this value by passing the new grain value in the `grain` argument to the **SETGRAIN** subroutine.

You can use the **ASKGRAIN** subroutine to report the current grain value.

Example: The following program, SGGrain.TRU, can be found in the directory TBDEMOS:

```
!  SGGrain   Demonstrate SetGrain.

LIBRARY "..\TBLibs\SGFunc.trc", "..\TBLibs\SGLib.trc"

OPEN #1: screen 0, .49, 0, 1

CALL SetGrain (10)
CALL SetTitle ("Grain = 10")
CALL Fgraph (-pi, pi, 1, "white white magenta")

OPEN #2: screen .5, 1, 0, 1

CALL SetGrain (100)
CALL SetTitle ("Grain = 100")
CALL Fgraph (-pi, pi, 1, "white white magenta")

GET KEY key

END

DEF F(x) = Sin(3*x)
```

demonstrates the use of the **SETGRAIN** subroutine by displaying two graphs of the same function side by side — one with a grain of 10 and the other with a grain of 100.

Exceptions: None

See also: **ASKGRAIN**, **FGRAPH**, **ADDFGRAPH**, **MANYFGRAPH**

SETGRAPHTYPE Subroutine

Library: SGLIB.TRC

Syntax: CALL SETGRAPHTYPE (*strex*)

Usage: `CALL SETGRAPHTYPE (type$)`

Summary: Controls the type of graph that will be drawn by subsequent data and function plotting subroutines.

Details: The **SETGRAPHTYPE** subroutine is used to control the type of graph that will be produced for subsequent data and function plots produced by the **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**, **ADDFGRAPH**, and **MANYFGRAPH** subroutines.

The type of subsequent graphs is determined by the value passed as `type$`. The possible values of `type$` are:

Types of Graphs

`Type$` **value**	**Description**
`"XY"`	Normal graph
`"LOGX"`	Semi-logarithmic graph with x-axis logarithmically scaled
`"LOGY"`	Semi-logarithmic graph with y-axis logarithmically scaled
`"LOGXY"`	Logarithmic graph with both x- and y-axes logarithmically scaled
`"POLAR"`	Polar graph

Logarithmic and semi-logarithmic graphs look very similar to normal graphs, but one or both of the axes is scaled logarithmically.

Polar graphs, however, look quite different from normal graphs in that they are circular. For this reason, the horizontal and vertical labels are ignored for polar graphs; only the title is shown.

When a graphing routine is used to draw a polar graph, what would normally be the x- and y-coordinates are interpreted as r and theta (or distance and angle) coordinates, respectively. Therefore, as you might expect, the function plotting subroutines expect to find an externally defined function in the form `r = F(theta)`.

Polar graphs interpret angle measures as radians by default, but you can change this interpretation using the **SETANGLE** subroutine.

You can use the **ASKGRAPHTYPE** subroutine to determine the type of graph that will be used for the next data or function plot.

Example: See the example program in the description of MANYDATAGRAPH (SGData3.TRU) for an example of the use of this subroutine.

Exceptions: None

See also: **ASKGRAPHTYPE**, **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**, **FGRAPH**, **ADDFGRAPH**, **MANYFGRAPH**

SETGRID Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL SETGRID (*strex*)

Usage: `CALL SETGRID (style$)`

Summary: Controls the presence, direction, and type of the grid within subsequently drawn charts and graphs.

Details: The **SETGRID** subroutine is used to control the presence, direction, and type of the grid within the frame of graphs or charts drawn by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH** subroutines.

The **SETGRID** subroutine allows you to control the presence and direction of the grid lines by passing one of the following values in `style$`:

Available Grid Directions

`Style$` **value**	**Description**
`""`	No grid lines
`"H"`	Horizontal grid lines only
`"V"`	Vertical grid lines only
`"HV"`	Both horizontal and vertical grid lines

The value of `style$` may be specified in any combination of uppercase and lowercase letters. In addition, the value of `style$` may include instructions that indicate the type of grid lines that you would like drawn. By default, grid lines are drawn as solid lines. However, you can append one of the following modifiers to a letter in the value of `style$` to specify a different line type for grid lines traveling in that direction:

Available Grid Type Modifiers

Modifier	**Description**
`-`	Dashed grid lines
`.`	Dotted grid lines
`-.`	Dash-dotted grid lines

For example, passing a value of `"H-.V"` for `style$` would result in dash-dotted grid lines in the horizontal direction and solid grid lines in the vertical direction.

If the value of `type$` does not represent a valid value, however, an error will be generated.

By default, the grid lines are turned off. You can use the **ASKGRID** subroutine to report the current grid setting.

Example: See the example program in the description of MULTIBAR (BGBar2.TRU) for an example of the use of this subroutine.

Exceptions: 113 No such SetGrid direction: *xxx*

See also: **ASKGRID**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

SETHLABEL Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL SETHLABEL (*strex*)

Usage: `CALL SETHLABEL (hlabel$)`

Summary: Sets the value of the horizontal label which will be displayed for subsequently drawn charts and graphs.

Details: The **SETHLABEL** subroutine is used to set the value of the horizontal label that will be used to label the frame of graphs or charts drawn by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, and **MANYDATAGRAPH** subroutines.

The **SETHLABEL** subroutine expects the value of the horizontal label to be passed as `hlabel$`. Passing a null string effectively eliminates the horizontal label.

If the value you set for the horizontal label exceeds the available room, the graphing subroutine which draws the next graph will generate an error.

There is no default value for the horizontal label. Therefore, if you want it to appear, you will need to specify its values before drawing the graph.

You may specify new values for the title, the horizontal label, and the vertical label simultaneously using the **SETTEXT** subroutine. Use the **SETVLABEL** and **SETTITLE** subroutines to set the values of the vertical label and the title, respectively.

You may use the **ASKHLABEL** subroutine to report the current value of the horizontal label.

Example: None

Exceptions: None

See also: **ASKHLABEL**, **SETTEXT**, **SETVLABEL**, **SETTITLE**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **PIECHART**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

SETLAYOUT Subroutine

Library: BGLIB.TRC

Syntax: CALL SETLAYOUT (*strex*)

Usage: `CALL SETLAYOUT (direction$)`

Summary: Controls the direction of the bars within subsequently drawn bar charts and histograms.

Details: The **SETLAYOUT** subroutine is used to control the direction of the bars within each bar chart or histogram produced by a subsequent invocation of the **MULTIBAR** or **MULTIHIST** subroutine.

The **SETLAYOUT** subroutine allows you to control the direction in which the bars will be drawn by passing one of the following values in `direction$`:

Types of Bar Groupings

`Direction$` **value**	**Description**
`"HORIZONTAL"`	Bars oriented horizontally
`"VERTICAL"`	Bars oriented vertically

The value of `type$` may be specified in any combination of uppercase and lowercase letters. In addition, the value of `type$` may be truncated to any number of letters. That is, values of `"H"` and `"V"` will suffice. If the value of `type$` does not represent a valid value, however, an error will be generated.

By default, the bar direction is set to a value of `"VERTICAL"`. You can use the **ASKLAYOUT** subroutine to report the current bar layout setting.

Example: See the example program in the description of MULTIBAR (BGBar2.TRU) for an example of the use of this subroutine.

Exceptions: 131 No such barchart direction: *xxx*

See also: **ASKLAYOUT**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**

SETLS Subroutine

Library: SGLIB.TRC

Syntax: CALL SETLS (*numex*)

Usage: `CALL SETLS (flag)`

Summary: Controls whether least-squares linear fits will be drawn automatically for subsequent data plots.

Details: The **SETLS** subroutine is used to control whether or not least-squares linear fits will be drawn automatically for subsequent data plots produced by the **DATAGRAPH**, **ADDDATAGRAPH**, and **MANYDATAGRAPH** subroutines.

The least-squares linear fit of a data plot is the straight line which best fits the locations of the data points. That is, the least-squares linear fit of a data plot is the straight line which minimizes the vertical distance between itself and each of the data points which form the plot. Such a line may be used to help predict where data points might lie in areas of the graph for which data is unavailable.

By default, the **DATAGRAPH**, **ADDDATAGRAPH**, and **MANYDATAGRAPH** subroutines draw the data plots without displaying the least-squares linear fit of the data points. However, invoking the **SETLS** subroutine with the value of `flag` equal to 1 will instruct subsequent invocations of these routines to add such a linear fit to each graph they draw. You may then turn off this line fitting by invoking the **SETLS** subroutine again with the value of `flag` equal to 0.

When the **DATAGRAPH** and **ADDDATAGRAPH** subroutines draw a linear fit, they draw a solid line in the data color. The **MANYDATAGRAPH** subroutine draws each graph's linear fit in the same color and line style as the lines connecting that graph's data points.

You can use the **ASKLS** subroutine to determine whether least-squares linear fitting is currently active or inactive.

Example: None

Exceptions: None

See also: **ASKLS**, **ADDLSGRAPH**, **DATAGRAPH**, **ADDDATAGRAPH**, **MANYDATAGRAPH**

SETTEXT Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL SETTEXT (*strex*, *strex*, *strex*)

Usage: `CALL SETTEXT (title$, hlabel$, vlabel$)`

Summary: Sets the values of the title, horizontal label, and vertical label which will be displayed for subsequently drawn charts and graphs.

Details: The **SETTEXT** subroutine is used to set the values of the title, horizontal label, and vertical label that will be used to label the frame of graphs or charts drawn by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, and **MANYDATAGRAPH** subroutines. (These values also apply to the **PIECHART** subroutine, but only the value of the title is used.)

The **SETTEXT** subroutine expects the value of the title to be passed as `title$`, the value of the horizontal label to be passed as `hlabel$`, and the value of the vertical label to be passed as `vlabel$`. Passing a null string for any of these values effectively eliminates that label.

If the values you set for one or more of these labels exceeds the available room, the graphing subroutine which draws the next graph will generate an error.

There are no default values for the title, the horizontal label, or the vertical label. Therefore, if you want any of them to appear, you will need to specify their values before drawing the graph.

You may specify a new value for the title, the horizontal label, or the vertical label individually using the **SETTITLE**, **SETHLABEL**, or **SETVLABEL** subroutines, respectively.

You may use the **ASKTEXT** subroutine to report the current values of the title, the horizontal label, and the vertical label.

Example: See almost all the example programs described in this section for examples of the use of this subroutine.

Exceptions: None

See also: **ASKTEXT**, **SETTITLE**, **SETHLABEL**, **SETVLABEL**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **PIECHART**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

SETTITLE Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL SETTITLE (*strex*)

Usage: `CALL SETTITLE (title$)`

Summary: Sets the value of the title which will be displayed for subsequently drawn charts and graphs.

Details: The **SETTITLE** subroutine is used to set the value of the title that will be used to label the frame of graphs or charts drawn by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**, and **PIECHART** subroutines.

The **SETTITLE** subroutine expects the value of the title to be passed as `title$`. Passing a null string effectively eliminates the title.

If the value you set for the title exceeds the available room, the graphing subroutine which draws the next graph will generate an error.

There is no default value for the title. Therefore, if you want it to appear, you will need to specify its values before drawing the graph.

You may specify new values for the title, the horizontal label, and the vertical label simultaneously using the **SETTEXT** subroutine. Use the **SETHLABEL** and **SETVLABEL** subroutines to set the values of the horizontal label and the vertical label, respectively.

You may use the **ASKTITLE** subroutine to report the current value of the title.

Example: See the examples programs in the descriptions of SETGRAIN (SGGrain.TRU) and SORTPOINTS (SGSortPt.TRU) for examples of the use of this subroutine.

Exceptions: None

See also: **ASKTITLE**, **SETTEXT**, **SETHLABEL**, **SETVLABEL**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **PIECHART**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

SETVLABEL Subroutine

Library: BGLIB.TRC or SGLIB.TRC

Syntax: CALL SETVLABEL (*strex*)

Usage: `CALL SETVLABEL (vlabel$)`

Summary: Sets the value of the vertical label which will be displayed for subsequently drawn charts and graphs.

Details: The **SETVLABEL** subroutine is used to set the value of the vertical label that will be used to label the frame of graphs or charts drawn by subsequent invocations of the **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, and **MANYDATAGRAPH** subroutines.

The **SETVLABEL** subroutine expects the value of the vertical label to be passed as `vlabel$`. Passing a null string effectively eliminates the vertical label.

If the value you set for the vertical label exceeds the available room, the graphing subroutine which draws the next graph will generate an error.

There is no default value for the vertical label. Therefore, if you want it to appear, you will need to specify its values before drawing the graph.

You may specify new values for the title, the horizontal label, and the vertical label simultaneously using the **SETTEXT** subroutine. Use the **SETHLABEL** and **SETTITLE** subroutines to set the values of the horizontal label and the title, respectively.

You may use the **ASKVLABEL** subroutine to report the current value of the vertical label.

Example: None

Exceptions: None

See also: **ASKVLABEL**, **SETTEXT**, **SETHLABEL**, **SETTITLE**, **BARCHART**, **MULTIBAR**, **HISTOGRAM**, **MULTIHIST**, **IBEAM**, **PIECHART**, **FGRAPH**, **MANYFGRAPH**, **DATAGRAPH**, **MANYDATAGRAPH**

SETXSCALE Subroutine

Library: SGLIB.TRC

Syntax: CALL SETXSCALE (*numex, numex*)

Usage: `CALL SETXSCALE (70, 170)`

Summary: Turns off auto-scaling and sets the x-range for subsequent graphs.

Details: The **SETXSCALE** subroutine is used to set the value of the x-scale for subsequent graphs. It turns off auto-scaling. The actual x-range may be slightly different as this subroutine may round to "good-looking" numbers.

Example: None.

Exceptions: None

See also: **SETYSCALE**

SETYSCALE Subroutine

Library: SGLIB.TRC

Syntax: CALL SETYSCALE (*numex, numex*)

Usage: `CALL SETYSCALE (70, 170)`

Summary: Turns off auto-scaling and sets the y-range for subsequent graphs.

Details: The **SETYSCALE** subroutine is used to set the value of the y-scale for subsequent graphs. It turns off auto-scaling. The actual y-range may be slightly different as this subroutine may round to "good-looking" numbers.

Example: See the example in the description of ADDLSGRAPH (SLSquar.TRU) for an example of the use of this subroutine.

Exceptions: None

See also: **ADDLSGRAPH, SETXSCALE**

SORTPOINTS Subroutine

Library: SGLIB.TRC

Syntax: CALL SORTPOINTS (*numarrarg*, *numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL SORTPOINTS (x(), y())`

Summary: Sorts the one-dimensional parallel arrays `x` and `y` into ascending order by the values of stored in the `x` array.

Details: The **SORTPOINTS** subroutine sorts the parallel, one-dimensional arrays `x` and `y` into ascending order by the values stored in the `x` array.

Parallel arrays are simply arrays in which elements with identical subscripts are related. For instance, the `x` and `y` arrays are considered to be parallel if the first element of the `x` array is related to the first element of the `y` array, the second to the second, and so forth for each element in both arrays. When parallel arrays are sorted, the elements in both arrays are rearranged in an identical manner so as to maintain these relationships.

The **SORTPOINTS** subroutine is useful for sorting the arrays of coordinates passed into the **DATAGRAPH** and **ADDDATAGRAPH** subroutines, but it can be used to sort any pair of one-dimensional parallel arrays.

To sort two-dimensional arrays in a similar fashion, use the **SORTPOINTS2** subroutine. To sort a single one-dimensional array, use the **SORTN** subroutine.

Example: The following program, SGSortPt.TRU, can be found in the directory TBDEMOS:

```
!  SGSortPt   Display unsorted vs. sorted data points.

LIBRARY "..\TBLibs\SGLib.trc"

DIM x(10), y(10)

FOR i = 1 to 10                     ! Get some unsorted data points
    LET x(i) = rnd
    LET y(i) = rnd
```

```
NEXT i

OPEN #1: screen 0, .49, 0, 1      ! Left: unsorted points
CALL SetTitle ("Unsorted")
CALL DataGraph (x, y, 10, 3, "")

OPEN #2: screen .5, 1, 0, 1       ! Right: sorted points
CALL SetTitle ("Sorted")
CALL SortPoints (x, y)
CALL DataGraph (x, y, 10, 3, "")

GET KEY key

END
```

demonstrates the usefulness of using the **SORTPOINTS** subroutine with the **DATAGRAPH** subroutine.

Exceptions: None

See also: **SORTPOINTS2**, **DATAGRAPH**, **ADDDATAGRAPH**, **SORTN**

SORTPOINTS2 Subroutine

Library: SGLIB.TRC

Syntax: CALL SORTPOINTS2 (*numarrarg*, *numarrarg*)

numarrarg:: *numarr*
numarr bowlegs

Usage: `CALL SORTPOINTS2 (x(,), y(,))`

Summary: Sorts the parallel rows of the two-dimensional arrays `x` and `y` into ascending order by the values of stored in rows of the `x` array.

Details: The **SORTPOINTS2** subroutine sorts the elements of the parallel rows of the two-dimensional arrays `x` and `y` into ascending order by the values stored in the rows of the `x` array.

Parallel arrays are simply arrays in which elements with identical subscripts are related. For instance, the `x` and `y` arrays are considered to be parallel if the first element of the `x` array is related to the first element of the `y` array, the second to the second, and so forth for each element in both arrays. When parallel arrays are sorted, the elements in both arrays are rearranged in an identical manner so as to maintain these relationships.

The **SORTPOINTS2** subroutine treats corresponding rows of the `x` and `y` arrays as individually parallel one-dimensional arrays. That is, the elements of each pair of corresponding rows are rearranged independently of the other rows.

The **SORTPOINTS2** subroutine is useful for sorting the arrays of coordinates passed into the **MANYDATAGRAPH** subroutine, but it can be used to sort any pair of two-dimensional arrays with parallel rows.

To sort one-dimensional arrays in a similar fashion, use the **SORTPOINTS** subroutine. To sort a single one-dimensional array, use the **SORTN** subroutine.

Example: None

Exceptions: None

See also: **SORTPOINTS**, **MANYDATAGRAPH**, **SORTN**

CHAPTER

24

Calling C Routines in True BASIC

True BASIC is a powerful language, and can accomplish many things for you. However, you may find on occasion that you would like to extend the language to do something that can't easily or quickly be done in True BASIC. You may wish to rewrite a routine to execute more quickly, or you may wish to interact with the environment in a way that True BASIC does not support. With the Gold Edition of True BASIC, you can write these routines in C, link the routines into your True BASIC program, and call them using the processes outlined below.

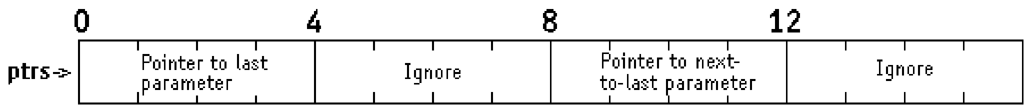
There are two basic strategies used for linking and calling C routines in True BASIC: one for the Macintosh, one for Windows and OS/2. Those of you familiar with using C or assembly routines in older versions of the Macintosh will find the current strategy for Macintoshes similar to that used in the older versions of True BASIC.

In the notes below, we assume that you are familiar with the appropriate C compilers and linkers. If you are not, you will need to familiarize yourself with C in general, and the specific compilers and linkers you will use before you will be able to get to much out of these notes. At the moment, we use CodeWarrior for the Macintosh, Visual C/C++ 6.0 for Win32, and VisualAge C++ for OS/2. Other compilers should work as well, though.

General notes:

1 Your C routine will be passed one parameter: a pointer to the list of pointers to the parameters passed by your True BASIC program. This list of pointers contains a long word with a pointer to the parameter, followed by a long word you can ignore, followed by the next long word with a pointer to a parameter, followed by another long word you can ignore, etc.

```
main(char *ptrs)
{
}
```



2 You need to know how many parameters there are and in what order they will be.

3 Note that the pointers are in the list in the reverse order from the calling sequence. Thus, if you issue the statement `CALL MyCRoutine(a,b$)`, the first parameter you find on the list passed to the C routine is `b$`, followed by `a`.

4 Each parameter will be either a True BASIC string, a True BASIC number, or a True BASIC array. We have included a file named `tbcdefs.h` with definitions of each of these items. Note that the definitions are different for the Mac than for Windows or OS/2. The fields all have the same names, but the difference in byte-ordering means the definitions appear in different orders.

5 We have included a simple example on the diskette or CD for each platform.

6 Note that you don't *have* to use C--any language that will let you create a CODE resource for the Mac or a DLL for Windows or OS/2, and will let you access the pointer to the True BASIC variables will do fine.

Macintosh:

The Process:

1 Write your C routine. (See the The Rules below).

2 Compile and link it as a CODE resource with both Type and ResType of TRU2 and ResID of 0. Give it an appropriate library name.

3 Run the Mac version of `finaltouch` over it.

4 Use a `LIBRARY` statement in your True BASIC source code to use the result.

The Rules:

1 Your routine should be of type `int`.

2 You will be limited in what you will be able to do in the C routine because it must fit in a code resource. For example, you will probably be unable to use most of the routines in the standard C runtime library. You will, however, be able to use Toolbox routines directly.

3 True BASIC uses 4-byte ints, 8-byte doubles, and 68K struct alignment. You'll need to follow the same standards.

Finaltouch:

When running `finaltouch`, you will be asked for the `SUB` or `DEF` statement that defines your routine. Just type the `SUB` or `DEF` line that your routine would have if it were written in True BASIC, complete with its list of arguments. `Finaltouch` then asks for the name of the file that contains your C routine. Supply the name you gave it in item 2 of "The Process" above.

You'll also need to know:

1 Remember that True BASIC for the Mac is a Fat binary--it will run on either PPC or 68K machines. Your C routine, however, will be compiled and linked for one or the other. If the user tries to run it on the wrong platform, it will crash. You can get around this by creating two libraries, one for 68K and one for PPC. Link both of them into your True BASIC program with `LIBRARY` statements. Then, use `TC_GetSysInfo` or `Object` with `OBJM_SYSINFO` to get the value of the attribute `ENV`, passing in `"ISA"` as the variable you're looking for. If the result contains "PPC", you are running on PPC. Otherwise you're running on 68K:

```
dim v(1)
let env$ = "ISA"
let v(1) = 0
call object(9,0,"ENV",env$,v)
if pos(env$,"PPC") > 0 then
      call... ! PPC version
else
      call... ! 68K version
end if
```

Sample C Routine:

A sample routine has been provided, `finfo.c`, which allows you to get and set the file type and creator for a file.

Windows and OS/2:

The Process:

1 Write your C routine. (See the The Rules below).

2 Compile and link it as a DLL.

3 Run the Windows or OS/2 version of `finaltouch` over it. (**Finaltouch** is distributed as `FTOUCH.EXE`.)

4 Use a `LIBRARY` statement in your True BASIC source code to use the result.

The Rules:

1 Your routine should be of type `void`.

Finaltouch:

When running `finaltouch`, you will be asked for the `SUB` or `DEF` statement that defines your routine. Just type the `SUB` or `DEF` line that your routine would have if it were written in True

BASIC, complete with its list of arguments. `Finaltouch` then asks for the name of the library file. This is the name that you want to use in the `LIBRARY` statement, for example, `myfile.trc`. `Finaltouch` next asks for the name of the DLL. Give just the basename: don't add the ".DLL". Thus, for `MYRTN.DLL`, you would just answer `MYRTN`. Last, finaltouch asks for the name of the routine in the DLL. This is the name that you gave the C routine (or the name that you exported, if they are different). The resulting file will be the library file for True BASIC.

You'll also need to know:

1. The DLL will need to be in the same directory as True BASIC or in the path defined by the PATH (Windows) or LIBPATH (OS/2) environment variable when you run your True BASIC program. If the system can't find a DLL of the appropriate name, a "File not found" exception (9003) will be raised when the C subroutine is called.
2. If the DLL does not contain a routine of the name you specified when running finaltouch, a "No such function or subroutine" exception (-6) will be raised.

Sample C Routine:

A sample routine has been provided, `tbplysnd.c`, which allows you to play a given .WAV sound file (Windows example; no OS/2 example currently provided).

Additional Resources:

Here are several web resources that contain useful information about C in general, and the specific compilers and linkers we mention in this Guide.

`http://www.metrowerks.com/` (CodeWarrior)

`http://msdn.microsoft.com/visualc/` (Visual C++)

`http://www.software.ibm.com/ad/visualage-c++/index.html` (VisualAge C++)

CHAPTER

25

Using SOCKET Routines in True BASIC

The Internet is everywhere today. True BASIC's socket routines will allow you to use True BASIC to write programs to access other machines and servers on the Internet or any intranet running the TCP/IP protocol.

True BASIC's implementation generally follows the standard Berkeley sockets implementation, so those familiar with Berkeley sockets should feel right at home. The socket library written by Charlie Reiman of NCSA was helpful in developing these routines.

The first few paragraphs of this document will be a summary of how True BASIC matches up with Berkeley sockets. Those less familiar should read past for a somewhat more in-depth description, although these notes are not meant to serve as a sockets primer. There are several primers both in print and on the web. A good place to start might be:

`http://world.std.com/~jimf/papers/sockets/sockets.html`

True BASIC's socket library is called `TrueSock.trc`. Make sure you include this in a `LIBRARY` statement in your program, as well as `DECLARE DEF` statements for any of the routines below defined as `DEF`s.

SUB TS_Init

No analog in the Berkeley sockets world. Call this subroutine first to set up the constants mentioned below.

DEF TS_Socket(family,type,protocol)

Analogous to the `socket()` function. `family` should be one of `AF_UNIX` or `AF_INET`. `type` should be one of `SOCK_STREAM` or `SOCK_DGRAM`. protocol should be one of `IPPROTO_TCP` or `IP_PROTO_UDP`. Return value is the socket id. On some platforms, in addition to socket errors, you may get the error "No such file." This means that the platform's socket DLLs, necessary to use the socket features of True BASIC, are not installed, and you will need to install them. If you are unsure how to do this, you will need review your operating system's documentation for information on how to find and install socket support. For Windows, this must be Winsock 2.x, which is WS2_32.DLL, not Winsock 1.1.

Some Windows 95 installations may not have Winsock 2 installed. An update is available at:

`http://www.microsoft.com/windows/downloads/bin/W95ws2setup.exe`

SUB TS_Bind(tb_socket,family,port,addr$)

Analogous to the `bind()` function. `tb_socket` should be a valid socket id. `family` should be one of `AF_UNIX` or `AF_INET`. `addr$` takes one of several forms. For `AF_UNIX`, it is simply a string with a filename. `TS_Bind` will take care of combining it with the other information to make a standard address. For `AF_INET`, we expect either a string version of the address as an integer, or the dotted notation. For example, for the address 192.168.0.12, the value of addr$ should be either "192.168.0.12" or "3232235532". If you have no preference for a port and address, pass "0" for these arguments. Again, `TS_Bind` will take care of turning it into a standard address.

SUB TS_Connect(tb_socket,family,port,addr$)

Analogous to the `connect()` function. Arguments are the same as for `TS_Bind`.

DEF TS_Receive$(tb_socket,num_bytes)

Analogous to the `recv()` function. `tb_socket` is a valid socket id. `num_bytes` is the number of bytes you want to receive. The return value is a string containing the data received.

SUB TS_Send(tb_socket,s$)

Analogous to the `send()` function. `tb_socket` is a valid socket id. `s$` is the data to send.

SUB TS_Listen(tb_socket,backlog)

Analogous to the `listen()` function. `tb_socket` is a valid socket id. `backlog` is maximum length of the queue of pending connections.

DEF TS_Accept(tb_socket,family,port,addr$)

Analogous to the `accept()` function. Arguments are the same as for `TS_Bind`. Return value is the id for the accepted socket.

SUB TS_Close(tb_socket)

Analogous to the `close()` function (or, for those of you more used to WinSock, the `closesocket()` function.) `tb_socket` is a valid socket id.

DEF TS_GetHostByName$(name$)

Analogous to the `gethostbyname()` function. `name$` is the name of host you want an address for. The return value is an address in dotted notation suitable for passing to any of the routines above which require such an address.

For those who would like a bit more in the way of explanation about how to use the True BASIC sockets library, we present the following example, which we will explain in the paragraphs following. The point of the example is to connect briefly to an ftp server, then disconnect again. In order for the example to work, you must have an active connection to the Internet. First, the full example:

```
LIBRARY "truesock.trc"
DECLARE PUBLIC AF_INET,SOCK_STREAM,IPPROTO_TCP
DECLARE DEF TS_Socket,TS_Receive$,TS_GetHostByName$

CALL TS_Init

LET s = TS_Socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)

CALL TS_Connect(s,AF_INET,21,TS_GetHostByName$("ftp.truebasic.com"))

LET r$ = TS_Receive$(s,100)
PRINT "> ";r$

CALL TS_Send(s,"QUIT"&chr$(10))
LET r$ = TS_Receive$(s,75)
PRINT "> ";r$

CALL TS_Close(s)

END
```

Now, a section-by-section explanation:

```
LIBRARY "truesock.trc"
DECLARE PUBLIC AF_INET,SOCK_STREAM,IPPROTO_TCP
DECLARE DEF TS_Socket,TS_Receive$,TS_GetHostByName$
```

We use the `LIBRARY` statement to link in True BASIC's socket routines. The `DECLARE PUBLIC` statement allows us to reference common "constants" set up in the library. These constants will be explained further below. We use `DECLARE DEF` as True BASIC requires to declare those functions we are going to use.

```
CALL TS_Init
```

We call the routine `TS_Init` to initialize all of the "constants" necessary to communicate with the sockets library. These constants include those referenced above in the `DECLARE PUBLIC` statement. This routine should always be called before any other socket routines are called.

```
LET s = TS_Socket(AF_INET,SOCK_STREAM,IPPROTO_TCP)
```

Now, we create a socket. We need to supply three pieces of information: the protocol *family*, the socket *type*, and the socket *protocol*.

The protocol family can be one of either `AF_INET` or `AF_UNIX`. Other families are available in some implementations of sockets, but at the moment, these are the only two that True BASIC supports. Pass `AF_INET` to use Internet protocols, generally used for communications between machines; pass AF_UNIX to use internal protocols, generally used for creating local pipes or process-to-process communications. Because we want to connect to an ftp server on another machine, we pass AF_INET.

The socket type can be one of either SOCK_STREAM or SOCK_DGRAM. Again, other types are available in some implementations of sockets, but at the moment, these are the only two that True BASIC supports. The type SOCK_STREAM provides a reliable means to create a two-way connection that supports byte streams where data of any size may be sent and received. The type SOCK_DGRAM provides unreliable messages of a fixed maximum size. For most applications for which you will use True BASIC's sockets implemetation, you will use SOCK_STREAM, as we will here.

The socket protocol can be one of either IPPROTO_TCP or IPPROTO_UDP. The protocol you use will be determined largely by the type of socket you create. IPPROTO_TCP is a transmission control protocol; IPPROTO_UDP is a user datagram protocol. We are using SOCK_STREAM, so we will use IPPROTO_TCP. Just as with the socket type, for most applications for which you will use True BASIC's sockets implemetation, you will use IPPROTO_TCP,

```
  CALL TS_Connect(s,AF_INET,21,TS_GetHostByName$("ftp.truebasic.com"))
```

Having set up a socket to use, it is now time to actually make a connection to the ftp server. We need to supply four pieces of information: the *socket*, the protocol *family*, the *port* to connect to on the host, and the *address* of the host to connect to.

The first, the socket, is easy – we pass the socket id we received in the previous step. The protocol family is easy as well – we pass the same protocol family that we specified in the previous step.

The value for the port tells True BASIC what port you want to connect to on the host. One host may run a number of different servers (ftp, http, telnet, etc.) While they all have the same address, they run on different ports. We pass port number 21, as that is the normal port for an ftp server. Some common ports are:

ftp:	21
telnet:	23
smtp:	25
gopher:	70
http:	80
pop3:	110

The address of the host is also not too hard. You may know the address in dotted notation, in which case you can use it directly; for example, 192.168.0.1. However, if you're actually connecting via the Internet instead of an internal network, the address may change, and in many cases you won't know the address. So, we use the TB_GetHostByName$ function to turn a host's name into a numeric address, which we can then pass as the address.

```
LET r$ = TS_Receive$(s,100)
PRINT "> ";r$
```

Having connected to an ftp server, we know we are going to get a message. So, we call the function `TS_Receive$`, telling it which socket to use (the same we've been using) and how many bytes to look for, in this case, 100. If there is nothing to receive, it will wait until there is something; if there is something, it will return whatever there is, even if it is fewer than 100 bytes. The bytes sent by the server are returned by the function. In our example, they are placed in `r$`. In this case, if you are running the program, you should see something like this:

```
220 zaphod FTP server (Version wu-2.4.2-academ[BETA-15](1) Sun Dec
14 12:16:04 EST 1997) ready.
```

What you see (if there is anything to receive) will vary from server to server and from port to port. Note that you need to have some idea of what to expect from the server. You don't want to call `TS_Receive$` unless you expect there to be something to receive.

```
CALL TS_Send(s,"QUIT"&chr$(10))
LET r$ = TS_Receive$(s,75)
PRINT "> ";r$
```

Now, we are just going to quit. We know that the way to quit from an ftp server is to send the message "QUIT" with an end-of-line character (which on Unix is ASCII 10, although others will probably work.) So, we call `TS_Send`, once again passing the socket id, and passing the string "QUIT" with the end-of-line character appended.

We call `TS_Receive$` once again to get the ftp server's final message, which will look something like this:

```
221 Goodbye.
```

```
CALL TS_Close(s)
```

That's all we wanted to do with that socket, so we call TS_Close, passing the socket id, to close and release the socket.

CHAPTER

26

True BASIC SQL Libraries

TrueTrue BASIC provides a set of library routines for accessing SQL engines in a consistent way across platforms. This library is called `tbsqllib.trc`. To use the routines in this library, include the following statement at the beginning of your program or module:

```
LIBRARY "tbsqllib.trc"
```

This library has been written chiefly for the purpose of providing an interface to an SQL database engine. The routines let you establish a connection with a database, send queries to that database, and retrieve results. The results are returned as True BASIC arrays. This array format makes it easier for passing results to the toolkit routines for graphing or doing statistical analysis.

The examples presented in this manual, however, are designed to provide a template for using the database routines and are therefore limited to showing you how to select fields from a database table and insert entries into that table. As you learn about SQL, you can use this template for more complex queries that join results from different tables or that create intermediate result sets.

Overview of server-database-table relations.

The one idea to keep in mind as you work with these routines is that there are really four levels at which you can specify a connection: (a) the SQL server (e.g., MySQL), (b) the specific database, (c) the tables, and (d) the fields in the individual tables, referenced by row and column.

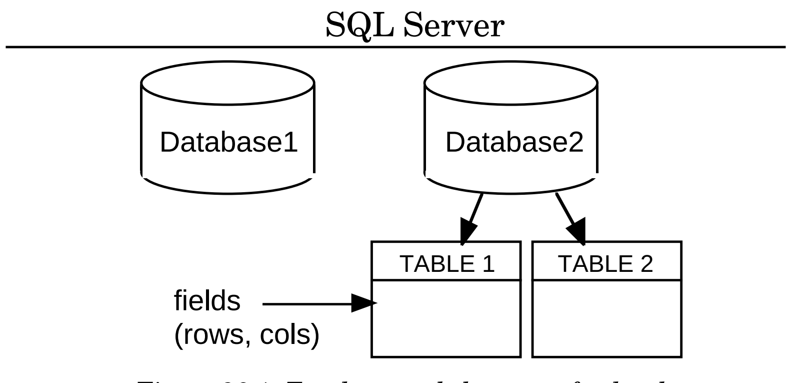


Figure 26-1. *Fundamental elements of a database.*

True BASIC's SQL routines use a "context" identifier (a number) to keep track of the connections to different databases even if they are on the same server. This context value gets returned to you when you connect to a SQL server and database. You then pass this value to all future TB SQL routines to indicate which connection your call refers to.

A single built-in function `DEF sql_connect(MYHOST$, DBNAME$, MYUSERNAME$, MYPASSWORD$)` connects you to the host server and database. You specify the table you want in the actual SQL statements that you issue.

The function returns an integer value called the context, which must be used in subsequent subroutine calls. Most SQL statements have the table name and fields embedded in the SQL statement, determining the exact server, database, table, and field that you are referencing.

For example, "`SELECT id_participant FROM participant_table`" will select all the participant ids from the participant table. The context identifier gets passed as part of the subroutine call to indicate the database:

```
CALL sql_query(context,query$,rows,cols)
```

To retrieve the results and place them into a True BASIC array called "results$(,)", you would use:

```
CALL sql_getallresults(context,result$(rows,cols))
```

See the QuickStart guide in this chapter for a complete template for establishing connections and issuing SQL queries.

Connecting to multiple databases.

The TB SQL interface lets you connect to multiple databases on the same or different servers. You can connect directly to the database engine for supported databases (e.g., MySQL) or through the ODBC interface to access most other databases. (For Win32, Mac OS, and OS/2, all connections will be made through the ODBC interface.) Note that the same set of routines is used for both situations. You simply need to call the appropriate routines in `TBSQLLIB`. True BASIC will automatically use a direct connection or the ODBC interface depending on the platform.

In many cases, you will be connecting to a single database engine on a specific machine. This is the scenario we will present in our examples; they can be modified easily to connect to multiple databases on different machines

ODBC, currently supported in our Macintosh, Windows, and OS/2 products, is a system of APIs and drivers for database access which allows one set of routines to access any ODBC-compliant database, eliminating the need to code for each database separately. This allows us, and you, to support any number of database engines with a minimum of effort. You must have installed the operating system's ODBC support before using True BASIC's ODBC routines.

Getting help on SQL & ODBC.

It is beyond the scope of this manual to provide a detailed overview of database management and SQL (the standard query language). Information on SQL and specific database engines is available at:

Learning SQL

Introduction to SQL: `http://w3.one.net/~jhoffman/sqltut.htm`

MySQL

Homepage for MySQL: `http://www.tcx.se`
Gamma release of MySQL manual: `http://www.tcx.se/Manual/manual_toc.html`

Oracle

Oracle: `http://temp.redhat.com/linux-info/ldp/HOWTO/Oracle-HOWTO.html`

ODBC

Microsoft: `http://www.microsoft.com/data/odbc`

ODBC for Linux

`http://www.dharma.com/`

More on SQL & ODBC for Linux

`http://www.linuxmall.com`
`http://www.xnet.com/`

Quick Start

The general sequence of steps for accessing a database is to connect to the database, query it, retrieve any results, possibly update an entry, and close the connection. For example,

```
LIBRARY "tbsqllib.trc"

DECLARE DEF sql_connect
DIM result$(0,0)

WHEN ERROR IN
  ! & at end and beginning of lines indicates line continuation
  LET context = sql_connect("localhost","inventory","myname", "mypassword")
  CALL sql_query(context,"SELECT product,quantity from WAREHOUSE", &
    & rows,cols) ! & at end and beginning indicates line continuation
  IF rows = 0 then
    ! no products in WAREHOUSE table.
    ! this is more likely when you have a WHERE clause.
  ELSE
    CALL sql_getallresults(context,result$(rows,cols))
    FOR i = 1 to rows
      ! display results
      PRINT "Product: " & result$(i,1)
      PRINT "Quantity: " & result$(i,2)
      PRINT
    NEXT i
  END IF
USE
  PRINT "ERROR: " & extext$
  PRINT exline$
END WHEN
CALL sql_close(context)
PRINT "done."
END
```

[!] **NOTE: You need to declare the function** `sql_connect` **in your programs because it is defined in the tbsqllib library as opposed to being built into the language. In the example above, the host is “localhost”, the database name is “inventory”, the username is “myname”, and the password is “mypassword”. The query returns the product and quantity for all items in the WAREHOUSE database and then prints a listing of them.**

The call to SUB sql_query does not return the actual results of the query but does tell you how many rows and columns matched your query.

The call to sql_getallresults returns all of the results for the query and places them in the array result$(,). For large result sets, you should use SUB sql_getresults which allows you to specify a subset of the rows to return. Otherwise, the call to sql_getallresults may take a while to return and will use up substantial amounts of memory.

[!] **NOTE: Some SQL statements such as INSERTs or UPDATEs will not generate a retrievable set of results, though the call the sql_query may return the number of affected rows depending on your SQL engine**.

The other important technique to notice here is that the code is enclosed in an error handler with the call to SUB sql_close after the handler. This format ensures that the connection gets terminated if an error occurs. If an error occurs, extext$ will contain the True BASIC error message and, in most cases, the specific error message returned from the database.

Summary of Functions and Subroutines

The following functions and subroutines are defined in TBSQLLIB.TRU and are explained in the following pages:

DEF sql_connect(MYHOST$,DBNAME$,MYUSERNAME$,MYPASSWORD$)

SUB sql_close(context)

SUB sql_matchfields(context,table$,pattern$,fieldlist$(,))

SUB sql_matchtables(context,pattern$,tablelist$())

SUB sql_gethandles(context,handles$) **(ODBC only)**

SUB sql_query(context,query$,rows,cols)

SUB sql_getresults(context,start row,rows,result$(rows,cols))

SUB sql_getallresults(context,result$(rows,cols))

The SQL subroutine is the only built-in subroutine. This routine gets called by the routines and functions in `TBSQLLIB.TRC`:

```
        SUB sql(option,context,in$,inlen(),out$,outlen())
```

Connecting To A Database

Connecting to a database occurs with a single subroutine call where you pass the host name, database name, username, and password. A context identifier (an integer) is returned. You will use this context identifier for future calls to the SQL library routines. You can make up to ten simultaneous connections to different databases, regardless of whether they are on the same or different servers.

DEF sql_connect(MYHOST$,DBNAME$,MYUSERNAME$,MYPASSWORD$)

MYHOST: the name of the server (e.g., truebasic.com, localhost)

DBNAME: the name of the database

MYUSERNAME: the username (e.g., nobody, guest)

MYPASSWORD: the password for the username

ODBC

`DBNAME` will be ignored. `MYHOST` will be the name of the data source set up in the ODBC control panel which includes a database name. `MYUSERNAME` and `MYPASSWORD` are the username and password (if any) set up for the data source in the ODBC control panel.

MySQL

The host, username, password combination is used to look up access permissions in the permission table. `MYHOST` will most likely be 'localhost' for Web access. Similarly, your CGI scripts will probably use "nobody" as the username. MySQL has three database tables that determine access: db, host, and user.

The MySQL User Database.

The values you pass to `DEF sql_connect` are checked against the user database which has the following architecture:

Field	Type	Null	Key	Default	Extra
User	char(60)		PRI		
password	char(16)		PRI		
Select_priv	char(16)				
Insert_priv	char(1)			N	
Update_priv	char(1)			N	
Delete_priv	char(1)			N	
Create_priv	char(1)			N	
Drop_priv	char(1			N	
Reload_priv	char(1)			N	
Shutdown_priv	char(1)			N	
Process_priv	char(1)			N	
File_priv	char(1)			N	

The User and password are the primary keys with all of the privileges, except SELECT, default to "N" or "no".

In most cases, you will give people select permission which is equivalent to read-only access. Insert permission is generally not destructive since it allows people to add data. Update, delete, and drop permissions should be restricted. You can control access to these functions by writing CGI scripts that perform specific rather than variable operations (e.g., only updates from scripts that require a password).

The most common error when working with the permission tables is to forget to reload the table after making a change. To reload the tables, type:

```
mysqladmin reload
```

Reloading the permission tables makes your changes active. Also note that the password is encryted so you can't just pass a string in your UPDATEs to the table. Use the password() function that is part of MySQL (e.g., password("mypassword")).

For more information on MySQL, and especially about the permission tables, see the MySQL documentation online at: `http://www.webware-inc.com/MySQL/toc.html`.

Error Messages related to database connections

SQL errors start at -14000. When an error occurs, the error message, calling sequence, and error number get stored in the reserved True BASIC variables extext$, exline$, and extype respectively.

Could not connect to the specified SQL host. (-14000)

No connection to SQL host established. (-14001)

Could not connect to the specified database. (-14002)

Null or badly formed DATABASE NAME. (-14006)

Error (MySQL): # (-14009)

Too many connections open. (-14011)

On some platforms, you may get the error "No such file.", as well. This means that the platform's ODBC DLLs or shared libraries, necessary to use the ODBC features of True BASIC, are not installed, and you will need to install them. These DLLs should have been installed with your operating system or database engine; if they are not installed, you will need to review your operating system or database engine documentation to find out how to install them.

Closing Connections

Don't forget to close the connection to your database engine at the end of your program.

SUB sql_close(context)

This routine closes the server connection. You pass the context identifier returned from the sql_connect function.

It is good practice to enclose your SQL (and other code) in an error handler. Call SUB sql_close after the error handler to ensure that the connection gets closed. See the example in the QuickStart section of this chapter.

Obtaining Information About Data Sources

In many cases, you will already know the names of the databases and tables along with the data types and default values for these fields. However, it is good practice to dynamically construct these by querying the database rather than hard-coding the settings and order of fields into your programs. Then you can modify your tables without having the changes ripple through all your code that accesses those tables.

For example, when you issue a query that returns all the fields in a table (e.g., SELECT * FROM myhugeartcollection), you will get the all the entries and all their fields in the order they appear in the table. If you have a PRIMARY KEY or other entry in the table definition, this field does not get returned. If you rearrange the table architecture, the fields returned by this SELECT statement will reflect the new order.

You can avoid this problem by specifying the fields you want returned. For example, `SELECT painter,masterpiece FROM myhugeartcollection` will return just the 'painter' and 'masterpiece' fields for each entry.

In some cases, you may need access to the names of the fields. For convenience, we provide the following routines for retrieving those field names.

SUB sql_matchfields(context,table$,pattern$,fieldlist$(,))

given a tablename, return the field names and data types matching pattern$.

SUB sql_matchtables(context,pattern$,tablelist$())

returns tablenames matching pattern$ for current database.

The pattern$ parameter consists of characters with the percent symbol "%" being a wildcard for multiple characters and the underscore symbol "_" serving as the wildcard for single characters.

Wildcards let you create patterns that don't look for exact matches. For example, a `pattern$="%Springfield,__"` would return matches for "Walnut Avenue, Springfield,IL", " Springfield,CA", etc. Using `pattern$="%"` would return all the entries. Using `pattern$="_"` would return all entries with a single character for the field or table.

To search for a wildcard character such as a table named “cmt_admin” you would use `"cmt\_admin"`. In general, life will be easier for you if you refrain from using underscores, percent symbols, hyphens, and quotation marks in table and field names!

Passing database handle to C routines

To extend True BASIC's support routines, you can get the handles for a database connection and pass them to a C routine, use the subroutine:

SUB sql_gethandles(context,handles$) (ODBC only)

See the section entitled *Using C routines with True BASIC* in Chapter 24.

These handles will be encoded in the string handle$ which you should pass to the C routine. You can retrieve them in the following way:

```
/* Assume "handles" is the True BASIC string already retrieved. */
SQLHANDLE    *tbsqlhandles;
SQLHENV       tbsqlhenv;    /* environment handle */
SQLHDBC       tbsqlhdbc;    /* connection handle */
SQLHSTMT      tbsqlhstmt;   /* statement handle */

tbsqlhandles = (SQLHANDLE *) handles->str->text;
tbsqlhenv = tbsqlhandles[0];
tbsqlhdbc = tbsqlhandles[1];
tbsqlhstmt = tbsqlhandles[2];
```

Executing Queries

There is only one routine, `SUB sql_query`, that executes SQL queries and returns the number of rows and columns affected. Two other routines, `SUB sql_getresults` and `SUB sql_getallresults`, let you retrieve a subset of the results or all the results, respectively.

You can check the number of rows affected by your query to determine which results routine to use.

SUB sql_query(context,query$,rows,cols)

returns the number of rows and columns affected. Note that some SQL statements like INSERT and UPDATE may not return the number of rows affected. Also, some ODBC databases may not know this value at the time for any operation. (See SUB sql_GetResults.)

The SQL queries can perform a wide range of operations from creating databases and tables, to selecting information from them, to joining results from two queries.

Debugging Queries

One useful strategy for debugging queries is to print out the query string along with the True BASIC error messages. For example:

```
LIBRARY "tbsqllib.trc"

DECLARE DEF sql_connect
DIM result$(0,0)
WHEN ERROR IN
    LET context = sql_connect("localhost","inventory","myname", &
      & "mypassword")
    LET QUERY$ = "SELECT product,quantity from WAREHOUSE"
    CALL sql_query(context,query$,rows,cols)
    ! get the results and do something with them
USE
    PRINT "ERROR: " & extext$
    PRINT exline$
    PRINT "Query: " & query$        ! print the query out too
END WHEN
CALL sql_close(context)
```

The True BASIC error messages, in most cases, will contain an error message from the database. However, it is sometimes difficult to interpret these messages without the query.

Error Messages related to queries

Null or badly formed SQL query. (-14003)
No results available for SQL query. (-14004)
Incomplete transfer of SQL query results (-14005).
Trouble creating DATABASE. (-14007)
Invalid argument for SUB SQL. (-14008)
Error (MySQL): # (-14009)

You should not get error -14008 if you use the routines in the library `tbsqllib.trc` since this error message indicates an invalid option being passed to the built-in True BASIC routine `SUB SQL` (see Internal Routines section below).

Quotation Marks In Queries

One common problem people have constructing queries involves quotation marks. Quotation marks need to be preceeded by a backslash for most SQL engines to handle them correctly.

The library `tbsqllib.trc` provides a routine for inserting backslashes before quotation marks.

```
SUB escape_quotes(entry$)
    LET p=posr(entry$,"""")
```

```
      do while p>0
        LET entry$(p:0)="\"
        LET p1=p
        LET p=posr(entry$,"""",p1)
      loop
  END SUB
```

You can modify this routine to handle other characters that cause problems with specific database engines.

Retrieving Results

SUB sql_getresults(context,startrow,rows,result$(rows,cols))

get results from last query. Specify the first row to return in `startrow` and the number of rows to return in "rows". If `startrow` is 0, then return all rows. If rows is 0, then return all rows after `startrow`. The actual number of rows returned will be placed in rows upon exit. This is *especially* handy for those instances when SQL_Query could not return the correct number of rows.

SUB sql_getallresults(context,result$(rows,cols))

returns all results from last query in array format. Calling this routine is equivalent to calling `sql_getresults(context,0,0,result$)`.

Tips for SQL Developers

Indexes as a way to speed up SELECT statements

SELECT statements can retrieve information selectively from a database using a WHERE clause to indicate the criteria. For example, to retrieve all entries with a product ID greater than twenty you could use the statement:

```
  SELECT product_name FROM catalog WHERE product_id>20
```

This query would return the product name from the catalog table where the `product_id` is greater than 20.

The database engine has to search through the catalog table to determine which entries have a product greater than 20. The comparison needs to be made for each row in the database. So the larger the table, the longer this query will take.

One way to greatly increase the speed of this type of query is to create an index for the product ID.

```
  ALTER TABLE catalog ADD INDEX (product_id)
```

When you create an index for a field in a database, most database engines will create a separate sorted list for that field. This separate, sorted list will be used for comparisons instead of the unsorted entries.

The UNIQUE attribute, as opposed to INDEX, can be used when you want the index to be a unique value.

In general for most large tables, you should add an index for any field that appears in a WHERE clause. Most database engines let you create indexes for text fields as well as numeric fields.

TRADE-OFF:
The disadvantage of adding indexes to a table is that it slows down INSERT queries. This slowdown is due to the fact that the sorted lists need to be updated as well. So if you are doing significantly more INSERTS than SELECTS, you may not want to add as many indexes. You can speed up INSERTS by LOCKING the table during INSERTs:

```
LOCK TABLES catalog WRITE
INSERT INTO catalog VALUES ("3D Graphics")
UNLOCK TABLES
```

On most database engines this reduces the number of times the index buffer gets flushed as well as the number of multi-connection tests that need to be performed. You should, however, check the documentation for your SQL engine for details if you are concerned with performance.

Ultimately the speed of inserts is affected by the time connecting, sending the query to server, parsing the query, inserting the record, inserting indexes, closing connection. The True BASIC libraries process and format the query, taking some additional time. Opening the table, which you should do once at the beginning of your program, takes some time too.

[!] **NOTE: Note that many databases require that INDEXES and PRIMARY KEYs be defined as NOT NULL. Also, you can use the EXPLAIN command to see which indexes are used in a specific SELECT statement.**

Adding a primary key

A primary key is typically an integer value that uniquely identifies a row.

You can have only one primary key. You can have this field auto increment so that a unique number identifies each record in your table.

```
CREATE TABLE catalog (id_product INT NOT NULL AUTO_INCREMENT,
PRIMARY KEY (id_product), product_id INT NOT NULL, product_name varchar(100))
```

This command creates the table catalog with id_product as the PRIMARY KEY.

Getting the value of an AUTO_INCREMENT column in ODBC

You have just inserted a set of values into a table and want to get the value the primary index that was just auto-incremented:

Getting the value of an AUTO_INCREMENT column can be done with a SELECT statement and a WHERE clause if you have a unique item for which you can search. The better method is to use LAST_INSERT_ID:

```
INSERT INTO catalog (auto,text) VALUES (null,'3D Graphics')
SELECT LAST_INSERT_ID()
```

[!] **NOTE: If you delete a row from the table, then the next INSERT will replace that row and assume its ID. Thus, assuming that your last INSERT will produce the highest value in the auto-incremented column is not reliable.**

True BASIC Internal SQL Routines

The built-in routines for handling the SQL interface should only be used if you are writing your own library of routines. True BASIC may in the future alter the internal routine or its parameters. By using the library tbsqllib.trc, you can ensure that the application code you develop will not need to be updated if changes are made. This strategy of providing library routines that access built-in functions is a standard procedure at True BASIC where we attempt to encapsulate functionality in modules and libraries.

Currently, there is a single built-in subroutine that supports the SQL interface:

```
SUB sql(option,context,in$,inlen(),out$,outlen())
```

The options for this routine are:

```
TBSQL_CONNECT     = 1
TBSQL_DATABASE    = 2
TBSQL_CLOSE       = 3
TBSQL_QUERY       = 4
TBSQL_GETRESULTS  = 5
TBSQL_CREATEDB    = 6     ! Not used for ODBC
TBSQL_DROPDB      = 7     ! Not used for ODBC
TBSQL_MATCHTABLES = 8
TBSQL_MATCHFIELDS = 9
TBSQL_GETHANDLES  = 10
```

The input and output strings are actually a concatenation of a series of strings. The arrays inlen() and outlen() contain the length of the substrings. In some cases, such as TBSQL_QUERY where the number of rows and columns affected gets returned, these numerical values are returned as the first and second elements in the array outlen().

```
SUB sql_query(context,query$,rows,cols)
  ! execute SQL query.
  ! returns rows,cols found.  (though in most cases you will
  ! know the number of cols and just be interested in the rows.)
```

```
    dim inlen(1),outlen(0)

    let inlen(1) = len(query$)
    call sql(TBSQL_QUERY,context,query$,inlen(),out$,outlen())
    if ubound(outlen)=2 then
      let result_rows = outlen(1)
      let result_cols = outlen(2)
    else
      let result_rows,result_cols = -1   ! error
    end if
    rows = result_rows
    cols = result_cols
  END SUB
```

In some cases, you will pass multiple strings to the SQL routine. For example, to connect to the server, you need to pass the host name, user name, and password:

```
let in$ = myhost$
let inlen(1) = len(myhost$)
let in$ = in$ & myusername$
let inlen(2) = len(myusername$)
let in$ = in$ & mypassword$
let inlen(3) = len(mypassword$)

call sql(TBSQL_CONNECT,context,in$,inlen(),out$,outlen())
```

You can examine the source code, `tbsqllib.tru` for more examples.

CHAPTER

27

True BASIC PostScript™ Support

PostScript Support

The Gold Edition of True BASIC allows you to journal True BASIC graphics to a PostScript file. This enables you to have True BASIC write PostScript commands for True BASIC graphics statements (such as BOX LINES) to a PostScript file at the same time as they are being executed on the screen. When you finish, you will have a PostScript file which can be printed on a PostScript printer in the printer's native resolution or which can be loaded into a number of page layout or drawing programs and manipulated or included in a complex document.

Using PostScript with True BASIC

To start this PostScript output, include in your program a call to the built-in subroutine BEGIN_POSTSCRIPT:

```
CALL Begin_PostScript("mypsfile",left,right,bottom,top)
```

The first parameter of this subroutine is the name of the file where the PostScript commands are to go. If the file does not exist, it will be created. If it does exist, it will be overwritten. The four numeric parameters define the region the PostScript output should occupy on the display device, whether it is a printer or a drawing program. The parameters are the left, right, bottom, and top edges of the region, and they are defined in **points**. A point is 1/72 of an inch.

For example, if you wish to map the image in the current *physical* window onto a full 8.5 by 11 inch page with one inch margins, use (..., 72, 540, 72, 720) for the parameters.

[!] **NOTE:** The PostScript journalling includes the full physical window even though you may be using a smaller logical window to produce your graphics.

BEGIN_POSTSCRIPT will begin the file with an Encapsulated PostScript (EPS) Specification Version 2.0 header so that other applications which can import EPS files will recognize and import it. While some EPS files have a preview section, not all versions of

True BASIC supply a preview section to files created with BEGIN_POSTSCRIPT. This will affect how the EPS file looks when imported into some applications.

When you have finished with the section of graphics that you wish to have journalled to PostScript file, include in your program a call to the built-in subroutine END_POSTSCRIPT:

```
CALL End_PostScript(n)
```

This will close the file and stop the journalling. Currently, the variable *n* means nothing. It is reserved for future use. If you call END_POSTSCRIPT without having called BEGIN_POSTSCRIPT, the call will return without effect.

If while journalling to the PostScript file you wish to append your own PostScript commands or comments to the current file, include a call to the built-in subroutine ADD_POSTSCRIPT:

```
CALL Add_PostScript(add_line$)
```

The variable *add_line$* represents the line you wish to place in the file. If *add_line$* = "" then a blank line will be placed in the file.

Note: Any added non-blank line must be a legitimate PostScript command or comment.

If you call ADD_POSTSCRIPT while no PostScript file is open, the call will return without effect. In some environments, the PostScript file is simply a text (ASCII) file, which means you can also edit the file easily after it is created.

You may journal to only one PostScript file at a time. Calling BEGIN_POSTSCRIPT while a PostScript file is already open will result in an error being caused.

Printing your PostScript File to a Printer

Perhaps the most common thing you will want to do with your PostScript file is obtain a nice printout of it from a PostScript printer. To do this you will need a PostScript printing utility that takes the raw PostScript file you have created and sends it to your printer.

M On the MacOS you can use the Apple Printer Utility which allows you to send a PostScript file directly to a printer.

W On Windows you can use a freeware utility called PrintFile. Other utilities that also can do the job are Drop*PS from Bare Bones Software, Ghostscript, and GSView. Each of these applications are available on a variety of operating systems and comes with its own instructions on how best to use it.

Example Program

```
!
! A simple program to demonstrate PostScript journalling. The
! program will create a star on-screen and a PostScript file
! that will create a star on a PostScript device.
!

SET WINDOW -0.5,0.5,-0.5,0.5  ! put origin in center of screen

!
! Open a PostScript file; point values will leave a one-inch
! margin when output on standard letter size paper later.
!
CALL Begin_PostScript("mypsfile",72,540,72,720)

!
! Now, add a comment to remind us what the picture is of.
! The comment will appear in the PostScript file, but not
! in a PostScript image. (A percent symbol [%] is the
! comment character in PostScript.)
!
CALL Add_PostScript("")! A blank line for readability
CALL Add_PostScript("% A demonstration PostScript file: a star.")
CALL Add_PostScript("")! A blank line for readability

!
! Now, draw our picture...
!
FOR rv = 0 to 2*pi step pi/6
   DRAW simple_line with rotate(rv)
NEXT rv

CALL End_PostScript(1) ! Finished PostScript journalling

PICTURE simple_line
   PLOT 0,0;0,0.5
END PICTURE

END
```

CHAIN Statement

> **CHAIN** *strex*
> **CHAIN** *strex*, **RETURN**
> **CHAIN** *strex arglist$*
> **CHAIN** *strex arglist$,* **RETURN**

The CHAIN statement stops the current program and starts up the program in the file named in *strex*. If the target program is not accessible, an exception does NOT occur. Instead a black DOS type SYSTEM COMMAND window opens with a white default OUTPUT window and True BASIC crashes. To avoid this happening you are advised to first check that *strex* exists BEFORE attempting to CHAIN to it.

IF *arglist$* is present then there MUST be a matching PROGRAM statement in the target program with the same number of variables as *arglist$.* IF the PROGRAM statement is missing or the number of variables does not match *arglist$* then an exception does NOT occur. Instead the *arglist$* is ignored if the PROGRAM statement is missing, or if the number of variables is incorrect then only those that are correct will be filled. For example if there are 3 parameters in *arglist$* and only 2 in the PROGRAM parameters then the first two from *arglist$* will be transferred to the PROGRAM parameters.

The parameters in *arglist$* consist of variables separated by spaces, or one string variable that includes spaces in the text, e.g.

```
LET arglist$ = “Mary had a little lamb” (5 parameters)
LET arglist$ = arg1$ & “ “ & arg2$ & “ “ & arg3$ (3 parameters)
```

If the individual parameter is a filename then it is likely the filename may contain spaces. CHAIN considers such spaces as extra parameters, e.g.

```
LET arglist$=”C:\Program files\myfolder\myfile”
```

This will count as 2 parameters “C:\Program” and “files\myfolder\myfile”
To avoid such problems you may use the following two sub-routines.
The parameter passing mechanism in CHAIN is by value, the same as defined functions.

This routine checks to see if the target program exists, and cleans up the filename *argfile$* that may be present in *arglist$.*

```
SUB check_CHAIN(targetfile$,argfile$)
    WHEN EXCEPTION IN
       OPEN #99:NAME targetfile$,create old,access outin,org byte
       ASK #99: FILESIZE bytes
       LET error$=””
       CLOSE #99
    USE
       CLOSE #99
       CAUSE EXCEPTION 10005
```

```
            EXIT SUB
        END WHEN

        LET length=len(argfile$)
        FOR n=1 to length
            IF argfile$[n:n]=chr$(32) then
                LET argfile$[n:n]=chr$(31)
            END IF
        NEXT n
END SUB
```

This routine should be used in the target file to restore the original filename.

```
SUB restore_name(argfile$)
        LET length=len(argfile$)
        FOR n=1 to length
            IF argfile$[n:n]=chr$(31) then
                LET argfile$[n:n]=chr$(32)
            END IF
        NEXT n
END SUB
```

If the RETURN clause is missing, all memory storage associated with the parent program is released, allowing the target program to occupy and use more memory.
If the RETURN clause is absent, any runtime error that occurs in the target program but is not handled within the target program by WHEN EXCETION IN will be handled and reported by the system.

If the RETURN clause is present, the parent program is retained in memory, and when the target program finishes, control is returned to the parent program at the first statement following the CHAIN statement.
If the RETURN clause is present, any runtime error that occurs in the target program but is not handled within the target program by WHEN EXCEPTION IN will be sent back to the parent program. If the target program is in source form and contains syntax errors, the exception number will be 10005, but the message will describe the actual error.

If the string-expression after the word CHAIN begins with a “!”, the rest of it will be taken as a command to the operating system. If the string-expression begins with a “!&”, and a RETURN clause is present, the parent program will continue immediately; and the command will be executed in background.

NOTE: On some Windows systems, "!" and "!&" behave effectively the same. There will still be a small difference, as the latter will return immediately, while the former will wait until the CHAINed application is launched before returning.

When the target program starts, all modules associated with it are initialised, even if the target program has already been chained to previously. However, loaded modules are not re-initialized.

On personal computer systems, chaining from source or compiled programs is permitted to either source or compiled target programs. Chaining to and from executable (bound) programs is also permitted except there are restrictions with UNIX based operating systems.

A target program can itself chain to another program. This process can continue indefinitely and is limited only the available memory.

APPENDIX

A

Character Set

This table lists only the ASCII character set (0-127). Each character in this table appears with its decimal and hexadecimal equivalent. These ASCII codes work on all True BASIC computers.

The names shown in this table can be used as arguments for the ORD function. Note that the names beginning with decimal number 095 (UND) are designed for use with computers that do not distinguish between upper- and lowercase letters.

Different operating platforms usually offer additional characters, but they are non-standard across platforms and no attempt is made here to summarize them. Consult your platform's user guide for details. Note: the CHR$ function allows arguments in the range 0..255, and so can generate the non-ASCII characters in the range 128..255, provided you know the decimal number of such characters.

Decimal	Name	Hex
000	nul	00
001	soh	01
002	stx	02
003	etx	03
004	eot	04
005	enq	05
006	ack	06
007	bel	07
008	bs	08
009	ht	09
010	lf	0A
011	vt	0B
012	ff	0C
013	cr	0D
014	so	0E
015	si	0F
016	dle	10
017	dc1	11
018	dc2	12
019	dc3	13
020	dc4	14
021	nak	15
022	syn	16
023	etb	17
024	can	18
025	em	19
026	sub	1A
027	escape	1B
028	fs	1C
029	gs	1D
030	rs	1E
031	us	1F
032	space	20
033	!	21
034	"	22
035	#	23
036	$	24
037	%	25
038	&	26
039	`	27
040	(	28
041	)	29
042	*	2A
043	+	2B
044	,	2C
045	—	2D
046	.	2E
047	/	2F
048	0	30
049	1	31
050	2	32
051	3	33
052	4	34
053	5	35
054	6	36
055	7	37
056	8	38
057	9	39
058	:	3A
059	;	3B
060	<	3C
061	=	3D
062	>	3E
063	?	3F
064	@	40
065	A	41

Decimal	Name	Hex
066	B	42
067	C	43
068	D	44
069	E	45
070	F	46
071	G	47
072	H	48
073	I	49
074	J	4A
075	K	4B
076	L	4C
077	M	4D
078	N	4E
079	O	4F
080	P	50
081	Q	51
082	R	52
083	S	53
084	T	54
085	U	55
086	V	56
087	W	57
088	X	58
089	Y	59
090	Z	5A
091	[	5B
092	\	5C
093	]	5D
094	^	5E
095	_ UND	5F
096	` GRA	60
097	a LCA	61
098	b LCB	62
099	c LCC	63
100	d LCD	64
101	e LCE	65
102	f LCF	66
103	g LCG	67
104	h LCH	68
105	i LCI	69
106	j LCJ	6A
107	k LCK	6B
108	l LCL	6C
109	m LCM	6D
110	n LCN	6E
111	o LCO	6F
112	p LCP	70
113	q LCQ	71
114	r LCR	72
115	s LCS	73
116	t LCT	74
117	u LCU	75
118	v LCV	76
119	w LCW	77
120	x LCX	78
121	y LCY	79
122	z LCZ	7A
123	{ LBR	7B
124	| VLN	7C
125	} RBR	7D
126	~ TIL	7E
127	DEL	7F

APPENDIX

B

Error Numbers and Messages

The following run-time error (exception) numbers and messages are provided by the EXTYPE and **EXTEXT$** functions (see Chapters 16 and 18). When there is no **WHEN USE** structure to intercept the error, the True BASIC system prints the message and halts the program.

Errors in the True BASIC source code are not included here. Further explanations of both source program errors and run-time errors can be found in Appendix C.

Extype	Extext$
1001	Overflow in numeric constant
1002	Overflow.
1003	Overflow in numeric function.
1004	Overflow in **VAL**.
1005	Overflow in **MAT** operation.
1006	Overflow in **READ**.
1007	Overflow in **INPUT** (nonfatal).
1008	Overflow in file **INPUT**.
1009	Overflow in **DET** or **DOT**.
1051	String too long.
1052	String too long in **MAT**.
1053	String too long in **READ**.
1054	String too long in **INPUT** (nonfatal).
1105	String too long in file **INPUT**.
1106	String too long in assignment.
2001	Subscript out of bounds.
-3000	Argument not in range
3001	Division by zero.
3002	Negative number to non-integral power.
3003	Zero to negative power.
3004	**LOG** of number <= 0.
3005	**SQR** of negative number.
3006	**MOD** and **REMAINDER** can't have 0 as 2nd argument.
3007	**ASIN** or **ACOS** argument must be between 1 and -1.
3008	Can't use **ANGLE**(0,0).
3009	Can't invert singular matrix.
-3050	Argument for **SIN**, **COS**, or **TAN** too large.
-3051	Argument too large or small for accurate result.

Extype	Extext$
4001	**VAL** string isn't a proper number.
4002	**CHR$** argument must be between 0 and 255.
4003	Improper **ORD** string.
4004	**SIZE** index out of range.
4005	**TAB** column less than 1 (nonfatal).
4006	**MARGIN** less than zonewidth.
4007	**ZONEWIDTH** out of range.
4008	**LBOUND** index out of range.
4009	**UBOUND** index out of range.
4010	**REPEAT$** count < 0.
-4020	Improper **NUM** string.
4102	Improper **TEXT JUSTIFY** value (nonfatal).
4301	Mismatched parameters for **CHAIN/PROGRAM**.
4302	Mismatched dimensions for **CHAIN/PROGRAM**.
-4303	Invalid numeric argument "#".
-4501	Error in **PLAY** string.
5000	Out of memory.
5001	Array too large.
6001	Mismatched array sizes.
6002	**DET** needs a square matrix.
6003	**INV** needs a square matrix.
6004	**IDN** must make a square matrix.
6005	Illegal array bounds.
6101	Mismatched string array sizes.
7001	Channel number must be 1 to 1000.
7002	Can't use #0 here (nonfatal).
7003	Channel is already open.
7004	Channel isn't open.

Extype	Extext$
7051	Record **LENGTH** <= 0.
7100	Unknown value for **OPEN** option.
7102	Too many channels open.
7103	File's record size doesn't match **OPEN RECSIZE**.
7104	Wrong type of file.
-7105	Wrong type of file for **LOCK** (**UNLOCK**.) (These accesses not available in V5)
-7106	Wrong type of file for **NETWORK** (**NETOUT**, **NETIN**) access. (Not available)
-7107	Can't **UNLOCK** range. (Not available)
-7108	Range or part of range is already **LOCK**ed. (Not available)
7202	Must be **RECORD** or **BYTE** for **SET RECORD**.
7204	Can't use **SAME** here.
-7250	Can't **SET RECSIZE** on non-empty **RECORD** file.
-7251	Must be **BYTE** file or empty for **SET RECSIZE**.
-7252	File pointer out of bounds.
7301	Can't **ERASE** file not opened as **OUTIN**.
7302	Can't output to **INPUT** file.
7303	Can't input from **OUTPUT** file.
7308	Can't **PRINT** or **WRITE** to middle of this file.
7312	Can't set **ZONEWIDTH** or **MARGIN** for this file.
7313	Can't set **ZONEWIDTH** or **MARGIN** for **INPUT** file.
7317	Can't **PRINT** to **INTERNAL** file.
7318	Can't **INPUT** from **INTERNAL** file.
7321	Can't **SKIP REST** on **STREAM** file.
-7351	Must be **BYTE** file for **READ BYTES**.
7401	Channel is not open for **TRACE**.
7402	Wrong file type for **TRACE**.
8001	Reading past end of data.
8002	Too few input items (nonfatal).
8003	Too many input items (nonfatal).
8011	Reading past end of file.
8012	Too few data in record.
8013	Too many data in record.
8101	Data item isn't a number.
8102	Badly formed input line (nonfatal).
8103	String given instead of number (nonfatal).
-8104	Data item isn't a string.
8105	Badly formed input line from file.
8201	Badly formed **USING** string.
8202	No **USING** item for output.
8203	**USING** value too large for field (nonfatal).
8208	**USING** exponent too large for field (nonfatal).
8301	Output item bigger than **RECSIZE**.
8302	Input item bigger than **RECSIZE**.
-8304	Must **SET RECSIZE** before **WRITE**.
8401	Input timeout.
8402	Timeout value < 0.
-8450	Nested **INPUT** statements with **TIMEOUT** clauses.
-8501	Must be a **TEXT** file.
-8502	Must be **RECORD** or **BYTE** file.
-8503	Can't use **READ** or **WRITE** for **TEXT** file.
9000	File i/o error.
9001	File is read or write protected.
9002	Trouble using disk or printer.
9003	No such file.
9004	File already exists.
9005	Diskette removed, or wrong diskette.
9006	Disk full.
9007	Too many channels open.
9008	No such directory.
9100	Can't open temporary file.
9101	Can't open **PRINTER**.
-9200	File i/o error (PostScript).
-9201	File is read or write protected (PostScript).
-9202	Trouble using disk or printer (PostScript).
-9203	No such file (PostScript).
-9205	Diskette removed, or wrong diskette (PostScript)
-9206	Disk full (PostScript).
-9207	Too many channels open (PostScript).
-9208	No such directory (PostScript).
-9209	Illegal PostScript area definition.
-9210	PostScript journalling already enabled.
9601	Cursor set out of bounds.
10001	**ON** index out of range, no **ELSE** given.
10002	**RETURN** without **GOSUB**.
10004	No **CASE** selected, but no **CASE ELSE**.
10005	Program not available for **CHAIN**. *
-10006	Exception in **CHAIN**ed program.
10007	Break statement encountered.
11000	Can't do graphics on this computer.
-11001	Window minimum = maximum.

Extype	Extext$
-11002	Screen minimum >= maximum.
-11003	Screen bounds must be 0 to 1.
11004	Can't **SET WINDOW** in picture.
-11005	Channel isn't a window.
-11008	No such color.
-11009	User-defined window coordinates too large for **GET MOUSE** (**POINT**)
11140	No **GET MOUSE** on this computer.
-11210	Invalid option for **SUB** Clipboard.
-11211	Invalid type for **SUB** Clipboard.
-11212	Error opening clipboard for reading.
-11213	Error closing clipboard.
-11214	Error opening clipboard for writing.
-11215	Error putting text onto clipboard.
-11220	Unknown or invalid object **ID**.
-11221	Cannot reference a freed object **ID**.
-11222	Unknown or invalid **SYSINFO** request.
-11223	Attribute not used for specified object.
-11224	Unknown or invalid group method.
-11225	Unknown or invalid attribute in **SET**/**GET**.
-11226	Unknown or invalid font name.
-11227	Unknown in invalid font style.
-11228	Font size must be greater than zero.
-11229	TextEdit method passed to non-TextEdit object.
-11232	Error adding paragraph.
-11233	Paragraph number is too large.
-11234	Error deleting paragraph.
-11235	Error appending paragraph.
-11236	Object **ID** out of range.
-11237	Unknown window method.
-11238	Unknown object method.
-11239	Unable to **SHOW** window.
-11240	Unknown or invalid object type specification in **CREATE**.
-11241	Too many **EXIT CHARS** for Edit Field.
-11242	Can't set **ACTIVE** until object is visible.
-11243	Can't set **NUM LINES**.
-11244	Can't set **NUM PARS**.
-11245	Can't set **NUM CHARS**.
-11246	Can't set **LINES IN PAR**.
-11248	Can't set **MAX WIDTH**.
-11249	Can't set **FONT METRICS**.
-11250	Too many trap chars for TextEdit.
-11251	Color must be >= 0.
-11252	Paragraph out of range for **GET LINE**.
-11253	Line out of range for **GET LINE**.
-11254	Unknown or invalid menu item type specification.
-11255	Can't check a menu separator.
-11256	Menu separators are not checkable.
-11257	Unknown or invalid control object type.
-11258	Unknown or invalid graphic object type.
-11259	Unknown or invalid window object type.
-11260	Unknown or invalid group object type.
-11261	Can't check a menubar item.
-11262	Can't make menubar item a separator.
-11263	Menu parent incorrect for menu type.
-11264	Can't **SELECT** an un**SHOW**n window.
-11265	Unknown or invalid brush pattern.
-11266	Unknown or invalid pen pattern.
-11267	Unknown or invalid directory.
-11268	Can't get current directory.
-11269	Unknown option for **SUB** System.
-11270	Can't get **STAT** info for file in directory.
-11271	**RECTANGLE** minimum = maximum.
-11272	No Help File opened.
-11273	Not enough values for attribute list in **SET**/**GET**.
-11300	Dialog box has no buttons specified.
-11301	Unknown or invalid dialog box specification.

*EXTYPE 10005 will occur either if the target program is not available, or if it is a BASIC source program that contains syntax errors. EXTYPE -10006 will occur only if **CHAIN** with **RETURN** is used, and is a catch-all **EXTYPE** in the original program for mismatched parameters (4301 and 4302) and all runtime exceptions in the target program that are not handled there.

[!] **Note:** Exceptions 4301 and 4302 cannot be intercepted with an exception handler in the target program, nor can they ever occur as the value of **EXTYPE**. They can appear only in the system error message on the screen.

Nonfatal exceptions do not halt the program, but allow it to continue. The nonfatal **INPUT** exceptions (1007, 1054, 8002, 8003, 8102, and 8103) request the user to reenter the entire input line. Exception 4005 causes the **TAB** column to be 1. Exception 4102 ignores the improper value, and retains the previous value. Exception 7002 is ignored. Exceptions 8203 and 8204 print the partially completed line, print the offending number on the next line, and continue the **USING** string on the third line. All nonfatal exceptions can be trapped with **WHEN** structures, as can fatal exceptions.

Nonfatal exceptions that occur *lexically* inside an exception handler are treated as fatal errors. That is, the line causing the exception must lie physically between the **WHEN EXCEPTION IN** and the **USE** lines for the exception to be treated as fatal.

APPENDIX

C

Explanation of Error Messages

This appendix contains a list of all True BASIC error messages, in alphabetic order.

There are two general classes of True BASIC errors: compile-time errors and runtime errors.

Compile-time errors are caught when True BASIC compiles your source program, which it does before each run. Runtime errors occur after your program has started running. Runtime error messages can be intercepted by an error handler in your program . In this appendix, each runtime error message is followed by its number (**EXTYPE** value). The runtime errors are summarized in Appendix B, in order of the **EXTYPE** value for each error.

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.

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 not in range (-3000)

The argument you have supplied to a function is not within the legal or defined range for that function. Check the definition of the function to make sure you are aware of any limitations. If your argument is the result of calculation in your program, make sure you have not made a mistake in setting up the calculation.

Argument too large or small for accurate result. (-3051)

The argument in a call to the **SIN**, **COS**, or **TAN** function is too large to permit accurate reduction.

Argument types don't match.

You're calling a routine with some arguments, but earlier in your program you defined or called the same routine with different arguments. Either you're giving a different number of arguments in the calls, or their types are different – that is, you're passing strings instead of numbers, or vice versa. Check this call against preceding calls, and against the routine's definition.

Bad FIND item; try using quotes.

When you're trying to find a string which contains a comma or quotation marks, you must enclose the entire string within quote marks. (These rules are the same as the rules for strings in **INPUT** replies or **DATA** statements.)

Badly formed input line (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. See the **INPUT** statement in Chapter 4 for a description of input replies.

Badly formed USING string. (8201)

The format string in your **USING$** function or **PRINT USING** statement is incorrect. Some format item doesn't follow True BASIC's rules. See Chapter 3 and Appendix D for a description of format strings.

Break statement encountered. (10007)

You've encountered a **BREAK** statement in a program-unit in which debugging is active.

Can't chain from bound program.

You cannot chain from a bound program (one that is directly executable) on some systems.

Can't continue.

You've just given a CONTINUE command, to resume running a suspended program. However, True BASIC cannot continue the program. There are several possible reasons. You cannot continue a program that you haven't yet started running, or one which you've just changed. You cannot continue a program which stopped because an error occurred. And you cannot continue a suspended program after using a DO command. If you are trying to debug a program which stopped because of an error, try using the BREAK command to insert breakpoints before the erroneous line, and then run the program again.

Can't copy region into itself.

The True BASIC editor does not let you copy a region into itself. For instance, you may not make a copy of some subroutine within that subroutine. If you really want to, you can put a copy of the region somewhere else, and then move this copy into the original region.

Can't do graphics on this computer. (-11000)

Your computer cannot draw graphics. Therefore you may not use the **SET WINDOW**, **PLOT**, **DRAW**, **BOX**, **FLOOD**, **GET POINT**, or **GET MOUSE** statements.

Can't edit compiled program.

Your program is compiled, and so cannot be changed. Once you've given a **COMPILE** command, you are only allowed to **RUN**, **SAVE**, or **REPLACE** the program. If a change needs to be made, call up the uncompiled version and change it.

Can't ERASE file not opened as OUTIN. (7301)

You may not use the **ERASE** statement on a file, unless the file has been opened with **ACCESS OUTIN**. See the **OPEN** statement in Chapters 12 and 18 for a description of the **OPEN** statement and file accesses.

Can't get STAT info for file in directory. (11270)

The user tried to usea a directory name in the template for Exec_Reader which is not allowed.

Can't INPUT from INTERNAL file. (7318)

You are trying to use the **INPUT** statement with an **INTERNAL** file; use **READ** instead.

Can't input from OUTPUT file. (7303)

You may not read input from a file which was opened with **ACCESS OUTPUT**. If you must read from this file, change the **OPEN** statement to use **ACCESS OUTIN**.

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 temporary file. (9100)

You are using the file statement ERASE REST. True BASIC requires a temporary file to carry out this instruction, and had trouble opening one.

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 PRINT to INTERNAL file. (7317)

You are trying to use the **PRINT** statement with an **INTERNAL** file; use **WRITE** instead.

Can’t PRINT or WRITE to middle of this file. (7308)

You may not overwrite data in a text file. Use the **RESET** statement (see Chapter 12) to move the file pointer to the end of the file before printing to it, or **ERASE** the file.

Can’t set ZONEWIDTH or MARGIN for INPUT file. (7313)

You are trying to set the zonewidth or margin on a file opened for input only. Change the file **OPEN** statement to include **ACCESS OUTPUT** or **ACCESS OUTIN**.

Can’t set ZONEWIDTH or MARGIN for this file. (7312)

You are trying to set the zonewidth or margin on a file whose type is not **TEXT** or **DISPLAY SEQUENTIAL**.

Can’t SET RECSIZE on non-empty RECORD file. (-7250)

Once a record file has been created, it has a fixed record size. You cannot change a file’s record size without unsaving the file and recreating it, or erasing the file. See the **UNSAVE** and **ERASE** statements.

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 SKIP REST on STREAM file. (7321)

You are trying to **SKIP REST** on a **STREAM** file. **SKIP REST** applies only to skipping the rest of a record in a **DISPLAY** or **INTERNAL SEQUENTIAL** file. **STREAM** files don’t have records.

Can’t UNLOCK range. (-7107)

LOCK and UNLOCK are not available in the initial release of Version 5.

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 READ or WRITE for TEXT file. (-8503)

You are trying to use **READ** or **WRITE** on a file whose type is **TEXT**; use **INPUT** or **PRINT** instead. Or, you can open the file with **ORG SEQUENTIAL** and **RECTYPE DISPLAY** (**TEXT** files and **DISPLAY SEQUENTIAL** files are indistinguishable in content.)

Can’t use SAME here. (7204)

You are using the record-setter **SAME** and either the most recent operation on that channel caused an exception, or no record operation has taken place since the channel was opened.

Can’t use this statement here.

You’ve used part of a True BASIC structure, but in the wrong place. For instance, you might have placed a **CASE** part outside of any **SELECT CASE** statement, or **ELSE IF** statement outside of any **IF-THEN** statement. True BASIC also prints this message if you add an extraneous statement between the **SELECT CASE** line and its first **CASE** part. Refer to the proper chapters of this manual to see how the structured statements are formed.

Can’t use #0 here (nonfatal). (7002)

You may not use channel #0 in **OPEN** or **CLOSE** statements, since #0 is always open. This exception is ignored unless trapped.

Channel is already open. (7003)

You are trying to open a channel which is already open. Check and make sure that you're not already using this channel number somewhere else in your program. Also remember to **CLOSE** a channel when you're done with it.

Channel is not open for TRACE. (7401)

You are trying to use **TRACE** to a file, but have not opened the file. Check to see that you have opened the file in the same program-unit.

Channel isn't a window. (-11005)

You are trying to switch to another window with the **WINDOW** statement, but the channel you've given is connected to a file (not a window). Check and make sure that you're giving the right channel number in the **WINDOW** statement.

Channel isn't open. (7004)

You're trying to use a channel to access a file or window, but that channel isn't open. Each channel must be opened by an **OPEN** statement before it can be used. Check to see that you've given the right channel number, and that you did open the channel. Also make sure you didn't accidentally close the channel with a **CLOSE** statement. Finally, make sure that if the channel was opened in a different program-unit, it was passed as an argument.

Channel number must be 1 to 1000. (7001)

Channel numbers must lie in the range 1 to 1000. Change your program so it doesn't use numbers outside this range.

CHR$ argument must be between 0 and 255. (4002)

You've tried to convert a number that is out-of-range to a character.

(Compiled program.)

This reminder appears in the editing window when you call up a compiled program, and after you give the **COMPILE** command. Since compiled programs have been digested into an internal format, you cannot see the program's text. Nor may you try to change the program in any way; instead, you must call up the uncompiled version and modify it.

Constant too large: constant in routine.

The numeric constant displayed is too large for your computer to handle. Use the built-in function **MAXNUM** to find the largest possible number on your computer, and change your program to use a smaller number.

Cursor set out of bounds. (9601)

Your **SET CURSOR** statement gives a row and column position that lie outside the current window's bounds. Remember that each window has its own cursor addresses – the top left position in each window is row 1, column 1. Use **ASK MAX CURSOR** to find out how many rows and columns are available.

Data item isn't a number. (8101)

You are trying to **READ** a number from a record file, but the current record in the file contains a string. You must read strings into string variables. Make sure that you are at the right spot in the file.

Data item isn't a string. (-8104)

You are trying to **READ** a string from a record file, but the current record in the file contains a number. You must read numbers into numeric variables. Make sure that you are at the right spot in the file.

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. On some operating systems, this error may be given if the directory containing the file has become full. Try removing other files from the disk. Your operating system may also lose track of storage on the disk, so consult your operating system guide to see if there is some special utility program you can run to recover lost storage.

Disk full (PostScript). (-9206)

This is the same as Disk full (9006), but specifically for a PostScript file opened with **COPY_POSTSCRIPT**. It could occur while journalling graphics to a PostScript file.

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!

Diskette removed, or wrong diskette (PostScript). (-9205)

This is the same as Diskette removed, or wrong diskette (9005), but specifically for a PostScript file opened with **COPY_POSTSCRIPT**. It could occur while journalling graphics to a PostScript file.

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 call up another file, start a new current file, or end your True BASIC session without saving or replacing your current file. Enter yes if you do want to save the file (replacing the current saved copy), no if you want to discard your changes, or cancel if you want to do something else (for example, save the file with a different name). If you're typing the response, you can abbreviate any of these replies to a single letter.

Doesn't belong here.

The cursor points to some word in your program that 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 Chapter 18 of this manual. 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. See the description of **PLAY** in Chapters 15 and 18 for a description of how to write melodies in **PLAY** strings.

Exception in CHAINed program. (-10006)

You have chained to a program with the **RETURN** option, and an exception has occurred in the target program and was not handled there. Also included are the parameter mismatch exception (4301 and 4302).

Expected "thing".

The cursor points to a spot where True BASIC expected some word or punctuation, but found something else. This message may jog your memory enough so that you can repair the statement. Otherwise, look up the statement in this manual, and then fix your program.

Expected relational operator.

The cursor points to a spot where you must put a relational operator, such as = or <. Finish writing out the comparison which must be there. (Note that True BASIC does not allow testing statements like *IF A THEN ...*, as some other BASICs do. Change such statements to *IF A<>0 THEN ....*)

File already exists. (9004)

You are trying to create a new file, but it already exists. Check to make sure that you've given the right file name. If you want to overwrite an existing file, change the **CREATE NEW** in the **OPEN** statement to be **CREATE NEWOLD**.

File already exists. Do you want to overwrite it?

You have tried to **SAVE** a file which already exists. Answering yes to this question replaces the file on disk. Any other response abandons the command. If you're typing the reply, you can abbreviate your response to the letter.

File i/o error. (9000)

You have encountered a file input or output error that is not covered by the other file error messages.

File i/o error (PostScript). (-9200)

This is the same as File i/o error (9000), but specifically for a PostScript file opened with **COPY_POSTSCRIPT**. It could occur while journalling graphics to a PostScript file.

File is read or write protected. (9001)

You are trying to read or write a file, but your operating system has write- or read-protected this file. True BASIC does not know how to handle read and write protection, so exit from True BASIC and use your operating system to remove the protection. This can also happen if your *disk* is write-protected; if so, you can remove the write protection from the disk and try again.

File is read or write protected (PostScript). (-9201)

This is the same as File is read or write protected (9001), but specifically for a PostScript file opened with **COPY_POSTSCRIPT**. It could occur while journalling graphics to a PostScript file.

File pointer out of bounds. (-7252)

You are trying to use the **RESET** or **SET POINTER** or **SET RECORD** statement to change a file's pointer. However, the position you've given is either less than 1, or past the end of the file. Try using the **ASK FILESIZE** statement to find how long the file is.

File's record size doesn't match OPEN RECSIZE. (7103)

Each record file has its **RECSIZE** built into it. The **RECSIZE** of the file you're trying to open doesn't match the **RECSIZE** given in your **OPEN** statement. Are you sure you're opening the right file? You may delete the **RECSIZE** part from your **OPEN** statement to simply use the file's record size. If you're sure you want the **RECSIZE** given in the **OPEN** statement, try performing a little test to discover the saved file's **RECSIZE**. Open the file (without giving a **RECSIZE**), and then use the **ASK RECSIZE** statement to determine the file's record size.

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. See the **DATA** statement in Chapters 7 and 18 for descriptions of how to use data items that contain quote marks or commas.

Illegal exit.

You've written an **EXIT DO** statement outside of any **DO-LOOP** structure. Or you have written an unknown kind of **EXIT** statement, such as "EXITFRED." Check to make sure that the **EXIT** statement lies properly within its structure, and that the word following **EXIT** is spelled correctly.

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 file.

You've written a **SET** or **ASK** statement that requires a channel number, but doesn't have one. Or you've added a channel number to a **SET** or **ASK** statement that doesn't allow one. Add or delete the channel number, as appropriate.

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 manual, 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 (see Chapter 17). 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** and **OPTION BASE** statements, and as line numbers.

Illegal option.

The only options supported by True BASIC are **OPTION ANGLE**, **OPTION BASE**, **OPTION NOLET**, and **OPTION TYPO**. Make sure you've spelled **ANGLE**, **BASE**, **DEGREES**, **RADIANS**, **NOLET**, or **TYPO** properly.

Illegal parameter.

You've written a **SUB** or **DEF** or **PICTURE** line, defining a routine. 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 PostScript area definition. (-9209)

The arguments supplied to **COPY_POSTSCRIPT** are incorrect. You have passed a right value that is less than or equal to the left value, or a top value less than or equal to the bottom value.

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. If you want to omit the keyword "LET", see the description of the **OPTION NOLET** statement in Chapters 2 and 18.

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.

Improper TEXT JUSTIFY value (nonfatal). (4102)

You've used an invalid word in **SET TEXT JUSTIFY**. See the **SET TEXT JUSTIFY** statement in Chapter 18 for the valid words. Invalid words will be ignored and the program will continue with the previous **TEXT JUSTIFY** values.

Input item bigger than RECSIZE. (8302)

This message is only given for damaged record files. It indicates that some record in the file is bigger than the file's record size. This message can only arise if the diskette containing the file has somehow decayed, or if the file was tampered with as a byte file, or from some application other than True BASIC.

Input timeout. (8401)

You have executed an **INPUT** or similar statement with a **TIMEOUT** clause, and the user has failed to respond within the allotted time.

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.

Invalid numeric argument (-4303)

Using the **CHAIN** statement, you have passed a non-numeric argument to a program which expects a numeric argument in that position. Check the **PROGRAM** statement in the program to which you are CHAINing to find out what arguments it expects.

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.

MARGIN less than zonewidth. (4006)

Your **SET MARGIN** statement is trying to set a margin less than the zonewidth. This is not allowed. Check your margin, or change the zonewidth before you change the margin.

Mismatched array sizes. (6001)

You're using a **MAT** statement that requires arrays of the same size, but the arrays are different sizes. For example, matrix addition requires the two arrays added together to have the same sizes. Matrix multiplication has slightly more complicated rules.

Mismatched string array sizes. (6101)

You're using a **MAT** statement with concatenation of string arrays, and the arrays are not the same size.

Mismatched dimensions for CHAIN/PROGRAM. (4302)

You are attempting to pass any array argument in the **WITH** clause of a **CHAIN** statement, but the number of its dimensions does not agree with that of the array parameter in the **PROGRAM** statement. (This exception cannot be handled by a **WHEN USE** structure.)

Mismatched parameters for CHAIN/PROGRAM. (4301)

The number and type of the arguments in a **WITH** clause in a **CHAIN** statement do not match the parameters in the corresponding **PROGRAM** statement. This exception will also occur if the **CHAIN** statement has a **WITH** clause while the **PROGRAM** statement is either missing or lacks parameters, and vice versa. (This exception cannot be handled by a **WHEN USE** structure.)

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**. See Chapter 10 for a description of how to define functions.

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 program name.

You have used a **PROGRAM** statement without supplying an identifier (name) after it. The format of the **PROGRAM** statement is PROGRAM *identifier* or PROGRAM *identifier*(*funparmlist*)..

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**, **INPUT**, **ASK**, and **GET** statements. Look up the statement in this manual 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.

Must be an integer.

The function specified must have an integer argument. Check to see that you have supplied an integer. If the argument is the result of calculation in your program, check to see if you need to use the **INT** or **ROUND** function to ensure that the result is an integer.

Must be BYTE file for READ BYTES. (-7351)

You're trying to use a **READ BYTE** statement on a text or record file. Files must be opened as byte files in order to be read or written as byte files. Change the **OPEN** statement for this file to include an **ORGANIZATION BYTE** part. (See Chapter 12 for a description of byte files.)

Must be BYTE file or empty for SET RECSIZE. (-7251)

SET RECSIZE can only be used on byte files or empty record files. Text files don't have record sizes, and non-empty record files have their record sizes built into them. Check to make sure that you're using the right channel.

Must be RECORD or BYTE file. (-8502)

You're using the **READ** or **WRITE** statement on a text file, or on a window. These statements work only for record and byte files. Check to make sure that you're using the proper channel. Use the **INPUT** and **PRINT** statements for text files and windows.

Must be RECORD or BYTE file for SET RECORD. (7202)

SET RECORD cannot be used on text files. You may reset a text file's pointer only to the very beginning or very end of the file.

Must be a TEXT file. (-8501)

You're using an **INPUT** or **PRINT** statement on a record file or a byte file. Record and byte files can only be read and written by **READ** and **WRITE** statements. Check to make sure that the file you opened is actually a text file, if you don't want to handle record files. (**ASK ORGANIZATION** can tell you a file's type.)

Must invoke with CHAIN to pass array argument other than command line.

You have specified an array in the *funparmlist* in the **PROGRAM** statement of your program but have invoked the program from the editor instead of from a **CHAIN** statement. Arrays may be passed into a program only through the use of the **CHAIN** statement.

Must SET RECSIZE before WRITE. (-8304)

You cannot write to an empty record file without somehow first indicating that file's record size. You may either execute a **SET RECSIZE** statement before the first **WRITE** statement, or you may specify the file's record size in the **OPEN** statement. See Chapters 12 and 18 for a description of **SET RECSIZE** and **OPEN**.

Name can't be redefined.

You can't use the same name for two different things. Thus, if you have a variable named *X*, you cannot also have a subroutine or array named *X*. Rename one of the things, so everything has its own unique name. True BASIC also prints this message when you try to use a "reserved word" as a variable.

Negative number to non-integral power. (3002)

You're trying to compute *n^x*, but *n* is negative and *x* is not an integer. The results are mathematically meaningless.

Nested INPUT statements with TIMEOUT clauses. (-8450)

While executing an **INPUT** or similar statement with a **TIMEOUT** clause, you have initiated another such statement, perhaps as a side effect of a function evaluation.

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 GET MOUSE on this computer. (11140)

You're trying to use a **GET MOUSE** statement, but your computer does not support a "mouse" input device or you have forgotten to "install the driver" for your mouse. You may be able to use the **GET POINT** statement instead.

No main program.

Your current file contains only functions, pictures, and/or subroutines – but doesn't contain a main program. Go back and write a main program!

No saved copy—still want to compile?

You've just given a compile command. Remember that True BASIC discards the uncompiled version of your program after compiling it. Here, True BASIC has noticed that you do not have a saved version of this program; either you've never saved a copy, or you've modified it since you last saved a copy. Answer "yes" if you want to go ahead and compile the program, or "no" if you want to stop and save a copy before trying again.

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. See Chapter 13 for a complete list of the color names known to True BASIC.

No such directory. (9008)

You have tried to enter a nonexistent directory, or given a name which isn't a directory.

No such directory (PostScript). (-9208)

This is the same as No such directory (9008), but specifically for a PostScript file being opened with **COPY_POSTSCRIPT**.

No such file. (9003)

You're trying to use a file that doesn't exist. Check to make sure you spelled the program's name properly, and to make sure you have inserted the correct disk in your computer.

(True BASIC attaches extensions to file names for some operating systems. For example, True BASIC programs are usually saved with a ".TRU" extension under the Windows and OS/2 systems. Check to make sure you've added the correct extension in an **OPEN** statement that tries to open programs.)

No such file (PostScript). (-9203)

This is the same as No such file (9003), but specifically for a PostScript file being opened with **COPY_POSTSCRIPT**.

No such line numbers.

You've given a range of line numbers in some command, but no lines have those numbers.

No USING item for output. (8202)

The format string in your **USING$** function or **PRINT USING** statement has no format items in it. That is, it contains only literal text. See Chapter 3 and Appendix D for descriptions of how to write format strings and add a format item.

ON index out of range, no ELSE given. (10001)

You've used an **ON GOTO** or **ON GOSUB** statement, and the number on which you're branching is out of range. For instance, you can get this message for "ON X GOSUB 100,120,130" if X is less than 1 or greater than 3. Check to make sure that the index's value is reasonable. If you want to handle indexes out of bounds, add an **ELSE** part to the **ON** statement. See Appendix E for a description.

Out of memory. (5000)

Your problem requires more memory than is attached to your computer. Since True BASIC will use all the memory supplied with your computer, you may be able to fix this problem by buying more memory. Otherwise, you must try to use less memory. Here are a few suggestions.

Use smaller arrays. Arrays can take up a surprising amount of space, especially if they have more than one dimension. If you have big arrays, see if you can solve your problem using smaller arrays.

Compile your program, and use the compiled version. The program text itself may take up a fair amount of room. Save your program and then compile it. This will compile your program into a much more compact form. Try running the compiled version of the program.

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. The same effect comes from a routine that **GOSUBs** into itself, but forgets to **RETURN**.

If you are an advanced programmer, you may wish to try the Packb and Unpackb routines as a last resort. They let you pack data, so that more data can be fit into memory. see the **PACKB** subroutine and the **UNPACKB** function for details.

Output item bigger than RECSIZE. (8301)

You're trying to **WRITE** an item to a record file, but it doesn't fit in the record size established for that file. Use **ASK RECSIZE** to find the file's record size. No item can be longer than the record size. Remember that numbers are stored using eight characters (bytes) per number.

Overflow. (1002)

You've computed a number bigger than the one your computer can handle. Use the built-in function **MAXNUM** to find 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 file INPUT. (1008)

You have generated an overflow in the course of inputting a number from a **TEXT** or **DISPLAY** SEQUENTIAL file.

Overflow in INPUT (nonfatal). (1007)

You have entered as input a number that is too large. You will need 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)

You have generated an overflow 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.

PostScript journalling already enabled. (-9210)

You have tried to call **BEGIN_POSTSCRIPT** after having already called **BEGIN_POSTSCRIPT** without subsequently calling **END_POSTSCRIPT**. You may journal to only one PostScript file at a time.

Program not available for CHAIN. (-10005)

You are attempting to chain to a program that is not available or perhaps does not exist. Check the name of the file in the CHAIN statement.

Program unit too large: name

A program unit in your program (usually the "Main program") has exceeded the 64K intermediate code limit. Move some code into external routines.

Range or part of range is already LOCKed. (-7108)

LOCK and UNLOCK are not available in the initial release of Version 5.

Reading past end of file. (8011)

You're trying to read more than exists in the file. This message can be given for any kind of file. Check carefully to see if the file contains everything you think it should. You may find the **MORE** #1 and **END** #1 tests useful for reading files of variable lengths.

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.

RETURN without GOSUB. (10002)

You've just executed a **RETURN** instruction, but there has been no corresponding **GOSUB** instruction. These things are rather difficult to diagnose and fix. Rewrite your program to use subroutines, and such problems cannot occur.

Record LENGTH <= 0. (7051)

You have specified a maximum record length less than or equal to 0 in the **RECSIZE** option in the **OPEN** statement.

Screen bounds must be 0 to 1. (-11003)

The bounds given on an **OPEN SCREEN** statement must lie in the range 0 to 1 (inclusive). No matter how big your screen is, the left and bottom edges are defined to be 0; the right and top edges are defined to be 1. See Chapter 13 for a description of how to open windows on the screen.

Screen minimum >= maximum. (-11002)

The **OPEN SCREEN** statement takes four numbers defining the edges of the new window: left, right, bottom, top. The right edge must be greater than the left edge, and the top edge must be greater than the bottom. Make sure you've typed the edges in the right order. If you're trying to get a reversed window, you can do this by reversing the edges in a **SET WINDOW** statement.

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 of 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.

String too long. (1051)

You've tried to create a string longer than the maximum size allowed on your computer.

String too long in assignment. (1106)

You've tried to use a string that is too long in a **LET** statement.

String too long in file INPUT. (1105)

You've tried to input from a file a string that is too long.

String too long in INPUT (nonfatal). (1054)

You've tried to input a string that is too long for the string variable or array. You will be required to enter the entire input line.

String too long in MAT. (1052)

You've tried to use a string that is too long in a **MAT** operation.

String too long in READ. (1053)

You've generated a string that is too long while reading from a data statement.

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. Contact tech support.

TAB column less than 1 (nonfatal). (4005)

You've used the Tab function in a **PRINT** statement, but its argument is less than 1. TAB(1) will be executed instead . If the **TAB** argument is greater than the current margin, it will be replaced by its value "modulo" the current margin.

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.

Timeout value < 0. (8402)

You have specified a timeout value < 0 in the **TIMEOUT** clause of an **INPUT** or similar statement.

Too few data in record. (8012)

You are attempting to **READ** or **INPUT** more elements than are present in a record from a **DISPLAY** or **INTERNAL SEQUENTIAL** file. Make sure the **READ** or **INPUT** statement contains the same number of variables as there are elements in the record.

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 channels open. (7102)

You have tried to open more than 25 channels — files or windows. True BASIC can only handle 25 channels being open at any time. Try closing some excess channels.

Too many channels open. (9007)

You've opened more files than the operating system allows. The maximum number that can be open at one time varies between machines, but generally is at least as large as 25.

Too many channels open (PostScript). (-9207)

This is the same as Too many channels open (9007), but specifically for a PostScript file being opened with **COPY_POSTSCRIPT**.

Too many data in record. (8013)

You are attempting to **READ** or **INPUT** fewer elements than are present in a record from a **DISPLAY** or **INTERNAL SEQUENTIAL** file. Make sure the **READ** or **INPUT** statement contains the same number of variables as there are elements in the record, or use **SKIP REST**.

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.

Too many shared channels in routine.

You can only have 1000 shared channels.

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. If you cannot find the error, try running your computer's diagnostic tests.

Trouble using disk or printer (PostScript). (-9202)

This is the same as Trouble using disk or printer (9002), but specifically for a PostScript file opened with **COPY_POST-SCRIPT**. It could occur while journalling graphics to a PostScript file.

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 public variable name in routine.

You're trying to use a public variable (in a **DECLARE PUBLIC** statement) that you haven't defined in a **PUBLIC** statement. Either add a **PUBLIC** statement for this variable in some appropriate routine, or change one of the **DECLARE PUBLIC** statements to a **PUBLIC** statement.

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 value for OPEN option. (7100)

The option you gave after the **ACCESS**, **CREATE**, or **ORGANIZATION** part of the **OPEN** statement doesn't exist. Print the string you used, and check it against the list of available options listed with the **OPEN** statement in Chapters 12 and 18. Although you can write the option in any mixture of upper-and lowercase letters, you may not abbreviate options or include excess spaces.

Unknown variable.

You are using **OPTION TYPO** to check for spelling mistakes, and it has found a variable name that you haven't declared anywhere. If True BASIC has found a typing mistake, just correct the spelling. Otherwise, add a **LOCAL** statement that lists this variable, or include the variable in its correct **DECLARE PUBLIC** or **SHARE** statement.

User-defined coordinates too large for GET MOUSE (POINT). (-11009)

The coordinates you have specified for the current window, while legal, are too large for **GET MOUSE** or **GET POINT** to perform the necessary arithmetic to return a value without causing an overflow. Try using a coordinate system with smaller extremes.

USING exponent too large for field (nonfatal). (8204)

You've tried to output a number with a **USING** string and the exponential field (i.e., ^^^) is not large enough. The partially completed line will be printed with asterisks in place of the using field, the numeric value will be printed on the next line, and printing will continue on the following line. You should extend the using field by adding one or more ^'s.

USING value too large for field (nonfatal). (8203)

You've tried to output a number or string with a **USING** string and the field (i.e. ###) is not large enough. The partially completed line will be printed, then the offending value will be printed on the next line, and the **USING** string continued on the third line. You should extend the field by adding one or more #'s.

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. See Chapters 2 and 17 for descriptions of how to write numbers for True BASIC.

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 file type for TRACE. (7402)

You have opened a file to use with **TRACE**, but it is the wrong type. It must be a **TEXT** or **DISPLAY SEQUENTIAL** file.

Wrong number of arguments.

You're calling a function, subprogram, or picture, but giving the wrong number of arguments. Look up the routine's definition in your program (if you've written it), or in this manual; then correct the call.

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 (+).

Wrong type of file. (7104)

Each kind of True BASIC file has its type built into it. Thus, True BASIC can tell text files from record files from compiled files. You have indicated, in your **OPEN** statement, that the file has a certain type, but it doesn't. Correct your **OPEN** statement. If you really mean to ignore the file's correct type, open it as a byte file.

Wrong type of file for LOCK (UNLOCK). (-7105)

LOCK and UNLOCK are not available in the initial release of Version 5.

You have two public variables called name.

You can't have more than one public variable with the same name. If you mean to have two different variables, change the name of one. Or if you want different parts of your program to refer to this same variable, change all but one of the related **PUBLIC** statements to be a **DECLARE PUBLIC** statement.

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.

ZONEWIDTH out of range. (4007)

You've executed a **SET ZONEWIDTH** statement, trying to set the zonewidth less than 1, or greater than the margin. Either fix the **SET ZONEWIDTH** statement, or use an **ASK MARGIN** statement to find out the current margin, and make the line wider with a **SET MARGIN** statement.

APPENDIX

D

PRINT USING Specifications

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 lets you 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; see Chapter 18, the **PRINT** statement.)

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. (See Chapter 18, for the **USING$** function.)

The following runtime errors can arise:

Exceptions:	8201	Badly formed USING string.
	8202	No USING item for output.
	8203	USING value too large for field. (nonfatal)
	8204	USING exponent too large for field. (nonfatal)

Formatting Numerical Values

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                  |***
```

Without **USING**, the number is printed left-justified with a leading space for a possible minus sign; it occupies 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 and 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. (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 String Values

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 Literals in Format Strings

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
```

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 .
```

ANSI -Standard Forms

True BASIC employs two forms of the **PRINT USING** and **USING$** functions. The first is the version used since version 1.0 of the Language System. This is the default in Version 5.

The other is a completely ANSI-standard version, which is slightly more restrictive. If you wish to use this version, you may include the statement **OPTION USING ANSI** in your program before the first **USING** statement that you wish to conform to the ANSI standard.

To switch back to the default version, you may include the statement **OPTION USING TRUE** before the first **USING** statement that you wish to conform to the True BASIC specifications.

APPENDIX

E

DO Programs

Do Programs in TB Silver again operate as they did in earlier DOS and Macintosh versions. That is, **Do Programs** are external subroutines with a certain calling sequence. They can operate on the source file in the editing window that was active when the Do program was invoked. Or they can take actions independent of the contents of the editing window. (This simple model was not used for versions 5.0 and 5.1.)

Do Program Calling Sequence

The Do program must be written as an external subroutine with the first two relevant lines as follows:

```
        EXTERNAL
        SUB MyDoProgram (lines$(), options$)
```

and with and END SUB somewhat later.

The actual names don't matter. In particular, the name of the subroutine is arbitrary. Do programs are accessed by the name of the file which contains them, not by the subroutine name.

Do programs can be invoked from the Run menu, or through a typed command. If the latter, TB Silver looks in the directory TBDo, unless you have established a different "alias" for {DO}.

This can be done by typing the command:

```
   Alias {do} c:\TBSilver\TBdo, C:\MyDir\Do
```

The Do command first examines the standard location for do programs; if not found, it examines additional directories.

When the Do subroutine starts, its first argument, the string array list$(), will contain the lines of the file currently displayed in the editing window that was active when the Do program was invoked. The string variable options$ will contain anything provided by the Do program dialog box, or following a comma with a typed command. An illustration of the latter is:

```
        do mydoprogram, 1 2 3
```

will supply the string "1 2 3" to the variable options$. Note that a comma must be used to separate the name of the do program file from the options.

You can modify the contents of the string array in the do program. When the do subroutine ends, the current editing window will contain the possibly modified lines.

Preparing Do programs is easy. Examine the source code in the subdirectory Sources in the directory TBDo for ideas.

An Example

Suppose you wish to convert the contents of an editing window to all uppercase letters, or to all lowercase letters. You might write:

```
        EXTERNAL
```

```
DO ChangeCase (lines$(), options$)

FOR i = 1 to ubound(lines$)
    IF options$ = "upper" then
       LET lines$(i) = Ucase$(lines$(i))
    ELSE IF options$ = "lower" then
       LET lines$(i) = lcase$(lines$(i))
    END IF
NEXT i

END SUB
```

The user might invoke this do subroutine with either of

```
do dofile, upper
do dofile, lower
```

A more sophisticated do subroutine would first check to see that the options provided were legal and spelled correctly.

APPENDIX

F

Scope and Aliasing

A program deals with various objects, such as variables, channels, and data. Some of these (specifically variables, arrays, functions, and data) are identified or referred to by their names. Channels are referred to by their channel numbers. Data is accessible according to its location in the program.

A program may contain statements whose effect may be the entire program, or only a part of it. For example, the margin and zonewidth of the default window are the same (until changed) throughout the entire program, while a **DIM** statement applies only to the program-unit that contains it.

The portion of the program over which an object can be accessed, or a statement has its effect, is called the scope of the object or statement. In this appendix we outline the scope of several parts of a True BASIC program. We also briefly describe aliasing, in which the same object is identified by two or different names.

Scope

The following list summarizes the ***scope*** of program elements and statements.

variable	program-unit
array	program-unit
procedure parameter	procedure
internal procedure	containing program-unit
program-unit	program
channel #0	program
channel #n, n>0	program-unit
GOSUB stack	containing procedure, exception handler, or program-unit
RND sequence	program
SET and **ASK** objects	program
line numbers	program-unit
DATA	program-unit
DIM	program-unit or module header
LOCAL	containing procedure or program-unit or module header
SHARE	module header and all module procedures, program unit
PUBLIC	program (**DECLARE PUBLIC** needed outside the module or program-unit containing the **PUBLIC** statement)
DECLARE PUBLIC	program-unit
DECLARE DEF	program-unit
OPTION TYPO	the rest of file
OPTION NOLET	the rest of file
OPTION ANGLE	the rest of program-unit, or the rest of the module (until overridden by the next **OPTION ANGLE**)
OPTION BASE	the rest of the program-unit, or the rest of the module (until overridden by the next **OPTION BASE**)

The following objects have as their scope the entire ***program*** (that is, the main program and all associated modules and external procedures).

1. Program-unit. The name of a program-unit is known throughout, and the program-unit can be accessed from any part of the program. (A program-unit is an external subroutine, defined function, or picture. A **DECLARE DEF** statement is needed to access an external defined function.)
2. Channel #0. Channel #0 always refers to the default window, which is used for input, output, and graphics.
3. **RND** sequence. The random number sequence provided by the **RND** function is the same for every program, in the absence of a **RANDOMIZE** statement.
4. **SET** and **ASK** objects. All file and graphics **SET** objects are known throughout the program and remain **SET** until changed. (File **SET** objects can be determined only by an **ASK** statement with the file channel number. Current window SET objects can be determined by an **ASK** statement without a channel number.)
5. **PUBLIC** variables and arrays. All variables and arrays whose names appear in **PUBLIC** statements are known throughout the program. (A **DECLARE PUBLIC** statement is needed if outside the module or program-unit containing the **PUBLIC** declaration.)

The following objects have as their scope the containing ***program-unit***.

1. Variables and arrays. Variables and arrays that appear in ***program-units*** are accessible throughout that ***program-unit.*** (This is true whether or not the variables or arrays appear in **LOCAL** statements, or whether the arrays are declared in **LOCAL** or **DIM** statements, as long as such **LOCAL** statements are not contained within an internal procedure.)
2. Parameters in external procedures. The parameters in an external **SUB**, **DEF**, or **PICTURE** statement have as their scope the associated external subroutine, defined function, or picture.
3. Internal procedure. An internal procedure contained within a ***program-unit*** can be accessed only from that ***program-unit***.
4. Channel #*n*, *n*>0. All channels, other than channel #0, are known to, and accessible from, only the ***program-unit*** containing their **OPEN** statement, or the ***program-unit*** containing them as parameters.
5. Line numbers. Line numbers are available as targets of **GOTO** and similar statements only from within the containing ***program-unit***.
6. **DATA** sequence. The ***data-sequence*** of a ***program-unit*** is formed from the **DATA** statements in the ***program-unit***, whether or not the **DATA** statements are contained within internal procedures.
7. **DECLARE DEF** and **DECLARE PUBLIC**. The **DECLARE DEF** and **DECLARE PUBLIC** statements are required in each *program-unit* that wishes to access public variables or arrays, or external defined functions. Such statements must appear before the first use of the public variable or defined function.
8. **SHARE** statements. Variables, arrays, and channel numbers included in **SHARE** statements are accessible throughout the containing ***program-unit*** or, if the **SHARE** statement occurs in a ***module-header***, throughout the module. Such variables and arrays retain their values between invocations of the ***program-unit***, if the ***program-unit*** is not a main program.

The following objects have as their scope the smaller of the containing internal procedure or ***program-unit***. That is, if they are found inside an internal procedure, they apply only to that procedure.

1. **GOSUB** stack. A separate “stack” of return addresses for use by **GOSUB** and **RETURN** statements is maintained for each internal procedure, and detached exception handler, as well as for each ***program-unit***.
2. **LOCAL** statement. A **LOCAL** statement in an internal procedure applies only to the containing internal procedure. (Since a **DIM** statement always applies to the containing ***program-unit***, a local array must always be declared in a **LOCAL** statement rather than a **DIM** statement.)

3. Parameters in internal procedures. The parameters in an internal **SUB**, **DEF**, or **PICTURE** statement have as their scope the associated internal subroutine, defined function, or picture. In addition, a variable or array having the same name in the containing ***program-unit*** is not available.

The ***scope*** of **OPTION** statements, being slightly different, is given separately.

1. **OPTION ANGLE**. Same as for **OPTION BASE** below.
2. **OPTION BASE**. An **OPTION BASE** statement in a ***program-unit*** that is not in a module is in effect from its location in a ***program-unit*** through the rest of the ***program-unit***, or until a new **OPTION BASE** statement is encountered. An **OPTION BASE** statement in a ***module*** is in effect through the rest of the ***module***, or until a new **OPTION BASE** statement is encountered, whether or not the **OPTION BASE** statement occurred in the ***module-header*** or in one of the procedures of the ***module***.
3. **OPTION NOLET**. Same as for **OPTION TYPO**.
4. **OPTION TYPO**. The **OPTION TYPO** statement is in effect from its location in a ***program-unit*** or ***module-header*** through the rest of the ***file***. The file may contain any number of external procedures and modules. It does not apply to statements in the ***program-unit*** or ***module-header*** that occur earlier.

Aliases

Aliases are instances whereby the same object, such as a variable, has two or more names. If care is not taken when using aliases, mysterious program behavior can result. Aliases can arise in the following ways.

1. The same variable or array appears in two or more positions in a **CALL** or **DRAW** statement, and the parameter is called reference.

```
CALL Zilch (a, a)
END

SUB Zilch (b, c)
    ! Here, b and c both refer to a.
    ! Changing b changes a and thus also c,
    ! and vice versa.
END SUB

CALL Zilch ((a), a)
END
SUB Zilch (b, c)
    ! Here, b and c initially have the same value.
    ! However, changing b does not change c,
    ! or vice versa.
END SUB
```

2. The same channel expression appears in two or more positions in a **CALL** or **DRAW** statement.

```
CALL Zilch (#2, #2)
END
SUB Zilch (#3, #4)
    ! Here, #3 and #4 refer to the same channel.
END SUB
```

3. An array is called by reference in a CALL or DRAW statement, as is an element of that array.

```
CALL Zilch (B(), B(1))
END
SUB Zilch (A(), X)
    ! Here, A(1) and X refer to the same quantity.
    ! Changing one will change the other.
END SUB
```

```
LET Y = Zilch(B(), B(1))
END

DEF Zilch (A(), X)
    ! Here, A(1) and X have the same initial value.
    ! However, no aliasing occurs because function
    ! parameters are always called "by value".
END DEF
```

4. A variable or array appears in a **CALL** or **DRAW** statement referring to an internal procedure, the same variable or array also being accessible to that procedure as a "global" variable or array.

```
CALL Zilch (a, aa())
SUB Zilch (b, bb())
    ! Here, b and a refer to the same quantity,
    ! and aa() and bb() refer to the same array.
END SUB

END
```

5. A channel appears in a **CALL** or **DRAW** statement referring to an internal procedure, the same channel also being accessible to that procedure as a "global" channel.

```
OPEN #1: ...
CALL Zilch (#1)
SUB Zilch (#2)
    ! Here, #1 and #2 refer to the same channel.
END SUB

END

CALL Zilch (#0)
SUB Zilch (#1)
    LET x = 17
    ! The following PRINT statements are equivalent.
    PRINT x .....
    PRINT #0: x         ! #0 is the default screen.
    PRINT #1: x         ! Here, #1 is the same as #0.
END SUB

END
```

In this last example, it does not matter whether the subroutine is internal or external.

APPENDIX

G

True BASIC Limits and Specifications

True BASIC was created to work as identically as possible on every computer. The architecture of various CPU chips, however, impose accuracy and size limitations on various operations.

System Limits

	Mac	**Win NT**	**Win 3.1**	**Win 95**	**OS/2**
Accuracy of numbers	<———		16 digits for all		———>
Accuracy of Sin, Cos, Tan, Atn, Log, Exp	<———		16 digits for all		———>
Smallest positive number Eps(0)	<———		2.2250739e-308 for all		———>
Largest positive number (Maxnum)	<———		1.7976931 e+308 for all		———>
Maximum string length	*	*	16-64 Mb	*	448 Mb
Maximum number of files open	<———		25 for all		———>
Maximum dimensions in an array	<———		255 for all		———>

* size of memory available on system

General True BASIC Limits

Length of variable names	31 characters
Largest line number	999,999
Number of channels open at once	25
Maximum **ZONEWIDTH**	Maxnum
Maximum **MARGIN**	Maxnum

APPENDIX

H

Line Numbers

There are seven statements in True BASIC that refer to line numbers. They are: **GOTO**, **IF GOSUB**, **RETURN**, **ON GOTO**, **ON GOSUB**, and **RESTORE** to a line-number. In addition, there are two input/output recovery statements that can appear only in certain input-output statements. They are: **IF MISSING THEN** and **IF THERE THEN**. The individual statements are discussed in Chapter 18; the valid forms are repeated here for convenience.

These statements can be used only with line-numbered programs. The rule is that a file containing one or more program-units must either have line-numbers on all lines, or no line-numbers. Thus, a given program can have parts which are line-numbered and parts which are not, provided the parts appear in different files.

goto::	GOTO *line-number*
if-then::	IF *logex* THEN *line-number*
	IF *logex* THEN *line-number* ELSE *line-number*
gosub::	GOSUB *line-number*
return::	RETURN
on-goto::	ON *rnumex* GOTO *line-number-list*
	ON *rnumex* GOTO *line-number-list* ELSE *line-number*
on-gosub::	ON *rnumex* GOSUB *line-number-list*
	ON *rnumex* GOSUB *line-number-list* ELSE *line-number*
restore::	RESTORE *line-number*
line-number-list::	*line-number …, line-number*
line-number::	*integer*
if-missing::	IF MISSING THEN *action*
if-there::	IF THERE THEN *action*
action::	EXIT DO
	EXIT FOR
	line-number

The keyword **GOTO** may be replaced by the keywords **GO TO**. The keyword **GOSUB** may be replaced by the keywords **GO SUB**.

A *simple-statement* may be used in place of a *line-number* in the **IF THEN** statement and in the **ELSE** part of the two **ON** statements, but not in a *line-number-list*. The *simple-statements* are listed in connection with the **IF** statement in Chapter 18. One of the *simple-statements* is the **GOTO** statement. A *line-number* in an **IF THEN** statement, in the **ELSE** part of an **ON** statement is equivalent to a GOTO statement with the same *line-number*.

Line-numbers must lie in the range 1 through 999999, inclusive. That is, they must not be 0 or negative, must not be one million or greater, and must not contain characters other than digits.

The target *line-numbers* must be within the "scope" of the statement. That is, the target *line-number* must be within the same *program-unit*.

In addition, the target *line-number* is restricted by additional rules, which are outlined below. A simple summary of the rules is that you can't jump *into* a control structure from any line outside it. And you can't even jump *out of* a **SUB**, **DEF**, or **PICTURE** from inside it. For other control structures, you may jump out of the structure.

Jumps and Procedures

If you are inside a *procedure* definition (*subroutine*, *defined-function*, or *picture*), you can't jump to the first line of the procedure (e.g., the **SUB** line) or to any line outside the procedure. You *can* jump to the last line, which is equivalent to an **EXIT** statement. These rules apply to both internal and external procedures.

If you are outside an internal procedure definition, you can jump to the first line of that procedure definition but not to other lines inside that procedure. In this case, the next statement to be executed will be the first one after the **END** statement for that internal procedure definition.

Jumps and Loops

For both **FOR** and **DO** loops, if you are outside a loop, you are not allowed to jump into the loop. You *can* jump to the first line of the loop (the **FOR** or **DO** statement), which isn't considered to be "inside" the loop. It is also legal to jump out of the loop from inside, including jumping to the first line, which will restart the loop.

Jumps and Decision Structures

If you are inside a part of a choice structure (**IF THEN ELSE** or **SELECT CASE**), you can't jump into a different section of the structure – from one **CASE** to another, or from the **THEN** clause to the **ELSE** part, etc. Also, you cannot jump into the structure from any line outside it.

If you are in a **WHEN** structure, you cannot jump from one part to the other. Also, you cannot jump into either part of the structure from any line outside it.

GOSUB RETURN Stacks

Each invocation of each procedure (internal or external) has its own private **GOSUB RETURN** stack, as does each *module-header*. This stack is empty upon entry to the procedure.

APPENDIX

I

Index of True BASIC Statements Functions and Subroutines

The main part of this section lists all keywords, functions, and subroutines included either in the True BASIC Language System itself or in one of the libraries. Following that list are sections on methods and attributes for the built-in **OBJECT** subroutine: “Object Methods,” “Control Object Attributes,” “Graphic Object Attributes,” “Menu Object Attributes,” “Window Object Attributes,” and “Events;” these are described in Chapter 19.

The following symbols are used for items other than True BASIC keywords (statements or parts of statements):

F	A function
S	A subroutine (use **CALL**)
AF	An array function
AC	An array constant

Library file names are also given for those functions, subroutines, and array functions and constants that are not built into the Language System. For example, **ACOSH** is a function provided in the FNHLIB.TRC library.

ABS (*n*)	F		Ch 8, 18
returns absolute value of *n*			
ACOS (*n*)	F		Ch 8, 18
returns arccosine of *n*			
ACOSH (*n*)	F	MathLib	Ch 23
returns hyperbolic arccosine of *n*			
ACOT (*x***)**	F	MathLib,	Ch 23
returns the arc cotangent of *x*			
ACOTH (*n*)	F	MathLib	Ch 23
returns hyperbolic arccotangent of *n*			
ACSC (*x***)**	F	MathLib	Ch 23
returns the arc cosecant of *x*			
ACSCH (*n*)	F	MathLib	Ch 23
returns hyperbolic arccosecant of *n*			
ADDDATAGRAPH (*x(), y(), pstyle, lstyle, col$*)	S	SGLib	Ch 23
draws another line graph of a data set over the current graph			
ADDFGRAPH (*style, col$*)	S	SGFunc	Ch 23
draws another line graph of the defined function F(x) over the current graph		SGLib	
ADDLSGRAPH (*x(), y(), style, col$*)	S	SGLib	Ch 23
computes and draws the least-squares linear fit for the specified data set			
ADD_POSTSCRIPT (*add_line$***)**	S	Gold	Ch 27
adds lines to a currently open PostScript file			

ALPHANUM$	returns set of all alphabetic and numeric characters	F	StrLib	Ch 23
AND (*a*, *b*)	returns bit-by-bit logical AND of *a* and *b*	F	HexLib	Ch 23
ANGLE (*x*, *y*)	returns counter-clockwise angle between positive x-axis and point (*x*, *y*)	F		Ch 8, 18
ASEC (*x*)	returns the arc secant of *x*	F	MathLib	Ch 23
ASECH (*n*)	returns hyperbolic arcsecant of *n*	F	MathLib	Ch 23
ASIN (*n*)	returns arcsine of *n*	F		Ch 8, 18
ASINH (*n*)	returns hyperbolic arcsine of *n*	F	MathLib	Ch 23
ASK #*n*: **ACCESS** *acc$*	reports access mode of file #*n* as INPUT, OUTPUT, or OUTIN			Ch 12, 18
ASK BACK *color*	reports current background color number			Ch 13, 18
ASK BACK *color$*	reports current background color name			Ch 13, 18
ASK COLOR *color*	reports current foreground color number			Ch 13, 18
ASK COLOR *color$*	reports current foreground color name			Ch 13,18
ASK COLOR MIX (*color*) *red*, *green*, *blue*	reports current mix of red, green, and blue intensities in specified color			Ch 13, 18
ASK CURSOR *row*, *column*	reports current position of text cursor			Ch 18
ASK CURSOR *status$*	reports current status of text cursor as ON or OFF			Ch 4, 18
ASK #*n*: **DATUM** *type$*	reports type of next item in stream file #*n* as NUMERIC, STRING, NONE, or UNKNOWN			Ch 12, 18
ASK DIRECTORY *directory$*	reports full pathname of current directory			Ch 12, 18
ASK #*n*: **ERASABLE** *ans$*	reports whether file #*n* may be erased as YES or NO			Ch 18
ASK #*n*: **FILESIZE** *size*	reports size of file #*n* in bytes or records			Ch 12, 18
ASK #*n*: **FILETYPE** *type$*	reports what is associated with channel #*n* as FILE or DEVICE			Ch 12, 18
ASK FREE MEMORY *freemem*	reports approximate free memory in bytes			Ch 18
ASK MARGIN *margin*	reports position of margin in current logical window			Ch 3, 18

ASK #*n*: **MARGIN** *margin* reports position of margin in file #*n*			Ch 12, 18
ASK MAX COLOR *colors* reports maximum color number			Ch 13, 18
ASK MAX CURSOR *rows*, *columns* reports number of rows and columns of text in current logical window			Ch 5, 13, 18
ASK MODE *mode$* reports current screen mode			Ch 18
ASK #*n*: **NAME** *name$* reports name of file #*n*			Ch 12, 18
ASK #*n*: **ORGANIZATION** *type$* reports the organization of file #*n* as TEXT, STREAM, RANDOM, RECORD, BYTE, or WINDOW			Ch 12, 18
ASK PIXELS *hor*, *vert* reports number of pixels in current logical window			Ch 13, 14, 18
ASK #*n*: **POINTER** *location$* reports position of pointer in file #*n* as BEGIN, END, or MIDDLE			Ch 12, 18
ASK #*n*: **RECORD** *recnum* reports number of current record in file #*n*			Ch 12, 18
ASK #*n*: **RECSIZE** *recsize* reports record size parameter of file #*n*			Ch 12, 18
ASK #*n*: **RECTYPE** *type$* reports type of file associated with #*n* as DISPLAY or INTERNAL			Ch 12, 18
ASK SCREEN *lft*, *rgt*, *btm*, *top* reports position of current logical window within physical window			Ch 14, 18
ASK #*n*: **SETTER** *ans$* reports whether file pointer may be set to a specific record in file #*n*			Ch 18
ASK TEXT JUSTIFY *hor$,vert$* reports text justification as LEFT, RIGHT, or CENTER and TOP, BOTTOM, BASE, or HALF			Ch 13, 18
ASK WINDOW *lft*, *rgt*, *btm*, *top* reports range of coordinates represented by current logical window			Ch 13, 14, 18
ASK ZONEWIDTH *width* reports width of print zones in current logical window			Ch 3, 18
ASK #*n*: **ZONEWIDTH** *width* reports width of print zones in file #*n*			Ch 12, 18
ASKANGLE (*measure$*) reports angle interpretation for subsequent polar graphs as DEG or RAD	S	SGLib	Ch 23
ASKBARTYPE (*type$*) reports grouping of bars in subsequent charts as SIDE, STACK, or OVER	S	BGLib	Ch 23
ASKGRAIN (*grain*) reports grain to be used for subsequent function plots	S	SGLib	Ch 23
ASKGRAPHTYPE (*type$*) reports type of graph to be used for subsequent plots as XY, LOGX, LOGY, LOGXY, or POLAR	S	SGLib	Ch 23

ASKGRID (*style$*) reports current presence, direction, and type of grid for charts and graphs	S	BGLib SGLib	Ch 23
ASKHLABEL (*hlabel$*) reports horizontal label to be used for subsequent charts and graphs	S	BGLib SGLib	Ch 23
ASKLAYOUT (*dir$*) reports direction in which bars of subsequent charts will be oriented as HORIZONTAL or VERTICAL	S	BGLib	Ch 23
ASKLS (*flag*) reports whether least-squares linear fits will be drawn for subsequent data graphs as 1 for yes or 0 for no	S	SGLib	Ch 23
ASKTEXT (*title$*, *hlabel$*, *vlabel$*) reports title and labels to be used for subsequent charts and graphs	S	BGLib SGLib	Ch 23
ASKTITLE (*title$*) reports title to be used for subsequent charts and graphs	S	BGLib SGLib	Ch 23
ASKVLABEL (*vlabel$*) reports vertical label to be used for subsequent charts and graphs	S	BGLib SGLib	Ch 23
ATANH (*n*) returns hyberbolic arctangent of *n*	F	MathLib	Ch 23
ATN (*n*) returns arctangent of *n*	F		Ch 8, 18
BALANCEBARS (*data()*, *units$()*, *colors$*) draws a balanced bar chart of the specified data	S	BGLib	Ch 23
BARCHART (*data1(,)*, *data2(,), units$()*, *legends$(), colors$*) draws a bar chart of specified data	S	BGLib	Ch 23
BEGIN_POSTSCRIPT (*"mypsfile",left,right,bottom,top***)** starts PostScript output to a file	S	Gold	Ch 27
BIN$ (*n*) returns *n* in binary representation	F	HexLib	Ch 23
BOX AREA *lft*, *rgt*, *btm*, *top* fills specified rectangular region with current foreground color			Ch 13, 18
BOX CIRCLE *lft*, *rgt*, *btm*, *top* inscribes ellipse in rectangular region			Ch 13, 18
BOX CLEAR *lft*, *rgt*, *btm*, *top* clears specified rectangular region with current background color			Ch 13, 18
BOX DISK *lft*, *rgt*, *btm*, *top* inscribes filled ellipse in rectangular region			Ch 13, 18
BOX ELLIPSE *lft*, *rgt*, *btm*, *top* inscribes ellipse in rectangular region			Ch 13, 18
BOX KEEP *lft*, *rgt*, *btm*, *top* **IN** *image$* stores contents of rectangular region in *image$*			Ch 13, 18
BOX LINES *lft*, *rgt*, *btm*, *top* draws outline of rectangular region			Ch 13, 18
BOX SHOW *image$* **AT** *x*, *y* displays rectangular region stored in *image$* with its lower left corner at point (*x*, *y*)			Ch 13, 18
BREAK breaks program execution			Ch 18

BREAKUP (*phrase$*, *word$*, *delim$*) extracts next word from phrase as delineated by specified delimiter	S	StrLib	Ch 23
CALL MySubroutine invokes specified subroutine			Ch 10, 18
CASE “n” specifies one case of SELECT CASE structure			Ch 5, 18
CAUSE ERROR *errnum*, *errmsg$* causes error number *errnum* with message *errmsg$*			Ch 16, 18
CAUSE EXCEPTION *errnum*, *errmsg$* causes error number *errnum* with message *errmsg$*			Ch 16, 18
CEIL (*n*) returns least integer greater than or equal to *n*	F		Ch 8, 18
**CENTER$** (*text$*, *width*, *back$*) returns *text$* centered within string of specified width	F	StrLib	Ch 23
CHAIN “!cc -o pgm pgm.c -lc”, **RETURN** chains to shell or other program			Ch 18
**CHARDIFF$** (*a$*, *b$*) returns all characters appearing in *a$* but not in *b$*	F	StrLib	Ch 23
**CHARINT$** (*a$*, *b$*) returns all characters appearing in both *a$* and *b$*	F	StrLib	Ch 23
CHARS$ (*from*, *to*) returns characters with ASCII codes in specified range, inclusive	F	StrLib	Ch 23
**CHARUNION$** (*a$*, *b$*) returns all characters appearing in either *a$* or *b$*	F	StrLib	Ch 23
CHR$ (*code*) returns character whose ASCII code corresponds to *code*	F		Ch 8, 18
CLEAR clears the current logical window with the current background color			Ch 3, 13, 14, 18
CLIPBOARD (*operation$*, *type$*, *item$*) stores or retrieves specified item using environment’s clipboard	S		Ch 18
CLOSE #*n* closes channel #*n*			Ch 3, 12, 18
COMLIB (*method, p1, p2, ps$*) general purpose communications subroutine	S		Ch 18, 22
COMOPEN (*method, #1, port, speed, options$*) opens a communications port	S		Ch 18, 22
COM_BUF (*type*) returns available buffer space	F	ComLib	Ch 22
COM_CONTROL (*opt$*) resets options and modem signals	S	ComLib	Ch 22
COM_OPEN (*#1, port, speed, opt$*) opens a communications port	S	ComLib	Ch 22
COM_RECEIVE (*buf$*) receives contents of input buffer	S	ComLib	Ch 22

COM_RECEIVELINE (*buf$*) receives input buffer up to first CR	S	ComLib	Ch 22
COM_SEND (*s$*) sends contents of s$ immediately	S	ComLib	Ch 22
COM_SENDBREAK sends a break	S	ComLIb	Ch 22
COM_SENDCR (*s$*) sends contents of s$ followed by a CR	S	ComLib	Ch 22
COM_SENDLINE (*s$*) sends contents of s$ followed by a CR/LF	S	ComLib	Ch 22
COM_STATUS (*type$*) returns the status of any of several types	F	ComLib	Ch 22
COM_SWITCH (*port*) switches to the port indicated	S	ComLib	Ch 22
COM_WAITLINE (*wtime, f, l$*) like COM_RECEIVELINE but with a timeout	S	ComLib	Ch 22
C**OM_WAITPROMPT** (*p$, wtime, f, s$*) waits for a specified prompt, with timeout	S	ComLib	Ch 22
CON returns numeric array containing ones in every element	AC		Ch 9, 18
CONTINUE continues with line following most recent error			Ch 16, 18
CONTROL$ returns set of all control characters	F	StrLib	Ch 23
CONVERT (*number$*) returns decimal value of hexadecimal, octal, binary, or decimal value	F	HexLib	Ch 23
COS (*n*) returns cosine of *n*	F		Ch 8, 18
COSH (*n*) returns hyperbolic cosine of *n*	F		Ch 8, 18
COT (*n*) returns cotangent of *n*	F		Ch 8, 18
COTH (*n*) returns hyperbolic cotangent of *n*	F	MathLib	Ch 23
CPOS (*string$*, *searchset$*) returns position of first occurrence of any character in *searchset$* within *string$*	F		Ch 8, 18
CPOS (*string$*, *searchset$*, *startpos*) returns position of first occurrence of any character in *searchset$* within *string$* starting at *startpos*	F		Ch 8, 18
CPOSR (*string$*, *searchset$*) returns position of last occurrence of any character in *searchset$* within *string$*	F		Ch 8, 18
CPOSR (*string$*, *searchset$*, *startpos*) returns position of last occurrence of any character in *searchset$* within *string$* starting at *startpos*	F		Ch 8, 18

CPSORTN (*values()*, *indices()*) performs customized pointer sort of *values()*	S	SortLib	Ch 23
CPSORTS (*values$()*, *indices()*) performs customized pointer sort of *values$()*	S	SortLib	Ch 23
CSC (*n*) returns cosecant of *n*	F		Ch 8, 18
CSCH (*n*) returns hyperbolic cosecant of *n*	F	MathLib	Ch 23
CSEARCHN (*array()*, *number*, *index*, *found*) performs customized search for *number* within *array()*	S	SortLib	Ch 23
CSEARCHS (*array()*, *string$*, *index*, *found*) performs customized search for *string$* within *array$()*	S	SortLib	Ch 23
CSORTN (*array()*) performs customized sort of *array()*	S	SortLib	Ch 23
CSORTS (*array$()*) performs customized sort of *array$()*	S	SortLib	Ch 23
DATA 1, 2, three contains program's built-in data			Ch 7, 18
DATAGRAPH (*x()*, *y()*, *pstyle*, *lstyle*, *col$*) draws a line graph of a data set	S	SGLib	Ch 23
DATE returns current date in form YYDDD	F		Ch 8, 18
DATE$ returns current date in form "YYYYMMDD"	F		Ch 8, 18
DEBUG on turns debugging on			Ch 18
DECLARE DEF MyDef declares specified function			Ch 10, 18
DECLARE FUNCTION MyFunction declares specified function			Ch 10, 18
DECLARE NUMERIC mynumvar declares specified variable as numeric			Ch 18
DECLARE PUBLIC mypubvar declares specified variable as public			Ch 11, 18
DECLARE STRING mystringvar$*20 declares specified variable as string with maximum length as specified			Ch 18
DECLARE SUB MySub has no effect			Ch 18
DEF MyDef(argument1, argument2$) begins defined function, single or multiple line			Ch 10, 18
DEF SQL_CONNECT (*myhost$, dbname$, myusername$, mypassword$***)** connects you to the ost server and database	F	Gold	Ch 26
DEF TS_ACCEPT (*tb_socket, family, port, addr$***)** returns	S	Gold	Ch 25
DEF TS_GETHOSTBYNAME$ (*name$***)** returns the name of the host for which you want an address	S	Gold	Ch 25

DEF TS_RECEIVE$ (*tb_socket, num_bytes***)** returns the number of bytes you specify as a data string	S	Gold	Ch 25
DEF TS_SOCKET (*family, type, protocol***)** analogous to the socket() function	S	Gold	Ch 25
DEG (*n*) returns the number of degrees in *n* radians	F		Ch 8, 18
**DELCHAR$** (*text$*, *oldchars$*) returns value of *text$* with all characters appearing in *oldchars$* removed	F	StrLib	Ch 23
**DELMIX$** (*text$*, *old$*) returns value of *text$* with all characters appearing in *oldchars$* removed regardless of case	F	StrLib	Ch 23
**DELSTR$** (*text$*, *old$*) returns value of *text$* with occurrences of *old$* removed	F	StrLib	Ch 23
DET (*matrix*) returns determinant of specified matrix	F		Ch 9, 18
DET returns determinant of last matrix inverted	F		Ch 9, 18
DIGITS$ returns set of all digit characters	F	StrLib	Ch 23
DIM array(10,10) dimensions the specified array(s)			Ch 9, 18
DIVIDE (*dividend*, *divisor*, *quotient*, *remainder*) divides *dividend* by *divisor* into *quotient* and *remainder*	S		Ch 8, 18
DO indicates beginning of do loop			Ch 6, 18
DOLLARS$ (*amount*) returns *amount* nicely formatted as a dollar amount	F	StrLib	Ch 23
DOLLARVAL (*string$*) returns numeric value of *string$* ignoring dollar signs, commas, asterisks, and spaces	F	StrLib	Ch 23
DOT (*a*, *b*) returns dot product of two vectors	F		Ch 9, 18
DRAW MyPicture draws specified picture			Ch 13, 18
ELSE part of IF structure			Ch 5, 18
ELSEIF part of IF structure			Ch 5, 18
END indicates end of main program			Ch 1, 18
END DEF indicates end of defined function			Ch 10, 18
END FUNCTION indicates end of defined function			Ch 10, 18
END HANDLER indicates end of detached error handler			Ch 16, 18

END IF indicates end of IF structure			Ch 5, 18
END MODULE indicates end of module			Ch 11, 18
END PICTURE indicates end of picture definition			Ch 13, 18
END_POSTSCRIPT (*n*) closes the file and stops journalling. The *n* is non-functional; reserved for future use.	S	Gold	Ch 27
END SELECT indicates end of SELECT structure			Ch 5, 18
END SUB indicates end of subroutine definition			Ch 10, 18
END WHEN indicates end of WHEN structure			Ch 16, 18
ENGLISHNUM$ (*n*) returns value of *n* represented in English	F	StrLib	Ch 23
EPS (*n*) returns epsilon of *n*	F		Ch 8, 18
ERASE #*n* erases contents of file #*n*			Ch 12, 18
EVAL (*expression$*) returns value of constant-based expression	F	StrLib	Ch 23
EXEC_ASKDIR (*dirname$*) returns the name of the current directory	S	ExecLib	Ch 12, 22
EXEC_CHDIR (*newdir$*) changes the current directory	S	ExecLib	Ch 12, 22
EXEC_CLIMBDIR *(dir, template$, name$(), size(), dlm$(), tlm$(), type$())* reports contents of current directory and its subdirectories	S	ExecLib	Ch 12, 22
EXEC_DISKSPACE (*used ,free*) returns the current hard disk use	S	ExecLib	Ch 12, 22
EXEC_MKDIR *(dirname$)* creates directory with specified name	S	ExecLib	Ch 12, 22
EXEC_READDIR *(template$, name$(), size(), dlm$(), tlm$(), type$(), vname$)* reports contents of current directory	S	ExecLib	Ch 12, 22
EXEC_RENAME *(oldname$, newname$)* renames file	S	ExecLib	Ch 12, 22
EXEC_RMDIR *(dirname$)* deletes directory with specified name	S	ExecLib	Ch 12,22
EXEC_SETDATE *(date$)* sets the system date	S	ExecLib	Ch 12,22
EXEC_SETTIME *(time$)* sets the system time	S	ExecLib	Ch 12,22
EXIT DEF exits enclosing defined function			Ch 10, 18

EXIT DO exits enclosing do loop			Ch 6, 18
EXIT FOR exits enclosing for loop			Ch 6, 18
EXIT FUNCTION exits enclosing defined function			Ch 10, 18
EXIT HANDLER exits enclosing error handler			Ch 16, 18
EXIT PICTURE exits enclosing picture			Ch 13, 18
EXIT SUB exits enclosing subroutine			Ch 10, 18
EXLINE returns line number of most recent error	F		Ch 16, 18
EXLINE$ returns line number lineage of most recent error	F		Ch 16, 18
EXP (*n*) returns natural exponent of *n*	F		Ch 8, 18
EXPLODE (*phrase$*, *words$()*, *delims$*) parses phrase into array of words using specified delimiters	S	StrLib	Ch 23
EXPLODEN (*phrase$*, *nums()*, *delims$*) parses phrase into array of numbers using specified delimiters	S	StrLib	Ch 23
EXTERNAL defines subsequent procedure definitions as external			Ch 11, 18
EXTEXT$ returns error message associated with most recent error	F		Ch 16, 18
EXTYPE returns error number associated with most recent error	F		Ch 16, 19
FGRAPH (*startx*, *endx*, *style*, *colors$*) draws line graph of externally defined function F(x) over specified domain	S	SGFunc SGLib	Ch 23
FILLARRAY (*from$()*, *to$()*, *width*) “fills” *to$()* array with contents of *from$()* array to specified width	S	StrLib	Ch 23
FILLFROM (#*n*, *width*, *result$*, *work$*) returns single “filled” line from file #*n*	S	StrLib	Ch 23
FLOOD *x*, *y* floods closed region containing point (*x*, *y*)			Ch 13, 18
FOR *index* = *start* **TO** *finish* **STEP** *increment* starts for loop			Ch 6, 18
FP (*n*) returns fractional part of *n*	F		Ch 8, 18
FUNCTION MyFunction(argument1, argument2$) begins defined function, single or multiple line			Ch 10, 18
GET KEY *code* gets single keystroke from keyboard buffer or user			Ch 4, 18

GET MOUSE *x*, *y*, *state*			Ch 13, 18
gets current position of mouse pointer and state of mouse button			
GET POINT *x*, *y*			Ch 13, 18
gets location on screen from user			
GOSUB 1000			Ch 18, App I
jumps to specified line number after pushing current line number on return stack			
GOTO 1000			Ch 18, App I
jumps to specified line number			
HANDLER MyHandler			Ch 16, 18
beginning of detached error handler definition			
**HEADER$** (*left$*, *center$*, *right$*, *width*, *back$*)	F	StrLib	Ch 23
returns “header” of specified width containing specified text items			
HEX$ (*n*)	F	HexLib	Ch 23
returns *n* in signed hexadecimal notation			
HEXW$ (*n*)	F	HexLib	Ch 23
returns *n* in four-byte unsigned hexadecimal notation			
HISTOGRAM (*data()*, *colors$*)	S	BGLib	Ch 23
draws histogram of specified data			
IBEAM (*high()*, *low()*, *units$()*, *colors$*)	S	BGLib	Ch 23
draws “I-beam” chart of specified data			
IDN	AF		Ch 9, 18
returns identity matrix			
IF			Ch 5, 18
beginning of IF statement or structure			
IMAGE: “###.###”			Ch 18, App I
defines format string for PRINT USING statement			
INPUT *number*, *string$*			Ch 4, 18
gets user input, after displaying default prompt, and assigns it to specified variable(s)			
INPUT #n: *number*, *string$*			Ch 12, 18
gets input from file and assigns it to specified variable(s)			
INPUT PROMPT *prompt$*: *number*, *string$*			Ch 4, 18
gets user input, after displaying specified prompt, and assigns it to specified variable(s)			
INT (*n*)	F		Ch 8, 18
returns greatest integer less than or equal to *n*			
**INTRIM$** (*text$*)	F	StrLib	Ch 23
returns value of *text$* with all series of spaces replaced with single spaces			
INV (*matrix*)	AF		Ch 9, 18
returns inverse of specified square matrix			
IP (*n*)	F		Ch 8, 18
returns integer part of *n*			
**JUSTIFY$** (*text$*, *width*)	F	StrLib	Ch 23
returns string containing *text$* justified to specified width			
JUSTIFYARRAY (*from$()*, *to$()*, *width*)	S	StrLib	Ch 23
“fills” and justifies *to$()* array with contents of *from$()* array to specified width			

JUSTIFYFROM (#*n*, *width*, *result$*, *work$*)	S	StrLib	Ch 23
returns single "filled" and justified line from file #*n*			
**KEEPCHAR$** (*text$*, *oldchars$*)	F	StrLib	Ch 23
returns value of *text$* with all characters not appearing in *oldchars$* removed			
LBOUND (*array*)	F		Ch 9, 18
returns lower bound of one-dimensional array			
LBOUND (*matrix*, *dimension*)	F		Ch 9, 18
returns lower bound of specified dimension of multi-dimensional matrix			
**LCASE$** (*text$*)	F		Ch 8, 18
returns value of *text$* with all letters converted to lowercase			
**LEFT$** (*text$*, *n*)	F	StrLib	Ch 23
returns leftmost *n* characters of *text$*			
LEN (*text$*)	F		Ch 8, 18
returns length of *text$* in bytes (or characters)			
LET *variable* = *expression*	F		Ch 2, 18
assigns value of expression on right to variable on left			
LETTERS$	F	StrLib	Ch 23
returns set of all alphabetic characters, uppercase and lowercase			
LIBRARY "Module.tru"			Ch 11, 18
names the file(s) containing procedures needed by program			
LINE INPUT *line$*			Ch 4, 18
gets line of user input, after displaying default prompt, and assigns it to specified variable			
LINE INPUT #n: *line$*			Ch 12, 18
gets line of input from file and assigns it to specified variable			
LINE INPUT PROMPT *prompt$*: *line$*			Ch 12, 18
gets line of user input, after displaying specified prompt, and assigns it to specified variable			
**LJUST$** (*text$*, *width*, *back$*)	F	StrLib	Ch 23
returns string of specified length containing value of *text$* left-justified			
LOCAL variable			Ch 10, 11, 18
defines specified variable(s) as local to enclosing program-unit			
LOG (*n*)	F		Ch 8, 18
returns natural logarithm of *n*			
LOG10 (*n*)	F		Ch 8, 18
returns common logarithm (to base 10) of *n*			
LOG2 (n)	F		Ch 8, 18
returns logarithm to base 2 of *n*			
LOOP			Ch 6, 18
indicates end of do loop			
LOWER$	F	StrLib	Ch 23
returns set of lowercase alphabetic characters			
**LTRIM$** (*text$*)	F		Ch 8, 18
returns value of *text$* with all leading spaces removed			
LVAL (*number$*)	F	StrLib	Ch 23
returns numeric value represented by leading characters of *number$*			

MAX (*a*, *b*)	F		Ch 8, 18
returns maximum of a and b			
MAXLEN (*string$*)	F		Ch 8, 18
returns maximum length of *string$*			
MAXNUM	F		Ch 8, 18
returns largest numeric value possible			
MAXSIZE (*array*)	F		Ch 8, 18
returns 2,147,483,648			
**MID$** (*string$*, *start*, *chars*)	F	StrLib	Ch 23
returns specified number of characters from within *string$* beginning at *start*			
MIN (*a*, *b*)	F		Ch 8, 18
returns minimum of a and b			
MOD (*x*, *y*)	F		Ch 8, 18
returns *x* modulo *y*			
MODULE foo			Ch 11, 18
beginning of module structure			
MULTIBAR (*data(,), units$(), legends$(), col$*)	S	BGLib	Ch 23
draws multi-bar chart of specified data sets			
MULTIHIST (*data(,), legends$(), col$*)	S	BGLib	Ch 23
draws multiple histograms of specified data sets in single frame			
NCPOS (*string$*, *searchset$*)	F		Ch 8, 18
returns position of first occurrence of any character not in *searchset$* within *string$*			
NCPOS (*string$*, *searchset$*, *startpos*)	F		Ch 8, 18
returns position of first occurrence of any character not in *searchset$* within *string$* starting at *startpos*			
NCPOSR (*string$*, *searchset$*)	F		Ch 8, 18
returns position of last occurrence of any character not in *searchset$* within *string$*			
NCPOSR (*string$*, *searchset$*, *startpos*)	F		Ch 8, 18
returns position of last occurrence of any character not in *searchset$* within *string$* starting at *startpos*			
NEXT *index*			Ch 6, 18
end of for loop			
NEXTWORD (*phrase$*, *word$*, *delims$*)	S	StrLib	Ch 23
gets next word from phrase, as delineated by any series of delimiter characters			
**NICEDATE$** (*date$*)	F	StrLib	Ch 23
returns full date represented by *date$*, including month name, day, and year			
**NICETIME$** (*time$*)	F	StrLib	Ch 23
returns time represented by *time$* as hour and minute followed by a.m. or p.m.			
**NOSPACE$** (*text$*)	F	StrLib	Ch 23
returns value of *text$* with all spaces removed			

NOW$ returns current as hour and minute followed by a.m. or p.m.	F	StrLib	Ch 23
**NPLUGCHAR$** (*text$*, *chars$*, *template$*) returns value of *text$* with characters not appearing in *chars$* replaced by specified template	F	StrLib	Ch 23
**NREPCHAR$** (*text$*, *oldchars$*, *new$*) returns value of *text$* with all characters not appearing in *oldchars$* replaced with *new$*	F	StrLib	
NUL$ returns array of null strings	AC		Ch 9, 18
NUM (*IEEEstring$*) returns numeric value represented by 8-byte *IEEEstring$*	F		Ch 8, 18
**NUM$** (*number$*) returns value of *number* represented as string in 8-byte IEEE format	F		Ch 8, 18
OBJECT (*method*, *id*, *attributes$*, *values$*, *values()*) invokes method for object; see methods and attributes listings at end of this Index	S		Ch 19
OCT$ (*n*) returns *n* in octal representation	F	HexLib	Ch 23
ON *n* **GOSUB** 10, 20, 30 causes jump to *n*-th line number, after placing current line number on return stack			Ch 18, App I
ON *n* **GOTO** 10, 20, 30 causes jump to *n*-th line number			Ch 18, App I
OPEN #1: **NAME** "foo", **CREATE** newold, **ACCESS** outin, **ORG** text opens channel to named file			Ch 12, 18
OPEN #1: **PRINTER** opens channel to printer			Ch 12, 18
OPEN #1: **SCREEN** *lft*, *rgt*, *btm*, *top* opens channel to logical window			Ch 13, 18
OPTION ANGLE degrees assumes angles measured in degrees for subsequent trigonometric functions			Ch 8, 18
OPTION ANGLE radians assumes angles measured in radians for subsequent trigonometric functions			Ch 8, 18
OPTION ARITHMETIC native has no effect			Ch 18
OPTION ARITHMETIC standard has no effect			Ch 18
OPTION BASE 7 sets default lower bound for arrays			Ch 18
OPTION COLLATE native has no effect			Ch 18
OPTION COLLATE standard has no effect			Ch 18
OPTION NOLET allows LET statements without LET keyword			Ch 1, 18

OPTION TYPO forces all variables to be declared prior to use			Ch 10, 11, 18
OPTION USING true assumes True BASIC rules for subsequent format strings			Ch 18
OPTION USING ansi assumes ANSI standard rules for subsequent format strings			Ch 18
OR (*a*, *b*) returns bit-by-bit logical OR of *a* and *b*	F	HexLib	Ch 23
ORD (*character$*) returns ASCII code of specified character	F		Ch 8, 18
PACKB (*string$*, *startbit*, *bitlength*, *integer*) packs value of *integer* into *bitlength* bits of *string$* beginning at *startbit*-th bit	S		Ch 12, 18
PAUSE *seconds* pauses for specified duration			Ch 13, 18
PI returns 3.14159...			Ch 8, 18
PICTURE MyPicture beginning of picture structure			Ch 13, 18
PIECHART (*data()*, *units$()*, *colors$*, *wedge*, *percent*) draws pie chart of specified data	S	BGLib	Ch 23
PLAY *music$* plays musical tune defined by *music$*			Ch 15, 18
PLOT *x*, *y* plots specified point			Ch 13, 18
PLOT *x1*, *y1*; *x2*, *y2* plots line segment connecting specified points			Ch 13, 18
PLOT AREA: *x1*, *y1*; *x2*, *y2*; *x3*, *y3* plots filled area defined by line segments connecting specified points			Ch 13, 18
PLOT LINES: x1,y1; y1,y2 plots line segments connecting specified points			Ch 13, 18
PLOT POINTS: *x1*, *y1*; *x2*, *y2* plots specified points			Ch 13, 18
PLOT TEXT, **AT** *x*, *y*: *string$* plots value of *string$* at specified point			Ch 13, 18
**PLUGCHAR$** (*text$*, *chars$*, *template$*) returns value of *text$* with characters appearing in *chars$* replaced by specified template	F	StrLib	Ch 23
**PLUGMIX$** (*text$*, *old$*, *template$*) returns value of *text$* with occurrences of *old$*, in any mix of case, replaced by specified template	F	StrLib	Ch 23
**PLUGSTR$** (*text$*, *old$*, *template$*) returns value of *text$* with occurrences of *old$* replaced by specified template	F	StrLib	Ch 23
POS (*string$*, *searchstr$*) returns position of first occurrence of *searchstr$* within *string$*	F		Ch 8, 18

POS (*string$*, *searchstr$*, *startpos*)	F		Ch 8, 18
returns position of first occurrence of *searchstr$* within *string$* starting at *startpos*			
POSR (*string$*, *searchstr$*)	F		Ch 8, 18
returns position of last occurrence of *searchstr$* within *string$*			
POSR (*string$*, *searchstr$*, *startpos*)	F		Ch 8, 18
returns position of last occurrence of *searchstr$* within *string$* starting at *startpos*			
PRINT *expression*			Ch 3, 18
prints expression(s) to screen			
PRINT USING *format$*: *expression*			Ch 3, 18
prints expression(s) to screen using specified format			
PRINT #*n*: *expression*			Ch 12, 18
prints expression(s) to channel #*n*			
PRINT #*n*, **USING** *format$*: *expression*			Ch 12, 18
prints expression(s) to channel #*n* using specified format			
PRIVATE MyProcedure			Ch 11, 18
defines specified procedure(s) as private to containing module			
PROGRAM MyProg (*parameter$*)			Ch 18
identifies current program and its chain parameters			
PSORTN (*values(), indices()*)	S	SortLib	Ch 23
performs standard pointer sort of *values()*			
PSORTS (*values$(), indices()*)	S	SortLib	Ch 23
performs standard pointer sort of *values$()*			
PUBLIC *variable*			Ch 11, 18
defines specified variable(s) as public, or global, throughout containing program-unit or module			
PUNCT$	F	StrLib	Ch 23
returns set of all punctuation characters			
RAD (*n*)	F		Ch 8, 18
returns the number of radians in *n* degrees			
RANDOMIZE			Ch 8, 18
seeds random number generator			
READ *variable*			Ch 7, 18
reads specified variable(s) from DATA statements			
READ #*n*: *variable*			Ch 12, 18
reads specified variable(s) from file #*n*			
READ #*n*, **BYTES** *b*: *variable$*			Ch 12, 18
reads *b* bytes from file #*n* to specified variable			
READCPIXEL (*x, y, r, g, b*)	S		Ch 18
reads the RGB color values at the user selected coordinate			
READ_IMAGE *(filetype$,boxkeepstring$, Filename$)*			Ch 14, 18
reads image from a file into a box keep string			
READPIXEL (*x, y*)	F		Ch 18
returns the color number of the pixel at the user selected coordinate			

REM This is a comment			Ch 1, 18
makes line comment which is ignored by compiler (may also use exclamation point (!))			
REMAINDER (*x*, *y*)	F		Ch 8, 18
returns remainder resulting from division of *x* by *y*			
**REPCHAR$** (*text$*, *oldchars$*, *new$*)	F	StrLib	Ch 23
returns value of *text$* with all characters appearing in *oldchars$* replaced with *new$*			
**REPEAT$** (*string$*, *n*)	F		Ch 8, 18
returns string containing *n* copies of *string$*			
**REPMIX$** (*text$*, *old$*, *new$*)	F	StrLib	Ch 23
returns value of *text$* with occurrences of *old$*, in any mix of case, replaced by *new$*			
**REPSTR$** (*text$*, *old$*, *new$*)	F	StrLib	Ch 23
returns value of *text$* with occurrences of *old$* replaced by *new$*			
RESET #*n*: BEGIN			Ch 12, 18
sets pointer in file #*n* to BEGIN, END, NEXT, or SAME record			
RESTORE			Ch 18, App E
restores pointer in data pool to first item			
RETRY			Ch 16, 18
retries line generating most recent error			
RETURN			Ch 18
jumps to line number at top of return stack			
**REVERSE$** (*text$*)	F	StrLib	Ch 23
returns value of *text$* with order of characters reversed			
REVERSEN (*array()*)	S	SortLib	Ch 23
reverses order of elements in *array()*			
REVERSES (*array$()*)	S	SortLib	Ch 23
reverses order of elements in *array$()*			
**RIGHT$** (*text$*, *n*)	F	StrLib	Ch 23
returns rightmost *n* characters of *text$*			
**RJUST$** (*text$*, *width*, *back$*)	F	StrLib	Ch 23
returns string of specified length containing value of *text$* right-justified			
RND	F		Ch 8, 18
returns psuedo-random number greater than or equal to 0 and less than 1			
**RNDSTR$** (*chars$*, *length*)	F	StrLib	Ch 23
returns string of specified length consisting of characters randomly chosen from *chars$*			
ROMAN$ (*n*)	F	StrLib	Ch 23
returns Roman numeral representation of *n*			
ROUND (*n*)	F		Ch 8, 18
returns *n* rounded to nearest integer			
ROUND (*n*, *places*)	F		Ch 8, 18
returns *n* rounded to specified number of places			
**RTRIM$** (s$)	F		Ch 8, 18
returns value of *text$* with all trailing spaces removed			
RUNTIME	F		Ch 18
returns seconds of processor time used since start of execution (or –1 if not applicable)			

SEARCHN (*array()*, *number*, *index*, *found*) searches sorted array for specified number, returning *index* and *found* flag	S	SortLib	Ch 23
SEARCHS (*array$()*, *string$*, *index*, *found*) searches sorted array for specified string, returning *index* and *found* flag	S	SortLib	Ch 23
SEC (*n*) returns secant of *n*	F		Ch 8, 18
SECH (*a*) returns hyperbolic secant of *n*	F	MathLib	Ch 23
SELECT CASE s$ start of a select case structure			Ch 5, 18
SET BACK *color* sets background color using specified color number			Ch 13, 18
SET BACK *color$* sets background color using specified color name			Ch 13, 18
SET COLOR *color* sets foreground color using specified color number			Ch 13, 18
SET COLOR *color$* sets foreground color using specified color name			Ch 13, 18
SET COLOR MIX (*color*) *red*, *green*, *blue* sets mix of red, green, and blue intensities used to form color with specified number			Ch 13, 18
SET CURSOR *row*, *column* sets position of text cursor			Ch 3, 13, 18
SET CURSOR *status$* sets status of text cursor as ON or OFF			Ch 4, 18
SET DIRECTORY *dirname$* sets current directory			Ch 12, 18
SET MARGIN *margin* sets position of margin in current logical window			Ch 3, 13, 18
SET #*n*: **MARGIN** *margin* sets position of margin in file #*n*			Ch 12, 18
SET MODE *mode$* sets screen mode			Ch 18
SET #*n*: **POINTER** BEGIN sets pointer in file #*n* to BEGIN, END, NEXT, or SAME record			Ch 12, 18
SET #*n*: **RECORD** *recnum* sets current record in file #*n* to record *recnum*			Ch 12, 18
SET #*n*: **RECSIZE** *recsize* sets record size parameter of empty file #*n*			Ch 12, 18
SET TEXT JUSTIFY *hor$,vert$* sets text justification as LEFT, RIGHT, or CENTER and TOP, BOTTOM, BASE, or HALF			Ch 13, 18
SET WINDOW *lft*, *rgt*, *btm*, *top* sets range of coordinates represented by current logical window			Ch 13, 14, 18
SET ZONEWIDTH *width* sets width of print zones in current logical window			Ch 3, 18

SET #*n*: **ZONEWIDTH** *width* sets width of print zones in file #*n*			Ch 12, 18
SETANGLE (*measure$*) sets angle interpretation for subsequent polar graphs as DEG or RAD	S	SGLib	Ch 23
SETBARTYPE (*type$*) sets grouping of bars in subsequent charts as SIDE, STACK, or OVER	S	BGLib	Ch 23
SETGRAIN (*grain*) sets grain to be used for subsequent function plots	S	SGLib	Ch 23
SETGRAPHTYPE (*type$*) sets type of graph to be used for subsequent plots as XY, LOGX, LOGY, LOGXY, or POLAR	S	SGLib	Ch 23
SETGRID (*style$*) sets presence, direction, and type of grid to be used for subsequent charts and graphs	S	BGLib SGLib	Ch 23
SETHLABEL (*hlabel$*) sets horizontal label to be used for subsequent charts and graphs	S	BGLib SGLib	Ch 23
SETLAYOUT (*dir$*) sets direction in which bars of subsequent charts will be oriented as HORIZONTAL or VERTICAL	S	BGLib	Ch 23
SETLS (*flag*) sets whether least-squares linear fits will be drawn for subsequent data graphs using 1 for yes or 0 for no	S	SGLib	Ch 23
SETTEXT (*title$*, *hlabel$*, *vlabel$*) sets title and labels to be used for subsequent charts and graphs	S	BGLib SGLib	Ch 23
SETTITLE (*title$*) sets title to be used for subsequent charts and graphs	S	BGLib SGLib	Ch 23
SETVLABEL (*vlabel$*) sets vertical label to be used for subsequent charts and graphs	S	BGLib SGLib	Ch 23
SETYSCALE (*lowy, highy*) sets vertical scale to be used for subsequent charts and graphs	S	SGLib	Ch 23
SGN (*n*) returns sign of *n* as –1, 0, or 1	F		Ch 8, 18
SHARE *variable* defines specified variable(s) as shared throughout containing program-unit or module			Ch 11, 18
**SHORTDATE$** (*date$*) returns date represented by *date$* as day of month, three-letter abbreviation of month name, and last two digits of year	F	StrLib	Ch 23
SIN (*n*) returns sine of *n*	F		Ch 8,.18
SINH (*n*) returns hyperbolic sine of *n*	F		Ch 8, 18
SIZE (*array*) returns number of elements in specified array	F		Ch 9 18
SIZE (*matrix*, *dimension*) returns number of elements in specified dimension of specified matrix	F		Ch 9, 18

SORTN (*array()*)	S	SortLib	Ch 23
sorts *array()* into ascending order			
SORTPOINTS (*x()*, *y()*)	S	SGLib	Ch 23
sorts parallel arrays *x()* and *y()* into ascending order by elements of *x()*			
SORTPOINTS2 (*x(,)*, *y(,)*)	S	SGLib	Ch 23
sorts parallel rows of *x(,)* and *y(,)* into ascending order by elements in rows of *x(,)*			
SORTS (array$())	S	SortLib	Ch 23
sorts *array$()* into ascending order			
SORT_OFF	S	SortLib	Ch 23
restores normal sorting			
SORT_OBSERVECASE	S	SortLib	Ch 23
sorts with upper- lowercase different (default)			
SORT_IGNORECASE	S	SortLib	Ch 23
Treats upper- lowercase as the same			
SORT_NICENUMBERS_ON	S	SortLib	Ch 23
initiates special nice number sorting			
SORT_NICENUMBERS_OFF	S	SortLib	Ch 23
restores normal ASCII sorting (default)			
SORT_NOKEYS	S	SortLib	Ch 23
restores entire string sorting (default)			
SORT_ONE KEY (*from, to*)	S	SortLib	Ch 23
sorts on substring field			
SORT_TWOKEYS (*f1, t1, f2, t2*)	S	SortLib	Ch 23
sorts on two substring fields			
SOUND *frequency*, *duration*			Ch 15, 18
sounds note with specified frequency and duration			
SQL_CLOSE (*context***)**	S	Gold	Ch 26
ends a SQL query			
SQL_GETALLRESULTS (*context, result$(rows, cols)***)**	S	Gold	Ch 26
returns results in a True BASIC array			
SQL_GETRESULTS (*context, startrow, rows, result$(rows, cols)***)**	S	Gold	Ch 26
returns results from last query			
SQL_GETHANDLES (*context, handles$***) - OBDC only**	S	Gold	Ch 26
used for access to OBDC databases			
SQL_MATCHFIELDS (*context, table$, pattern$, tablelist$()***)**	S	Gold	Ch 26
returns the field names and matching data types from a given table			
SQL_MATCHTABLES (*context, pattern$, tablelist$()***)**	S	Gold	Ch 26
returns table names from database			
SQL_QUERY (*context, query$,(rows, cols)***)**	S	Gold	Ch 26
returns results of query			
SQR (*n*)	F		Ch 8, 18
returns square root of *n*			
STOP			Ch 18
stops program			

STR$ (*n*)	returns value of *n* converted to string value	F		Ch 8, 18
STRWIDTH (*id*, *string$*)	returns length of *string$* measured in pixels within specified physical window	F		Ch 18
SUB MySub (*argument1, argument2$*)	begins subroutine definition			Ch 10, 18
SUB TS_BIND (*tb_socket, family, port, addr$***)**	combines information to make a standard address	S	Gold	Ch 25
SUB TS_CLOSE (*tb_socket***)**	close function for TB_Socket routine	S	Gold	Ch 25
SUB TS_CONNECT (*tb_socket, num_bytes***)**	returns a string containing data	S	Gold	Ch 25
SUB TS_INIT	used to initiate the use of the True BASIC socket routines	S	Gold	Ch 25
SUB TS_LISTEN (*tb_socket, backlog***)**	returns the maximum length of the queue of pending connections	S	Gold	Ch 25
SUB TS_SEND (*tb_socket, s$***)**	used to send data via the open socket	S	Gold	Ch 25
SUPERVAL (*table$*, *expression$*, *value*)	returns value of variable-based expression	S	StrLib	Ch 23
SYS_EVENT (*timer*, *type$*, *window*, *misc1*, *misc2*)	gets next system event from operating system's event queue	S		Ch 18, 20
SYSTEM (*operation*, *result1$, result2$, result3$*)	performs one of several operating system functions	S		Ch 18
TAB (*column*)	repositions text cursor at specified column of current print line	F		Ch 3, 18
TAB (*row*, *column*)	repositions text cursor at specified column of specified row	F		Ch 3, 18
TAN (*n*)	returns tangent of *n*	F		Ch 8, 18
TANH (*n*)	returns hyperbolic tangent of *n*	F		Ch 8, 18
TBD (*e, t, type*, *title$*, *msg$*, *btn$*, *name$*, *text$*, *start*, *dflt*, *timeout*, *result)*	creates and displays modal dialog box, returning user's response	S		Ch 18, 21
TBDX (*e, r, b, t, parm$,parm( ), type*, *title$*, *msg$*, *btn$*, *name$*, *text$*, *start*, *dflt*, *timeout*, *result)*	creates and displays a custom dialog box, returning user's response	S		Ch 18, 21
TC_CHECKBOX_CREATE (*cid, text$, lft, rgt, btm, top*)	creates a check box along with its associated text	S	TrueCtrl	Ch 14, 22
TC_CHECKBOX_GET (*cid, state*)	returns the status of a check box	S	TrueCtrl	Ch 14, 22
TC_CHECKBOX_SET (*cid, state*)	sets the status of a check box	S	TrueCtrl	Ch 14, 22
TC_CLEANUP	deactivates event handling, required before termination	S	TrueCtrl	Ch 14, 22

TC_EDIT_CHECKFIELD (*cid, errormess$*) checks the edit field contents against the edit field format	S	TrueCtrl	Ch 14, 22
TC_EDIT_CREATE (*cid, text$, lft, rgt, btm, top*) creates an editable field with initial text	S	TrueCtrl	Ch 14, 22
TC_EDIT_GETTEXT (*cid, text$*) returns the current text in an edit field	S	TrueCtrl	Ch 14, 22
TC_EDIT_SETFORMAT (*cid, format$*) sets the format for an edit field	S	TrueCtrl	Ch 14, 22
TC_EDIT_SETTEXT (*cid, newtext$*) sets the text for an edit field	S	TrueCtrl	Ch 14, 22
TC_ENV_SET (*attribute$, value$*) sets certain environment attributes; Unix only	S	TrueCtrl	Ch 14, 22
TC_ERASE (*id*) hides (makes invisible) an object or control; opposite of TC_SHOW	S	TrueCtrl	Ch 14, 22
TC_EVENT (*timer, event$, window, x1, x2*) returns the next event on the event queue	S	TrueCtrl	Ch 14, 22
TC_FONTSAVAILABLE (*fonts$()*) returns names of fonts available on the system	S	TrueCtrl	Ch 14, 22
TC_FREE (*id*) deletes (releases) an object or control; opposite of TC_XXX_CREATE	S	TrueCtrl	Ch 14, 22
TC_GET (*id, attributes$, value$, values()*) returns current value(s) of attributes	S	TrueCtrl	Ch 14, 22
TC_GETRECT (*id, xl, xr, yb, yt*) returns current coordinates in pixels of object or control	S	TrueCtrl	Ch 14, 22
TC_GETSCREENSIZE (*left, right, bottom, top*) returns the coordinates of the entire screen in pixels	S	TrueCtrl	Ch 14, 22
TC_GETSYSINFO (*attribute$, value$, values()*) returns certain system environment values	S	TrueCtrl	Ch 14, 22
TC_GETTEXT (*id, text$*) returns current text of a control	S	TrueCtrl	Ch 22
TC_GRAPH_CREATE (*gid, type$, lft, rgt, btm, top*) creates a graphical object	S	TrueCtrl	Ch 14, 22
TC_GRAPH_GETIMAGETOBOX (*cid, boxstring$*) converts graphical image into a box keep string	S	TrueCtrl	Ch14, 22
TC_GRAPH_SCALE (*gid, scalex, scaley*) changes the size of a graphical object	S	TrueCtrl	Ch 22
TC_GRAPH_SETALINE (*gid, start, end*) sets arrow heads at either end of an arrow line	S	TrueCtrl	Ch 14, 22
TC_GRAPH_SETARC (*gid, starta, enda*) sets the extent of an arc or pie segment	S	TrueCtrl	Ch 14, 22
TC_GRAPH_SETBRUSH (*gid, backcolor, color, pattern$*) sets the back color and brush properties for graphical object	S	TrueCtrl	Ch 14, 22
TC_GRAPH_SETDRAWMODE (*gid, mode$*) sets the logical drawing mode for a graphical object	S	TrueCtrl	Ch 14, 22

TC_GRAPH_SETIMAGE (*gid, filename$, adjustflag*) sets and displays an image from a file	S	TrueCtrl	Ch 22
TC_GRAPH_SETIMAGEFROMBOX (*cid, boxstring$*) sets a graphical image from a box keep string	S	TrueCtrl	Ch 14, 22
TC_GRAPH_SETIMAGEFROMFILE (*cid, filename$, filetype$, adjustflag*) sets a graphical image from a file (similar to TC_GRAPH_SETIMAGE)	S	TrueCtrl	Ch 14, 22
TC_GRAPH_SETPEN (*gid, width, color, style$, pattern$*) sets the pen properties for a graphical object	S	TrueCtrl	Ch 14, 22
TC_GRAPH_SETPOLY (*gid, points(,)*) sets the points for a polyline or polygon	S	TrueCtrl	Ch 14, 22
TC_GRAPH_SETROUNDRECT (*gid, owidth, oheight*) sets the curvature for a rounded rectangle	S	TrueCtrl	Ch 14, 22
TC_GRAPH_SHIFT (*gid, deltax, deltay*) moves a graphical object	S	TrueCtrl	Ch 22
TC_GROUPBOX_CREATE (*cid, title$, lft, rgt, btm, top*) creates a group box with title	S	TrueCtrl	Ch 14, 22
TC_INIT activates event handling, required by True Controls	S	TrueCtrl	Ch 14, 22
TC_LISTBOX_CREATE (*cid, mode$(), lft, rgt, btm, top*) creates a selection list box; specify entries with TC_SetList	S	TrueCtrl	Ch 14, 22
TC_LISTBOX_GET (*cid, selection()*) returns all currently selected element positions	S	TrueCtrl	Ch 14, 22
TC_LISTBOX_SET (*cid, selection*) selects a specified element in a list box (use TC_SetList to enter elements)	S	TrueCtrl	Ch 14, 22
TC_LISTBTN_CREATE (*cid, text$(), lft, rgt, btm, top*) creates a list button with scrollable text	S	TrueCtrl	Ch 14, 22
TC_LISTBTN_GET (*cid, selection*) returns currently selected list element	S	TrueCtrl	Ch 14, 22
TC_LISTBTN_SET (*cid, selection*) selects currently selected list element	S	TrueCtrl	Ch 14, 22
TC_LISTEDIT_CREATE (*cid, text$(), lft, rgt, btm, top*) creates a list edit button with scrollable text	S	TrueCtrl	Ch 14, 22
TC_LISTEDIT_GET (*cid, text$*) returns current text in edit field	S	TrueCtrl	Ch 14, 22
TC_LISTEDIT_SET (*cid, text$*) selects current text in edit field	S	TrueCtrl	Ch 14, 22
TC_MENU_ADDITEM (*wid, menu, text$*) adds a menu item to the end of a menu	S	TrueCtrl	Ch 14, 22
TC_MENU_ADDMENU (*wid, menu$()*) adds new menu at end of current menu structure	S	TrueCtrl	Ch 14, 22
TC_MENU_DELITEM (*wid, menu, item*) deletes a specific menu item from a menu	S	TrueCtrl	Ch 14, 22
TC_MENU_DELMENU (*wid*) deletes last menu of existing menu structure	S	TrueCtrl	Ch 14, 22

TC_MENU_FREE (*wid*) deletes or frees (releases) an entire menu structure	S	TrueCtrl	Ch 14, 22
TC_MENU_GETCHECK (*wid, menu, item, flag*) returns state of check mark on a menu item	S	TrueCtrl	Ch 22
TC_MENU_GETENABLE (*wid, menu, item, flag*) returns state of menu item (enabled or disabled)	S	TrueCtrl	Ch 14, 22
TC_MENU_GETTEXT (*wid, menu, item, text$*) returns text in specified menu position	S	TrueCtrl	Ch 14, 22
TC_MENU_SET (*wid, menu$(,)*) sets the menu structure for a physical window	S	TrueCtrl	Ch 14, 22
TC_MENU_SETCHECK (*wid, menu, item, flag*) sets or removes a menu check mark	S	TrueCtrl	Ch 14, 22
TC_MENU_SETENABLE (*wid, menu, item, flag*) enables or disables a menu item	S	TrueCtrl	Ch 14, 22
TC_MENU_SETTEXT (*wid, menu, item, newtext$*) resets the text of a menu item	S	TrueCtrl	Ch 14, 22
TC_PIXTOUSER (*px, py, ws, wy*) converts from pixel to world (user) coordinates	S	TrueCtrl	Ch 14, 22
TC_PUSHBTN_CREATE (*cid, text$, lft, rgt, btm, top*) creates a push button with text	S	TrueCtrl	Ch 14, 22
TC_RADIOGROUP_CREATE (*rbid, text$(), lft, rgt, btm, top*) creates a radio button group with text	S	TrueCtrl	Ch 14, 22
TC_RADIOGROUP_ON (*rgid, button*) returns which radio button is currently on	S	TrueCtrl	Ch 14, 22
TC_RADIOGROUP_SET (*rbid, button*) sets specified radio button to on	S	TrueCtrl	Ch 14, 22
TC_RADIOGROUP_SETTEXT (*rid, button, newtext$*) sets the text of a radio group button	S	TrueCtrl	Ch 22
TC_SBAR_CREATE (*cid, type$, lft, rgt, btm, top*) creates a scroll bar, vertical or horizontal	S	TrueCtrl	Ch 14, 22
TC_SBAR_GETINCREMENTS (*cid, single, page*) returns the current increments for a scroll bar	S	TrueCtrl	Ch 14, 22
TC_SBAR_GETPOSITION (*cid, position*) returns the current scroll bar slider position	S	TrueCtrl	Ch 14, 22
TC_SBAR_GETRANGE (*cid, srange, erange, prop*) returns the current range and slider parameters for a scroll bar	S	TrueCtrl	Ch 14, 22
TC_SBAR_SETINCREMENTS (*cid, single, page*) sets the increments for a scroll bar	S	TrueCtrl	Ch 14, 22
TC_SBAR_SETPOSITION (*cid, position)* sets the scroll bar slider position	S	TrueCtrl	Ch 14, 22
TC_SBAR_SETRANGE(*cid, srange, erange, prop*) sets the range and slider parameters for a scroll bar	S	TrueCtrl	Ch 14, 22
TC_SELECT (*id*) selects a selectable control	S	TrueCtrl	Ch 22

TC_SENSITIZE (*id, flag*) determines whether a control responds to mouse clicks	S	TrueCtrl	Ch 22
TC_SET (*id, attributes$, value$, values()*) sets the value(s) of certain attributes	S	TrueCtrl	Ch 14, 22
TC_SETLIST (*id, text$()*) sets the text list of a list button, list edit button, or list box	S	TrueCtrl	Ch 14, 22
TC_SETRECT (*id, newxl, newxr, newyb, newyt*) resets the coordinates of an object or control	S	TrueCtrl	Ch 14, 22
TC_SETRECTPIXELS (*id, xl, xr, yb, yt*) resets the location of an object or control using pixels	S	TrueCtrl	Ch 22
TC_SETRECTUSERS (*id, xl, xr, yb, yt*) resets the location of an object or control using user coordinates	S	TrueCtrl	Ch 22
TC_SETTEXT (*id, text$*) sets the text of any control that allows setting its text	S	TrueCtrl	Ch 22
TC_SETTEXTJUSTIFY (*id, justify$*) resets the text justify parameter for a control	S	TrueCtrl	Ch 14, 22
TC_SETUNITSTOPIXELS sets the units flag to pixels	S	TrueCtrl	Ch 22
TC_SETUNITSTOUSERS sets the units flag to user coordinates	S	TrueCtrl	Ch 22
TC_SHOW (*id*) shows (makes visible) an object or control; opposite of TC_ERASE	S	TrueCtrl	Ch 14, 22
TC_SHOW_DEFAULT (*defaultflag*) sets the flag for a control or graphical object to be shown upon creation	S	TrueCtrl	Ch 14, 22
TC_STEXT_CREATE (*cid, text$, lft, rgt, btm, top*) creates a static text field	S	TrueCtrl	Ch 14, 22
TC_TXED_APPEND (*cid, text$, revealflag*) appends a line of text at the end of a text edit control	S	TrueCtrl	Ch 22
TC_TXED_COPY (*cid*) copies selected text to the clipboard	S	TrueCtrl	Ch 14, 22
TC_TXED_CREATE (*cid, options$, lft, rgt, btm, top*) creates a text edit control, with options$	S	TrueCtrl	Ch 14, 22
TC_TXED_CUT (*cid*) cuts selected text to the clipboard	S	TrueCtrl	Ch 14, 22
TC_TXED_FIND (*cid, case, word, key$, p, l1, c1, l2, c2, found*) finds key word in the text, returns its position	S	TrueCtrl	Ch 14, 22
TC_TXED_GETCURSOR (*cid, p, l, c*) returns the current cursor position of a text edit control	S	TrueCtrl	Ch 22
TC_TXED_GETSELECTION (*cid, p1, l1, c1, p2, l2, c2*) returns the extent of selected text in a text edit control	S	TrueCtrl	Ch 14, 22
TC_TXED_GETTEXT (*cid, text$*) returns the current text of a text edit control	S	TrueCtrl	Ch 14, 22
TC_TXED_PASTE (*cid*) inserts clipboard contents to current cursor position or replaces selected text	S	TrueCtrl	Ch 14, 22

TC_WIN_PRINT (*wid*) S TrueCtrl Ch 22
prints contents of current window to current printer

TC_WIN_REALIZEPALETTE (*win*) S TrueCtrl Ch 22
“realizes” the True BASIC palette to the system palette

TC_WIN_SETBRUSH (*wid, backcolor, color, pattern$*) S TrueCtrl Ch 14, 22
sets the back color, brush properties for a phsyical window

TC_WIN_SETCURSOR (*wid, cursor$*) S TrueCtrl Ch 14, 22
sets the cursor style for a phsyical window

TC_WIN_SETDRAWMODE (*wid, mode$*) S TrueCtrl Ch 14, 22
sets the logical drawing mode for a phsyical window

TC_WIN_SETFONT (*wid, fontname$, fontsize, fontsytle$*) S TrueCtrl Ch 14, 22
sets the font properties for a phsyical window

TC_WIN_SETPEN (*wid, width, color, style$, pattern$*) S TrueCtrl Ch 14, 22
sets the pen properties for a phsyical window

TC_WIN_SETTITLE (*wid, title$*) S TrueCtrl Ch 14, 22
sets the title of a phsyical window

TC_WIN_SWITCH (*wid*) S TrueCtrl Ch 14, 22
makes a phsyical window both active (in front) and target for ouput

TC_WIN_SWITCHCURRENT S TrueCtrl Ch 22
switches to the full logical window of the current physical window

TC_WIN_TARGET (*wid*) S TrueCtrl Ch 14, 22
directs output to a phsyical window

TC_WIN_UPDATE (*wid, left, right, bottom, left*) S TrueCtrl Ch 22
updates contents of physical window or portion thereof

TC_WIN_VALID (*wid*) S TrueCtrl Ch 14, 22
causes error if phsyical window is not valid

TC_WINHSBAR_GETINCREMENTS (*wid, single, page*) S TrueCtrl Ch 22
returns the current values of the increments for an attached horizontal scroll bar

TC_WINHSBAR_GETPOSITION (*wid, position*) S TrueCtrl Ch 22
returnsthe current location of an attached horizontal scroll bar slider
in terms of the scrollbar parameters `srange`, `erange`, and `prop`

TC_WINHSBAR_GETRANGE (*wid, srange, erange, prop*) S TrueCtrl Ch 22
returns the current values of the scrollbar parameters
for an attached horizontal scroll bar

TC_WINHSBAR_SETINCREMENTS (*wid, single, page*) S TrueCtrl Ch 22
sets the scrollbar increments for an attached horizontal scroll bar

TC_WINHSBAR_SETPOSITION (*wid, position*) S TrueCtrl Ch 22
sets the position of the slider (the thumb) of an attached horizontal scroll
bar in terms of the scrollbar parameters `srange`, `erange`, and `prop`

TC_WINHSBAR_SETRANGE (*wid, srange, erange, prop*) S TrueCtrl Ch 22
sets the scrollbar parameters for extreme positions and proportional size
(if allowed) of the slider for an attached horizontal scroll bar

TC_WINVSBAR_GETINCREMENTS (*wid, single, page*) S TrueCtrl Ch 22
returns the current values of the increments
for an attached vertical scroll bar

TC_WINVSBAR_GETPOSITION (*wid, position*) returns the current location of an attached vertical scroll bar slider in terms of the scrollbar parameters `srange`, `erange`, and `prop`	S	TrueCtrl	Ch 22
TC_WINVSBAR_GETRANGE (*wid, srange, erange, prop*) returns the current values of the scrollbar parameters for an attached vertical scroll bar	S	TrueCtrl	Ch 22
TC_WINVSBAR_SETINCREMENTS (*wid, single, page*) sets the scrollbar increments for an attached vertical scroll bar	S	TrueCtrl	Ch 22
TC_WINVSBAR_SETPOSITION (*wid, position*) sets the position of the slider (the thumb) of an attached vertical scroll bar in terms of the scrollbar parameters `srange`, `erange`, and `prop`	S	TrueCtrl	Ch 22
TC_WINVSBAR_SETRANGE (*wid, srange, erange, prop*) sets the scrollbar parameters for extreme positions and proportional size (if allowed) of the slider for an attached vertical scroll bar	S	TrueCtrl	Ch 22
TD_GETFILE (*extension$, filename$, changedir*) displays a standard open file dialog box	S	TrueDial	Ch 14, 22
TD_ASKDELIMITER (*delim$*) returns the delimiter used as a line break	S	TrueDial	Ch 22
TD_GETTIMEOUT (*timeout*) returns the current timeout for dialog boxes	S	TrueDial	Ch 14, 22
TD_INPUT (*message, button$, text$, default, result*) displays a single-line input dialog box	S	TrueDial	Ch 14, 22
TD_INPUTM (*title$, message$, button$, labels$(), text$(), highlight, default, result*) displays a multiple-line input dialog box	S	TrueDial	Ch 14, 22
TD_LINEINPUT (*message$, text$*) displays a line input dialog box	S	TrueDial	Ch 14, 22
TD_LIST (*message$, button$, list$(), choice, default, result*) displays a selection list box	S	TrueDial	Ch 14, 22
TD_MESSAGE (*title$, message$, button$, default, result*) displays a simple message dialog box with a title	S	TrueDial	Ch 14, 22
TD_SAVEFILE (*extension$, filename$*) displays a standard save file dialog box	S	TrueDial	Ch 14, 22
TD_SETDELIMITER (*delim$*) sets the delimiter used as a line break	S	TrueDial	Ch 22
TD_SETLOCATION (*x loc, y loc*) sets upper-left corner for type 1 and 4 dialog boxes in pixels	S	TrueDial	Ch 22
TD_SETTIMEOUT (*timeout*) sets the timeout for all subsequent dialog boxes	S	TrueDial	Ch 14, 22
TD_WARN (*message$, button$, default, result*) displays a simple message dialog box (without a title)	S	TrueDial	Ch 14, 22
TD_YN (*message$, default, result*) displays a simple Yes-No dialog box	S	TrueDial	Ch 14, 22
TD_YNC (*message$, default, result*) displays a simple Yes-No-Cancel dialaog box	S	TrueDial	Ch 14, 22
TIME returns current time as number of seconds since midnight	F		Ch 8, 18
TIME$ returns current time in form "HH:MM:SS"	F		Ch 8, 18

TODAY$	F	StrLib	Ch 23
returns current date consisting of weekday, month name, day of month, and year			
TRACE on			Ch 18
turns trace mode on			
**TRIM$** (*text$*)	F		Ch 8, 18
returns value of *text$* with all leading and trailing spaces removed			
TRN (*matrix*)	AF		Ch 9, 18
returns transpose of *matrix*			
TRUNCATE (*value*, *n*)	F		Ch 8, 18
returns value of *value* truncated to *n* places			
UBOUND (*array*)	F		Ch 9, 18
returns upper bound of one-dimensional array			
UBOUND (*matrix*, *dimension*)	F		Ch 9, 18
returns upper bound of specified dimension of multi-dimensional matrix			
**UCASE$** (*text$*)	F		Ch 9, 18
returns value of *text$* with all letters converted to uppercase			
**UNIQ$** (*text$*)	F	StrLib	Ch 23
returns set of all characters contained in *text$*			
UNPACKB (*string$*, *startbit*, *bitlength*)	F		Ch 12, 18
returns integer value stored within specified bits of *string$*			
UNSAVE *filename$*			Ch 12, 18
deletes specified file from current directory			
UPPER$	F	StrLib	Ch 23
returns set of uppercase alphabetic characters			
USE			Ch 16, 18
separates protected portion from handler portion in WHEN structure			
**USING$** (*format$*, *value*)	F		Ch 8, 18
returns specified value(s) formatted according to specified format string			
VAL (*number$*)	F		Ch 8, 18
returns numerical value of contents of *number$*			
WEEKDAY (*adate$*)	F	StrLib	Ch 23
returns number of weekday on which specified date falls			
**WEEKDAY$** (*adate$*)	F	StrLib	Ch 23
returns name of weekday on which specified date falls			
WHEN ERROR IN			Ch 16, 18
beginning of error handler with attached handler portion			
WHEN ERROR USE MyHandler			Ch 16, 18
beginning of error handler with detached handler portion			
WINDOW #*n*			Ch 13, 14, 18
switches to logical window associated with #*n*			
WRITE #*n*: *variable*			Ch 12, 18
writes specified variable(s) to internal file #*n*			
WRITE_IMAGE (*filetype$, boxkeepstring#, filename$*)			Ch 12, 18
writes to an image file from a box keep string			
XOR (*a*, *b*)	F	HexLib	Ch 23
returns bit-by-bit logical XOR of *a* and *b*			
ZER (*a*)	AC		Ch 9, 18
returns array of zeros			

Object Methods (see Chapter 19 for full explanations)

`CALL Object (method, id, attributes$, values$, values())`

OBJM_COPY
creates copy of specified object
ID of copy returned in `values(1)`

OBJM_CREATE
creates new object; must be one of following types:

`OBJT_CONTROL`	control
`OBJT_GRAPHIC`	graphic image
`OBJT_GROUP`	group of radio buttons
`OBJT_MENU`	menu or menu item
`OBJT_WINDOW`	physical window

returns ID of created object in `id`

OBJM_ERASE
hides specified object

OBJM_FREE
destroys specified object and frees memory it occupied

OBJM_GET
gets specified attributes of specified object

OBJM_PAGESETUP
displays page setup dialog box for PRINT method parameters

OBJM_PRINT (applies to windows only)
prints contents of specified window to printer

OBJM_SCROLL
scrolls contents of window

OBJM_SELECT (applies to windows only)
selects specified physical window
passes type of selection in `values(1)`:

1	make window current (target)
2	make window active (moves to front)
3	make window current (target) and active

OBJM_SET
sets specified attributes of specified object

OBJM_SHOW
displays specified object

OBJM_SYSINFO
reports useful, system-wide information; reports or sets operating system parameters

OBJM_TXE_ADD_PAR (applies to text edit controls only)
adds text in `values$` as new paragraph following paragraph specified in `values(1)`

OBJM_TXE_APPEND_PAR (applies to text edit controls only)
appends text in `values$` to end of paragraph specified in `values(1)`

OBJM_TXE_DEL_PAR (applies to text edit controls only)
deletes paragraph specified in `values(1)`

OBJM_TXE_HSCROLL (applies to text edit controls only)
scrolls contents of text edit control horizontally by pixels specified in `values(1)`

OBJM_TXE_RESUME (applies to text edit controls only)
resumes processing by specified text edit control

OBJM_TXE_SUSPEND (applies to text edit controls only)
suspends processing by specified text edit control

OBJM_TXE_VSCROLL (applies to text edit controls only)
scrolls contents of text edit control vertically by pixels specified in `values(1)`

OBJM_UPDATE (applies to windows only)
updates contents of specified physical window
passes region to update as left, right, bottom, and top edges in `values(2)`…`values(5)`
specifies units used to interpret these coordinates by `values(1)`:

0	pixel coordinates
1	user coordinates

Control Object Attributes (see Chapter 19 for full explanations)

ACTIVE
determines whether text editor is active or inactive

0	inactive
1	active (default)

BACK COLOR
determines color of background within text editor; default is white (-2)

BORDER
determines whether text editor is displayed with border

0	no border (default)
1	border

BORDER COLOR
determines color of border surrounding text editor; default is black (-1)

BOTTOM RELATIVE
determines whether bottom edge of control is defined relative to bottom edge of window

0	absolute (default)
1	relative

CHAR LIMIT
determines total number of characters user may enter into text editor; default is 65535

CONTROL TYPE
determines type of control

`CTLT_PUSHBUTTON`	push button
`CTLT_RADIOBUTTON`	radio button
`CTLT_CHECKBOX`	check box
`CTLT_HSCROLL`	horizontal scroll bar
`CTLT_VSCROLL`	vertical scroll bar
`CTLT_EDIT`	edit text field
`CTLT_TEXT`	static text field
`CTLT_LBOX`	list box
`CTLT_LISTBUTTON`	list button
`CTLT_LISTEDIT`	list edit text field
`CTLT_GROUPBOX`	group box
`CTLT_TXED`	text editor
`CTLT_ICON`	icon type (may be ignored)

DEFAULT
determines appearance of active button

1	special outline
0	normal

END RANGE
determines end of range represented by scroll bar

EXIT CHAR
specifies characters that can deselect and exit field

FONT METRICS
reports leading, ascender, descender, xsize, ysize, and bearing of text editor's font in pixels
may not be set

FONT NAME
determines font used in text editor. Portable font names are HELVETICA, FIXED (default), TIMES, and SYSTEM. Other names may be available in specific operating environments

FONT SIZE
determines size of font used in text editor
specified in points (1 point = 1/72 inch); default is 10 points

FONT STYLE
determines style of font used in text editor
allowable styles are PLAIN (default), BOLD, ITALIC, and BOLDITALIC

FORE COLOR
determines color of text within text editor; default is black

FORMAT
determines the format associated with a particular edit field

HSCROLL
provides storage for ID of horizontal scroll bar associated with text editor
ignored by True BASIC

INSERTION
determines position of insertion point within text editor
specified as paragraph, line, and character passed in `values()`

KEY EVENTS
determines whether text editor generates TXE_KEYPRESS events

0	no TXE_KEYPRESS events (default)
1	generate TXE_KEYPRESS events

LEFT RELATIVE
determines whether left edge of control is defined relative to left edge of window

0	absolute (default)
1	relative

LINE
reports text of specified line within text editor's contents
passes number of paragraph and line effected in `values()`
passes text of line in `values$`

LINES IN PAR
reports number of lines in specified paragraph of text editor's contents
returns number of paragraph in `values(1)`
returns number of lines in paragraph in `values(2)`

LIST
determines item in list box, list button, or list edit text field
items passed as flat array

MARGIN
determines margin, in pixels, of text editor

MAX WIDTH
reports length, in pixels, of longest line in text editor's contents

MOUSE EVENTS

determines whether text editor generates TXE_MOUSE events

0	no TXE_MOUSE events (default)
1	generate TXE_MOUSE events

NAME

name of control; ignored by True BASIC

NUM CHARS

reports number of characters in text editor's contents

NUM LINES

reports number of lines in text editor's contents

NUM PARS

reports number of paragraphs in text editor's contents

ON

returns the ID (in `values(1)`) and ordinal number (in `values(2)`) of the radio button that is on, if any

ORIGIN

reports position of upper, left-hand corner of text visible in text editor
returned as paragraph, relative line, absolute line, and pixel offset in `values()`

PAGE INCREMENT

provides storage for page distance to scroll for scroll bar; ignored by True BASIC

POSITION

determines or returns position of scroll bar's slider. Must be in appropriate range

PROPORTION

determines the proportional size of the scroll bar slider, if possible

READONLY

determines whether text editor allows editing

0	allow editing (default)
1	read only

RECTANGLE

determines size and position of control
passed as four numeric values representing positions of left, right, bottom, and top edges

RELATIVE

determines whether all edges of control are defined relative to edges of window

0	absolute (default)
1	relative

RIGHT RELATIVE

determines whether right edge of control is defined relative to right edge of window

0	absolute (default)
1	relative

SELECTION

reports or sets item(s) currently selected in list box or list button in `values()`; may not be set for list button

or

determines currently selected text within text editor; passed as starting and ending paragraphs, lines, and characters in `values()`

SELECTION MODE

determines how user may select items in list box

`LBXM_SINGLE`	one item at a time
`LBXM_MULTIPLE`	one or more items at a time
`LBXM_READONLY`	no items; view only

SENSITIVE

determines whether control is sensitive or insensitive

0	insensitive
1	sensitive (default)

SINGLE INCREMENT

provides storage for arrow distance to scroll for scroll bar; ignored by True BASIC

START RANGE

determines start of range represented by scroll bar

STATE

determines visual state of radio button or check box

0	off (default)
1	on

TEXT

determines text within buttons, text or edit fields, or text editor

TEXT JUSTIFY

determines justification of text within button or static text field

0	left-aligned
1	centered (default)
2	right-aligned

TEXTEDIT

provides storage for ID of text edit control associated with scroll bar
ignored by True BASIC

TITLE

determines title of group box

TOP RELATIVE

determines whether top edge of control is defined relative to top edge of window

0	absolute (default)
1	relative

TRAP CHAR

specifies keys to be trapped by text editor
number of keys to be trapped passed first in `values()`
each key to be trapped passed as key code and trap code as follows:

1	suspend text editor on keystroke
2	do not suspend; let text editor handle keystroke
3	suspend text editor only if text is selected

UNITS

determines coordinate system used for positioning control

0	pixel coordinates (default)
1	user coordinates

VISIBLE

reports whether control is currently visible; may not be set

1	visible
0	hidden

VSCROLL

provides storage for ID of vertical scroll bar associated with text editor
ignored by True BASIC

WRAP

determines whether text editor wraps paragraphs into lines

0	do not wrap paragraphs (default)
1	wrap paragraphs

Graphic Object Attributes

BACKGROUND COLOR
determines the background color to be used inside a graphics object; default is white (-2)

BOTTOM RELATIVE
determines whether bottom edge of image is defined relative to bottom edge of window

0	absolute (default)
1	relative

BRUSH COLOR
determines number of color used for image's brush; default is black (-1)

BRUSH PATTERN
determines pattern of image's pen

`PBP_HOLLOW`	no visible pattern
`PBP_SOLID`	solid pattern (default)
`PBP_HORZ`	pattern of horizontal lines
`PBP_VERT`	pattern vertical lines
`PBP_FDIAG`	pattern of "forward" diagonal lines
`PBP_BDIAG`	pattern of "backward" diagonal lines
`PBP_CROSS`	pattern of crossing horizontal and vertical lines
`PBP_DIAGCROSS`	pattern of crossing diagonal lines

DRAWMODE
determines drawing mode of image's pen and brush

`DM_COPY`	ignores current contents of window (default)
`DM_OR`	bitwise OR between bit planes
`DM_XOR`	bitwise XOR between bit planes
`DM_CLEAR`	clears current contents to color 0
`DM_NOT_COPY`	logical negation of COPY
`DM_NOT_OR`	logical negation of OR
`DM_NOT_XOR`	logical negation of XOR
`DM_NOT_CLEAR`	logical negation of CLEAR

END ARROW
determines if arrow head is at ending point of arrow line

0	no arrow at ending point (default)
1	arrow at ending point

FILENAME
specifies name of file containing graphical image

FORCE PALETTE
specifies whether to use existing palette (0), or to use palette of the image (1)

GRAPHIC TYPE
determines type of image

`GRFT_ALINE`	line segment with arrow head(s)
`GRFT_ARC`	arc
`GRFT_CIRCLE`	ellipse
`GRFT_IMAGE`	import a graphical image
`GRFT_LINE`	line segment
`GRFT_PIE`	pie
`GRFT_POLYGON`	closed polygon
`GRFT_POLYLINE`	open polygon
`GRFT_RECTANGLE`	rectangle
`GRFT_ROUNDRECT`	rounded rectangle

IMAGE HEIGHT
returns the height of a graphical image

IMAGE WIDTH
returns the width of a graphical image

LEFT RELATIVE
determines whether left edge of image is defined relative to left edge of window

0	absolute (default)
1	relative

NAME
name of image; ignored by True BASIC

OVAL HEIGHT
determines height of oval defining roundness of rounded rectangle's corners
interpreted as pixel or user coordinates as determined by UNITS attribute

OVAL WIDTH
determines width of oval defining roundness of rounded rectangle's corners
interpreted as pixel or user coordinates as determined by UNITS attribute

PEN COLOR
determines number of color used for image's pen
default is black (-1)

PEN PATTERN
determines pattern of image's pen

`PBP_HOLLOW`	no visible pattern
`PBP_SOLID`	solid pattern (default)
`PBP_RUBBER`	grayish, or dappled, pattern

PEN STYLE
determines style of image's pen

`PENS_SOLID`	solid line (default)
`PENS_DOT`	dotted line
`PENS_DASH`	dashed line

POINTS
determines number and locations of vertices of polygon
number of points passed first, then x-y pairs defining points

RECTANGLE
determines size and position of image
passed as four numeric values representing positions of left, right, bottom, and top edges

RELATIVE
determines whether all edges of image are defined relative to edges of window

0	absolute (default)
1	relative

RIGHT RELATIVE
determines whether right edge of image is defined relative to right edge of window

0	absolute (default)
1	relative

START ARROW
determines if arrow head is at starting point of arrow line

0	no arrow at starting point (default)
1	arrow at starting point

START X
x-coordinate of point marking start of arc or pie

START Y
y-coordinate of point marking start of arc or pie

STOP X
x-coordinate of point marking end of arc or pie

STOP Y
y-coordinate of point marking end of arc or pie

TOP RELATIVE
determines whether top edge of image is defined relative to top edge of window

0	absolute (default)
1	relative

UNITS
determines coordinate system used for positioning image

0	pixel coordinates (default)
1	user coordinates

VISIBLE
reports whether image is currently visible; may not be set

1	visible
0	hidden

WIDTH
determines width of image's pen in pixels
default is 1

Menu Object Attributes

CHECKABLE
reserves space to left of item for check mark (in some environments)

0	no space reserved
1	space reserved (default)

CHECKED
determines whether menu item is preceded by check mark

0	no check mark (default)
1	check mark

MENU TYPE

`MENT_BAR`	**element is a new menu**
`MENT_ITME`	**element is a new item in existing menu**

ENABLED
determines whether menu or menu item will be enabled or disabled

0	disabled
1	enabled (default)

MKEY
determines keyboard equivalent for menu or menu item
passed as numeric key code

SEPARATOR
determines whether menu item will be displayed as separator

0	standard menu item (default)
1	separator

TEXT
determines text displayed as menu title or item

Window Object Attributes

BACKGROUND COLOR

determines number of color used for window's background; default is white (-2)

BRUSH COLOR

determines number of color used for window's brush; default is black (-1)

BRUSH PATTERN

determines pattern of window's pen

`PBP_HOLLOW`	no visible pattern
`PBP_SOLID`	solid pattern (default)
`PBP_HORZ`	pattern of horizontal lines
`PBP_VERT`	pattern vertical lines
`PBP_FDIAG`	pattern of "forward" diagonal lines
`PBP_BDIAG`	pattern of "backward" diagonal lines
`PBP_CROSS`	pattern of crossing horizontal and vertical lines
`PBP_DIAGCROSS`	pattern of crossing diagonal lines

CLOSE BOX

determines whether window contains close box

0	no close box (default)
1	close box

CURSOR

determines the type of cursor in use, or to be used

DRAWMODE

determines drawing mode of image's pen and brush

`DM_COPY`	ignores current contents of window (default)
`DM_OR`	bitwise OR between bit planes
`DM_XOR`	bitwise XOR between bit planes
`DM_CLEAR`	clears current contents to color 0
`DM_NOT_COPY`	logical negation of COPY
`DM_NOT_OR`	logical negation of OR
`DM_NOT_XOR`	logical negation of XOR
`DM_NOT_CLEAR`	logical negation of CLEAR

END RANGE HORIZONTAL

determines end of range represented by horizontal scroll bar

END RANGE VERTICAL

determines end of range represented by vertical scroll bar

FOCUS ORDER

determines the focus order of all controls in a window

FONT METRICS

reports leading, ascender, descender, xsize, ysize, and bearing of window's current font in pixels
may not be set

FONT NAME

determines font used in window
portable font names are HELVETICA, FIXED (default), TIMES, and SYSTEM
other names may be available in specific operating environments

FONT SIZE

determines size of font used in window
specified in points (1 point = 1/72 inch); default is 10 points

FONT STYLE

determines style of font used in window
allowable styles are PLAIN (default), BOLD, ITALIC, and BOLDITALIC

HSCROLL

determines whether window contains attached horizontal scroll bar

0	no attached horizontal scroll bar (default)
1	attached horizontal scroll bar

ICONIZABLE

determines whether window may be iconized

0	not iconizable (default)
1	iconizable

IMMUNE

determines whether window is immune

0	not immune
1	immune (default)

MOUSE MOVE

controls whether MOUSE MOVE events are returned by **Sys_Event**

0	no events returned (default)
1	events returned

NAME

name of window; ignored by True BASIC

PAGE HORIZONTAL

provides storage for page distance for horizontal scroll bar; ignored by True BASIC

PAGE VERTICAL

provides storage for page distance for vertical scroll bar; ignored by True BASIC

PEN COLOR

determines number of color used for window's pen
default is black (-1)

PEN PATTERN

determines pattern of window's pen

`PBP_HOLLOW`	no visible pattern
`PBP_SOLID`	solid pattern (default)
`PBP_RUBBER`	grayish, or dappled, pattern

PEN STYLE

determines style of window's pen

`PENS_SOLID`	solid line (default)
`PENS_DOT`	dotted line
`PENS_DASH`	dashed line

POSITION HORIZONTAL

determines position of horizontal scroll bar's slider
must be in appropriate range

POSITION VERTICAL

determines position of vertical scroll bar's slider
must be in appropriate range

PROPORTION HORIZONTAL

determines the proportional size of the scroll bar slider, if possible

PROPORTION VERTICAL
determines the proportional size of the scroll bar slider, if possible

RECTANGLE
determines size and position of window
passed as four numeric values representing positions of left, right, bottom, and top edges

RESIZE BOX
determines whether window contains resize box

0	no resize box (default)
1	resize box

SINGLE HORIZONTAL
provides storage for arrow distance for horizontal scroll bar; ignored by True BASIC

SINGLE VERTICAL
provides storage for arrow distance for vertical scroll bar; ignored by True BASIC

SOLID MIX
determines whether colors within window may be patterns (Windows versions only)

0	allow patterns (default)
1	solid colors only

START RANGE HORIZONTAL
determines start of range represented by horizontal scroll bar

START RANGE VERTICAL
determines start of range represented by vertical scroll bar

TEXTEDIT
reports ID of attached text edit control, if there is one

TITLE
determines text of window's title

TYPE
determines type of window

`WINT_DOC`	1	standard document window
`WINT_PLAIN`	2	plain window; single line border
`WINT_DOUBLE`	3	plain window; double line border
`WINT_NOBORDER`	7	no border

VISIBLE
determines or reports whether window is visible; may only be set during creation

1	visible
0	hidden

VSCROLL
determines whether window contains attached vertical scroll bar

0	no attached vertical scroll bar (default)
1	attached vertical scroll bar

WIDTH
determines width of window's pen in pixels
default is 1

Events

`CALL Sys_Event (timer, event$, window, x1, x2)`

`""` (NULL Event)
no event in event queue

CONTROL DESELECTED
control in window with indicated ID has been released or has lost input focus
control ID returned in `x2`

CONTROL DOUBLE
item in list control within window with indicated ID has been double-clicked
list control ID will be returned in `x2`

CONTROL SELECT
control in window with indicated ID has been clicked or has gained input focus
control ID returned in `x2`

CONTROL SINGLE
item in list control within window with indicated ID has been clicked once
list control ID will be returned in `x2`

DOUBLE
left (or only) mouse button has been double-clicked in window with indicated ID
x-coordinate, in user coordinates, returned in `x1`; y-coordinate in `x2`

DOUBLE MIDDLE
middle mouse button has been double-clicked in window with indicated ID
x-coordinate, in user coordinates, returned in `x1`; y-coordinate in `x2`

DOUBLE RIGHT
right mouse button has been double-clicked in window with indicated ID
x-coordinate, in user coordinates, returned in `x1`; y-coordinate in `x2`

DOWN
arrow at bottom of scroll bar in window with indicated ID has been clicked
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

END HSCROLL
slider in a horizontal scroll bar is no longer being moved or selected by the mouse

END VSCROLL
slider in a vertical scroll bar is no longer being moved or selected by the mouse

EXTEND
left (or only) mouse button has been shift-clicked in window with indicated ID
x-coordinate, in user coordinates, returned in `x1`; y-coordinate in `x2`

EXTEND MIDDLE
middle mouse button has been shift-clicked in window with indicated ID
x-coordinate, in user coordinates, returned in `x1`; y-coordinate in `x2`

EXTEND RIGHT
right mouse button has been shift-clicked in window with indicated ID
x-coordinate, in user coordinates, returned in `x1`; y-coordinate in `x2`

HIDE
window with indicated ID has been closed by user

HSCROLL
slider in horizontal scroll bar in window with indicated ID has been dragged
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

KEYPRESS
key has been pressed in window with indicated ID
numeric key code returned in `x1`

LEFT
arrow at left end of scroll bar in window with indicated ID has been clicked
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

MENU
menu item in window with indicated ID has been selected
menu item ID will be returned in `x2`

PAGEDOWN
area below slider in scroll bar in window with indicated ID has been clicked
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

PAGELEFT
area left of slider in scroll bar in window with indicated ID has been clicked
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

PAGERIGHT
area right of slider in scroll bar in window with indicated ID has been clicked
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

PAGEUP
area above slider in scroll bar in window with indicated ID has been clicked
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

REFRESH
contents of window with indicated ID need to be redrawn
will be handled automatically for immune windows

RIGHT
arrow at right end of scroll bar in window with indicated ID has been clicked
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

SELECT
window with indicated ID has gained input focus or has been opened

SINGLE
left (or only) mouse button has been clicked once in window with indicated ID
x-coordinate, in user coordinates, returned in `x1`; y-coordinate in `x2`

SINGLE MIDDLE
middle mouse button has been clicked once in window with indicated ID
x-coordinate, in user coordinates, returned in `x1`; y-coordinate in `x2`

SINGLE RIGHT
right mouse button has been clicked once in window with indicated ID
x-coordinate, in user coordinates, returned in `x1`; y-coordinate in `x2`

SIZE
window with indicated ID has been resized or opened

TXE HSCROLL
user is scrolling text horizontally by holding down mouse near the left or right of the text edit control

TXE KEYPRESS
key has been pressed within text editor in window with indicated ID
numeric key code returned in `x1`
text editor ID will be returned in `x2`
only generated when text editor's KEY EVENTS or TRAP CHAR attributes set

TXE MOUSE
mouse has been clicked within text editor in window with indicated ID
text editor ID will be returned in `x2`
only generated when text editor's MOUSE EVENTS attributes set

TXE VSCROLL
user is scrolling text vertically by holding down mouse near the top or bottom of the text edit control

UP
arrow at top of scroll bar in window with indicated ID has been clicked
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

VSCROLL
slider in vertical scroll bar in window with indicated ID has been dragged
scroll bar ID will be returned in `x2`; value of 0 returned if scroll bar attached

APPENDIX

J

Debugging and Correcting Errors

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.

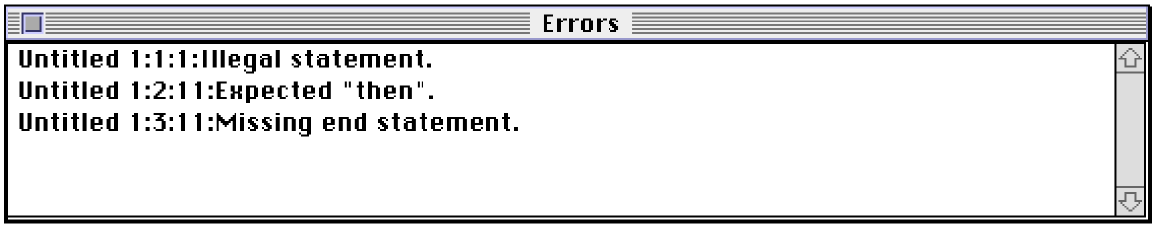
Illegal Statements

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 C 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 C 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 and the EXLINE, EXLINE$, EXTEXT$, and EXTYPE functions are explained in Chapter 16.

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.
- Next, get a printed listing of your program and read it carefully.
- As you read, check your variable names. Have you spelled them correctly and consistently throughout the program? The OPTION TYPO and LOCAL statements described below can help you catch spelling errors in variable names.

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 10 tells more about 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. The first step is to turn debugging on by selecting Debug Mode 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.

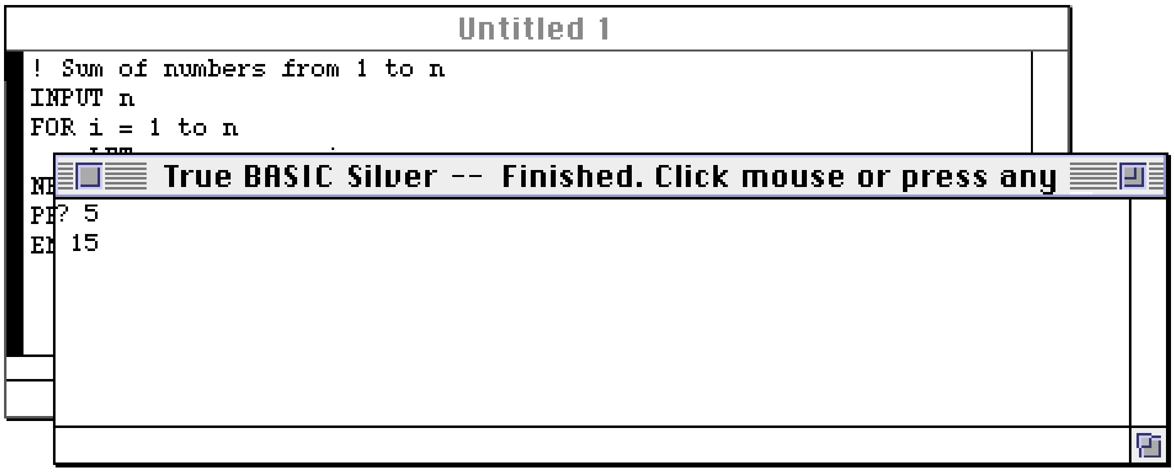
If you accidentally close the Variable window, you can reopen it by selecting it from the Windows menu of Editing window.

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.)



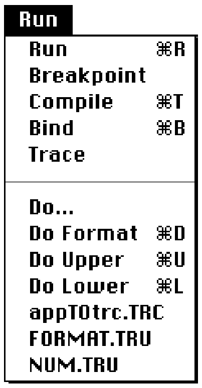
```
! Sum of numbers from 1 to n
INPUT n
FOR i = 1 to n
```

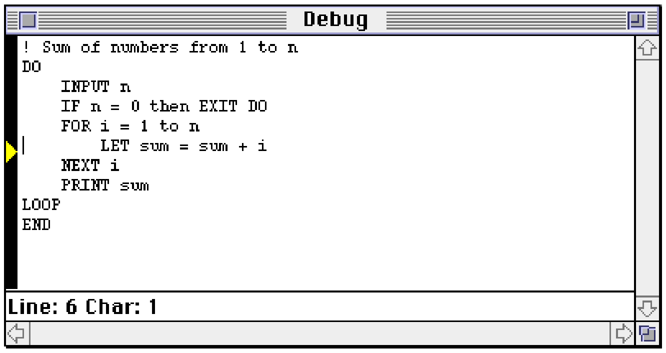
```
? 5
 15
```

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.

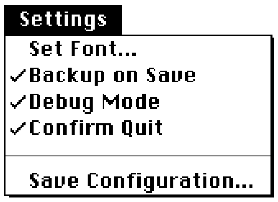




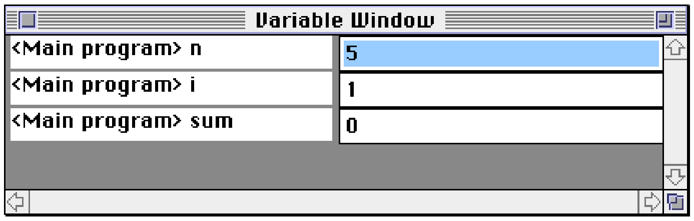
```
! Sum of numbers from 1 to n
DO
    INPUT n
    IF n = 0 then EXIT DO
    FOR i = 1 to n
        LET sum = sum + i
    NEXT i
    PRINT sum
LOOP
END
```

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.



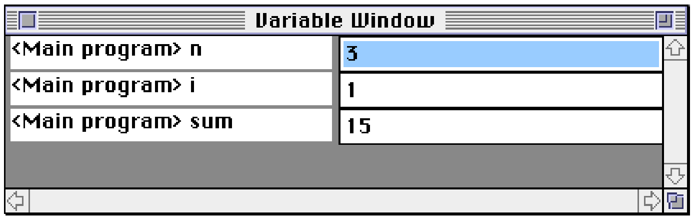
Make sure Debug Mode is checked in the Settings menu. Now place the cursor in front of the line 'LET sum = sum + 1', which is the workhorse line in the program. Next, choose Breakpoint from the Run Menu. Now choose Run from the Run menu. The program will stop almost immediately at the breakpoint.



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

The Variable Window will look like this. Everything looks okay. Continue the program by selecting Continue from the Run of the Variable Window, or by typing 'continue' in the command line, until it prints the result 15, in the Output Window.

Now, enter 3 when the '?' appears. Notice the current status of the Variable Window.



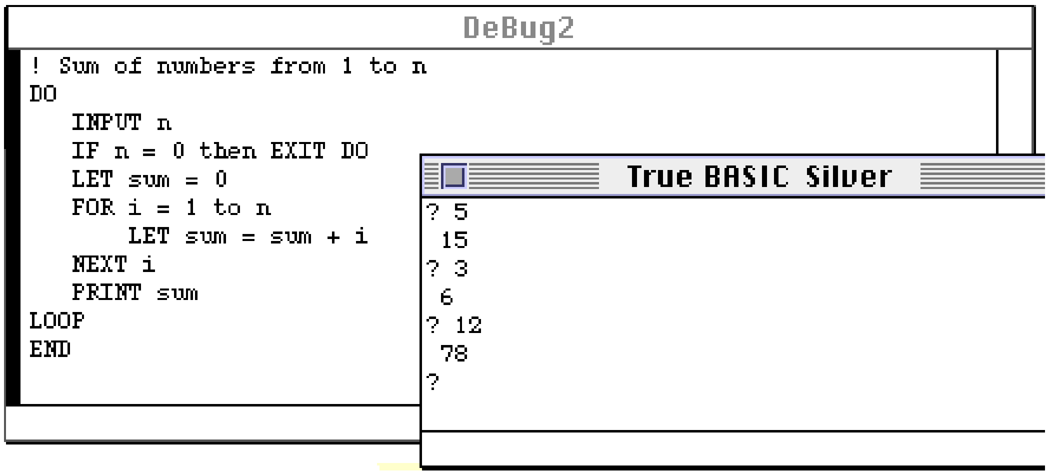
<Main program> n	3
<Main program> i	1
<Main program> sum	15

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

K

Features for Advanced Programmers

LOADED WORKSPACES

True BASIC provides a LOAD command that allows you to extend the True BASIC language. You may load any library or module and the procedures of the library or module will, in effect, be added to the language. This allows both strengthening the language and customizing for a particular use. To accomplish this:.

- Write the procedures in True BASIC.
- Save them in a LIBRARY file.
- Use the LOAD command to bring that library into the language.
- Use STORE/RESTORE to save and restore workspaces that you have loaded, on demand, or as part of your Language System STARTUP.TRU file.

At this point, your functions, subroutines, and pictures behave exactly as if they were built into True BASIC.

See Chapter 11 for more information on writing your own libraries and modules.

Loading

Loading is a means of customizing the language to your particular needs. The following example illustrates how loading works and how useful it can be. Suppose you have a library called STRLIB.TRU, which contains a number of useful string routines. One of these routines produces a nicely formatted date:

```
DEF Nicedate$(d$)
    WHEN error in
         FOR i = 1 to Val(d$[5:6])
             READ month$
             DATA January,February,March,April,May,June,July,
             DATA August,September,October,November,December
         NEXT i

         LET day$ = d$[7:8]
         IF day$[1:1] = "0" then LET day$[1:1] = ""
         LET Nicedate$ = month$ & " " & day$ & ", " & d$[1:4]
    USE
         CAUSE ERROR 1,"Bad date given to Nicedate$: " & d$
    END WHEN
END DEF
```

While the built-in function *Date$* produces a date of the form “19881026”, `NiceDate$ (date$)` produces “October 26, 1988”. One way of using this is to include a LIBRARY statement in the main program as well as DECLARE DEF *NiceDate$*. But there is a simpler and more efficient method.

You may type the command:

```
LOAD STRLIB.TRU
```

which loads all the routines of STRLIB into memory, and, in effect, makes them part of the True BASIC language. Any program may then contain a statement:

```
PRINT NiceDate$(date$)
```

without requiring a LIBRARY or DECLARE DEF statement. One may even type the above as a direct-mode command and have the date printed.

Avoiding declarations is, of course, a convenience. But loading provides a much greater benefit. Programs that use large libraries must wait until those libraries are brought into memory and all the appropriate linkages are performed. If, instead, the libraries were loaded, they stay in memory and the linkages are already performed. Thus the main program will begin to run much sooner.

As another example, suppose that you load the library MATHLIB.TRU from the TBLIBS directory. True BASIC now knows all the trigonometric functions, not just the ones built into the language. In addition to SIN, COS, TAN, and ATN, you may now use SECH and ASEC (hyperbolic secant and arcsecant) just as if they had been built-in.

If you have a compiled version of the library available, it will load faster. But once it is loaded, it makes no difference which version you used for loading, since loading compiles your library if necessary.

Loading a Module

Like any other library, a module may be loaded into memory with a LOAD command. All the routines in the module are rapidly available to any program, without the need for LIBRARY or DECLARE DEF statements. If the module defines a data structure, loading it makes the data structure "part of the language."

If one or more modules have been loaded, the order of initialization is: loaded modules are initialized first, then those being used from libraries, and then any in your current file. While modules in libraries are initialized only if they are to be used, all loaded modules are initialized.

Thought must be given as to what to include in the initialization of a module if the module is likely to be loaded. While automatic initialization of modules has great advantages, it can be confusing if a loaded module clears the screen or performs a graphics task as part of initialization. Since the module will be initialized before every run, the graphics task would be carried out even if the main program has nothing to do with graphics.

More on Loading

A library or module can be loaded only if there are no "loose ends" left. Doing this is easy because several libraries may be loaded with a single LOAD command:

```
LOAD STRLIB.TRC, MATHLIB.TRC
```

As long as all calls to subroutines and references to functions are resolved among the collection of libraries, the load will succeed.

Loaded libraries may be removed from memory by a FORGET command. This frees up the memory for a new LOAD or for any other use. If a second LOAD command is issued, without a FORGET, the new libraries and modules are added to those already loaded.

In contrast with LIBRARIES, you are not allowed to have duplicate procedure names in the library files you name in the LOAD statement. The reason is that all procedures in the libraries named in a LOAD statement are loaded; with libraries named in LIBRARY statements, only the needed procedures are loaded – once the first of a duplicate procedure is found, no notice is taken of the second.

Frequently used LOAD and RESTORE commands may be saved in a ***script file***. Or they may be in the startup file, which is already a script file and is automatically executed when True BASIC is first invoked.

Once your personal workspaces are defined and created, two commands, entered from the Command Window, are used:

STORE

The STORE command is used to store the current workspace, created with previous RESTORE or LOAD statements, to a disk file. This disk file can later be used in the RESTORE command. The format of the command is:

```
STORE mywspace
```

If you do not specify a file name, True BASIC will display the message "**Trouble using disk or printer**". If you specify a file name which already exists, True BASIC will ask you if you wish to overwrite it.

The STORE command saves the workspace as is, including all restored workspaces, loaded libraries, links between libraries, and assembly language subroutines. These will all be restored as is by issuing the command RESTORE with the same filename. The files created by the STORE command are much easier and faster to load than individual files using the LOAD command.

RESTORE

The RESTORE command restores a workspace from the specified workspace file created using the STORE command *(see the* STORE *command above)*. The format of the command is:

```
RESTORE mywspace
```

If you do not specify a filename, True BASIC will display the message "**Trouble using disk or printer**". If you specify the name of a file which is not a workspace, True BASIC will abort the RESTORE and issue an error message, "***xyz* is not a stored file**".

The RESTORE command first erases any existing workspace in memory created with LOAD or RESTORE commands and replaces it with the workspace in the file specifed.

This is a much faster method of loading a workspace than using the LOAD command. The libraries and subroutines which were used to create the workspace which was stored need not be available for the RESTORE command.

PROGRAMMER'S WORKBENCH

The True BASIC WorkBench provides tools to assist in the development of large programs, and to perform other common tasks that may otherwise require leaving the True BASIC Language System or writing small utility programs.

Its tools fall into five categories:

Source Code Control System

Search and Find

Other Programming Aids

Operations on the Current File

Utilities

Most of the tools are modeled on similar ones found in the Unix operating system. Since such facilities are abundant there, the TB WorkBench is not available for Unix versions of True BASIC. Another reason is that they are invoked by the True BASIC DO command, which is available only on personal computers. All commands are described by so-called **man** pages, which are also available on line.

[!] **WARNING: The use of these tools requires, in most cases, a thorough understanding of the file and pathname conventions on your operating system and the concept of aliases in True BASIC, as well as a willingness to learn the command syntax, and to accept very brief error messages.**

To use these tools you must also understand the use of DO commands within the True BASIC environment. Each command should be entered as the the name of a DO program.

Source Code Control System

The SCCS tools include **create**, **get**, **delta**, **checksum**, **header**, and **unget**.

The True BASIC Source Code Control System (SCCS) allows keeping different versions of a large body of source code without keeping full copies of each version. Only the “deltas”, or changes since the last update, are retained. The logic and general structure is based on the Unix SCCS utility.

To set up your SCCS, create a directory in which you wish to store the source code. This directory need not be the same for all projects, as each of your project directories will contain a pointer to the SCCS directory for that project.

[!] **WARNING: The SCCS directories are not protected (read only), and should not be used in shared environments.**

The fundamental tools are: **create**, **get**, **delta**.

Subsidiary tools are: **checksum**, **header**, and **unget**.

For each source code file in your working directory that you wish to control, make that file your “current” file, and use the **create** tool. It allows you to set the SCCS directory name, and to specify the new release number. The name of the SCCS directory is stored as the only entry in a small text file named “sccsprfx” in your project’s working directory.

Use the **get** tool to get a copy of a particular source code file for examination or modification. If you wish a version that is not the latest version, you can specify the version number. If you wish to make changes to the file for a later **delta** operation, you must so specify with an option on the **get**.

After making and testing changes, you can create a new version by using the **delta** tool. Only the changes (deletions and additions) are kept; the source code that is not changed is left alone. There are no options on the delta command.

There are several subsidiary operations that you can perform. The **checksum** tool recomputes the checksum of an SCCS file and compares it with the stored checksum. You may be able to detect file damage in this way. The use of this tool does not change the checksum in the stored SCCS file.

The **header** tool will print the header information for the files named. You can thereby review the history of the versions, with dates and reasons.

The **unget** tool can be used to negate a previous “get with edit permission” command, thus prohibiting a subsequent **delta**.

Search and Find

The Search and Find tools include **find**, **findsub**, **plan**, and **xref**.

The True BASIC Language System includes a "find" command that can locate any string of characters in the "current" file. The WorkBench **find** tool can search for a string of characters in several files, all files in a certain directory, and can even examine all subdirectories. (Similar to the Unix **grep** command, **find** does not include regular string expressions, but does climb directories.) **Find** allows searching for whole words, and allows searching with or without regard to case (upper- and lowercase.) Displayed are the lines containing a match of the search string, the ordinal number of each line in the file, and the file name. Lines longer than the display screen are folded. Search strings that contain spaces or other special characters must be quoted.

Related to the **find** tool are two others: **findsub** and **plan**. **Findsub** locates all procedure definitions in the files named. A procedure is a True BASIC subroutine, function or def, or picture. The search can be limited to just one of the three types, or any two of the three types, or all three. Like **find**, **findsub** can examine any or all files in the specified directory, and possibly all subdirectories.

The **plan** tool examines each file named and produces a list of the locations of all procedure declarations and definitions. Also included are those procedures that appear in PRIVATE statements, and the last use of the procedure in the file. It is useful, for example, for identifying procedures declared or defined but never used in that file.

Finally, the **xref** tool produces a traditional cross-reference of all the keywords and variable names in a file.

Other Programming Aids

Other Programming Aids include the **diff**, **format**, **tbpp**, **make**, **show** and **dhex** tools.

The **diff** tool compares two text files, or the current file against another text file, and reports all lines that are different.

The **format** tool "formats" the True BASIC source code in the current file. Similar to the command DO FORMAT of the regular True BASIC Language System, this tool provides additional capabilities. It permits specifying the indentation column for on-line comments, and allows bypassing capitalization of certain key words.

The **tbpp** tool invokes a preprocessor similar to the Unix cpp or m4. It permits including or excluding blocks of code based on whether certain words are defined or not, or on other simple conditions. It also provides for "include files," something not provided by the True BASIC Language System.

The **make** tool checks the dates and times of the most recent changes to the files specified in the makefile, and constructs a script for bringing the dependent files up to date. The usual use is to make sure that compiled versions bear a date and time stamp that is more recent than that of the corresponding source file, but other uses are possible. (Since a DO program cannot issue True BASIC commands, **make** produces a script file which can later be invoked by the SCRIPT command. Thus, the **make** tool is simpler and less powerful than the Unix make, but nonetheless may be useful for very large projects.)

The **show** tool allows displaying a text file in the output window, starting at the line you specify.

The **dhex** tool provides a traditional interactive hex dump of any file.

Operations on the Current File

The WorkBench provides several useful operations on the current file. The **format** tool was mentioned previously.

The **sort** tool sorts the current file. The default is sorting the lines of the file using the ASCII code sequence. One option permits sorting in dictionary order, where "a" follows "A" but precedes "B." Another permits using a particular substring (like the fifth through the tenth character) as the sort key. The sort is not stable; that is, the original order of ties is not necessarily maintained.

The **column** tool arranges the words in the current file into a single column, or produces a single column arrangement of the file names in a directory.

The **mkscr** tool allows constructing a script file from a simple columnar list of words in the current file. (The abbreviaton "mkscr" stands for "make script".) As an example, to list all the files in your current directory, use these four commands:

```
    do column,-d
    do mkscr,"old ?; list"
    save doit
    script doit
```

The **keep** tool retains all lines in the current file that contain the key string you specify.

The **omit** tool throws away all lines in the current file that contain the key string you specify.

Utilities

Three other tools are included.

The **man** tool lists the man page or pages for the tool or tools you specify. To list all the topics for which man pages are available, use

```
    do man, topics
```

The **mkdir** tool creates a new directory, a function that is not included in the regular True BASIC Language System.

The **dir** tool lists the files in a particular directory in a particular directory along with their sizes, and dates and times last modified.

Output

Output on all commands is normally directed to the screen in pages. Pressing any key advances to the next page. Pressing the ESC-key terminates the tool.

Output can be directed to a file instead by using a single ">" followed by an output file name at the end of the DO command. Output can be directed to the screen and to a file or the printer by using a double ">>" at the end of the command line. This second option is a regular True BASIC feature and is available for all True BASIC typed commands. For example,

```
    do find, zilch . > outfile
```

will send the output from the find command to the file named "outfile." New information will be appended at the end. (To start with an empty file, you must first unsave it or remove its contents.)

```
    do find, zilch . >> outfile
```

will send the output both to the screen and to the file named. Again, new information will be appended. (The use of the double ">>" is a True BASIC Language System convention on personal computers. It merely sends all information that appears on the screen to the file named.)

```
    do find, zilch . > outfile1 >> outfile2
```

will send the output from the find tool to "outfile1," but practically nothing to "outfile2," since the information has been sent to the "outfile1" rather than to the screen. Of course,

```
    do find, zilch .
```

will send the information to the screen only.

Aborting a Tool

Aborting a tool before its activity has been completed can be done by selecting STOP in the File menu of the Command Window. To abort during screen output, use the ESCAPE-key.

Public Names

If you load the workbench routines into your workspace, certain workbench subroutines are publicly available.

You should avoid using their names for other purposes. All the other workbench subroutines are private. These subroutines are the actual tools.

WB_checksum	WB_findsub	WB_mkscr
WB_column	WB_format	WB_omit
WB_create	WB_get	WB_plan
WB_delta	WB_header	WB_show
WB_dhex	WB_keep	WB_sort
WB_diff	WB_make	WB_tbpp
WB_dir	WB_man	WB_unget
WB_find	WB_mkdir	WB_xref

These subroutines are support utilities:

WB_AskInputFileType	WB_Error	WB_MakeLegalDir
WB_CheckDir	WB_GetEOL$	WB_More
WB_Decode	WB_KeyMouse	

Using the Tools

All of the tools are invoked with the True BASIC Language System DO command. Many DO commands are designed to operate on the "current" file in the editing window; an example is DO FORMAT. But DO commands are more general in that they can invoke a complete program without leaving the editor. Many of the Work-Bench tools perform tasks not related to the current file, such as making a new directory (DO MKDIR).

The True BASIC Language System (referred to as TBLS) looks for the DO command code in one of several directories or folders, which are specified with the TBLS ALIAS command. In addition, the WorkBench requires two additional aliases, one for its compiled code and one for the online manual.

Setting up the workbench tools for use requires (a) the proper aliases, and (b) loading the workbench utilities into your workspace. (Actually, loading is not necessary, but it saves time and is recommended.)

Remember, on both Windows systems and the MacOS, True BASIC is insensitive to case, UPPER- versus lower-case. The same is true for the Workbench commands. Thus, the following two commands are the same:

```
do finds,-c searchstring wbfinds
DO FINDS,-C SearchString WBfinds
```

The option "-c" guarantees that the search string will match any string in the file that contains the same letter sequence, regardless of case.

W Installing in the Windows Environment

In True BASIC's home directory there is a subdirectory wrkbench. Enter it and bring up the file loadalia. You'll need to change the entries to correspond to the name of True BASIC's home directory and its location on you disk. For example, it might be

```
alias {do},    C:\TB\WrkBench\Tools\, ""
alias {WB_c},  C:\TB\WrkBench\C\
alias {WB_man},C:\TB\WrkBench\Man\
```

Suppose the home directory of True BASIC is

```
 D:\TRUBASIC
```

You'll want to change the loadalia file to look like this:

```
alias {do},    D:\TRUBASIC\WrkBench\Tools\, ""
alias {WB_c},  D:\TRUBASIC\WrkBench\C\
alias {WB_man},D:\TRUBASIC\WrkBench\Man\
```

(You can also add these new aliases to the STARTUP.TRU file in the home directory of True BASIC.)

Next, type

```
script loadall
```

This plays a dual role of exercising the new aliases, and also loading the workbench as a workspace.

You can now save this workspace for later use by typing

```
store myws
```

using a file name of your choice.

Next, you can create a new script file named, for example, myload

```
forget
scr loadalia
restore myws
```

and save it.

Finally, add the following lines to the STARTUP.TRU file in the home directory:

```
cd wrkbench
scr myload
```

Now, whenever you start True BASIC, your workbench workspace will be loaded and all aliases will be established.

Installing on the MacOS

The process is essentially the same. The only major difference is the file name convention.

In True BASIC's home directory there is a subdirectory wrkbench. Enter it and bring up the file loadalias. You'll need to change the entries to correspond to the name of True BASIC's home directory and its location on you disk. For example, it might be

```
alias {do},    HD:TB:WrkBench:Tools:, ""
alias {WB_c},  HD:TB:WrkBench:C:
alias {WB_man},HD:TB:WrkBench:Man:
```

Suppose the home directory of True BASIC is

```
MyHD:True BASIC:
```

You'll want to change the loadalia file to look like this:

```
alias {do},    MyHD:True BASIC:WrkBench:Tools:, ""
alias {WB_c},  MyHD:True BASIC:WrkBench:C:
alias {WB_man},MyHD:True BASIC:WrkBench:Man:
```

(You can also add these new aliases to the STARTUP.TRU file in the home directory of True BASIC.)

Next, type

```
script loadall
```

This plays a dual role of exercising the new aliases, and also loading the workbench as a workspace.

You can now save this workspace for later use by typing

```
store myws
```

using a file name of your choice.

Next, you can create a new script file named, for example, myload

```
forget
scr loadalia
restore myws
```

and save it.

Finally, add the following lines to the STARTUP.TRU file in the home directory:

```
cd :wrkbench
scr myload
```

Now, whenever you start True BASIC, your workbench workspace will be loaded and all aliases will be established.

Additional Details

If you prefer to have a smaller loaded workspace in order to, say, leave more room in main memory for your program, you can remove the unneeded file names from the file LOADALL.TRU (loadall). For example, if you do not plan to use the SCCS routines, you do not need to have WBSCCS.TRC loaded. The dependency of the tools on the WBXXXX.TRC files is as follows:

WBUTILIT.TRC	all tools except the man tool
WBSCCS.TRC	checksum, create, delta, get, header, unget
WBCOLUMN.TRC	column
WBDHEX.TRC	dhex

WBDIFF.TRC	diff
WBDIR.TRC	dir
WBFINDS.TRC	find, findsub, plan, show
WBFORMAT.TRC	format
WBKEEP.TRC	keep, omit
WBMAKE.TRC	make
WBMKDIR.TRC	mkdir
WBSORT.TRC	sort
WBTBPP.TRC	tbpp
WBXREF.TRC	xref

The mkscr tool is located in the WBUTILIT.TRC file. The man tool is self-contained.

If you have loaded all the tools, then the WB_C alias is no longer needed. If you never plan to use the man tool, then the WB_MAN alias is not needed.

The tools are designed so they can be used without being preloaded. Just make sure that the WB_C alias is properly set. The advantage of not preloading is that it saves main memory. The disadvantage is that the startup time for some of the tools will be noticeably long.

See the README.TRU (ReadMe) file in the main workbench directory (folder) for more information and changes since the date of this documentation.

checksum

NAME **checksum** — compares the checksums of SCCS files

USAGE
```
    do checksum, filename..filename
```

DOES Reads the checksum of an SCCS file, recomputes the checksum, and prints both.

OPTIONS None.

EXAMPLE Check and report the checksums in the named SCCS files:
```
    do checksum, main subs utilities
```

RELATED TOOLS **create**, **delta**

column

NAME **column** — produces a single-column directory list

USAGE
```
do column [,[-d[DIR]] [-f] [-tTemp]
```

DOES Arranges “words” into a single column, using spaces as the delimiters. Useful for constructing a single-column list of file names, for example.

OPTIONS

`-dDIR` If present, then forms a single column of the filenames in the specified directory, denoted by DIR,using the complete path names.

`-d` If -d without DIR is present, then forms a single column of the filenames in the current directory, using the short names. If -d is absent, then forms a single column of the words in the current file. In all cases, the result replaces the current file.

`-f` If present, forms a column of filenames only (omitting directories on DOS systems.

`-tTEMP` If present, uses TEMP as a template for selecting file names. If missing, includes all file names. An asterisk in TEMP matches any character or characters. Thus, to match file names ending with ".tru", use "-t*.tru". To match file names that start with "WB", use "-tWB*".

LIMITATIONS Make sure the edit (current file) window does not contain a "(compiled file)".
Will not include folder names on the Macintosh; thus, the -f option produces no result on the Macintosh.

EXAMPLE To list all True BASIC text files in the current directory on DOS systems:

```
do column, -d -f -t.tru
do mkscr, "old ?; list"
save doit; script doit
```

create

NAME **create** — creates an SCCS file from a source file

USAGE `do create, [-dDIR] [-rSID] filename`

DOES Creates an SCCS file from a source file.

OPTIONS

`-dDIR` If present, sets the SCCS directory to DIR, and changes the SCCS directory name saved in the canonical file "sccsprfx" in the current directory. If absent, retains the SCCS directory name from the file "sccsprfx."

`-rSID` If present, sets the new release number to SID. If absent, the new release number is 1.1.

LIMITATIONS Enter the user name as a single word or a quoted string; otherwise all characters after a non-leading blank will be ignored.

EXAMPLE To create a new SCCS entry with a new SCCS directory:

```
    do create, -dSCCSDIR newfile
```

The name of the new SCCS directory, 'SCCSDIR', may contain a full pathname and must conform to the directory naming convention of your operating system.

To create a new SCCS entry with a specified release number:

```
    do create, -r2.1 newfile
```

RELATED TOOLS **checksum, delta, get, header, unget**

delta

NAME **delta** — modifies an SCCS file from a current file

USAGE `do delta, filename`

DOES Modifies an SCCS file from a current file. The file must have been obtained by a get command with the -e option.

OPTIONS None.

LIMITATIONS None.

EXAMPLE To do a delta:

```
    do delta, filename
```

RELATED TOOLS **Get, unget**

dhex

NAME **dhex** — provides hex dumps

USAGE `do dhex, filename`

DOES Provides a hex dump of any portion of the file. The user will be asked for an address range in decimal, which should be provided in the form

```
first-byte, last-byte
```

The first byte in the file is numbered 0.

OPTIONS None.

LIMITATIONS None.

diff

NAME **diff** — identifies differences between text files

USAGE `do diff, file1 [file2]`

DOES Determines the differences between two files, one of which can be the current file.

OPTIONS If file2 is absent, compares the current file with file1. If file2 is present, compares file1 with file2.

LIMITATIONS Works only with text files consisting of lines that end with EOLs.
(May not work if last line does not end with an EOL.)

EXAMPLES To compare the current file with the file oldf, output going to the screen:

```
do diff, oldf
```

To compare file1 with file2, output going only to the file outfile:

```
do diff, file1 file2 > outfile
```

To compare the current file with file xyz, output going to both the screen and the printer:

```
do diff, xyz >>
```

dir

NAME **dir** — prints directory information

USAGE `do dir [,[-dDIR] [-tTEMP]]`

DOES Prints a directory showing file names, sizes, date last modified, and time last modified.

OPTIONS

`-dDIR` If present, then prints information for the directory DIR. If missing, prints information for the current directory.

`-tTEMP` If present, uses TEMP as a template for selecting file names.

If missing, includes all file names. An asterisk in TEMP matches any character or characters. Thus, to match file names ending with ".tru", use "-t*.tru". To match file names that start with "WB", use "-tWB*".

LIMITATIONS	Will not show folders on the Macintosh.
EXAMPLES	To list the current directory:

```
do dir
```

To list the directory HD:foo

```
do dir, -dHD:foo
```

To list only files whose names begin with "x" in the current directory:

```
do dir, -tx*
```

find

NAME **find** — searches for a string in a file(s) or directory

USAGE `do find, [-c] [-w] [-d] searchstr [files] [dir]`

DOES Searches one or more files, or a directory, for a search string. Prints the file name, the line number, and the line itself that contains the string.

OPTIONS

`-c` If present, ignores case (lower vs. upper) in determining matches. If absent, treats lower and upper case as different.

`-w` If present, searches for the "whole word"; that is, does not report a match if the search string is contained within a larger string. If absent, ignore words.

`-d` If present, searches the directory named, if any, and all subdirectories, etc. If absent, searches only the directory named, if any.

`searchstr` The search string. If it contains spaces or quotes, it itself must be quoted. That is, to search for the string

```
he said "Hello"
```

use as the search string

```
"he said ""Hello"""
```

`files` The name(s) of a file(s) to be searched. The file will be examined in the directory named, if any. If no file is named, searches all files in the directory named.

`dir` Names the directory to be searched. The current directory is denoted with a "." (without the quotes). Directories are named using the standard pathname conventions for the system.

LIMITATIONS There must be at least one file or one directory named. To search all files in the current directory, use no file names, but use `"."` for the directory name.

NOTES On the MacOS, the current folder is denoted with a single `"."`. Subdirectories start with a `":"`. A trailing `":"` must also be included. Superdirectories are denoted with `"::"`, with an additional `":"` for each additional level. A pathname starting with a disk name must be preceded by a ".", to distinguish it from an ordinary file name.
Thus, if the directory structure is:

```
disk:main:task:source
```

and the current directory is "task", then each of the directories can be named as follows:

```
disk    :::    or    .disk:
main    ::     or    .disk:main:
task     .
source  :source:
```

On Windows systems, the current directory is denoted with a single ".". Subdirectories are denoted with a "..", with an additional "\.." for each additional level. A pathname starting with a disk name must start with a "." to distinguish it from an ordinary file name. Thus, if the directory structure is:

```
c:\main\task\source
```

and the durrent directory is “task”, then each of the directories can be named as follows:

```
disk    ..\..   or   .C:
main    ..      or   .C\main
task
source  .\source
```

RELATED TOOLS **findsub**

findsub

NAME **findsub** — searches for sub (function, def, picture) lines in a file(s) or directory

USAGE `do findsub, [-s] [-f] [-p] [-d] [files] [dir]`

DOES Searches one or more files, or a directory, for a search string. Prints the file name, the line number, and the line itself that contains the sub (function, def, picture) statement.

OPTIONS

`-s` If present, searches for subroutine definitions.

`-f` If present, searches for function or def definitions.

`-p` If present, searches for picture definitions.
Note: if all three are absent, searches for all three types of subroutines. i.e., all three missing is the same as all three present.

`-d` If present, searches the directory named, if any, and all subdirectories, etc. If absent, searches only the directory named, if any.

`files` The name(s) of a file(s) to be searched. The file will be examined in the directory named, if any. If no file is named, searches all files in the directory named.

`dir` Names the directory to be searched. The current directory is denoted with a "." (without the quotes). Directories are named using the standard pathname conventions for the system.

LIMITATIONS There must be at least one file or one directory named. To search all files in the current directory, use no file names, but use "." for the directory name.

NOTES See **find**, for a discussion of DIR conventions.

RELATED TOOLS **find**

format

NAME	**format** — indents and capitalizes a source file
USAGE	`do format [, [-n] [-idd]]`
DOES	Indents the current file to show the structure. Capitalizes certain leading key words. If the first line contains a "!", the comment is replaced with the current date.
OPTIONS	
`-n`	Does not capitalize key words, but leaves them in their original form.
`-idd`	Sets the standard indentation for online comments to dd. If this option is absent, uses dd = 35.
LIMITATIONS	Uses a standard indentation, which can be changed by changing the data near the start of the subroutine.

get

NAME	**get** — gets an SCCS file
USAGE	`do get, [-rSID] [-e] filename`
DOES	Gets an SCCS file and both makes it the current file and saves it in the current directory.
OPTIONS	
`-rSID`	If present, gets the version that matches SID. If SID does not match, defaults to the latest SID. If absent, defaults to the latest SID.
`-e`	If present, gets the SCCS file with edit (delta) permission. It creates a "p." file in the SCCS directory containing key information about the update. This file must be present to permit a later delta. This file may be removed by the unget command.
LIMITATIONS	If the -e and -r options are both present, the SID must match the most recent SID in the file. In other words, branching of versions is not allowed. With the `-e` option, enter a single word as the user name; all characters after a non-leading blank will be ignored.
RELATED TOOLS	**create, delta, unget**

header

NAME	**header** — displays the header(s) on SCCS file(s)
USAGE	`do header, files`
DOES	Prints the header(s) of one or more SCCS file(s). Use the local names of the files, not the full SCCS name; i.e., no ".s".
OPTIONS	None.
LIMITATIONS	None.
RELATED TOOLS	**create, get, delta**

keep

NAME	**keep** — keeps certain lines in the current file
USAGE	`do keep, [-c] search$`

DOES	Keep all lines in the current file that contain the search string.
OPTIONS	
`-c`	If present, matches case (lower vs. upper) in determining matches. If absent, treats lower and upper case as the same.
LIMITATIONS	None.

make

NAME **make** — prepares a script file for make

USAGE `do make [, file]`

DOES Imitates the Unix make command by replacing the current file with a script file containing the necessary commands. Also saves the file with the canonical name 'makeit'. Carry out the script file by typing 'script makeit', or by first saving it with another name and then using the script command.

OPTIONS If the file is present, uses it as the make file. If absent, assumes the make file is named "makefile."

RULES Each line in the make file 'makefile' is a command to the make system.

There are several types of lines:

define
dependency
system commands
comment lines

Define lines have the following form:

```
   identifier   =   definition
```

As an example, if your source files are all in a directory :source:, then you can provide an abbreviation with

```
   s        =  :source:
   filelist =  Main Strings Numbers Booleans
```

Subsequently, use "{s}" (not quoted) in any pathname to refer to the directory containing the source files.

Dependency lines (typically) have the following form:

```
compiled-files   <=   source-files
```

A single file may be named in each, or a list of files.
Abbreviations of the {} type may be used. As an example,

```
{c}${filelist}*  <=  {s}${?}-s
```

assumes the source files in the directory referred to by {s} have the names in the list followed by "-s", and that the files in the directory referred to by {c} have the names in the list following by "*". Date and time stamps are checked. If updating is necessary, the subsequent indented commands are added to the script file with each "?" in the script file replaced by one of the names from the list of names. If recompilation is not necessary, the subsequent indented commands are ignored. If the compiled file is not present, it will be created. If the source file is not present, an error occurs. The dollar-sign ($) is required to indicate that the following definition is a list rather than a single item.

Subsequent indented commands can contain identifiers in braces, but question marks "?" should not be in braces.

Dependency lines need not involve source and compiled files. They can involve any files for which the date and time stamps are critical.

System command lines that are left-justified are added to the script file AS IS, and terminate any dependency of later-occurring indented command lines on a previous dependency line.

Comment lines are lines whose first nonblank character is a "!".
They are ignored completely.

Non-comment lines whose last nonblank character is a "&" are continued onto the next line.

LIMITATIONS Cannot output to a file, since it makes and saves a script file.

If the edit (current file) window contains "(compiled file)", the "makeit" file will be saved but not displayed in the edit window.

EXAMPLE The following make file has been used to recompile the workbench tools on the MacOS, and is included in the Workbench folder. (A similar file is in the Windows Workbench directory).

```
! We establish our compiled and source directories

c       = :c:
t       = :tools:
s       = :s:
cfilelist  = column dhex diff dir finds format make mkdir
               & sccs sort TBpp utilit xref

tfilelist  = checksum column create delta dhex diff dir
               find & findsub format get header make man
               mkdir mkscr & plan sort tbpp unget xref

! Here is our dependency, and the resulting conditional command

{c}WB${cfilelist}* <= {s}WB${?}
     cd {s}; old WB?; do tbpp,Mac; compile; ren WB?*; cd :{c};
     saverep; cd ::

{t}${tfilelist} <= {s}${?}-s
     cd {s}; old ?-s; do tbpp,Mac; compile; ren ?; cd :{t};
     saverep; cd ::
```

man

NAME **man** — prints manual pages

USAGE
```
do man, command..command
```

PURPOSE Prints the manual page for the command specified.

OPTIONS None.

LIMITATIONS Man pages exist for these commands only:

checksum, column, create, delta, dhex, diff, dir, find, findsub, format, get, header, make, man, mkdir, mkscr, output, plan, show, sort, tbpp, topics, unget, xref.

mkdir

NAME **mkdir** — creates new directories

USAGE
```
do mkdir, directory..directory
```

DOES Creates one or more new directories in the current directory.

OPTIONS None.

LIMITATIONS Does not work in non-hierarchical directories. Only one level of directory is allowed. Directory names must be legal.

mkscr

NAME **mkscr** — builds a script file from a list

USAGE
```
do mkscr, pattern
```

DOES Replaces the current file with a file in which each line consists of the pattern with the original line in the current file replacing each question-mark "?" in the pattern.

OPTIONS None.

LIMITATIONS If the pattern contains semicolons or ">>", or if leading and trailing spaces are required, then the entire pattern string must be enclosed in quote-marks.
Cannot direct output to a file.

EXAMPLES Assume that a line in the current file contains "master." Then

```
do mkscr,"old ?-s; compile; rep ?*"
```

will change that line to

```
"old master-s; compile; rep master*
```

Used in conjunction with the column command, this tool can construct a script file to list all files in the current directory.

```
do column, -d
do mkscr, "old ?; list"
save doit; script doit
```

omit

NAME	**omit** – omits certain lines in the current file
USAGE	`do omit, [-c] searchstr`
DOES	Omit all lines in the current file that DO NOT contain the search string.
OPTIONS	
`-c`	If present, matches case (lower vs. upper) in determining matches. If absent, treats lower and upper case as the same.
LIMITATIONS	None.

output

NAME	**output** — THIS IS NOT A TOOL
USEAGE	Serves only as a man page.
DOES	Nothing.
NOTES	As with all workbench commands, the output will normally be directed to the screen.

Other options:

> outfile
Output goes only to the designated file.

>> outfile
Output goes to the designated file as well as to the screen.

>>
Output goes to the attached printer as well as to the screen.

> outfile >>
Output goes to the designated file. Only the "Done with xxx" message goes to the printer.

>> outfile1 > outfile2
Not useful. The ">>" sends all output to the screen and to the file "outfile1 > outfile2", which may or may not be a legal file name.

plan

NAME	**plan** — locates routines in a file
USAGE	`do plan, filenames [dir]`
DOES	For each file, prepares a table showing all declarations, definitions, uses, and appearance in PRIVATE statements of subroutines, functions, and pictures.
OPTIONS	If dir is present, examines all the files in the directory named that match one of the filenames. If there are no filenames, examines all the files in the directory. If dir is not present, examines all the files in the current directory
LIMITATIONS	Will fail to identify all but the first uses of a defined function on the same line. Organizes procedures as encountered, not alphabetically.
NOTES	See **find,** for a discussion of DIR conventions.
BUGS	May fail to identify a name ending at a buffer boundary. May not exclude certain keywords in comments.

EXAMPLES

To construct a plan for a single file:

```
do plan, mainfile
```

To construct a plan for all files in the current directory, and direct output to a file only:

```
do plan, . > outfile
```

show

NAME	**show** — shows a text file, starting at the line designated
USAGE	`do show, [-lnnn] file`
DOES	Displays the text file in the output screen. Starts the display at the line designated. Use the ESC key to terminate the display.
OPTIONS	
`-lnnn`	If present, starts the display at line nnn. If absent, starts the display at line 1.
LIMITATIONS	None.
EXAMPLE	Use findsub to locate the file and starting line of the desired subroutine definition. Suppose the file is “subs” and the starting line is 120. Then use: `do show,-l120 subs`
RELATED TOOLS	**Findsub**

sort

NAME	**sort** — sorts the current file
USAGE	`do sort, [-d] [-sSSEXP]`
DOES	Sorts the current file.
OPTIONS	
`-d`	If present, uses the pseudo-dictionary order; i.e., AaBbCd... If absent, sorts according to the ASCII character sequence; i.e., ABC...abc...
`-sSSEXP`	If present, sorts using the substring given by the SSEXP. If absent, sorts the entire string. SSEXP must be of the form: `[f:t]` where f is the integer giving the “from” character, and t is the integer giving the “to” character. No spaces are allowed.

tbpp

NAME	**tbpp** — invokes the True BASIC preprocessor
USAGE	`do tbpp [, word .. word]`
DOES	Invokes the True BASIC Preprocessor on the current file. The preprocessor commands are based on the Unix cpp commands, and are also similar in function to several of the m4 commands. Words can be predefined in the command line, as well as with the #define command.

Commands are:

`#include filename`

Includes the contents of filename into the current file at that point. Filename may be unquoted or quoted.

`#define word [value]`

Defines the word, and if value is present, assigns that value to it. Words can be any contiguous sequence of letters and digits, and are converted to lowercase. Values are treated as strings without spaces, and are converted to lowercase.

`#undef word`

Removes the word from the defined list.

`#if expression`

Includes the subsequent lines of the current file if the expression is true. Omits the lines if the expression is false. Expressions can be of the form:

`word rel-op value`

where word is an identifier that has been previously defined in a #define, and value is a string that either relates or not to the value, if any, of the defined word. Rel-op is one of: <, <=, >, >=, =, <>.

`#ifdef word`

Includes the subsequent lines of the current file if word is defined. Omits the lines if it is not.

`#ifndef word`

Includes the subsequent lines of the current file if word is not defined. Omits the lines if it is.

`#elif expression`

Includes the subsequent lines of the current file if the expression is true, and if no previous lines in the same if-block have been included. Omits the lines otherwise.

`#else`

Includes the subsequent lines of the current file if no previous lines in the same if-block have been included. Omits the lines otherwise.

`#endif`

Ends the if-block

NOTES If-blocks may be nested. Include files can include other files.

OPTIONS Cannot direct output to a file, since `tbpp` works only with the current file.

LIMITATIONS The "#" must be in the first character position of the line. (Note: Do format automatically moves the # to the left margin if it is the initial nonblank character on a line.) Words "defined" in the command line have the null string as their value.

unget

NAME **unget —** removes the editing permission on an SCCS file

USAGE `do unget, files`

DOES Removes the editing permission on the SCCS file(s) corresponding to each of the files by removing the ".p" file in the SCCS directory.

OPTIONS None.

RELATED TOOLS **Get, delta**

xref

NAME **xref** — constructs a cross reference table

USE
```
do xref [, [-s] [files]
```

DOES Produces a cross reference table for all the numbers, keywords, variable names, etc., in the current file, or one or more saved files. Ignores REM statements, online comments, and quoted strings. Output is normally directed to the printer.

OPTIONS

`-s`	If present, uses a special sort so that "2" comes before "100"; this sort is slower. If missing uses an ASCII sort in which "2" comes after "100."
`files`	If present, constructs a cross-reference for one or more files. If missing, produces a cross-reference for the current file.

LIMITATIONS None.

EXAMPLE To do a cross-reference of a file and direct the output to another file (which is a good idea since the output is lengthy):
```
    do xref, filename > outfile
```

Index