

INTRODUCTION TO PROGRAMMING



BY BEN EZZELL
SECOND EDITION

Copyright © 1999, 2011 by Wilson WindowWare
All Rights Reserved

No part of this manual may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, for any purpose without the express written permission of Wilson WindowWare, Inc. Information in this document is subject to change without notice and does not represent a commitment by Wilson WindowWare, Inc.

The software described herein is furnished under a license agreement. It is against the law to copy this software under any circumstances except as provided by the license agreement.

TRADEMARKS

Microsoft and MS-DOS are registered trademarks of Microsoft Corporation.

Windows is a trademark of Microsoft Corporation.

WinBatch is a trademark of Wilson WindowWare, Inc.

TABLE OF CONTENTS

TABLE OF CONTENTS	I
INTRODUCTION : BECOMING A PROGRAMMER	XI
THERE WILL BE A TEST... ..	XI
CHAPTER 1 : THE GOLEM PRINCIPLE.....	13
BASICS OF PROGRAMMING	13
HOW COMPUTER APPLICATIONS WORK	14
LINGUA CYBER	14
THE SORCERER'S APPRENTICE.....	16
PRINCIPLES OF DESIGNING APPLICATIONS	19
Step One: Defining Objectives.....	19
Step Two: Defining Communications	20
Step Three: Defining Tasks	21
Step Four: Writing the Code.....	22
Step Five: Testing, Testing, and Retesting.....	22
Step Six: Getting Outside Opinions	23
Step Seven: Documenting.....	23
SUMMARY	24
CHAPTER 2 : THE PROGRAMMER'S WORKROOM	25
THE INTEGRATED DEVELOPMENT ENVIRONMENT	25
HOW TO USE THIS BOOK	25
COMPILED VERSUS INTERPRETED PROGRAMS	26
A BRIEF HISTORY OF THE IDE	27
An Introduction to the WinBatch IDE.....	28
A FIRST PROGRAM: HELLO, WORLD	29
WINBATCH STUDIO FEATURES AND TOOLS	32
File Operations	33
Cut, Copy, and Paste Operations.....	34
Undo and Redo Operations.....	34
Find, Find Next, and Replace Operations	34
Bookmark Operations	35
Tools.....	35
Debugging Tools.....	36
SUMMARY	38
CHAPTER 3 : DIALOGS AND THE DIALOG EDITOR.....	39
HOLDING A FORMAL CONVERSATION	39
AN INTRODUCTION TO THE DIALOG EDITOR	39
PREDEFINED DIALOGS	40
WIL DIALOG EDITOR	40
A SAMPLE DIALOG: WILDIALOG.WBT.....	44
DIALOG CONTROLS.....	48
Pushbuttons <PUSHBUTTON>	50
Radiobuttons <RADIOBUTTON>.....	51
Checkboxes <CHECKBOX>.....	53
Edit Boxes <EDITBOX>.....	54

Introduction to Programming

Static (Fixed) Text <STATICTEXT>	55
Variable Text <VARYTEXT>	55
Item (List) Boxes <ITEMBOX>	56
File List Boxes <FILELISTBOX>	57
Calendar <CALENDAR>	58
ComControl <COMCONTROL>	58
The DropList Box <DROPLISTBOX>	59
The GroupBox control <GROUPBOX>	60
The Spinner Control <SPINNER>	60
The Multi-Line Box <MULTILINEBOX>	60
The Picture Button Control <PICTUREBUTTON>	61
The Picture Control <PICTURE>	61
MENUS	61
TAB ORDER	62
SUMMARY	63
CHAPTER 4 : COMPUTER VOCABULARY – PART I	65
SIMPLE NOUNS – DATA TYPES AND VARIABLES	65
VARIABLES VERSUS CONSTANTS	66
WINBATCH DATA TYPES	66
Integer Constants	66
Floating-Point Constants	66
String Constants	67
Array	68
Huge Numbers	73
PREDEFINED CONSTANTS	73
Predefined String Constants	74
Predefined Floating-Point Constants	75
WINBATCH PROGRAM VARIABLES	75
VARIABLE NAMES	76
STRING VARIABLE CONVERSION	77
SUBSTITUTION	77
LISTS	79
KEYWORDS	79
SUMMARY	80
CHAPTER 5 : COMPUTER VOCABULARY – PART II	81
SIMPLE VERBS – OPERATORS AND OPERATIONS	81
MATH OPERATORS	82
GROUPING OPERATORS ()	82
THE ASSIGNMENT OPERATOR (=)	83
THE ADDITION AND SUBTRACTION OPERATORS (+ AND –)	84
THE MULTIPLICATION AND DIVISION OPERATORS (* AND /)	86
THE MODULUS OPERATOR (MOD)	87
THE EXPONENTIAL OPERATOR (**)	87
LOGICAL OPERATORS	88
The Logical AND Operator (& &)	88
The Logical OR Operator ()	88
The Logical NOT Operator (!)	89
RELATIONAL OPERATORS	89
The Equality and Inequality Operators (== and != or <>)	89
The Greater-Than and Less-Than Operators (>, >=, <, and <=)	90
BITWISE OPERATORS	90

The Left-Shift and Right-Shift Operators (<< and >>)	92
The Bitwise AND, OR, and XOR Operators (&, , and ^)	93
The Bitwise NOT Operator (~)	94
PRECEDENCE AND EVALUATION ORDER	94
COMMENTS	96
UNARY OPERATORS (VARIABLE REFERENCE OPERATORS)	96
BINARY STRING OPERATIONS	97
SUMMARY	97
CHAPTER 6 : COMPUTER VOCABULARY – PART III	99
STRINGS AND TEXT OPERATIONS	99
STRING-MANIPULATION FUNCTIONS	99
STRING-PARSING OPERATIONS	100
The ParseData Function	101
The StrScan and StrSub Functions.....	102
The ItemCount and ItemExtract Functions	104
Using the ArrayFileGet function.....	104
DIFFERENCES IN THE STRING-PARSING TECHNIQUES	105
SEARCH-AND-REPLACE OPERATIONS.....	105
The StrIndex and StrIndexNc Functions	105
The StrReplace Function	107
Selective Search and Replace.....	107
STRING-CONVERSION OPERATIONS	109
OTHER STRING CONVERSIONS	110
STRING-COMPARISON OPERATIONS.....	111
OTHER STRING OPERATIONS	113
The StrCharCount Function	113
The StrFill Function.....	113
The StrFix Functions	113
The StrTrim Function.....	114
Additional String Functions	114
LISTS AND LIST-SELECTION OPERATIONS.....	114
List Initialization	115
List Creation	116
List Display.....	119
List-Selection Handling.....	120
Lists of Lists	122
List Item Removal.....	124
PASSWORDS	124
KEYBOARD INPUT	125
SUMMARY	125
CHAPTER 7 : A TOOLKIT FOR OPERATIONS	127
FUNCTIONS AND SUBROUTINES	127
FUNCTIONS.....	127
User Defined Functions	129
SUBROUTINES	131
The gosub Statements.....	133
The Subroutines	134
Subprocedure Execution	137
EXTERNAL BATCH FILES	138
The First External Program.....	139
The Second External Program	141

Introduction to Programming

EXECUTABLE PROGRAMS	142
SUMMARY	143
CHAPTER 8 : GOING WITH THE FLOW.....	145
CONTROLLING OPERATIONS	145
BRANCHING AND PROGRAM CONTROL MECHANISMS	145
GOTO AND GOSUB BRANCHES.....	146
FORMS OF CONTROLLED BRANCHING	148
IF DECISIONS.....	148
TRUE OR FALSE.....	150
SIMPLE TESTS.....	151
COMPOUND TESTS.....	153
COMPLEX TESTS.....	154
Nested If..Else..Endif Statements	156
SWITCH/CASE DECISIONS	158
FALL-THROUGH EXECUTION	160
DUPLICATE CASE STATEMENTS	163
DEFAULT CASES	164
LOOPS	165
For Loops.....	165
ForEach Loop.....	167
For Loop Interruption.....	168
While Loops.....	169
While Loops Interruption	170
SUMMARY	170
CHAPTER 9 : IT'S ALL IN THE NUMBERS	173
MATHEMATICAL OPERATIONS	173
SIMPLE NUMERICAL MANIPULATIONS.....	173
The Abs and Fabs Functions	173
The Average Function	174
The Ceiling and Floor Functions	175
The Decimals Function	176
The Int Function	177
The Min and Max Functions	177
NUMBER TESTING	177
Pseudo-Random Numbers	178
LARGE AND TRANSCENDENTAL NUMBERS	180
The Exp Function.....	180
The LogE Function	180
The Log10 Function	181
The Sqrt Function	181
TRIGONOMETRIC OPERATIONS	182
The Sin, Cos, and Tan Functions.....	183
The ASin, ACos, and ATan Functions.....	184
The ASin Function.....	185
The ACos Function.....	185
The ATan Function	186
The Hyperbolic Functions: SinH, CosH and TanH	187
DATE AND TIME OPERATIONS.....	188
Date/Time Format.....	188
The TimeDate Function	188
The TimeYmdHms Function	189

The TimeJulianDay Function and the Day of the Week	189
The TimeJulToYmd Function	191
Time-Difference Calculations	191
Pause and Wait Functions	192
MATHEMATICS IN THE REAL WORLD	193
Accepting Input Variants	193
DOING THE MATH	197
FORMATTING VALUES	199
Formatting Currency	200
Formatting a Date	201
SUMMARY	202
CHAPTER 10 : SHOE BOXES AND FILE CABINETS.....	203
DATA STORAGE AND FILE OPERATIONS	203
FILE AND DIRECTORY CONCEPTS	203
HARD DRIVE MANAGEMENT	204
A UTILITY FOR DIRECTORY OPERATIONS	204
Planning the Utility	205
File Specification	205
Drive/Directory Specification	206
Exploring the CallFileList Utility	206
THE WINDOWS COMMON FILE DIALOG	209
Features of the Common File Dialog	210
Invoking the Common File Dialog	212
The Label Parameter	213
The Directory Parameter	213
The Filetypes Parameter	213
The Default Filename Parameter	214
The Flag Parameter	214
Returning a File Name	214
DIRECTORY INFORMATION FUNCTIONS.....	215
Converting Long and Short File Names	215
Locating Default Directories	215
Getting Drive Information	217
The DiskScan Function	217
The DiskSize and DiskFree Functions	219
HUGE NUMBERS	222
FILE MANAGEMENT	222
A SHORTCUT FOR LISTS	223
FILE-OPERATION FUNCTIONS	224
Handling File I/O	225
The FileOpen Function	226
The FileRead Function	226
The FileWrite Function	227
The FileClose Function	228
Manipulating Files	228
The FileAppend Function	228
The FileCopy Function	229
The FileMove Function	229
The FileRename Function	229
The FileCompare Function	230
The FileDelete Function	230
Additional File Functions	230

Introduction to Programming

BINARY FILE OPERATIONS	232
SUMMARY	233
CHAPTER 11 : WINDOWS AND GUI OPERATIONS	235
PAINTS, PENS, AND WINDOW BOXES	235
CREATING A WINDOW	235
WINDOW COORDINATES	240
LABELING THE WINDOW.....	241
WINDOWS WITHIN WINDOWS	241
DISPLAYING TEXT	244
Font Styles.....	244
Pitch and Family	245
Displaying the Message.....	245
ADDING BUTTONS.....	247
A QUICK REVIEW	248
MORE ABOUT COLORS.....	249
DRAWING IN A WINDOW	253
AN ALTERNATIVE TO BOXBUTTONWAIT	256
THE BOXDRAWCIRCLE AND BOXDRAWRECT FUNCTIONS	257
DRAWING STACK MANAGEMENT	259
PARTIAL CLEARING	259
FORMATTING TEXT IN WINDOWS.....	260
SUMMARY	263
CHAPTER 12 : MOUSING AROUND	265
GETTING AWAY FROM THE KEYBOARD.....	265
MOUSE OPERATIONS IN WINDOWS.....	265
Forcing Mouse Operations	265
Tracking the Mouse.....	267
SUMMARY	271
CHAPTER 13 : POKING INSIDE THE BOX.....	273
THE INTCONTROL FUNCTIONS.....	273
THE INTERNAL CONTROL TEST FUNCTION	273
GENERAL-PURPOSE FUNCTIONS.....	274
WINDOW INTERACTIONS AND WINDOW HANDLES	274
SYSTEM FONT SELECTION	274
FILE OPERATIONS.....	275
File Moves	275
File-Sharing Mode	275
FILE LIST BOX SETTINGS	276
GENERAL LIST BOX SETTINGS	277
APPLICATION CONTROL FUNCTIONS.....	277
Windows Messages.....	277
WINBATCH CONTROL	279
Exit Code	279
Icon States.....	279
Program File Names	280
WIL Termination Codes	280
WINDOWS RESTARTING	280
Windows System Restart	281
APPLICATION CLOSING	281
ERROR MESSAGES.....	281

ERROR HANDLING	282
CREATEPROCESS FLAGS.....	283
MEMORY ACCESS	283
INPUT TIMING AND WAITS.....	283
Timeouts	284
The SendKey Function.....	284
MISCELLANEOUS OPERATIONS.....	285
Language Control	285
System Menus.....	285
SUMMARY	286
CHAPTER 14 : DYNAMIC DIALOGS, MENUS, CALLBACKS	287
MAKING DYNAMIC DIALOGS	287
ADDING DIALOG PROCEDURE CODE.....	289
Functions vs Subroutines	297
Exercise_C	300
Exercise_D.....	301
Exercise_E	303
Exercise_F.....	304
SUMMARY	304
CHAPTER 15 : WHEN THINGS GO WRONG.....	307
DEBUGGING APPLICATIONS	307
LEARNING TO DEBUG.....	308
DEBUGGING IN AN IDE	308
DEBUGGING TOOLS	311
Debugging during Execution	311
Tracing Execution Step by Step	314
Terminating Execution	316
DEBUGGING OPTIONS.....	318
Debug.....	318
DebugTrace	319
Modes:	320
Mode Option Flags:.....	321
Immediate Action Codes:.....	322
Message or Pause Statements	323
DebugData	324
DEBUGGING AS A PROCESS	324
Debug Step 1: Run the Program.....	324
Debug Step 2: Test the Elements	324
Debug Step 3: Watch Branches, Tests, and Variable Tests	325
UNAVOIDABLE BUGS	326
SUMMARY	326
CHAPTER 16 : WINBATCH EXTENDERS.....	327
ADDING EXTENDED AND CUSTOM FUNCTIONS	327
THE WILX LANGUAGE EXTENDER	327
Getting Library Information	328
Converting between Numeric Systems	328
Accessing Drives	329
Accessing Windows API Functions	329
Verifying Credit Card Numbers	330
Using Utility Functions	330

Introduction to Programming

NETWORK EXTENDERS	331
Identifying the Network	331
Windows Platform Version	332
QUERYING ACROSS THE NETWORK	332
ADSI EXTENDER: WWADS44I.DLL	334
COMPILER OPTIONS FOR WIL EXTENDERS	334
CUSTOM EXTENDER DLLS.....	335
SUMMARY	335
APPENDIX A : WINBATCH DEMOS.....	337
REAL WORLD WIL SCRIPTS	337
CHAPTER 1 SAMPLES	337
WordCnt.wbt.....	337
CHAPTER 2 SAMPLES	338
Hello World.wbt	338
CHAPTER 3 SAMPLES	338
AskYesNo.wbt	338
AskLine.wbt	339
WILDdialog.wbt	339
PushButton.wbt	341
RadioButton.wbt	343
CheckBox.wbt.....	346
EditBox.wbt	347
ListBox.wbt	348
FileListBox.wbt	349
ComControl.wbt.....	350
CHAPTER 4 SAMPLES	351
StringTest.wbt	351
ArrayTest.wbt.....	352
HugeMath.wbt	355
VariTest.wbt	355
ListTest.wbt	356
CHAPTER 5 SAMPLES	356
MathTest.wbt.....	356
SimpleCalculator.wbt	361
CHAPTER 6 SAMPLES	363
SearchList.txt.....	363
SearchTest.wbt.....	363
SearchTest2.wbt.....	364
SearchTest3.wbt.....	365
SearchTest4.wbt.....	366
StrIndex.wbt.....	367
Blake.txt	367
SearchReplace.wbt.....	367
StrCmp.wbt	369
RelationalOperators.wbt.....	370
Parts.lst.....	371
ListSelection.wbt	372
ListSelection2.wbt	374
Password.wbt.....	376
WaitForKey.wbt	378
CHAPTER 7 SAMPLES	378
Music.txt.....	378

HyperLink.wbt	379
GetData.wbt	382
Parts.lst	384
ExternCall.wbt	384
SortData.wbt	384
Run_EXE.wbt	385
CHAPTER 8 SAMPLES	385
Logic.wbt	385
Select1.wbt	387
Select2.wbt	388
Select3.wbt	390
Prime.wbt	391
ForEach.wbt	391
Prime2.wbt	392
CHAPTER 9 SAMPLES	393
Average.wbt	393
Floor_Ceiling.wbt	393
Decimals.wbt	394
Min_Max.wbt	394
TestNumber.wbt	394
Random.wbt	396
Exponential.wbt	397
LogE.wbt	399
Log10.wbt	400
SquareRoot.wbt	401
Trig.wbt	402
ArcSin.wbt	403
ArcCosine.wbt	405
ArcTangent.wbt	406
HyperTrig.wbt	407
TimeCheck.wbt	409
TimeCheck2.wbt	410
TimeCheck3.wbt	411
Mortgage.wbt	412
FormatCurrency.wbt	417
CHAPTER 10 SAMPLES	419
CallFileList.wbt	419
DirTest.wbt	421
DirTest2.wbt	422
Free Disk Space.wbt	422
FormatNumber.wbt	424
Phone.lst	425
ShowList.wbt	426
PhoneList.wbt	426
Phone.lst	431
FileAttr.wbt	431
Binary.wbt	433
CHAPTER 11 SAMPLES	434
Hello Windows.wbt	434
Progress.wbt	437
Text Fonts.wbt	445
Colors.wbt	454
Buttons.wbt	459

Introduction to Programming

Lines.wbt	461
Shapes.wbt	469
Phone.lst	476
PhoneListBox.wbt.....	476
CHAPTER 12 SAMPLES	483
Freehand.wbt.....	483
CHAPTER 13 SAMPLES	488
SelfTest.wbt.....	488
CHAPTER 14 SAMPLES	489
Exercise A.wbt	489
Exercise B.wbt	494
Exercise C.wbt	498
Exercise D.wbt.....	503
Exercise E.wbt	508
Exercise F.wbt	513
CHAPTER 15 SAMPLES	519
Debug01.wbt.....	519
Debug02.wbt.....	520
Parts List.lst	521
ExternCall.wbt	521
GetData.wbt	522
SortData.wbt	523
Debug03a.wbt.....	523
Debug03b.wbt.....	524
Debug03c.wbt	525
Debug03d.wbt.....	525
CHAPTER 16 SAMPLES	526
PlatformInfo.wbt.....	526
NetTest.wbt.....	527

INTRODUCTION : BECOMING A PROGRAMMER

THERE WILL BE A TEST...

According to the popular myths, all programmers, commonly called “geeks” (even though the term itself applies to a certain type of performer in an old-style sideshow at the carnival) are strange individuals with eccentric habits. Programmers are solitary individuals who communicate better with computers than with other people, who work strange hours (usually in strange surroundings), and who are only happy when they are cracking into Pentagon or Kremlin computers and starting World War III.

Obviously, this last assertion is patently untrue and is simply a rumor fostered by the Gnomes of Zurich and the Trilateral Commission after certain rogue programmers blocked their collective efforts to take over the world. (The rest of the rumors, however, hold some elements of truth.)

This book is your invitation to join the ranks of those weird individuals who form the cyber-intelligencia comprising the true rulers of the world. As such, you will be expected to give up your three-piece suits, increase your intake of caffeinated beverages, and trade your usual 9-to-5 working hours for ... well, let's just say that programmers consider 9-to-5 to be slacker's hours.

However, as a reward, you will be allowed to mutter strange curses concerning devices and processes that the rest of the world has never heard of. You will be able to use such esoteric terms as *FOOBAR*, *bytes*, *registers* (which do not involve paper), *bitmasks*, *parameters*, and *dwords*; and to appear to be devoutly contemplating the soul of a silicon chip while others about you are devoting their time and energies to the mundanities of everyday business (including such tortures as conferences and committee meetings).

In this book, *Introduction to Programming*, we're going to cover a lot of ground. I hope that you'll find the journey interesting and the destination rewarding. You are on your way to joining the elite of the new order (and assuming your proper position on the “bleeding edge” of technology). Your instructions are quite simple: Sit back, flex your typing fingers, and ... read on.

The programming language we will use, WinBatch or WIL (Windows Interface Language), is a high-level language. This means that WinBatch allows you to perform tasks using relatively simple instructions that accomplish a great deal. And, perhaps more important, WinBatch allows you to write programs without needing to know all of the finer details of the Windows operating system.

Since the assumption is that you, the reader, are not an experienced programmer (or, in any case, not experienced with WinBatch), this book will start with the basics. We begin by showing you how to use the tools in the form of the WinBatch IDE (Integrated Development Environment) and then how to write simple programs.

Introduction to Programming

We will not, however, confine ourselves to only simple elements. In the course of this book, you will learn about a wide variety of programming capabilities—or *functions* as they are commonly called—as well as how to create data-storage elements, how to manipulate data elements, how to present them, and a host of other matters.

Along the way, you'll learn how to manipulate strings, create lists of data, read and write files, play with numbers, create buttons and controls, draw images, paint fancy windows, and even how to debug your programs. And, naturally enough, given the prevalence of networks in today's world, you'll learn how to access networks.

In short, you will learn both the basics of programming and a fair measure of the tricks of the trade—enough to put you well on the road to becoming a competent programmer.

Given this scope of material to be covered, it would be well to stop talking about it and to simply get started. Still, before proceeding, one word of warning:

There will be a test...

And, that said, let's get on with the real topic of programming applications.

CHAPTER 1 : THE GOLEM PRINCIPLE

BASICS OF PROGRAMMING

golem \gô-lam\ n. [Yiddish *goylem*]: an artificial human being of Hebrew folklore endowed with life.

In Hebrew folklore, a golem was a manufactured creature in the form of a human, constructed from clay and baked in an oven. The golem was sculpted with a cavity in its head. The constructor of the golem would place a script in the cavity. According to legend, this script was normally a holy scripture, and it was by the power of the words that the golem was brought to a semblance of life.

Once activated, the golem was able to perform such tasks as the animation's creator directed. From a religious standpoint, since a golem was a soulless creature, the construct was not subject to Levitical restrictions concerning the Sabbath and was not prohibited from working during the Sabbath.

Likewise, not being flesh and blood, a golem was not subject to exhaustion; did not become bored; did not require rest, sleep, or sustenance; and was not injured by fire or other dangerous or debilitating conditions. In short, a golem was the equivalent of our contemporary concept of a robot.

So, why are we talking about golems?

The main reason is that the stories of golems can tell us a great deal about their modern counterparts, the computer.

computer \kam-\ n. : one that computes; *specif.* an automatic electronic machine for performing calculations.

You'll notice that this definition of a computer (taken from *Webster's Seventh New Collegiate Dictionary*) doesn't have much in common with the definition of a golem. However, when you go beyond the mere dictionary definitions, you find that there are strong similarities.

Both are constructed of similar materials—clay begins as a mixture of hydrous aluminum silicates; a computer chip is also silicon-based but uses aluminum circuitry. A golem is baked (to produce anhydrous silicates), and a computer chip is also created using heat (both to initially purify the silicon and then to add specific and highly controlled impurities, which give it functional capabilities).

Introduction to Programming

And, of course, like the golem, the computer is also a soulless machine. It is not subject to exhaustion, does not suffer from boredom, and is not governed by Levitical restrictions.

But, for our purposes, the most important parallel is how they are both controlled by “words.” The golem is given life by the words—the script or scripture placed in its head. The computer chip also gains its life from the words—the instructions or machine code that tell the computer how to perform tasks.

How Computer Applications Work

Without getting into the details of CPUs and chips, file systems and memory allocation, or of bus speeds and duty cycles, what makes a computer application work is simply a series of instructions—words. These are not, however, words in English (or Russian, Yiddish, Thai, or Urdu). Instead, these instructions are in a binary language that the computer (or, more precisely, the computer’s CPU) can understand directly.

The term *word*, as used in the computer and software industry, has a special meaning. It refers to a piece of data that consists of 16 bits (or **binary digits**, which are the smallest unit of computer information). For this reason, this chapter refers to *instructions* rather than *words*.

Because what the computer can understand and what the programmer can readily understand are two quite different languages, programs are commonly written in what are called *high-level* languages, which are languages that humans can more easily comprehend. Note, however, that these high-level languages are not precisely English. It simply is not practical to tell the computer, “Add up this column of figures and give me the total, the average value, and a breakdown of expenses by month.”

Lingua Cyber

In order to “talk” to the computer and to create sets of instructions to produce your custom application, an initial requirement will be for you to learn to speak a *lingua franca*—or, if you prefer, a *lingua cyber*. You need to know how to use a high-level language that you, as the programmer, can understand and that can be translated into instructions that the computer can recognize and execute.

Remember, there is nothing “sissy” or unmanly/unwomanly about resorting to these high-level languages rather than struggling with the machine language instructions directly. The real point of developing any application is to write it in a reasonable length of time, to have it work when finished, and to have it accomplish the desired task.

For example, would you prefer to program using assembly language (a low-level language) instructions such as:

```
DISPLAY      EQU    2H          ; output function
DOSCALL      EQU    21H        ; DOS interrupt number
```

```

COUNT      EQU      12D          ; string length

DATASEG      SEGMENT              ; define data segment
STRING      DB      'Hello World!'
DATASEG      ENDS

CODESEG      SEGMENT              ; define code segment
MAIN        PROC      FAR
    ASSUME    CS:CODESEG, DS:DATASEG
START:
    PUSH     DS                    ; save old data segment
    XOR      AX,AX                 ; ax = 0
    PUSH     AX                    ; save ax on stack
    MOV      AX,DATASEG            ; store data segment in DS register
    MOV      DS,AX                ;
    MOV      CX,COUNT              ; store string length in CS register
    MOV      BX,OFFSET STRING      ; address of string

REPEAT:
    MOV      DL,[BX]               ; put one character in DL register
    MOV      AH,DISPLAY            ; display character function
    INT      DOSCALL               ; call DOS
    INC      BX                    ; advance the pointer
    LOOP     REPEAT                ; loop until done
    RET                            ; return to DOS
MAIN        ENDP
CODESEG      ENDS
MAIN        END                    ; end of assembly

```

Or would you find it easier to write:

```
print "Hello, World!"
```

Granted, this is a rhetorical question. We assume that your preference is a high-level language that allows simple instructions like those above to be used to accomplish relatively complex tasks.

Both of these code fragments perform the same task: assigning values to two variables, then adding the two variables with the sum stored in a third variable. And, finally, the third variable is printed to the console (DOS) display.

For programming applications, there are a wide variety of high-level languages, and these are commonly referred to as *computer languages*. BASIC, COBOL, FORTRAN, C/C++, and Java are just some of the multitude of computer languages.

This book is about using a high-level language called WIL (Windows Interface Language) or simply WinBatch. Although this language is less powerful than some other languages, it is also much easier to learn and use than many other languages, such as C/C++ or BASIC.

High-Level versus Low-Level Programming Languages

In the next chapter, you will see how a single WinBatch instruction provides a window, a frame, a title bar, and a message display on the screen. In addition, behind the scenes, this same instruction provides a timer that will close the window after a specified interval. Compare this with the assembly language example shown in this chapter, and you can get an idea of the relative power supplied by a high-level language.

Naturally, there is a trade-off. A low-level language like assembly allows the programmer to design anything—absolutely anything. A high-level language like WinBatch, while granting a great deal of power in simple instructions, also constrains the developer to using the functions provided.

Of course, learning to use a low-level language is considerably harder than learning to use a high-level language. Then, even after you’ve become familiar with the language, the work required to create even a simple application using a language like assembly is many, many times greater than creating a similar application using a high-level language like WinBatch.

This is not to suggest that either high-level or low-level languages are preferable; each has different uses. However, high-level languages offer two important advantages: They are easier to learn, and they can be used for rapid application development.

The Sorcerer’s Apprentice

Computers are fast and accurate, but they are also complete idiots. **Computers do not think!** The only thing that a computer can do is to follow instructions. A computer can accomplish its tasks very rapidly, efficiently, and patiently. But the computer does not take any initiative; it does not possess even a modicum of common sense. A computer will do something totally stupid **if this is what it has been instructed to do!**

If you saw Walt Disney’s film *Fantasia* (and probably also if you watched the Disney show on TV), you’ve seen the story “The Sorcerer’s Apprentice.” In this story, Mickey plays the part of an apprentice whose job is to sweep out the sorcerer’s laboratory and to fill the cistern with water. While the sorcerer is away, Mickey decides to try out one of his magic spells to animate the broom and have it carry buckets of water. Then, Mickey falls asleep. The animated broom continues to carry water, repeating the task long after the cistern is filled to overflowing. Mickey finally awakens when the entire laboratory is underwater.

Mickey, frantically trying to stop his creation, uses an ax to split the broom into a hundred pieces, each of which proceeds to carry more buckets of water into the structure. (This part of the story goes a bit beyond our immediate caution—an ax generally will stop your computer from executing its tasks.)

The point is that a computer application—any computer application—is very much like the broom animated by Mickey. Once an application is told to do a task, it will continue to do so until it is instructed otherwise. In computer applications, everything must be explicitly identified, and nothing can be assumed.

For a simple example, suppose that we’ve included instructions in a program to break a sentence, called a *string* in computer terminology, into individual words (in the human sense). We’re going to send each of these words to another process called *dictionary*.

A *subroutine* is a block of code containing a set of instructions which may be called from any point in a program to perform a specific task. (Subroutines are discussed in [Chapter 7](#).)

To do this, we would create a *loop*, which is computer terminology for a group of instructions (called a *block* of code) that are repeated *ad infinitum* until a test condition is satisfied. (Loops and other control conditions are discussed in [Chapter 8](#).)

The term *string* is commonly used to refer to a string of characters, called *chars*, but remember that the spaces between words are also characters.

Here are the instructions for the proposed process expressed as “dummy” code in plain language, with comments in italics (later, you’ll learn how to construct the same type of process in WinBatch):

```
Let location equal 1.
Repeat
  Is character at location in test_string equal to space?
  If NO
  Then
    Increase location by one.      (to test the next character)
  Else
  Begin
    Copy location minus one number of characters (don't include spaces)
    from first of test_string into new_string.
    Send new_string to dictionary.
    Trim location number of characters from first of test_string.
  End.
Until test_string is empty.
```

In this example, the indentations are simply to group lines (blocks) visually to show which commands execute together. This type of formatting is not required by the computer but should be used simply because it makes it much easier for you to read your own code at a later time—like an hour or more after you’ve written it, not to mention next month when you find that you need to make a change (and this **will** happen!). Also note that the practice of writing dummy code in plain language like this is often useful when designing an application.

Introduction to Programming

Now, these instructions should parse (separate) our sentence (string of characters) into individual words, sending each one to the dictionary subroutine and finishing when no more words remain, correct?

Does this question sound like it is loaded? Is it a trap?

Well, yes it is. The process, as written, will never finish. This is routine will become what is called an *infinite loop*, because, once initiated, it will attempt to continue until the power is shut off or the computer is reset. Obviously, this is not a desirable condition in an application.

In this example, the error is a simple one. We've created a loop that is set to terminate when a specific condition is satisfied. Within this loop, we have also created a test that looks for a space character to identify individual words. The failure will occur when the last word in the sentence is reached **unless, against all reasonable expectations, the sentence ends with a space character.**

In other words, as the routine is presently constructed, the final word in `test_string` will never be parsed unless there is a space following it to allow it to be identified as a word. And, unless this identification is made, `test_string` will never be empty, the test used to terminate the loop will never be satisfied, and the loop will never finish.

WinBatch provides a function called `ParseData` that breaks a line into its individual items, similar to the example discussed here (without becoming embroiled in an infinite loop, of course). An example of the `ParseData` function is included in the [WordCnt.wbt](#) demo program.

The moral of this example is a computer is only as intelligent as the application. And, of course, the application is only as intelligent as the instructions provided by the programmer.

Does it seem as if the point is being belabored excessively; as if a sledgehammer were being used to drive a thumbtack or an elephant gun used to shoot a mosquito? Please believe that, if anything, we have been constrained in pressing this point home. Murphy's Law is never more evident than in programming and never more annoying when it occurs.

☛ **Murphy's Law** (short version): Mistakes **will** happen! Repeatedly! When a program doesn't run as planned (and, at some point or another, this will happen to you, just as surely as the sun rises), we use a process called *debugging* to locate the point where an error occurs and to discover why. In other words, we debug programs to find our mistakes. This topic will be covered in [Chapter 15](#).

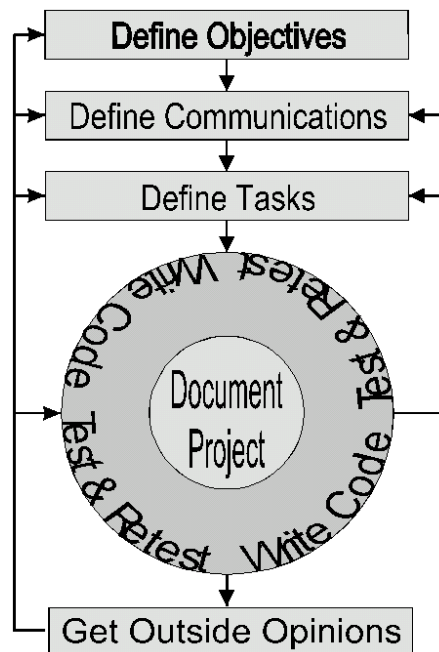
For the moment, you may consider the lecture terminated. We will now move on to the subject of application design.

Principles of Designing Applications

The one sure secret of success for a programmer—whether you are a professional software engineer or simply writing a few utility applications for your own use—is attention to detail. While it is not the only requirement (otherwise, we could all be replaced by cats), this one factor greatly affects whether you are successful or only frustrated in your programming endeavors. Furthermore, attention to detail is never more important than at the start of your project, in the planning and designing stage.

Therefore, before you learn how to create an application using a programming language, you need to know how to design the application. Application design is important no matter how simple your application is.

There are seven major steps involved in the design process, as illustrated below and discussed in the following sections. (Note that in practice, the development steps are not always distinct; they frequently overlap.)



The application development process

Step One: Defining Objectives

The first step, and the first principle, in designing any application is to decide what you want the application to do. This can be done formally, such as by writing a mission objective, or informally, as a series of notes or a paragraph or two of text. The exact form is unimportant; the content is important.

Introduction to Programming

Begin with a one-sentence description that is a broad overview of what the application is intended to accomplish. For example, suppose that we start with this stated objective for a simple application:

- Track the movements of the mouse.

Once you have an overview, this broad statement should be expanded to encompass and further define the purposes to be served by your design. Our sample overview objective could be expanded to include these further objectives:

- Provide an explanation to the user.
- Display the mouse position in screen coordinates.
- Update the displayed coordinates at regular intervals.
- Provide an exit option.

This represents a reasonable set of objectives. Later, perhaps we'll wish to revise this list. Typically, as we progress with developing an application, we find new features or elements that we want to add. It's also possible that some of the features initially planned later turn out to be unnecessary or even undesirable. That's fine as well. Objective definitions (and good programmers) are flexible; stating an objective doesn't mean that it is carved in stone.

Step Two: Defining Communications

Our objectives definition implies a couple of elements of communication:

- Find out the position of the mouse from the system.
- Report the position to the user.

Notice that there are two quite different types of communication included in this specification: one to retrieve information from the system, and a second to write information to the display.

Depending on the application's purpose, there may be quite a variety of communication elements within the program's scope. These can include:

- Retrieving data from a file
- Retrieving information from the system or from other ongoing processes
- Retrieving input and responses from the user
- Writing data to a file
- Passing information to the system or to other processes
- Writing information to the display (which can include providing audible cues or other types of feedback)

Each of these communication elements can be further subdivided into individual transactions or, perhaps more relevant in many cases, they can be further defined as the type of data being read or written.

Suppose that we expect to read and write a data record from a file. If so, the definition will need to include the structure of the data (how the data is formatted) and what type of record file is used for storage. Defining the structure of the data, whether it is very simple

or extremely complex, is every bit as important as recognizing that the data exchange will occur.

For example, let's assume that the application needs to read a record containing name and address information. Exactly what does this record consist of? Will it be just a name and a street address? That format is unlikely. More typically, the record will need to contain either one or two lines for the street address, plus the city, the state, and zip code. And if we expect to contact anyone beyond our national borders, the record will need to hold international address information as well.

And what about telephone numbers, area codes, and international country codes? Are there any extensions associated with entries? For that matter, do we need any type of PBX codes with certain types of calls or billing log codes?

Once we've exhausted the address and telephone number variations, are there any other fields that we should consider? How about notes, contact lists, e-mail addresses, records of previous calls, and so on. The list of considerations could be extended indefinitely, or it could be shortened if these elements are not relevant to the task—it all depends on the purposes of the application.

The point is that data elements and data structures require careful planning and design. Planning these elements at the beginning of your project can save you a lot of work later. Or, more accurately, it can save you a lot of *rework*.

Step Three: Defining Tasks

The next step in the process is to define the tasks that the application needs to accomplish. In our simple mouse-tracking example, we've already stated a couple of the tasks when we defined objectives:

- Display the mouse position in screen coordinates. This objective implies an element of communication but also implies a task: writing this information to the screen.
- Update the displayed coordinates at regular intervals. This objective also implies a task: creating a mechanism that will sample the mouse coordinates by requesting this information from the system, at specific intervals, and then pass this information to the display task.
- Provide an exit option.

You may think that this third task implied in the objectives is rather obvious. However, it's the obvious that is overlooked nine times out of ten!

This particular "obvious" task is one for which you normally do not need to make special provisions. Most applications have inherent exit provisions and, even if you don't plan for them, the application shell commonly supplied by the development tools will include an exit provision.

However, one element that is overlooked frequently is the need to provide a response to an exit command. This response can take a variety of forms and might include saving critical information before an exit, performing cleanup operations, or simply requesting confirmation before allowing the exit.

Introduction to Programming

Now we need to consider if there any other tasks that this simple application needs to handle. Actually, there are two more tasks, which are not likely to occur to the novice programmer.

One is to “capture” the mouse input. Understanding this task requires some knowledge of event messages. Very briefly, *event messages* are the means used under Windows for the operating system to pass data and notifications to applications. The operating system tracks the mouse position and then makes this information available, as event messages, to whichever application the mouse cursor is positioned over.

There are a great many other types of event messages that are being passed to all open applications at all times. However, applications are not required to respond to all event messages and, normally, a program will be written to respond to only certain messages and ignore all those that do not apply. Much of this message handling will be transparent to you, as the programmer, and will not require your specific attention. You’ll learn more about message handling and when you do need to be familiar with event messages in [Chapter 11](#).

Normally, an application only “owns” the mouse—and, therefore, is ready to receive event messages from the mouse to report mouse positions—while the mouse (the mouse cursor) is positioned over the application’s window. Capturing the mouse input is a method that allows an application to monitor the mouse-event (movement and button) messages at all times. (Note that although mouse capture does not prevent other open applications from being operated by the mouse, applications should not intercept mouse-event messages unnecessarily.)

The final task warranting mention here is predicated on the mouse-capture task. It is to ensure that the mouse capture is released before the application exits, which ties back into the need to provide exit responses.

Now, in this or any other application, there may or may not be other tasks requiring handling, and it may or may not be practical or possible to think of all of these in advance. However, you should do your best to define all the tasks for your application before moving on to the next step.

Step Four: Writing the Code

Perhaps this is point where you expected to start. But, by now, you should understand why this is the fourth step and not the first. Planning prevents problems.

Since writing the code is the main topic of this book, it will be covered in more than a little detail in subsequent chapters. It is mentioned here simply as a step in the overall process of application design.

Step Five: Testing, Testing, and Retesting

Once you have written your application, and during the process as well, you need to test, test again, and retest. Simply testing the application once is not enough.

You need to test several times because not all errors are obvious and some are downright obscure. Also, the more complex an application becomes, the harder it becomes to ensure that all possible conditions have been tested. And, quite frequently, an application may pass a test until just the right (or wrong) conditions are applied.

Testing applications incrementally during development and when they are finished will be discussed in following chapters in parallel with application development and then in greater detail in [Chapter 15](#).

Step Six: Getting Outside Opinions

Even after you've tested an application to frustration and you're absolutely sure that there are no possible errors remaining, your application is still not ready to go. At this point, you need an outside opinion, or better, multiple outside opinions.

For this purpose, you need to hand the application to someone who does not know how it works, has no expectations for its performance, and, ideally, has not been involved in the application's development. Then let that person try to break it!

Chances are that such outside testing will reveal flaws that you, who are very familiar with the application and have expectations of what it will do, would not likely discover. An outside tester is more likely to discover performance or interaction flaws, to find awkward elements in usage or input, or to come up with ideas for valuable enhancements. This is not to denigrate your own talents, but you should recognize the simple fact that it is difficult to survey the forest while you are standing in the midst of the trees.

Here's an important point: Whatever your outside testers tell you, **listen!** Never argue, explain, or justify ... just listen carefully. Otherwise, you'll miss one of the most valuable resources you have access to.

Step Seven: Documenting

Strictly speaking, documenting your application is not a separate step but should be numbered Step 1B, Step 2B, Step 3B, and so on.

This is not to imply that you necessarily need to publish a manual with four-color process photos and screen shots or that you need to produce elaborate online documentation. Your application may or may not require extensive documentation.

This is to suggest (or even direct) that you keep notes during the development process to remind you of why a particular subroutine was created, what the purpose of a feature was, what the input expectations are for a process—in short, what is going on within the application. These notes can be separate from the application as a text file or as separate memos, or they can be brief notations embedded in the source code as you write it. Ideally, you documentation should be both separate notes and embedded notations.

In this form, the documentation is intended less for the end user (although it can be used while preparing more formal documentation) than it is intended as a memory aid for you or for other programmers at a later time when the application requires revision, correction, or maintenance.

Introduction to Programming

Incidentally, the phrase “at a later time” does not necessarily mean next month or next year; it can also mean two hours from now, tomorrow, or next week! It doesn’t take long to forget precisely why you elected to do something in a particular fashion. A few minutes spent making notes while you are designing a portion of the application can save many hours a day later.

Summary

We began this chapter by talking about the nature of computers, computer languages, and computer applications. We offer the cautionary reminder that computers are both stupid and single-minded. Our point is that the computer will always try to do exactly what it is told to do, irrespective of whether this is what you *want* it to do.

Then, having tempered your expectations somewhat (but, we trust, without having discouraged you), we turned to the process of creating a computer application. We outlined seven basic steps for application design, stressing the importance of planning, testing, and documenting the application.

Perhaps these preliminary discussions have struck you as unnecessary, and you're anxious to get on to the point of actually creating something. If so, the next chapter should satisfy your desire to quicken the pace, as we move to hands-on programming. In [Chapter 2](#), you'll create an application while you become familiar with the tool set (the programming language) and the workspace (the IDE, or Integrated Development Environment).

As you continue into the details of programming, keep in mind what you've read in this chapter. Although the steps in the development process may seem dry and theoretical, applying them to your own projects will save you a great deal of trouble.

CHAPTER 2 : THE PROGRAMMER'S WORKROOM

THE INTEGRATED DEVELOPMENT ENVIRONMENT

Back of the beating hammer,
By which the steel is wrought,
Back of the workshop's clamor
The seeker may find the Thought.

Braley – *The Thinker*, Stanza 1

programmer – a person who designs, writes, tests and documents a computer program.
– The PC User's Pocket Dictionary

integrated development environment – abbreviated IDE. A complete set of program-development tools, all run from a common user interface. – The PC User's Pocket Dictionary

All the sample programs presented in this and the following chapters are available in [Appendix A](#).

Before we begin writing WinBatch programs, there is one rather obvious prerequisite: you must have WinBatch installed on your computer. If you have not yet installed WinBatch, you can follow the installation instructions in your WinBatch help file, under “WinBatch Setup,” to set it up on your computer. Then you will be able to follow the exercises and create the sample programs presented in this book.

First, we will look at the differences between compiled and interpreted programs, then we will introduce your programming workspace—the integrated development environment (IDE). During the course of this chapter, you will create a simple WinBatch program and learn how to generate dialog boxes.

How To Use This Book

Throughout this book, you will find references to example programs. All of the examples presented in this and following chapters will be listed in [Appendix A](#). Ideally, you should open these programs and watch how they function while reading the explanations for each.

Try using the Debugger mode (explained later in this chapter) to execute your own applications step by step. You are also encouraged to experiment with the example programs, to make changes to these and to observe how changes affect operations.

Introduction to Programming

Within the text of this book, the names of functions and variables commonly appear in the `Courier` font to make them easier to distinguish.

Another item for you to note is that, in the WinBatch editor, color coding is applied to the program. The color coding causes the names of functions to appear in `blue` and comments in `green` while reserved words such as `for`, `endif`, `case` or `gosub` appear in purple; quoted strings are shown in `red`.

Once your source file is saved as a `.WBT` file, this color coding appears automatically as you add or make revisions and should, we hope, help to prevent errors. Or, at the very least, make some errors more readily visible.

You will see similar color code syntax used throughout this book.

Compiled versus Interpreted Programs

WinBatch is marketed in two versions: an interpreter version and a compiler version.

An interpreter is simply a program which reads a source file – a.k.a. a script – and interprets each instruction in the source file to create a set of machine language instructions which can be executed by the computer. The source file – or script – is written in a form which can be read – and written – by humans but which is not intelligible to the computer. Therefore, in order to execute the program, the interpreter reads one set of instructions – normally a single line of code – and converts these into a machine-compatible format. The resulting machine code is then executed by the computer before proceeding to the next line of the source file.

While an interpreted program can be executed, it cannot be distributed – given to another user or run on another computer – without also distributing and installing the program interpreter on every machine where the program should execute.

In contrast, a compiler converts the entire source file into a machine language program which can, subsequently, be executed without requiring a copy of the compiler or interpreter. This means that the executable version of the program can be distributed and executed on any (compatible) system without the need for a copy of the WinBatch software itself. For example, a compiled WinBatch script can be distributed across a network without the individual workstations requiring a copy of WinBatch or access to WinBatch for the applications to function. You can distribute a compiled WinBatch program to anyone, and there are no licensing fees required for such distribution.

Also, as a secondary advantage, when you distribute a compiled `.EXE` (executable) file rather than a interpreted WinBatch script, the distributed product can be used but not modified by those third parties. (Of course, you can still modify the program, using the original source script.) In this fashion, your original concept for the program, as exemplified by the source code, remains proprietary and protected.

If you create an interpreted WinBatch script (with the interpreter version of WinBatch), the WinBatch software must be installed on the systems that will run that script. In other words, running your interpreted program requires WinBatch. Of course, the interpreter

version of WinBatch does cost less and, if your WinBatch programs are only for your own use on your own computer, there may be no great need for the compiler version.

For the purposes of this book, it does not matter whether you are using the interpreter version or compiler version of WinBatch. The principles of application development and the demo applications presented here apply to either version.

A Brief History of the IDE

Back in the olden days of yore, before graphic environments such as Windows became the preferred standard, application development was a quite different process. Source files for applications—the pseudo-English text files containing the instruction codes—were written using whatever editor was most convenient or the one that the programmer preferred.

Once the source file was written, or when the programmer had reached a point where he or she felt it was ready for testing, the instruction file would be saved as a disk file, and the compiler would be invoked to turn the instructions into machine language.

The *source code* file for a program or application may also be referred to as the *instruction file*, *program file*, *source file* or, in the case of a WinBatch application, as a *program script* or simply a *script file*. The terms used are interchangeable and all refer to a document file that can be read or modified using any editor or word processor program. Some languages, such as Visual Basic, use some proprietary file format (.frx form files in Visual Basic), but most languages rely on plain-text source files, even though they may use special file extensions to indicate the purpose of the file.

In most cases, at this point, the compiler would report a series of errors in the code, listing these by line number and offering a very brief error explanation. When this happened, the programmer would then reopen the source file, locate the offending lines, and attempt to correct the mistakes before repeating the compilation.

Those who remember the old ways first hand may object to our over simplification of a rather more laborious process; however, keep in mind that this coverage is intended as an introduction and comparison rather than as an in-depth history lesson.

When the compiler finally did not find any more errors—usually after several compile-and-correct cycles—the compiler would work its magic and build the executable (runnable) program. The resulting product could be tested. This testing inevitably uncovered other errors that occurred during program execution. Then the programmer would reopen the editor, load the text file containing the program instructions, and begin the process again.

The introduction of the IDE (integrated development environment) greatly improved the lot of the application developer. The first popular IDE came on the market in the mid-

Introduction to Programming

1980s, when Phillipe Kahn, founder of Borland International, put two new products in a single package: the Turbo Pascal compiler and the Turbo Pascal Integrated Development Environment.

Actually, the term *IDE* came later, but the concept itself was an instant hit. The combined editor/compiler/linker allowed programmers to write their code, compile and link, test the results interactively, locate errors instantly, correct and recompile, and continue without tedious interruptions.

Turbo Pascal was itself faster than any other compiler and better documented, and it also produced smaller executable code. Even better, Borland did not demand royalties from developers. At one time, Turbo Pascal and the subsequent Turbo C compiler threatened to put Microsoft out of the programming language business entirely. Microsoft's share of the development language market did not really recover until more than a decade later, when Microsoft finally introduced its own version of the IDE, under the name Visual C++.

Since then, the IDE has become the standard that few programmers are willing to forego. The fact is that an IDE makes application development much faster, easier, and cleaner (some consider not using one tantamount to proof of severe brain damage).

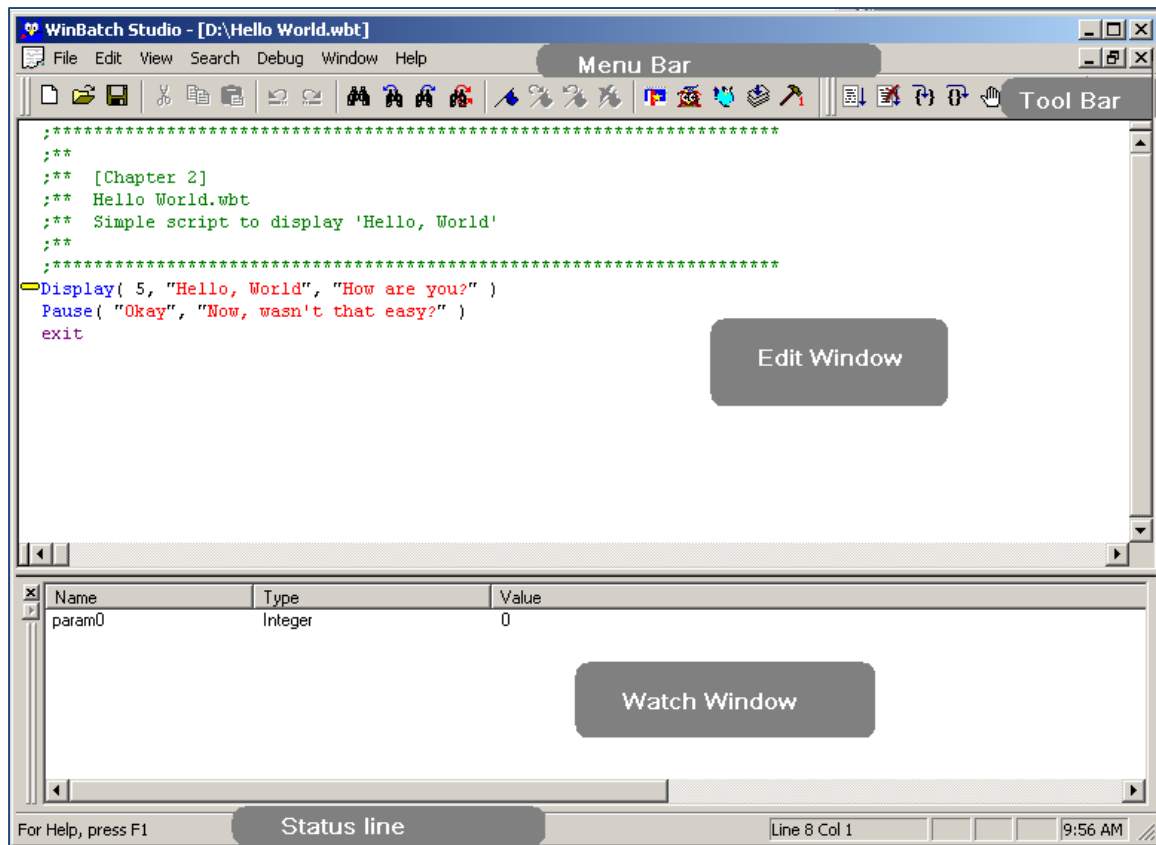
An Introduction to the WinBatch IDE

After you've installed WinBatch, you should find 'WinBatch Studio exe' on your program menu (the Start menu) where it can be launched like any other application.

In either case, when you launch WinBatch Studio, you should see something similar to the following window where the WinBatch IDE (WinBatch Studio) appears with labels added to identify its main elements:

- The Menu Bar
- The Tool Bar
- The Edit Window containing the program source code
- (Not shown) The build window, which will display error messages in the code and, on a second tab, the search result window
- The Watch Window, where program variables and their values will appear
- The Status Line shows a variety of information including the position of the cursor (line 5, col 1), the execution status (Script completed) and various details of the keyboard status (none appear in the illustration) as well as the current time.

In the illustration, the program edit window shows the complete source code for a simple program. Before going into the details of the WinBatch IDE – a.k.a., the WinBatch Studio – we'll construct and execute this program as an introduction to WinBatch programming.



The WinBatch Studio with the Hello, World example

A First Program: Hello, World

Hello, World is a traditional first program for teaching a computer language. This program is intended to do very little except for displaying a message. In this case, as you will see, we've added a small twist...

To get started, open WinBatch Studio and then select New from the File menu to create a blank document.

Since WinBatch Studio makes no assumptions about the type of document you are creating – or, more accurately, defaults to the assumption that this is a blank text document – you'll need to save the document with a “.wbt” extension before WinBatch will recognize this as a program script.

Next, enter the following source code:

```

; Hello, World.wbt
Display( 5, "Hello, World", "How are you?" )
Pause( "Okay", "Now, wasn't that easy?" )
exit

```

Introduction to Programming

The first line is a comment, which begins with a semicolon (;) to tell the interpreter to ignore everything following the semicolon on the current line. The comment in this example simply identifies the name of the source file.

The second line contains a `Display` instruction, which produces the image shown below:



The `Display` function is called with three arguments or parameters. The terms *argument* and *parameter* are effectively interchangeable and simply refer to a unit of information that can take several forms. The possible forms, or types, of data will be introduced in [Chapter 4](#).

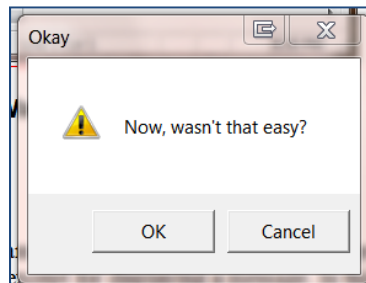
In this example, the first argument is a number that tells the `Display` function how many seconds to display the message window. Since the `Display` window does not have any controls, without specifying a time, the window would remain on the screen indefinitely—not a desired result in this case.

The second and third parameters are each strings. A *string* is simply a sequence of characters, such as a word or a sentence, enclosed in quotation marks (quotes) to indicate where the string begins and ends.

When a string needs to include quotation marks as part of the text, the usual quotation characters (") enclosing the string can be replaced with alternate quote marks, such as single quotes (') or back quotes (`). For example, ``"I quote, of course"`` would be displayed as "I quote, of course".

In the example, the second `Display` parameter (the first string) appears as the window title or caption. The third parameter (the second string) is the text to be displayed inside the window.

After the requested 5-second delay, the window closes, the `Display` function returns, and the next instruction, `Pause`, is executed, producing this result:



The `Pause` instruction is called with two string parameters. As with the `Display` instructions string parameters, the first parameter is the string that appears as the caption, and the second is the one displayed within the window. Unlike the `Display` instruction, the `Pause` instruction does not require an argument to set the time interval. Its display

automatically contains two button controls: OK and Cancel. Selecting either of these or the close button (the x) on the title bar will close the window.

Finally, once the `Pause` instruction returns, the last instruction in `Hello, World` is executed. This is the `exit` statement. Strictly speaking, this final instruction is not needed here. The program will terminate just fine without being explicitly told to do so.

Since you've already saved this script as a file with the `.wbt` extension – without doing so, you would have been unable to test the program since WinBatch would not have recognized this as a WIL script – WinBatch Studio will automatically save any changes made before executing the program.

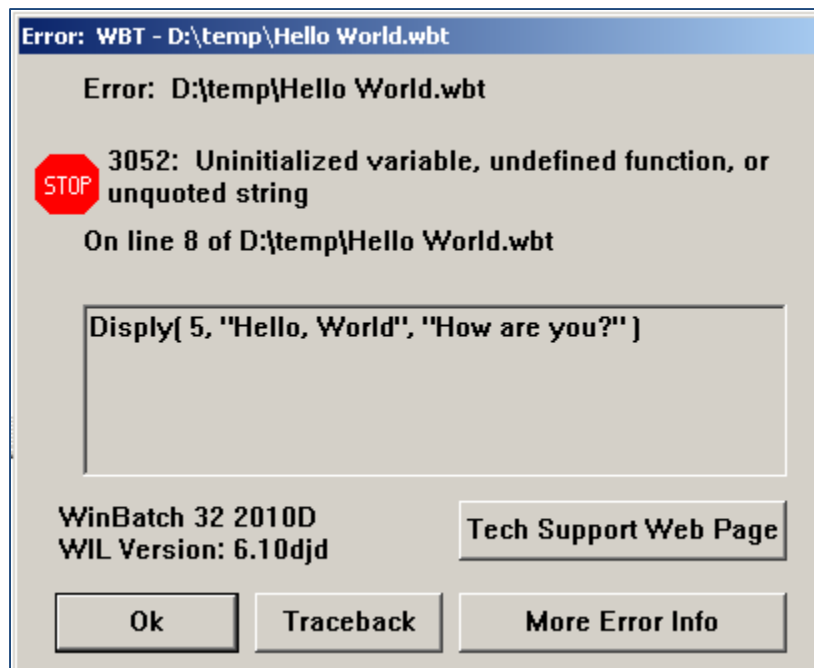
You can now run your copy of `Hello, World`. Simply click on the Go button on the toolbar (the seventh one from the right end of the toolbar).



The application should run, displaying the first message for 5 seconds, then the next message will appear until you click on one of the buttons to clear it.

If you happened to make a mistake while entering the code, after you click on the Go button, WinBatch will let you know, politely enough but firmly.

For example, suppose that the second instruction in [Hello World.wbt](#) has typographical error and “Display” is misspelled as “Disply”. Then, when you try to execute the program, a dialog will appear as:

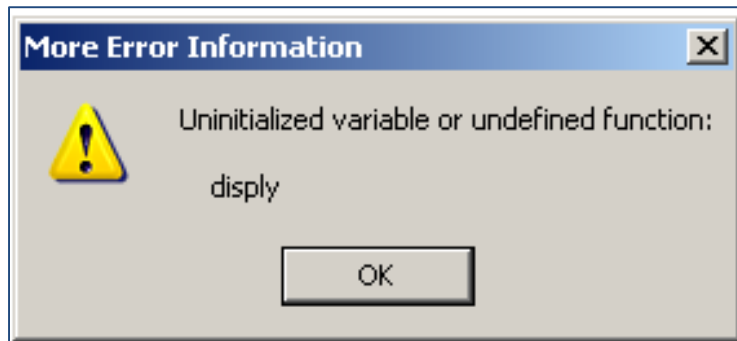


Introduction to Programming

Note: after making a change to the script, make sure to save the file before execution.

The dialog caption provides the error number and a brief explanation of the error while the first line in the dialog shows the program line where the error has occurred. If you simply click the OK button, you'll be back in the WinBatch Studio editor with the cursor positioned on the line where the mistake was recognized, ready to correct the script.

While the error message has shown you where the error occurs in the script, the More Error Info button (bottom right) will provide you with additional information as shown following.



Alternately, if you are on-line, you can click on the "Tech Support Web Page" button to connect to Wilson WindowWare's support site where you can locate further information relating to the error.

Our recommendation, however, would be to begin by simply looking at the source code and trying to figure out what it is that WinBatch doesn't recognize. Most errors tend to be quite simple ones and don't require any kind of extensive search to locate or understand.

And this concludes your introduction to the construction of a simple program. Now let's take a tour of some of the features and tools in the WinBatch IDE.

WinBatch Studio Features and Tools

The WinBatch Studio menus and toolbar provide a wide variety of tools for creating, manipulating, and testing applications. The WinBatch toolbar is shown below:



Many of the WinBatch toolbar buttons and menu options should be familiar to Windows users, since they are standard functions supplied by most Windows applications.

Most of the functions on the toolbar are also available on the menus. However, space limitations dictate that the toolbar contains only a subset of the 72 menu items. We won't discuss each of these menu options individually; most of them are self-explanatory. These menu items have also been documented in the WinBatch Studio .chm help file.

File Operations

The New, Open, and Save buttons on the toolbar are used to create source (or document) files, open files for editing, and save files to the disk.



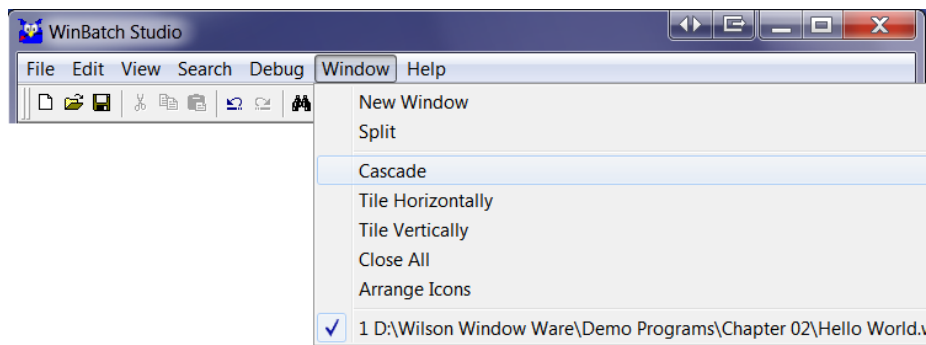
The first button on the left, the New tool, creates a new, blank and unnamed file. Until the file receives a name—by saving the file to disk—the file is given a temporary name, such as “Text1”.

The second button, the Open tool, displays a file selection dialog box for locating and opening files.

The third button, the Save tool, saves the file to disk. For a new, unnamed file, the Save tool calls the File Save dialog box, where you supply a name and file extension, as well as select a location to save the file. For a file that has previously been saved, the Save tool automatically saves the named file, without displaying a dialog box.

These operations and other associated file functions are also available as options on the File pull-down menu. In addition, the File menu will contain a list of recently used files.

The WinBatch editor can open multiple files at any time. The Windows pull-down menu will display a list of the currently open files, with a checkmark by the file currently selected (the active file):



If more than ten files were open in this illustration, the last option listed would be More Windows which would bring up a dialog box with a complete list of all the open files.

Introduction to Programming

The Windows menu also offers options to create a new window and to split a window, as well as a series of options to arrange windows and window icons (icons representing shrunk windows).

Cut, Copy, and Paste Operations

The `Cut`, `Copy`, and `Paste` buttons on the toolbar perform the standard Windows cut, copy, and paste operations.



Following standard Windows usage, the `Cut` and `Copy` buttons will be grayed-out (unavailable) until a selection has been highlighted, and the `Paste` button will be grayed-out unless there is material on the Clipboard available for insertion. Each of these tools, plus other editing functions, are also available on the Edit pull-down menu.

Undo and Redo Operations

The `Undo` and `Redo` buttons on the toolbar allow you to recall (undo) recent editing actions or to repeat (redo) them.



When there is no action that can be undone or redone, these buttons will be grayed-out.

The `Undo` and `Redo` functions are also found on the Edit pull-down menu.

Find, Find Next, and Replace Operations

The `Find`, `Find Previous`, `Find Next`, and `Replace` buttons on the toolbar perform search and replace functions.



Like the other tools discussed so far, these functions are standard for Windows applications and should be familiar to most Windows users.

The `Find` button displays a dialog box that allows you to enter a string (a text entry) and then locate a matching entry in the current file, scrolling the edit window to that point. The `Find` operation includes the option of finding and marking all matches within the file.

The `Find Previous` and `Find Next` buttons simply repeat the previous `Find` operation searching either backward or forward. You can use these to either step through matching entries or, if you have multiple source files open, to repeat the `Find` operation on subsequent files.

The `Replace` button executes a `Find` operation but adds the ability to replace the target string with new text. A `Replace` operation can replace a single entry, replace all matching entries, or find a matching entry and query before replacement.

The `Find`, `Find Prev`, `Find Next`, and `Replace` functions are also available on the `Search` pull-down menu.

Bookmark Operations

In a long source file, setting bookmarks at critical points allows you to return to those locations quickly and to rapidly scroll between locations. The toolbar includes four `Bookmark` buttons.



In order from left to right, the first of these four buttons is the `Toggle Bookmark` tool. Clicking this button either sets a bookmark at the cursor position within the source file or clears a bookmark at the cursor position.

The next button is the `Next Bookmark` tool. It allows you to step from one bookmark to the next in the order they appear in the source file. The button to its right is the `Previous Bookmark` tool, which steps through the bookmarks in the reverse order.

Last in this group, the `Clear All Bookmarks` button simply clears all bookmarks from the source file.

The bookmark operations appear only on the toolbar; they are not included as menu options.

Note: Bookmarks are not saved with the file. Once the file has been closed, the bookmarks are cleared.

Tools

The `Dialog Editor`, `Compile`, `WIL Type Viewer` and `Run` buttons on the toolbar are unique to IDEs.



The `Dialog Editor` button (the first of the four from left to right) starts the WIL Dialog Editor, which will be discussed later in this chapter.

The second button, for the `Compiler` tool, may or may not be available on your system. If this button is grayed-out, you have the interpreter version of WinBatch. As explained at the beginning of this chapter, this does not prevent you from creating applications using WinBatch; you just cannot produce compiled versions of these applications that run as stand-alone executables.

Introduction to Programming

The third button simply launches the `WIL Type Viewer`. See `WILTypeViewer.chm` for details.

The final button in this group, `Run`, is available in both versions of WinBatch. It simply executes the script (program) in the current source file.

These functions are all available as menu selections as well as toolbar buttons.

Whether you have WinBatch installed with or without the compiler option really doesn't matter during development. Until you're absolutely certain that your application is finished, you're going to use the `Go` button from the toolbar or the `Debug` option from the `Debug` menu to execute the application. The `Run` button on the toolbar (and the `Run application name` option on the `Debug` menu) executes your application in a non-debug mode. This option is not recommended until you are relatively sure that your script will function reasonably well.

Debugging Tools

The final group on the toolbar contains buttons for the application debugging tools. During application development, the tools in this group will be your most useful and most important assets for testing and debugging your application code.



The `Go` button (the first from the left) is similar to the `Run` button described in the previous section, in that they both initiate execution of the program being developed. The difference is that the `Go` button starts execution in a debug mode, allowing you to use breakpoints to halt execution, to perform step execution, and to watch variables and values during execution. In brief, the `Go` function permits you to examine the innards of the application during operation.

The second button, `Stop Debugging`, is used to halt (abort) execution. This is useful after you find a point where an error occurs or a point where you would like to revise the code.

The third button, `Step Into`, executes the application step by step, with a marker showing each line of code as it executes. At the same time, the `watch window` lists the variables in use and their values, allowing you to see precisely what is happening at each point in the program. When a subroutine is reached, the `Step Into` function traces execution into the subroutine, returning to the calling procedure when the subroutine is completed.

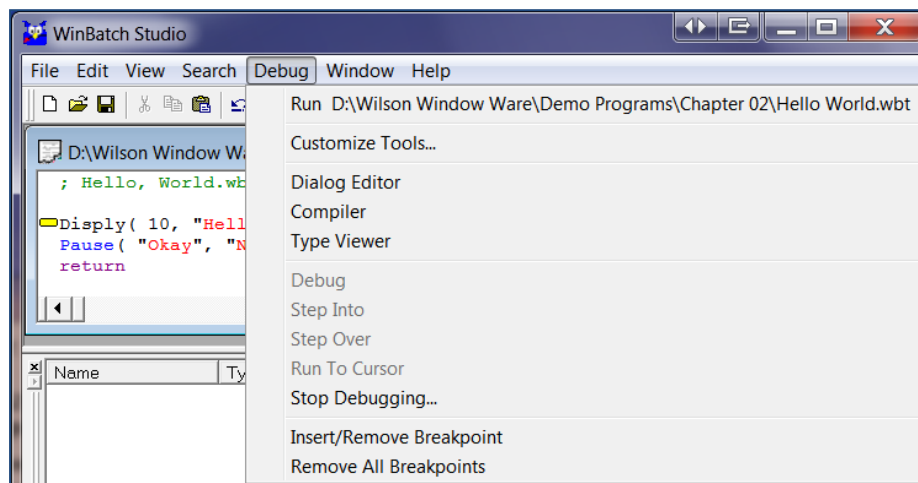
Program *variables* are used to store information or values for use by the application. Variables are discussed in [Chapter 4](#). *Subroutines* are sets of instructions that are called by name from a main program. Subroutines are the topic of [Chapter 7](#).

The fourth button, *Step Over*, is the same as *Step Into*, except that execution does not trace into any called subroutines.

The next button, *Insert/Remove Breakpoint*, sets or clears a breakpoint in the application code at the cursor position. During debug operations, execution will be suspended when a breakpoint-flagged operation is reached. The button to the right of *Insert/Remove Breakpoint* is the *Clear All Breakpoints* tool, which simply clears all breakpoint settings.

The final debug operation button, *Toggle Watch Window*, shows or hides the watch window. This window is used to display information about variables and values during application execution.

All of the debugging tools are also available on the following *Debug* pull-down menu:



In general, using the debugging tools is a multiple-step process. If you do not set any breakpoints in the application, when you use the *GO* function, the script will simply begin execution and run normally. This is subject, of course, to unexpected errors in the program. At this point, any experienced programmers who are reading this book probably have a wry smile on their faces, in response to the words “unexpected errors” in the previous sentence. For one, the phrase is a virtual oxymoron—if an error was expected, we would have done something about it already, right? So, most often, when developing an application, you’ll simply run it and watch to see what goes haywire and try to guess why. Then, after getting a rough idea of where things are going wrong, you’ll usually set a breakpoint in the code somewhere before things crash. And, after setting the breakpoint, you’ll execute the application again, letting it run to the breakpoint and then stepping through looking for our error or errors, as the case may be.

We will discuss debug operations in detail in [Chapter 15](#), after you have learned more about creating applications.

Summary

This chapter has offered an introduction to a number of important topics, beginning with the WinBatch IDE, which provides your workspace and tool room for application development. The WinBatch Studio introduction including creating a simple [Hello World](#) program and a tour of the IDE features and tools.

Now that you have been introduced to the WinBatch IDE—your basic workbench—it's time to move on to creating an application that goes somewhere beyond the simple [Hello World](#) example. To do this, in [Chapter 3](#), we'll continue the discussion by looking at dialog boxes and the Dialog Editor.

CHAPTER 3 : DIALOGS AND THE DIALOG EDITOR

HOLDING A FORMAL CONVERSATION

dialog – a window in a GUI-based application used to present a message, to offer controls for selection, to provide a means for data entry or to request a simple response.

dialog box – a dialog box opens when more information is needed from the user before the program can continue. A dialog box may contain several different elements, including text boxes, list boxes, command buttons and drop-down list boxes, depending on the purpose of the dialog box, but it does not have to contain all these elements at the same time. – The PC User's Pocket Dictionary

In plain English usage, a *dialogue* (note the difference in spelling) refers to a structured conversation between two individuals. In a computer application, a dialog is also a structured exchange where the computer program is presenting information to the user and requesting a response of one sort or another. Dialogs can be very simple, limited to displaying a message and waiting for a simple response such as pressing an [Ok] button, waiting for a choice between [Ok] and [Cancel] buttons or, in other cases, may offer a complex set of choices or options for direct data entry.

While dialog boxes are often seen as auxiliary input devices to supplement programs, dialogs can actually be the primary program interface and the presentation window for the application.

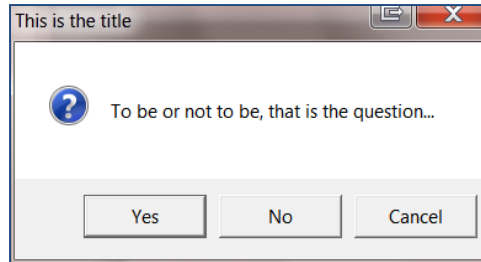
An Introduction to the Dialog Editor

Dialog boxes, or dialogs for short, are integral to graphical user interfaces (GUIs), including the Windows, OS/2, and Macintosh operating systems. For this reason, virtually all computer languages offer some means of creating and formatting dialog boxes; WinBatch is no exception.

Dialog boxes provide a pop-up information display, as well as a method of requesting user input, feedback, or information.

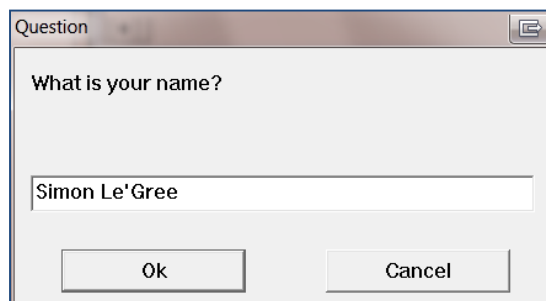
In WinBatch, several predefined dialog boxes offer simple input tools. For example, the `AskYesNo` dialog poses a simple question and offers three buttons for response, as shown in the following dialog.

Introduction to Programming



[AskYesNo.wbt](#)

Another example of a predefined dialog box is the `AskLine` dialog, which presents a text box for a string (character) response as shown here.



[AskLine.wbt](#)

Predefined Dialogs

Since both the `AskYesNo` and `AskLine` dialogs are predefined, all that is required to display either is to ask, as:

```
Answer = AskYesNo( "This is the title", "To be or not to be, that is  
the question ..." )
```

— or —

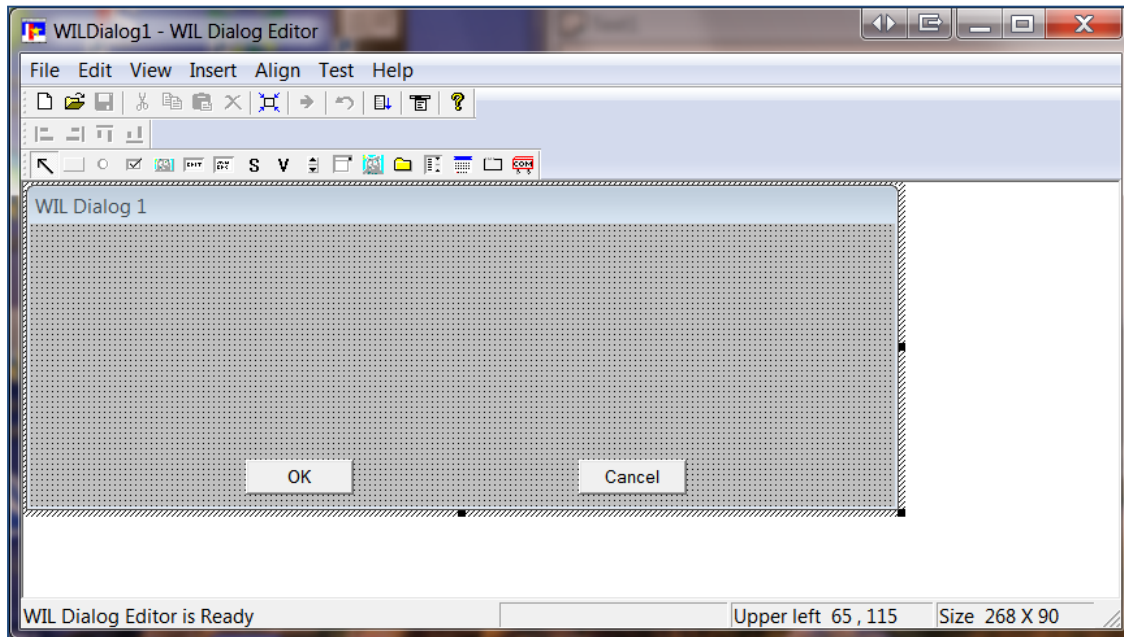
```
sName = AskLine( "Question", "What is your name?", "Simon Le'Gree" )
```

Other predefined dialogs are provided by WinBatch through the `AskColor`, `AskDirectory`, `AskFileName`, `AskFileText`, `AskFont`, `AskItemList`, `AskPassword` and `AskTextBox` functions.

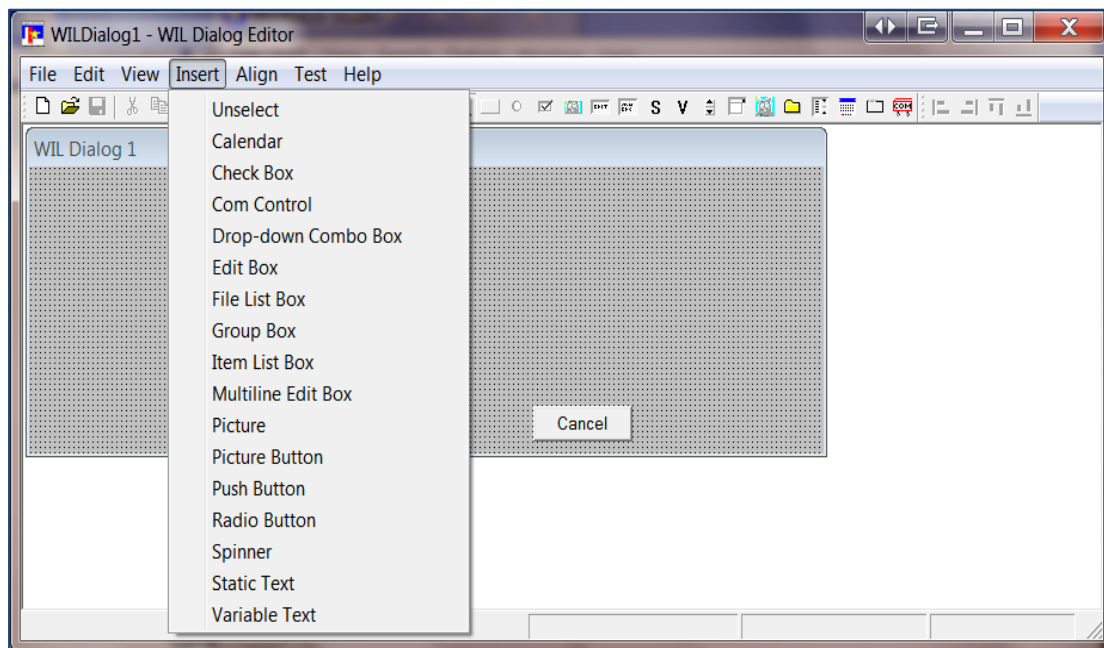
WIL Dialog Editor

More often, application programmers want to create their own dialogs rather than using predefined ones. The easiest way to create dialogs with WinBatch is to use the WIL Dialog Editor to graphically create dialog format instructions. (We'll cover the other ways to create dialogs after this introduction to the WIL Dialog Editor and dialog components.)

To start the WIL Dialog Editor, select it from the WinBatch Studio's Debug menu or from the toolbar. The WIL Dialog Editor window appears with a blank dialog box



After entering a name for the dialog box and setting any other global options desired, you can proceed to customize your dialog box by adding the elements, or *controls*, that you want this dialog to have. Controls include the buttons, boxes, and text that appear in the dialog box. To insert a control, select Insert from the WIL Dialog Editor's menu bar and a menu will appear offering a choice of controls.

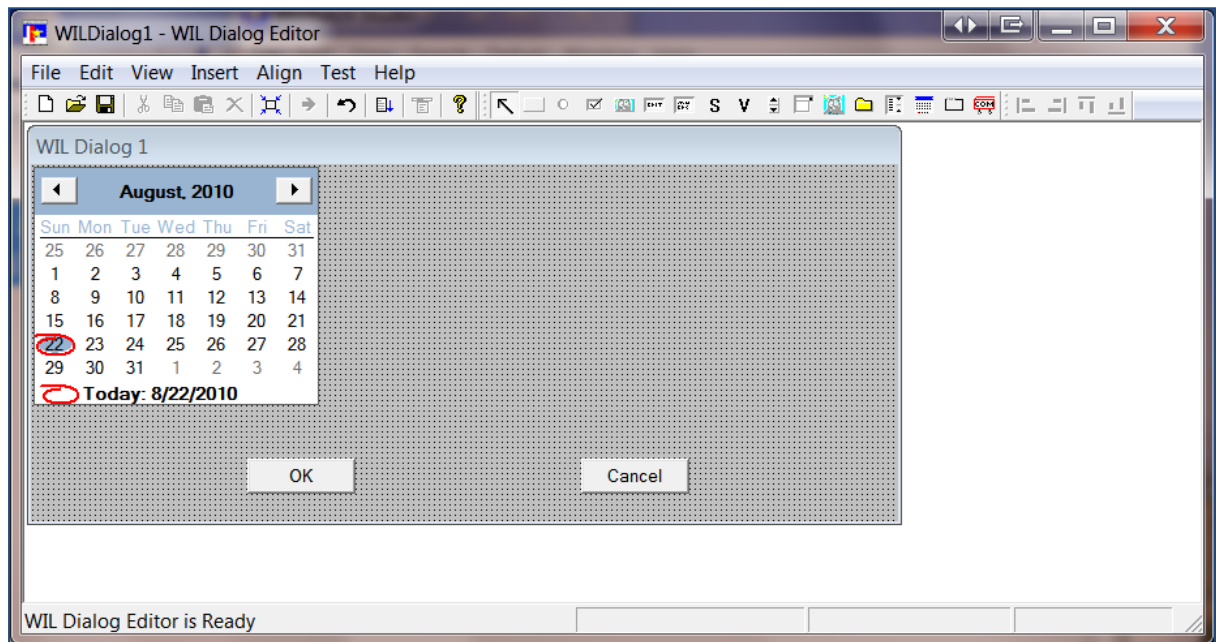


These can also be selected from the toolbar by clicking on an icon for the desired control.

Introduction to Programming

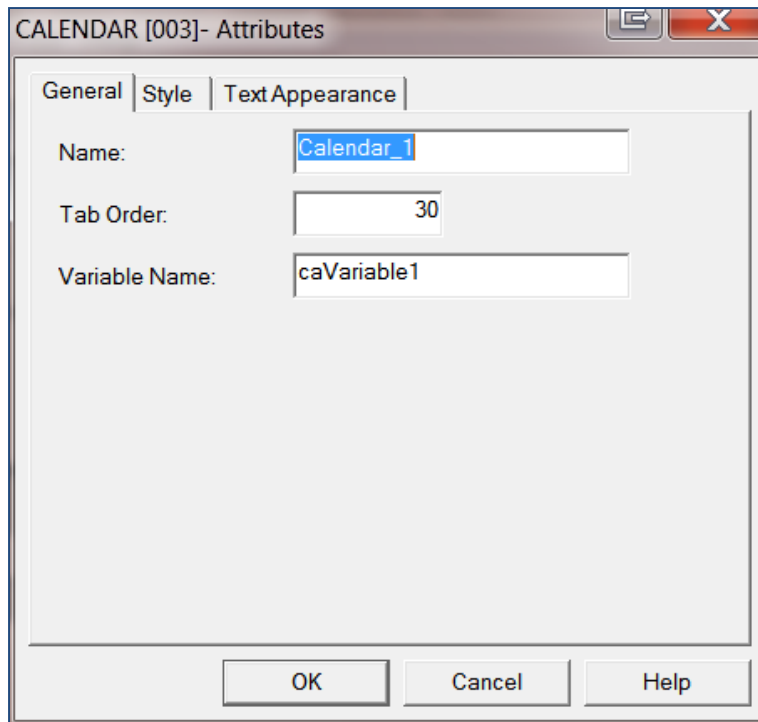
You set both the dialog caption – the name that will appear in its title bar – and the dialog name – the name used in the WinBatch program to display the dialog box – by double-clicking anywhere in the dialog (shaded portion) except on a control or other dialog object. In response, the Dialog Attributes and Control Defaults appears allowing you to modify attributes of the dialog box as a whole.

In the example following, a calendar control has been added to the dialog box. The calendar is created with default attributes and shows the current date – August 22, 2010 – both as the month and year at the top with the day of the month circled as well as the month/day/year at the bottom.



If the dialog box you are creating already contains a control that can appear only once in a dialog, such as a file list box, that control choice will be grayed-out and cannot be selected.

Each control can be modified – right click on the control and select Attributes – to bring up the Attributes dialog specific to the selected control.



As the Attributes dialog box indicates, you can assign a Label (name), the tab order (the order in which the tab key steps through controls) and a variable name to each control. We will explain the requirements for defining each type of control after looking at an example.

Constructing a dialog

The simplest way to construct a dialog is to begin by creating the individual controls – using the Insert menu option and the Attributes dialog to create each control – without worrying about the overall layout dialog.

Once you’ve created the basic controls, the mouse can be used to arrange and/or resize the controls as desired. Also, at the same time, the dialog as a whole can be resized.

Thus, if you begin with a large dialog to allow yourself adequate space for arranging your controls, you can do a rough layout, then – subsequently – you can resize the individual controls, optimize the layout and resize the dialog as needed.

To change a control after it is created; simply double-click on the control to bring up the Control Attributes dialog.

To remove a control from the dialog, click on the control and then hit the DELETE key.

Finally, when you are satisfied with the layout of the dialog, save your work.

Saving a dialog

To save a dialog, you have several options.

First, you can save your work as a separate .WBT (WinBatch scripT) file. This will be a flat-ASCII (text) file which contains the instructions for the dialog caption and the individual controls.

Second, you can also save your work to the clipboard and then paste the dialog format instructions directly into your .WBT script file (application source code). This is the simplest method since the dialog script appears within your program script making it easy to refer to or to modify as desired. This is also the format which will appear in virtually all of the demo programs in this book.

Note that the saved dialog format instructions will contain the dialog version header information, the layout instructions for the dialog controls and, last, the default command for executing the dialog as:

```
ButtonPushed = Dialog( "MyDialog" )
```

Editing a dialog layout

If the dialog has been saved as a .WBT file, then the file can be opened by the dialog editor using the File / Load command.

Alternately, the dialog format instructions can be copied from your .WBT script file (to the clipboard) and then copied from the clipboard into the dialog editor using File / Load from Clipboard.

The dialog format instructions can also be edited directly within the .WBT script.

A Sample Dialog: [WILDialog.wbt](#)

Below, the WIL Dialog shows an example dialog containing examples of most of the control types.



Notice that several of the dialog controls (at left) have been assigned non-default text colors by changing the individual attribute values for each.

The [WILDIALOG.wbt](#) example produces the following script:

```
DirChange( DirScript() )
ibVariable1 = "Red":@tab:"White":@tab:"Blue":@tab:"Green":@tab:
"Black":@tab:"Gray":@tab:"Orange":@tab:"Yellow":@tab:"Mauve":@tab:"Char
truse":@tab:"Peach":@tab:"Apricot"
MyDialogFormat=`WWWDLGED,6.2`
MyDialogCaption=`WIL Dialog`
MyDialogX=-1
MyDialogY=-1
MyDialogWidth=392
MyDialogHeight=332
```

Introduction to Programming

```
MyDialogNumControls=028
```

```
MyDialogProcedure=`DEFAULT`
```

```
MyDialogFont=`DEFAULT`
```

```
MyDialogTextColor=`DEFAULT`
```

```
MyDialogBackground=`DEFAULT, DEFAULT`
```

```
MyDialogConfig=0
```

```
MyDialog001=`023,009,180,192,GROUPBOX,"GroupBox_1",DEFAULT,"GroupBox",D  
EFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
MyDialog002=`039,023,038,010,RADIOBUTTON,"RadioButton_1",rbVariable,"Ra  
dioButton",1,20,DEFAULT,"Microsoft Sans  
Serif|5325|40|34","255|0|0",DEFAULT`
```

```
MyDialog003=`039,053,038,008,CHECKBOX,"CheckBox_1",cbVariable1,"CheckBo  
x",1,30,DEFAULT,"Microsoft Sans Serif|5325|40|34","0|255|0",DEFAULT`
```

```
MyDialog004=`039,075,030,010,EDITBOX,"EditBox_1",ebVariable1,"EditBox",  
DEFAULT,40,DEFAULT,"Microsoft Sans Serif|5325|40|34","0|0|255",DEFAULT`
```

```
MyDialog005=`039,101,038,010,STATICTEXT,"StaticText_1",DEFAULT,"StaticT  
ext",DEFAULT,50,DEFAULT,"Microsoft Sans  
Serif|5325|40|34","0|255|255",DEFAULT`
```

```
MyDialog006=`039,125,036,010,VARYTEXT,"VaryText_1",vtVariable1,"VaryTex  
t",DEFAULT,60,DEFAULT,"Modern|5632|40|65330","128|128|128",DEFAULT`
```

```
MyDialog007=`037,147,152,046,MULTILINEBOX,"MultiLineBox_1",mlVariable1,  
"MultiLineBox",DEFAULT,70,DEFAULT,"Microsoft Sans  
Serif|5325|140|34","255|0|255",DEFAULT`
```

```
MyDialog008=`143,023,048,020,PICTUREBUTTON,"PictureButton_1",DEFAULT,"P  
ict button 1",2,80,DEFAULT,DEFAULT,DEFAULT,"buddha_figure.bmp"`
```

```
MyDialog009=`143,053,048,023,DROPLISTBOX,"DropListBox_1",dlVariable1,DE  
FAULT,DEFAULT,90,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
MyDialog010=`143,075,046,010,SPINNER,"Spinner_1",spVariable1,"1",DEFAUL  
T,100,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
MyDialog011=`143,101,044,032,PICTURE,"Picture_1",DEFAULT,"Picture",DEFA  
ULT,110,DEFAULT,DEFAULT,DEFAULT,DEFAULT,"buddha_figure.bmp"`
```

```
MyDialog012=`261,013,100,032,FILELISTBOX,"FileListBox_1",flVariable1,"W  
ILDIALOG.wbt",DEFAULT,120,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
MyDialog013=`263,067,100,032,ITEMBOX,"ItemBox_1",ibVariable1,DEFAULT,DE  
FAULT,130,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
MyDialog014=`263,119,100,080,CALENDAR,"Calendar_1",caVariable1,DEFAULT,  
DEFAULT,140,DEFAULT,DEFAULT`
```

```
MyDialog015=`025,223,336,074,COMCONTROL,"ComControl_URL",DEFAULT,"http:  
//www.winbatch.com",DEFAULT,150,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
MyDialog016=`103,023,034,008,STATICTEXT,"StaticText_PictureButton",DEFA  
ULT,"PictureButton",DEFAULT,160,DEFAULT,"Microsoft Sans  
Serif|5632|40|34","0|128|0",DEFAULT`
```

```

MyDialog017=`103,053,036,008,STATICTEXT,"StaticText_DropListBox",DEFAULT
T,"DropListBox",DEFAULT,170,DEFAULT,"Microsoft Sans
Serif|5632|40|34","128|128|0",DEFAULT`

MyDialog018=`103,075,036,008,STATICTEXT,"StaticText_Spinner",DEFAULT,"S
pinner",DEFAULT,180,DEFAULT,"Microsoft Sans
Serif|5632|40|34","128|0|0",DEFAULT`

MyDialog019=`103,103,032,012,STATICTEXT,"StaticText_Picture",DEFAULT,"P
icture",DEFAULT,190,DEFAULT,"Microsoft Sans
Serif|5632|40|34","0|255|0",DEFAULT`

MyDialog020=`217,025,028,012,STATICTEXT,"StaticText_FileListBox",DEFAULT
T,"FileListBox",DEFAULT,200,DEFAULT,"Microsoft Sans
Serif|5632|40|34","0|0|128",DEFAULT`

MyDialog021=`217,075,028,012,STATICTEXT,"StaticText_ItemBox",DEFAULT,"I
temBox",DEFAULT,210,DEFAULT,"Microsoft Sans
Serif|5632|40|34","128|0|128",DEFAULT`

MyDialog022=`217,147,032,012,STATICTEXT,"StaticText_Calendar",DEFAULT,"
Calendar",DEFAULT,220,DEFAULT,"Microsoft Sans
Serif|5632|40|34","0|128|128",DEFAULT`

MyDialog023=`025,207,044,012,STATICTEXT,"StaticText_COMCONTROL",DEFAULT
,"COMControl",DEFAULT,230,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

MyDialog024=`000,000,000,000,MENUBAR,"Dialog_Bar"`

MyDialog025=`000,000,000,000,MENUITEM,"mbi1_Help","Dialog_Bar","Help",D
EFAULT,10,DEFAULT`

MyDialog026=`000,000,000,000,MENUITEM,"mbi2_About","mbi1_Help","About",
DEFAULT,10,DEFAULT`

MyDialog027=`121,303,044,014,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,
240,32,DEFAULT,DEFAULT,DEFAULT`

MyDialog028=`235,303,044,014,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Ca
ncel",0,250,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed = Dialog("MyDialog")

```

In the actual script, each control specification appears as a single line, the separate fields being comma-delimited for identification.

The syntax for a dialog format specification begins with a format identifier:

```
MyDialogFormat=`WWWDLGED,6.2`
```

This line provides the crucial information that the format used is the WIL interpreter version 6.2, a format which also imposes a limit of 30 characters for variable names.

The next section defines the format for the dialog box, beginning with the caption for the dialog box (WIL Dialog) and followed by the screen position, the size, the number of

Introduction to Programming

controls in the dialog (28) and the global attribute settings for procedure, text font, text color, background color:

```
MyDialogCaption=`WIL Dialog`  
MyDialogX=-1  
MyDialogY=-1  
MyDialogWidth=392  
MyDialogHeight=332  
MyDialogNumControls=028  
MyDialogProcedure=`DEFAULT`  
MyDialogFont=`DEFAULT`  
MyDialogTextColor=`DEFAULT`  
MyDialogBackground=`DEFAULT, DEFAULT`  
MyDialogConfig=0
```

The font and text colors can be customized for individual control objects by setting the attributes for specific controls. The background attribute specifies the .bmp image displayed by the PICTURE and PICTUREBUTTON controls.

This NumControls specification for the number of controls is important. If you edit a dialog script after it has been created and change the number of controls, you must also change the number specified in the NumControls line of the script. If you add controls and don't change the specification to match, the active dialog will display only the number of controls noted in this section of the format specification. On the other hand, if you increase the NumControls value beyond the number of controls actually defined, an error will occur when WinBatch fails to locate the correct number of controls.

Dialog Controls

Eighteen types of controls can be used in dialog boxes: CALENDAR, CHECKBOX, COMCONTROL, DROP LIST BOX, EDIT BOX, FILE LIST BOX, GROUP BOX, ITEM BOX, MENU BAR, MENU ITEM, MULTILINE BOX, PICTURE, PICTURE BUTTON, PUSH BUTTON, RADIO BUTTON, SPINNER, STATIC TEXT, and VARIABLE TEXT.

The dialog controls are defined using this format:

```
<dlg-variable>nn = `x, y, width, height, type, control-name,  
var/license-string, text/pre-selected item/progid/classid/moniker,  
value, tab-order, style, font, textcolor, background
```

<dlg-	The <dlg-variable> field identifies the dialog. In the WILDialog
-----------------	--

variable>	example, this field appears simply as MyDialogCaption at the top and as MyDialog001 through MyDialog028 for each of the controls defined.
nn	the numerical identifier of the control in the dialog. These are sequential numbers, from 001 through 200. (There is a limit of 200 controls in a dialog.)
x	horizontal position where the control appears in the dialog.
y	vertical position where the control appears in the dialog.
width	width of the control.
height	height of the control. This field should appear as DEFAULT for all controls except item boxes, file list boxes and pictures.
type	type identifies the control as one of the eighteen types of controls.
control-name	control-name is the name used to uniquely identify (reference) the control. The control-name must be unique (within the dialog) and cannot be more than 30 characters in length.
var/license-string	var/license-string is the variable name associated with the control; see Dialog in the WIL Reference Help file for details.
text / pre-selected / item / progid / classid / moniker	text: provides a text entry displayed inside or next to the control. pre-selected item: indicates which item in a list or range is the default item for the control. progid/classid: is used with COMCONTROL controls and is the programmatic identifier of an ActiveX, OLE or COM component control that supports integration into an automation client container. moniker: Two special monikers can be used in this attribute to implicitly use the WebBrowser control or MSHTML COM component provided by Internet Explorer.
value	It is the value returned by the control if selected and must be within the range 0..127.
tab-order	tab-order is the sequence in which the tab key steps selection through controls. This can be used to control the order of access independent of screen position and also to determine which of two (or more) overlapping controls appears on top. See Dialog in the WIL Reference Help file for further details.
style	controls the appearance and behavior of a control. Style options can be combined using the bit-wise OR () operator but are easier to set using the Style tab in the control's Attributes dialog where only the relevant options will appear. See Dialog in the WIL Reference Help file for further details.

Introduction to Programming

font	typeface used to display the control (normally default)
textcolor	color used for the text display as an RGB specification (normally default; i.e., black)
background	either the .bmp image used for the picture or picturebutton controls or the background color for a text field or default if neither is specified.

DEFAULT values can be overridden globally (for the entire dialog) or can be assigned individually for each control object.

The values used for control positions and sizes and for dialog positions and sizes are expressed in a unit of measure known as *dialog units*. These units vary according to the system font selected by the user. In brief, the units are:

1 dialog unit width	= ¼ the average width of the system font
1 dialog unit height	= 1/8 the height of the system font
4 dialog units width	= average width of the system font
8 dialog units height	= height of the system font

WIL interpreter restrictions limit variable names to 30 characters in length. A name longer than this may not be recognized correctly.

Pushbuttons <PUSHBUTTON>

As a general rule, selecting a pushbutton in a dialog will close the dialog. However, you can have your program recognize the button, take an action or set a value, and then reopen the dialog to show the modifications.

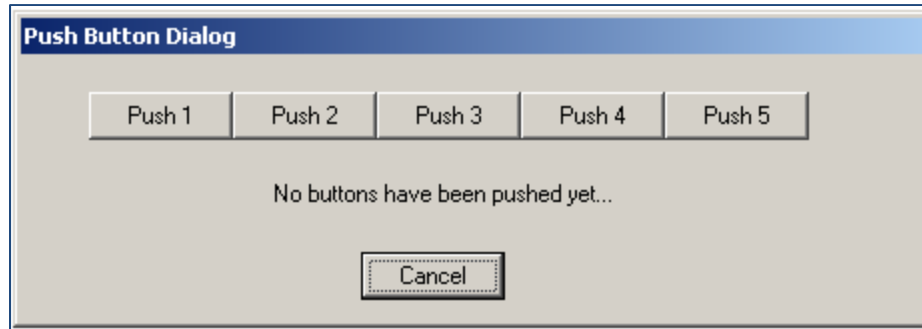
In the [WILDdialog.wbt](#) example, two pushbuttons are defined, one of which returns 0 (zero) as a Cancel button.

```
MyDialog027=`121,303,044,014,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,240,32,DEFAULT,DEFAULT,DEFAULT`
```

```
MyDialog028=`235,303,044,014,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel",0,250,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

Each pushbutton in the example has a different vertical position and returns a different value. The width for each button is specified as 44 dialog units; the height for each button is specified as 14 units. In addition, the `var` field for each is also `DEFAULT`, which is a requirement for pushbutton controls.

As another example, the [PushButton.wbt](#) script displays six pushbuttons:



Clicking on any of these buttons, with the exception of the Cancel button, causes the dialog to re-display, showing which button was pushed. Clicking on the Cancel button closes the dialog (without reopening it) but does pop up a message box reporting the cancellation.

A few rules apply to pushbuttons:

- The `var` field for all pushbuttons should be `DEFAULT`. Pushbuttons are identified only by the `value` returned by the dialog when the button is activated. For proper identification, each pushbutton should return a unique number value.
- Any pushbutton returning a `value` of 0 will either terminate the application or, if a label marked `:CANCEL` is supplied in the program; execution will go to the label. (This element is also demonstrated in the `PushButton` program, where it allows us to report that the Cancel button was selected before terminating.)

Using labels for program flow control is a topic introduced and discussed in [Chapter 8](#).

***Radiobuttons* <RADIOBUTTON>**

Radiobuttons normally appear as groups of two or more and are used to select one item from a group of options. Only one radiobutton in a group can be selected at any time, but one radiobutton must always be selected.

For radiobuttons, grouping is provided by giving each button in a group the same variable name but assigning each button a unique value. By default, when the dialog appears, the button with a value of 1 will already be selected. To have a different radiobutton appear as the initial selection, the variable associated with the group (called the `var` member) is assigned the value of the desired button before the dialog is called.

Changing the radiobutton default

Assume that we have a dialog with three radiobuttons each associated with the variable `rbRadiobutton` but with values 1, 2 and 3. If your program code reads:

```
rbRadiobutton = 2
```

Introduction to Programming

...before calling the dialog as:

```
ButtonPushed = Dialog( "MyDialog" )
```

... the dialog will appear with the second radiobutton as the default selection.

In the [WILDdialog](#) example, the specification for a radiobutton control appears as:

```
MyDialog002=`039,023,038,010,RADIOBUTTON,"RadioButton_1",rbVariable,"Radio  
dioButton",1,20,DEFAULT,"Microsoft Sans  
Serif|5325|40|34","255|0|0",DEFAULT`
```

In this instance, since there is only one radio button displayed, this button is checked by default. Like pushbutton controls, radiobutton controls are specified with a position and width, but the control height is left as `DEFAULT`. Unlike the pushbutton controls, however, the radiobutton has a `value` member identified as `rbVariable`. If there were other buttons in this group, all of them would have the same variable name but would have different values assigned.

Alternately, we might have several groups of radiobuttons where the groupings were created by the variable members associated with each button. For example, [RadioButton.wbt](#) defines three groups of radio buttons, identified by the variables: `rbColor`, `rbSize` and `rbStyle`:

```
MyDialog003=`001,004,041,011,RADIOBUTTON,"RadioButton_1",rbColor,"Red",  
1,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog004=`001,022,041,010,RADIOBUTTON,"RadioButton_2",rbColor,"Blue"  
,2,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog005=`001,041,041,011,RADIOBUTTON,"RadioButton_3",rbColor,"Green"  
,3,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog006=`042,004,042,011,RADIOBUTTON,"RadioButton_4",rbSize,"Small"  
,1,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog007=`042,022,042,010,RADIOBUTTON,"RadioButton_5",rbSize,"Medium"  
,2,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog008=`042,041,042,011,RADIOBUTTON,"RadioButton_6",rbSize,"Large"  
,3,80,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog009=`087,004,041,011,RADIOBUTTON,"RadioButton_7",rbStyle,"Moder  
n",1,90,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog010=`087,022,041,010,RADIOBUTTON,"RadioButton_8",rbStyle,"Class  
ic",2,100,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog011=`087,041,041,011,RADIOBUTTON,"RadioButton_9",rbStyle,"Fancy"  
,3,110,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

Here each group could have a different default radiobutton set as, for example:


```
rbColor = 2
rbSize = 3
rbStyle = 1
```

When the dialog exits, the variable for each group contains the value for the currently selected radiobutton.

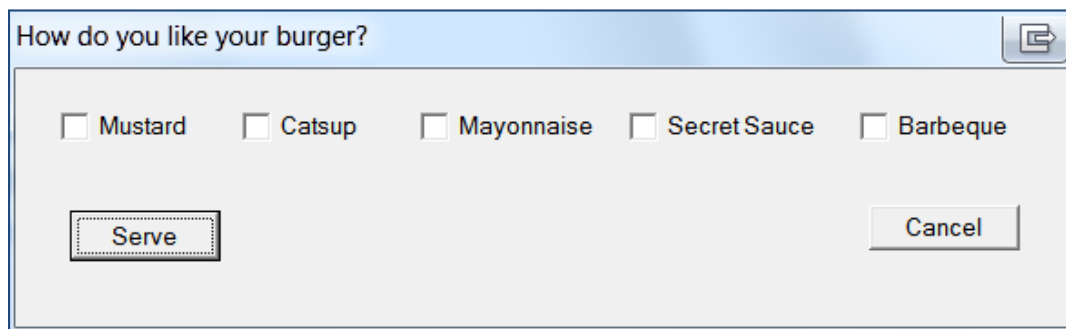
Checkboxes <CHECKBOX>

In the [WILDDialog](#) example, the checkbox specification appears as:

```
MyDialog003=`039,053,038,008,CHECKBOX,"CheckBox_1",cbVariable1,"CheckBo
x",1,30,DEFAULT,"Microsoft Sans Serif|5325|40|34","0|255|0",DEFAULT`
```

Checkboxes may appear singly or in groups, and selections may include none, one, or many. The only restrictions are that each checkbox must have a unique variable. Each checkbox has a value of zero (0) when unchecked and a value of one (1) when checked. By default, all checkboxes are cleared when a dialog opens but they may be set to show as checked by giving them a value of 1.

The [CheckBox.WBT](#) script displays five checkboxes and two pushbuttons:



When the dialog exits, the checkboxes can be polled by checking the value of each variable. If the checkbox is clear, the value will be 0; if set (marked), the value will be the numerical value assigned to the checkbox.

Alternatively, a group of checkboxes may share a single variable name but have different values, with each value a power of two (1, 2, 4, ... , 32, 64, and so on). When a single variable name is used, the value returned will be a combination of the values of the selected checkboxes. After return, individual flag values can be identified by **XORing** the return value with the values assigned to specific checkboxes.

Bit flags and binary XOR operations will be discussed in [Chapter 5](#) under Bitwise Operators.

Edit Boxes <EDITBOX>

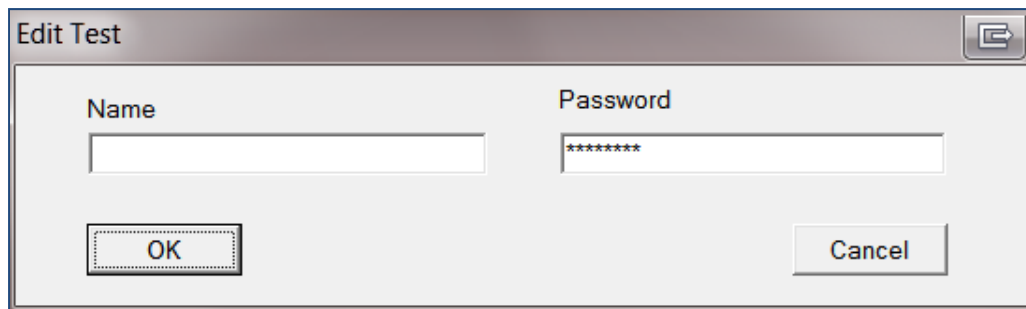
An edit box is a window in a dialog where the user may enter text or numerical values. There are no restrictions on the nature of the text entry, and any entry will be assigned to the associated variable.

The entry for the edit box in the [WILDdialog](#) example is:

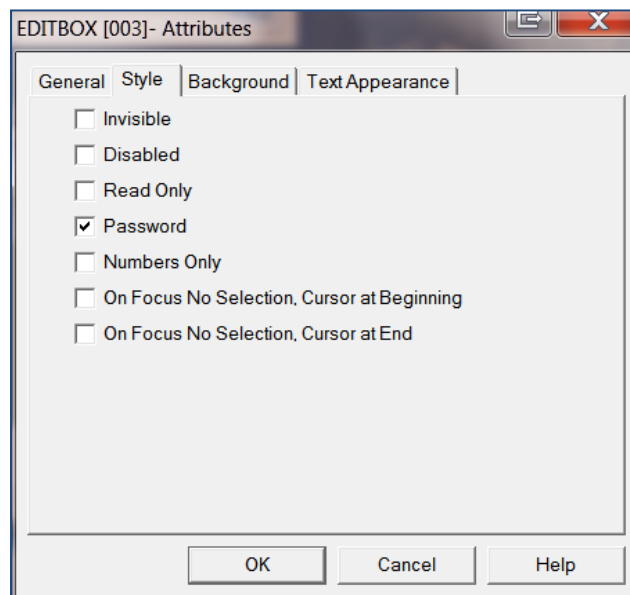
```
MyDialog004=`039,075,030,010,EDITBOX,"EditBox_1",ebVariable1,"EditBox",  
DEFAULT,40,DEFAULT,"Microsoft Sans Serif|5325|40|34","0|0|255",DEFAULT`
```

By default, when a dialog is displayed, edit boxes will be empty. Alternatively, you can assign a string or number value to the edit box before calling the dialog, and that will appear as the default display.

The [EditBox.WBT](#) script displays both a standard edit field and a password entry field:



The two edit boxes shown above are effectively the same control; both are simply edit boxes. The edit box for the password, however, has the Password attribute set (below) in the Style tab of the Attributes dialog.



This setting insures that any entry in the password edit box – including any default entry set in the General tab of the Attributes dialog – is displayed as a series of asterisks.

Alternately, if the variable name for an edit box begins with PW_, the editbox entry is treated as a password.

Note also that the edit box can be set to permit only a numerical entry.

The variable associated with an edit box may be treated as one of the following:

- A string — an entry containing alphabetic characters or nonnumeric punctuation is recognized as a string.
- An integer — an entry containing only numbers is identified as an integer. Note that entries with leading spaces will be identified as integers; those with embedded or trailing spaces will be recognized as strings.
- A floating-point number — an entry containing only numbers and a single decimal point is identified as a floating-point value.

Static (Fixed) Text <STATICTEXT>

A static text control doesn't actually do anything aside from providing a method of displaying descriptive text (labels) or instructions. One static text control specification in the [WILDDialog](#) example is:

```
MyDialog005=`039,101,038,010,STATICTEXT,"StaticText_1",DEFAULT,"StaticText",DEFAULT,50,DEFAULT,"Microsoft Sans Serif|5325|40|34","0|255|255",DEFAULT`
```

A text entry is limited to a single line with a maximum of 150 characters. The `var` field for a static text control is always `DEFAULT`.

If you need to display more than 150 characters as fixed text, such as for a lengthy explanation, use multiple static text controls.

Variable Text <VARYTEXT>

A variable text control is used to display a label or instruction that can be changed by the application by assigning a new string to the associated `var` member. If no assignment has been made, the control defaults to the string entered in the text field when the control was defined. In the [WILDDialog](#) example, the variable text control specification is:

```
MyDialog006=`039,125,036,010,VARYTEXT,"VaryText_1",vtVariable1,"VaryText",DEFAULT,60,DEFAULT,"Modern|5632|40|65330","128|128|128",DEFAULT`
```

Another example of a variable text control appears in the [PushButton.wbt](#) demo (see the “Pushbuttons” section earlier). In that example, the variable text control in the dialog reports which button was pushed.

Introduction to Programming

Item (List) Boxes <ITEMBOX>

Item, or list, boxes are used to display a list of items for selection. The item box control in the [WILDdialog](#) example is specified as:

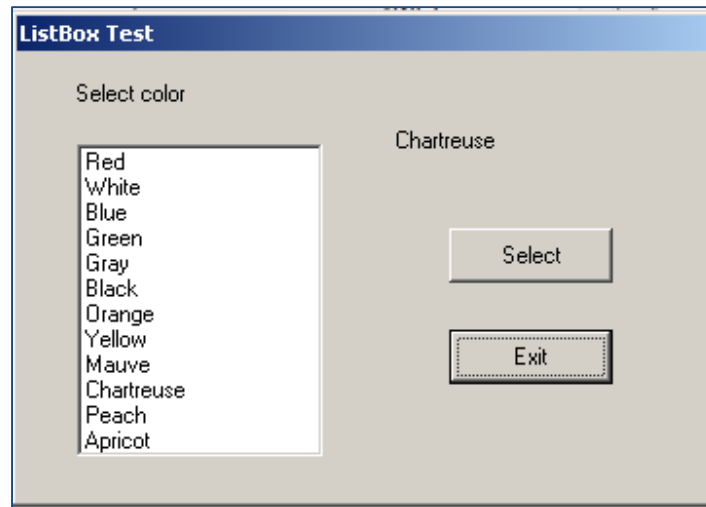
```
MyDialog013=`263,067,100,032,ITEMBOX,"ItemBox_1",ibVariable1,DEFAULT,DE  
FAULT,130,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

Items are presented for selection by assigning a tab-delimited list to the var member associated with the list box control. A tab-delimited list is simply a list with the entries separated by tab characters. In WinBatch Studio, the default tab spacing is three characters and the tab character appears as a small, double-arrow (»). In WinBatch specify @tab when building the string. Here's an example:

```
ibVariable1 = "Red":@TAB:"White":@TAB:"Blue":@TAB:"Green":@TAB:  
"Black":@TAB:"Gray":@TAB:"Orange":@TAB:"Yellow":@TAB:"Mauve":@TAB:"Char  
truse":@TAB:"Peach":@TAB:"Apricot"
```

The list is loaded in the original order, and users may select none, one, or several items from the list. (If a sorted list is desired, use the `ItemSort` function introduced and demonstrated in [Chapter 6](#).)

The [ListBox.wbt](#) script demonstrates a list box entry and selection:



When the dialog closes, item selections from the list box are returned in the var member associated with the list as a tab-delimited list.

While there is no limit to the number of items appearing in a list box, there is a limit to the number of items which can be selected. Selection is limited to 99 items and, if more items are selected, an error will occur.

File List Boxes <FILELISTBOX>

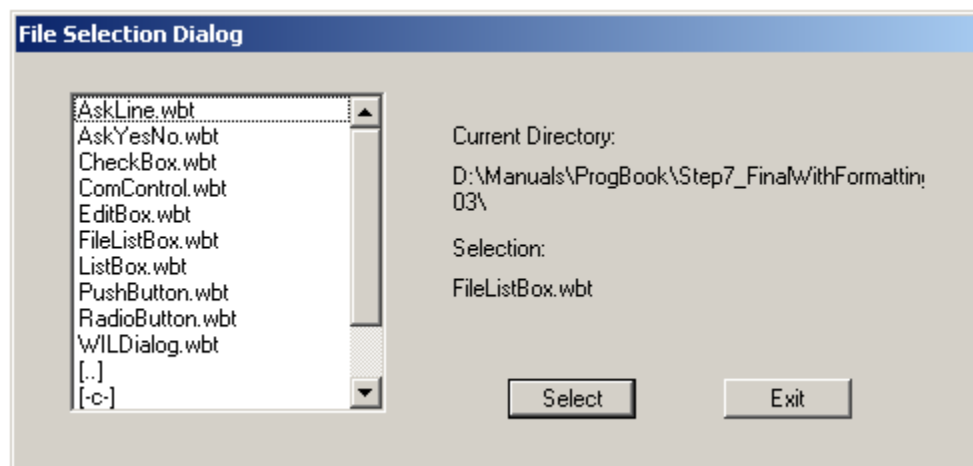
The FILELISTBOX control creates a drive/directory/file selection list, allowing the user to navigate directories and drives and to select a file or directory. The file list box specification in the [WIL Dialog](#) example is:

```
MyDialog012=`261,013,100,032,FILELISTBOX,"FileListBox_1",flVariable1,"W
ILDIALOG.wbt",DEFAULT,120,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

The text field for a file list box is always DEFAULT.

The var member associated with the control returns the selection as a filename. The DirGet function can be used, after the dialog closes, to retrieve the drive/path specification for the active directory.

When the dialog opens, the file list box displays a list of all files matching the file mask. The default file mask is *.* (all files). To restrict the display to just certain files, you can assign a specific file mask to the var member before calling the dialog. This is done in the [FileListBox.wbt](#) demo program:



In this example, a file mask might be used as:

```
flVariable1 = "*.wbt"
```

This mask restricts the file display to show only WinBatch dialog files.

If a file list box is combined with an edit box control using the same var member name, the user can enter a file mask or file specification directly. This will cause the file list box to be redisplayed with the appropriate file list. The [FileListBox.wbt](#) demo also shows how a variable text (VARYTEXT) control using the FILELISTBOX var name can be included to show the drive/directory (path) currently displayed in the list box.

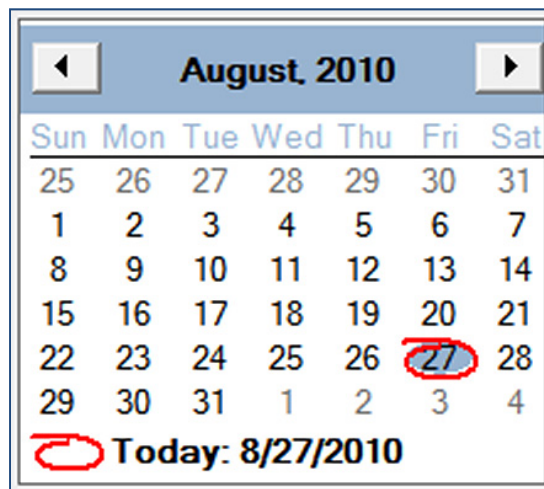
Introduction to Programming

Default behavior for a dialog containing a file list box requires a selection to be made before the dialog can close (except, of course, if the operation is canceled). The statement `IntControl(4,0,0,0,0)` can be included anywhere prior to calling the Dialog function to permit the dialog to close without a file selection.

Calendar <CALENDAR>

The CALENDAR control displays a calendar with the current date shown but allowing the user to select a different date by selecting a day from the displayed calendar or selecting a different month and/or year and then selecting a date. The calendar specification in the [WIL Dialog](#) example is:

```
MyDialog014=`263,119,100,080,CALENDAR,"Calendar_1",caVariable1,DEFAULT,
DEFAULT,140,DEFAULT,DEFAULT`
```



The current date is shown both at the bottom of the calendar and circled on the current date while the gray oval begins with the current date but will show any day selected by the user.

At the top, the two arrows step forward or backward by months; clicking on the year brings up a spinner to go forward or back by years. As the months and years scroll, the selected day of the month remains highlighted. Clicking on the Today or on the date shown at the bottom of the calendar takes you back to the current year, month, and day.

The selected date will be returned in the standard YYYY:MM:DD:HH:MM:SS format.

ComControl <COMCONTROL>

The COMCONTROL is a template to host an ActiveX, OLE, VB or COM component. If the WIL Dialog doesn't support a specific type of control, you can use a third party dialog component using the COMCONTROL. You indicate the specific control by placing a programmatic identifier (progid), class identifier (classid) or moniker in the text attribute

of the COMCONTROL definition string. The COMCONTROL specification in the [WIL Dialog](#) example is:

```
MyDialog015=`025,223,336,074,COMCONTROL,"ComControl_URL",DEFAULT,"http://www.winbatch.com",DEFAULT,150,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

The COMCONTROL supports rendered HTML in WIL Dialogs using either the WebBrowser Control or the MSHTML component of Internet Explorer. In order to use Internet Explorer controls in WIL dialogs you must have at least version 4.0 of Internet Explorer installed on the system. The WebBrowse feature is shown in the [ComControl.wbt](#) example.



Refer to Dialog entry in the WIL Reference Help file for further details about COMCONTROLS.

The DropList Box <DROPLISTBOX>

The DROPLISTBOX is a combination of the EDITBOX and a drop-down ITEMBOX. The DROPLISTBOX control allows the user to enter a value in the EDITBOX or select a suggested entry from the drop-down ITEMBOX.

Just as with a regular ITEMBOX, a tab-delimited list is used to load the drop-down ITEMBOX to the control's VARIABLE attribute; the EDITBOX portion of the control can be given an initial value by placing a string in the TEXT attribute. The user's selection is returned in the VARIABLE attribute.

The DROPLISTBOX specification in the [WIL Dialog](#) example is:

Introduction to Programming

```
MyDialog009=`143,053,048,023,DROPLISTBOX,"DropListBox_1",dlVariable1,DEFAULT,DEFAULT,90,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

The GroupBox control <GROUPBOX>

The GROUPBOX is simply a rectangular outline with a text label in the upper-left corner. A GROUPBOX is used to surround a group of controls serving a single purpose or aim with the function indicated by the label. The GROUPBOX is a visual aide only and any grouping of controls is up to the designer.

The GROUPBOX specification in the [WIL Dialog](#) example is:

```
MyDialog001=`023,009,180,192,GROUPBOX,"GroupBox_1",DEFAULT,"GroupBox",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

The Spinner Control <SPINNER>

The SPINNER control consists of a pair of arrow buttons which can be used to increment or decrement a value in the attached EDITBOX.

The control's VARIABLE attribute is used to set the SPINNER'S range of values. The variable is given a list delimited with a vertical bar (|) and containing two or three values as: "{MINIMUM}|{MAXIMUM}|{STEP}". The MINIMUM and MAXIMUM values must be in the range -32768..36767 and the difference cannot exceed 32767.

The STEP argument is the amount to add (increment) or subtract (decrement). If no STEP argument is specified, the STEP defaults to one (1).

An initial value for the SPINNER can be specified in the TEXT attribute of the control. If you do not specify an initial value or the initial value is outside the min/max range, the initial value will default to the minimum value.

The SPINNER specification in the [WIL Dialog](#) example is:

```
MyDialog010=`143,075,046,010,SPINNER,"Spinner_1",spVariable1,"1",DEFAULT,100,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

The Multi-Line Box <MULTILINEBOX>

The MULTILINE control permits the entry of multiple lines of text. In all other respects, the MULTILINE control behaves like an EDITBOX control.

The MULTILINE specification in the [WIL Dialog](#) example is:


```
MyDialog007=`037,147,152,046,MULTILINEBOX,"MultiLineBox_1",mlVariable1,
"MultiLineBox",DEFAULT,70,DEFAULT,"Microsoft Sans
Serif|5325|140|34","255|0|255",DEFAULT`
```

The Picture Button Control <PICTUREBUTTON>

The PICTUREBUTTON control is a pushbutton displaying an image instead of a text label. The image is set per the PICTURE control.

The PICTUREBUTTON specification in the [WIL Dialog](#) example is:

```
MyDialog008=`143,023,048,020,PICTUREBUTTON,"PictureButton_1",DEFAULT,"P
ict button 1",2,80,DEFAULT,DEFAULT,DEFAULT,"buddha_figure.bmp"`
```

The Picture Control <PICTURE>

The PICTURE control is used to display a bitmap image by specifying the filename and, optionally, the path where the bitmap is found, in the BACKGROUND attribute for the control. Be aware that bitmap will be resized to match the PICTURE control so the control should have an aspect ratio similar to the image to prevent distortion.

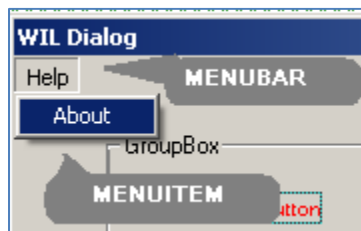
A text string can also be entered in the TEXT attribute and displayed when the image cannot be found.

The PICTURE specification in the [WIL Dialog](#) example is:

```
MyDialog011=`143,101,044,032,PICTURE,"Picture_1",DEFAULT,"Picture",DEFA
ULT,110,DEFAULT,DEFAULT,DEFAULT,"buddha_figure.bmp"`
```

Menus

WIL Dialogs support menus. Menus are made up of both a MENUBAR and MENUITEMS. The [WIL Dialog](#) example contains the Menu option Help |About:



A MENUBAR is a horizontal bar that appears at the top of your dialog just below the title bar. A menu bar contains menu items. Generally, menu items displayed in the menu bar cause dropdown menus to be displayed when selected by the user.

Introduction to Programming

`MENUITEMs` can be displayed on a menu bar or as a menu item associated with a drop-down, context menu or submenu. Dropdown menus are created by placing the name of a `MENUITEM` displayed in the menu bar in the parent attribute of the menu item's template entry. A submenu is started by placing the name of a `MENUITEM` other than a menu bar displayed menu item in the parent attribute.

Context menus are usually activated by right-clicking the client area of a control or dialog. Create context menus by specifying the name of a control in the parent attribute. If you use the `DEFAULT` keyword as the `MENUITEM` parent, the context menu will be associated with the dialog and display when the user right clicks on an 'empty' area of the dialog or on any control that does not already have a system or template supplied context menu.

The MENU specification in the [WIL Dialog](#) example is:

```
MyDialog024=`000,000,000,000,MENUBAR,"Dialog_Bar"\  
MyDialog025=`000,000,000,000,MENUITEM,"mbi1_Help","Dialog_Bar","Help",D  
EFAULT,10,DEFAULT\  
MyDialog026=`000,000,000,000,MENUITEM,"mbi2_About","mbi1_Help","About",  
DEFAULT,10,DEFAULT`
```

Tab Order

In a dialog box, *tab order* is the sequence in which the `Tab` key will step between controls. By default, the first control in the tab order always has the initial focus (as shown by a heavy outline) and will be the control activated when the Enter key is pressed.

Pressing the ESCAPE (or ESC) key is the same as selecting a button with the return value 0, normally the CANCEL or EXIT button.

The tab order is determined by the sequence of numbers in the control identifiers. For example, in the [PushButton.wbt](#) demo, there are 7 controls, defined as:

```
PushButtonDialog001=`085,049,036,012,PUSHBUTTON,"PushButton_Cancel",DEF  
AULT,"Cancel",0,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT\  
PushButtonDialog002=`017,009,036,012,PUSHBUTTON,"PushButton_1",DEFAULT,  
"Push 1",1,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT\  
PushButtonDialog003=`053,009,036,012,PUSHBUTTON,"PushButton_2",DEFAULT,  
"Push 2",2,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT\  
PushButtonDialog004=`089,009,036,012,PUSHBUTTON,"PushButton_3",DEFAULT,  
"Push 3",3,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT\  
PushButtonDialog005=`125,009,036,012,PUSHBUTTON,"PushButton_4",DEFAULT,  
"Push 4",4,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```

PushButtonDialog006=`161,009,036,012,PUSHBUTTON,"PushButton_5",DEFAULT,
"Push 5",5,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PushButtonDialog007=`063,031,088,010,VARYTEXT,"VaryText_1",vtVariable1,
"No buttons have been pushed
yet...",DEFAULT,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

```

The first control, PushButtonDialog001 (CANCEL button) will have focus initially because it has the lowest tab value (10).

PushButtonDialog002 through PushButtonDialog006 each have consecutive tab values and will each receive the focus, one after the other.

Last, PushButtonDialog007 is a varytext control, which cannot be edited or hold the focus at any time, so the tab key takes the focus to the next control in the sequence which ... surprise ... is the CANCEL pushbutton.

To observe the tab order in operation, run the [RadioButton.wbt](#) demo and use the Tab key to step through the controls. Notice how the focus is shown by highlighting (or an outline) as each control becomes the active control. Also notice that in the radiobutton group, only the selected button has the focus.

In the [FileListBox.wbt](#) demo, you can use the up and down arrow keys to change the selection, and the highlight will follow. Similarly, once the file list box has the focus, the up and down arrow keys can be used for selection within the list.

You can change the tab order in any of the demo programs simply by renumbering the tab values of the controls. As a suggestion, however, it may be easiest if you rearrange the control definitions to the order desired and then renumber them sequentially. But be careful, because errors in numbering will cause WinBatch to reject the script or to run the script incorrectly.

Summary

We've discussed using dialogs and how the Dialog Editor works. The descriptions of dialog controls included a variety of examples (source files for each are included in [Appendix A](#)), showing how the various controls function.

As for the Dialog Editor, your best resource for further instruction is quite simple: play with it, add and remove controls and experiment with their placement. A little familiarity will pay dividends in the long run.

In discussing the various dialog controls, we have not talked about the program code used to report or act on the selections made in the various examples. This has not been an oversight. Coverage of program reporting and responses has only been deferred until these elements of programming could be introduced and discussed in proper depth, as will be done in subsequent chapters. For the present, please experiment with the various dialog examples. Feel free to make changes and to observe the results.

Introduction to Programming

Next, in [Chapter 4](#), we will be looking at the basics of the programmer's computer vocabulary: the simple nouns that are used to define data types and variables.

CHAPTER 4 : COMPUTER VOCABULARY – PART I

SIMPLE NOUNS – DATA TYPES AND VARIABLES

"When *I* use a word," Humpty Dumpty said, in a rather scornful tone, "it means just what I choose it to mean— neither more nor less." (Lewis Carroll, *Through the Looking-Glass*)

"The time has come," the Walrus said, "To talk of many things:" (*ibid*)

One of the elements in programming that seems to confuse some people—especially lawyers and English majors—is that words can mean virtually anything we choose them to. What is called an *apple* at one point may suddenly become an *orange* at another point, or even become an animal or a mineral rather than a vegetable at all.

The reason is that in programming we use words as labels, in the manner expounded by H. Dumpty. We may change these labels at various times. Thus, if we so choose, it is perfectly legitimate to write instructions that say:

```
apples = oranges
```

or

```
pears = apples * oranges
```

Granted, these are deliberately “cute” examples, where the names of fruits are used to arbitrarily represent unspecified objects or values. In the second example, `apples` might be an item price, `oranges` the item count, and `pears` the total price, which would then be multiplied by `grapefruit` (the tax rate) to return `lemons` as the sales tax.

The point to recognize here is that the names used for various elements are simply labels; names **do not** always reflect what the elements actually contain or are used to represent. (Of course, as a matter of general practice, we normally attempt to choose names that do reflect the types of values or operations that we are undertaking; we’ll talk more about choosing names later in this chapter.)

The term *data types* refers to the kinds of values your WinBatch programs can use. While the data types themselves are fixed and immutable, the names we use to refer to instances of a particular data type are fluid and infinitely varied. Thus, it is important that you do not confuse the data type (fixed, or *constant*) with the data label (*variable*). Before we look at the various data types, let’s clarify the difference between a variable and a constant.

Variables versus Constants

Suppose we enter the following statement in a program:

```
nApples = 10
```

Because a new value could be assigned to `nApples` at any time, the term `nApples` is referred to as a *variable*. Program variables are used to store information or values for use by the application. Think of a variable as a box used to hold information. To access the information in that box, you refer to the box's label or name.

Variables are also referred to as variable members or as vars in some cases. The terms are identical in meaning and maybe used interchangeably.

The value of `10` currently assigned to the variable `nApples` is called a *constant*, because this value will change only if we rewrite the program. In like fashion, when a string is written as part of the program code, the string itself is a constant, but the `var` member containing the string is a variable.

A *constant variable* (or a variable constant) may sound like an oxymoron ... and in simple English, it is ... and the second form, in any sense, always is. The term *constant variable*, however, is not because we can create a *variable* – an element which has the potential to change – but give it a *constant value* – a value which does not change, ergo, a *constant variable*.

WinBatch Data Types

WinBatch supports five basic data types: integers, floating-point numbers, strings, arrays and huge numbers. The basic data types are constants, but keep in mind that all of these types may also refer to variable members containing values of these types.

Integer Constants

An integer is a whole number (such as 1, 2, 99, 256, 783, and so on) and can be a negative value (–1, –2, –99, and so on). Integers do not include fractions or decimal values. In other words, integers are values we can count on our fingers and toes (if we have enough hands and feet, of course).

Integer values can range from –2,147,483,648 through 2,147,483,648 (roughly plus or minus two billion) or, more compactly, from $-2^{31}+1$ to $+2^{31}-1$.

Floating-Point Constants

Floating-point numbers, called *floats* for short, are decimal or fractional numbers, such as 3.14159 (π), 2.7182 (e), 6.66666..., –4.576E23 (exponential), 0.000013579, or 27e197

(exponential). As you can see, floating-point numbers can be expressed in several formats and can include a plus (+) or minus (–) sign, as well as a decimal point (.) and an E or e (for exponential notation).

Floating-point numbers can range from negative to positive 1.0e+300 (a number too large to represent practically in a conventional numeric format).

☛ Assigning a floating-point constant outside the permitted range may produce unpredictable results.

A floating-point number must begin with a digit. For example, .00002 is not permitted, but 0.00002 is allowed.

String Constants

A string is simply text, such as a word, a sentence, or a paragraph. WinBatch does not support instructions carried over multiple lines but it does allow individual lines up to 2048 characters in length.

In a program, a string must be defined by enclosing it within quotation marks (called *quotes* for short). Three types of quote marks are used: double quotes ("), single quotes ('), and back quotes (`). The back quote is also referred to as an acute accent mark. All of these are found on standard (English) keyboards.

Although each type of quote is permissible, the opening and closing quote must be the same type. Thus, strings can be written as:

```
'a'
`tippie canoe and tyler too - whoop de do`
"The quick red fox jumped over the lazy brown dog"
```

If you want to include quotes within your string, use a quote mark different from the embedded quote to enclose the string. As an example, the [StringTest.wbt](#) demo includes the following two string formats:

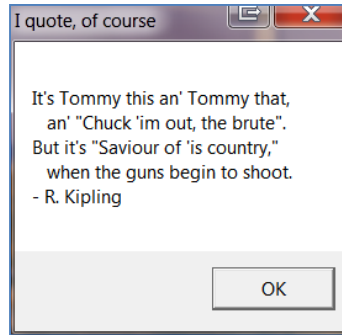
```
sQuote = 'A simple string'
Message( "This one's easy", sQuote )

sQuote = `It's Tommy this an' Tommy that,`:@CRLF:` an' "Chuck 'im
out, the brute".`:@CRLF:`But it's "Saviour of 'is country,"`:@CRLF:`
when the guns begin to shoot.`:@CRLF:`- R. Kipling`
Message( "I quote, of course", sQuote )
exit
```

In this book, many examples contain a line break to fit on the printed page. In the program code, the material must be written as a single line.

Introduction to Programming

In this example, the first string, `sQuote`, has been defined using single quotes and then passed to the `Message` function. In the second example, since both double and single quotes have been used in the string, the string has been enclosed using back quotes. In this case, when `sQuote` is passed as a parameter to the `Message` function, back quotes must be used again to enclose the argument. The predefined string constant `@crlf` (described in the next section) generates a carriage return/line feed to format the displayed string, producing this display:



If single or double quotes were used to enclose, or delimit, this string, WinBatch would become confused and halt with an error. (Open the [StringTest.wbt](#) script and try this yourself.)

The term *delimit* does not mean remove limits but is a contraction of denote limits. Strings are delimited by quotes. In the same fashion, when we speak of a tab-delimited list or a comma-delimited list, the reference is to the use of tab characters or comma characters to denote the separation between list items.

Because strings are important in many respects, [Chapter 6](#) is devoted to the topic of string variables and string operations.

Array

The array data type is used to specify a variable that can be indexed. An array is intended to describe a collection of *elements*.

If you think of a single variable as a box, an array is a shelf or shelves filled with identical boxes where an individual box can be found by its shelf number and its position on the shelf – such as second shelf, fourth box from the left.

Arrays are created using the `ArrDimension` function. An array may have from 1 to 5 dimensions, and can contain at least 10 million total *elements*, although this may be constrained by available memory.

Array elements are referenced with their subscripts enclosed in square brackets. If an array has more than one dimension, the subscripts are separated with commas.

Eg:

```
arrayvar[1]
```



```
arrayvar[1, 1]
arrayvar[0, 5, 2]
```

Array subscripts are 0-based. In other words, the first element in an array can be referenced with the subscript zero: `array[0]`.

Array elements can contain any type of WIL value: string, integer, float, etc. You can have different types of values within an array.

You may not pass an array as a parameter to a WIL function (except for functions which state they accept an array), or use it in any sort of operation.

For example:

```
Message("Value is", arrayvar) ; NOT legal
```

On the other hand, the following are all supported:

```
arrayvar[0] = 5
x = arrayvar[0]
Message("Value is", arrayvar[0])
arrayvar = 5 ; Redefines the array to integer type variable
x = arrayvar ; Creates a second variable that points to the array
```

You can pass arrays to user-defined functions, and you can return arrays with the `Return` command.

When you pass an array name (i.e., not an array element) as a parameter to a function, the array gets passed "by reference". That is, the function receives a pointer to the array, and is therefore able to make changes to it "in place". This is similar to passing a binary buffer handle to a function, where the function is then able to make wholesale changes to the binary buffer.

In contrast, passing an array element (i.e., with a subscript) to a function is like passing a regular string or integer parameter to a function -- it gets passed "by value". i.e. the function receives the value of the array element, but is not able to modify the array itself. By the same token, when you pass a string to a function like `StrUpper`:

```
newstring = StrUpper(oldstring)
```

The function does not modify the variable "oldstring" at all. If you want to modify the existing variable, you can assign to it the return value of the function, eg:

Introduction to Programming

```
mystring = StrUpper(mystring)
array[2] = StrUpper(array[2])
```

WinBatch offers an assortment of functions supporting arrays, including:

- `ArrayFileGet` converts a file to a one-dimensional array.
- `ArrayFileGetCsv` converts a CSV (comma separated value) file into a two-dimensional array.
- `ArrayFilePut` writes a one-dimension array to a file.
- `ArrayFilePutCsv` writes a two-dimension array to a CSV file.
- `ArrayFromStr` accepts a text string and returns a one dimension array with one character per array element.
- `ArrayInsert` Performs in-place insertion of an element into a single dimension array
- `Arrayize` converts a delimited list to an array.
- `ArrayLocate` searches an array for an element that matches a value.
- `ArrayRedim` changes array dimensions in-place.
- `ArrayRemove` performs in-place removal of an element from a single dimension array, or the in-place removal of a row or column from a two dimension array.
- `ArraySearch`
- `ArraySort` performs an in-place sort of arrays with one or two dimensions.
- `ArraySwapElements` swaps elements in an array.
- `ArrDimension` creates an array.
- `ArrayToStr` accepts a single dimension array and returns a text string constructed from the concatenation of each array element.
- `ArrInfo` gets information about an array.
- `ArrInitialize` initializes an array.

The [ArrayTest.wbt](#) demo shows how arrays can be used. First, we declare an array and populate it with ten colors:

```
; Create an array from a list of elements
ColorArray = Arrayize("White, Yellow, Magenta, Chartreuse, Light Blue, Dark
Blue, Green, Brown, Gray, Black", " , " )

; Get the number of elements
nMax = ArrInfo( ColorArray, 1 )
```

Next, the `ArrInfo` function is used to retrieve the number of entries in the array. Granted, we already know the size of the array – in this instance – but this information is not always known in advance ... and, as you will see, may change during execution.

The [ArrayTest.wbt](#) program presents a dialog requesting a number from one (1) to nMax, returning the entry as nNumber.

```
While @TRUE
    nSelectedColor = ""
    sPrompt = "Select a color by entering a number from 1 to " : nMax
    ButtonPushed = Dialog("ArrayTest")
```

Next it tests nNumber to ensure that it falls within the acceptable range (from 1 to nMax):

```
Switch ButtonPushed
    case 1 ; Test button
        If( nNumber > 0 && nNumber <= nMax )
            nSelectedColor = ColorArray[nNumber-1]
            sReport = "The color you selected was " : nSelectedColor
        Else
            sReport = "The color you selected was NOT VALID"
        Endif
    break
```

The chosen array element is stored into nSelectedColor, which is passed back to the dialog as the program loops.

Even though the size of the array was fixed when the array was allocated, we can resize the array using the ArrayRedim function and then add a new entry.

For example, the [ArrayTest.wbt](#) demo adds a new element to the array of colors by first resizing the array and then adding a new, user-defined color as the new array element, thus:

```
    case 2 ; Add button
        Gosub NEWCOLOR
        break
    EndSwitch
EndWhile
exit
```

The line Gosub NEWCOLOR causes the script execution to jump to the label :NEWCOLOR, which causes a new dialog to be displayed prompting the user to add a new color.

```
:NEWCOLOR
NewColorFormat=`WWWDLGED,6.2`
NewColorCaption=`New Color Entry`
```

Introduction to Programming

```
NewColorX=035
NewColorY=053
NewColorWidth=122
NewColorHeight=054
NewColorNumControls=005
NewColorProcedure=`DEFAULT`
NewColorFont=`DEFAULT`
NewColorTextColor=`DEFAULT`
NewColorBackground=`DEFAULT,DEFAULT`
NewColorConfig=0

NewColor001=`011,035,038,012,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,
10,32,DEFAULT,DEFAULT,DEFAULT`
NewColor002=`071,035,038,012,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Ca
ncel",0,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
NewColor003=`021,001,080,012,VARYTEXT,"VaryText_1",sNewColorPrompt,DEFA
ULT,DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
NewColor004=`013,017,038,012,STATICTEXT,"StaticText_1",DEFAULT,"Enter
color",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
NewColor005=`057,017,050,012,EDITBOX,"EditBox_1",sNewColor,DEFAULT,DEFA
ULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

nNext = nMax + 1
sNewColorPrompt = "The new item will be #" : nNext
sNewColor = ""
While @True
    ButtonPushed=Dialog("NewColor")
    if sNewColor == "" then continue ; loop until color is specified
    nMax = nNext
    ArrayRedim( ColorArray, nMax )
    ColorArray[nMax-1] = sNewColor
    break
Endwhile
return
```

Notice that the new entry is assigned to the element `nMax-1`, not to `nMax`. All arrays are indexed from 0, not from 1 so the first element in the array (element 1) actually has an index value of zero; thus the last (new) element in the array is indexed as `nMax-1` where `nMax` is the total number of elements (or entries).

```
ColorArray[nMax-1] = sNewColor
```

Likewise, attempting to access the array element `nMax` – which does not exist – will result in an error.

Huge Numbers

Huge number is a special data type. It is a long decimal number string, which may represent a number too large to be converted to an integer. This value cannot be modified with standard arithmetic operations, it requires the use of the Huge Math extender. Extenders are discussed in [Chapter 16](#).

The Huge Arithmetic Extender contains a few functions which allows simple high-precision math on extremely large numbers (up to 2000 digits). For example, the `Hugemath.wbt` adds two extremely large numbers.

```
AddExtender( "WWHUG34I.DLL" )
num1 = "12345678901234567890"
num2 = "98765432109876543210"
ret = huge_Add( num1, num2 )
Message( "Result of large number addition", ret )
```

Here are some WIL functions which can return huge numbers:

```
BinaryReadEx
BinaryWriteEx
DirInfoToArray
DirSize
DiskFree
DiskSize
FileInfoToArray
FileSize
FileSizeEx
WinResources
```

It is easy to see why many of these functions need to sometimes return huge number data types; many modern day computers support very large disk and file sizes.

Predefined Constants

In addition to constants composed of the data types, WinBatch also offers a variety of built-in constants. The predefined constants all begin with the at sign (@) character, as in these simple ones:

Introduction to Programming

@NO
@ARRANGE

@TILE
@WAIT

@TRUE
@STACK

@YES
@FALSE

The names of these constants are case-insensitive, so it does not matter whether they are spelled in uppercase, lowercase, or mixed case. For example, the predefined constant @YES is the same as @yes or @Yes.

For the complete list of predefined constants, see the Windows Interface Language Reference Help file.

In general, predefined constants are used for tests of various types where a return value or a condition is compared to a predefined constant. For example, you might test if an operation has completed successfully, if a condition has been satisfied, or if a specific result has been reported.

The benefits of using predefined constants are convenience and accuracy. For example, suppose that an operation returns a value of 1 if it is successful or 0 if a failure has occurred. These could be tested directly as:

```
if nResult == 1 then...
```

However, it's clearer to say:

```
if nResult == @TRUE then...
```

In this fashion, the code becomes easier to read, and there's less chance for confusion about what is being tested. (The `if` statement and other tests are discussed in [Chapter 8](#).)

In other cases, predefined constants offer greater accuracy and reduce the chance of error (by the programmer) where a test value is not a simple 0 or 1. For example, suppose that we need to test to determine if the middle button of the mouse has been double-clicked. Which would be easier: remembering the value 521 or referring to @MDBLCLICK? (You could also test for the binary flag values, looking for a result of 001000001001, but this is the kind of operation more suitable for binary-oriented machines than humans.)

Many of the values represented by the predefined constants are determined by the operating system. Operating system-supplied API (application program interface) functions return specific values as a response to operation requests.

Predefined String Constants

WinBatch defines four string constants, which are useful for formatting text or processing text data:

@CRLF	Carriage return/line feed
@CR	Carriage return
@LF	Line feed
@TAB	Tab

The @CRLF constant is used in the [StringTest.WBT](#) demo to break the displayed text over several lines, as shown in the “String Constants” section earlier in this chapter.

While the tab used in the WinBatch editor has a three-character width, the tab spacing for displayed text is determined by the characteristics of the font being used.

Predefined Floating-Point Constants

WinBatch supplies a variety of floating-point constants (20 entries), which can be used for engineering and scientific scripts. Here are some examples of predefined floating-point constants:

Constant	Value	Description
@E (ϵ)	2.718281828459045	Natural (Napierian) log base
@GFTSEC	32.174	Gravitational acceleration (ft/sec ²)
@PI (π)	3.141592653589793	Ratio of the diameter to the circumference of a circle
@RAD2DEG	57.29577951308232	Conversion constant for radians to degrees

WinBatch Program Variables

Many languages require variables to be declared as specific types before they can be used. At the same time, the declaration allocates a space of a specific size—like choosing a box of a particular size—to hold any data assigned to the variable. Once such a declaration is made, only data of the declared type can be copied to the variable (although data of other types may be typecast, or “converted,” to fit). This requirement is known as *strong typecasting*.

In WinBatch, no prior declaration for a data variable is required. A variable used for an integer at one point can become a floating-point value at another point, then be treated as a string a moment later. This flexibility is demonstrated in the [VariTest.wbt](#) demo.

Although variables do not need to be declared, they do need to be initialized before they can be used in any calculation except assignment. That is, a variable can be initialized by

Introduction to Programming

a calculation or function returning a value. In short, *initialization* means assigning a value to a variable, like this

```
a = 1.5
s = "This is a string"
```

Variable Names

In order to use a variable in a program, a variable identifier—a name—is required. In the early days of computers and programming, back when memory was literally worth more than its weight in gold, the practice was to use very short names, such as `a`, `b`, and `c2`. Fortunately, the old limitations on memory have long since vanished. The practice today is to use variable names that are denotative; that is, the name is an indication of what the variable contains or is used for.

It also has become a generally accepted practice to create names that include a prefix identifying the data type. For example, for integer, floating-point, and string variable types, the prefixes `n` (for number), `f`, and `s` might be used:

```
nCount = ItemCount( lListOfItems )
fAverage = nTotal / nCount
sFullName = sFirstName : " " : sLastName
```

Since WinBatch does not require explicit variable declarations and strong typecasting is not imposed, these indicative prefixes may or may not suit your purposes or practices—using them is a matter of personal preference. However, a type prefix can be a useful reminder of what kind of data a variable is intended to contain.

WinBatch variable names are limited to 30 characters in length. Variable names must begin with a letter and cannot contain any punctuation, spaces, or symbols (except underscores). Variable names may contain both letters and numbers.

Examples of legal variable names are:

```
a      n      nResult      MaxIterations      sMessage      fValue3      Total_Cost
```

Unacceptable variable names include:

```
2a      4n2len      ThisNameIsSimplyTooLongToBeAccepted
```

Variable names are not case-sensitive. The following variable names are each considered to be identical:

```
maxValue      MaxValue      maxvalue      MaXvAlUe      mAxVaLuE      mAXvALUE
```


String Variable Conversion

If a string variable is used in a mathematical operation, WinBatch will attempt to convert the string variable to an integer or floating-point value, as in this [VariTest.wbt](#) example:

```
n = "2"           ; this is a string
m = "2.02"        ; also a string representing a floating-point value
a = n * m         ; n is converted to an integer and m to a float
```

In this instance, the conversion can be accomplished because the strings contain only numbers or, in the case of `m`, numbers and a decimal point.

However, if the strings contain alphabetic characters or punctuation, a mathematical operation would result in an error because the strings cannot be converted to numeric values, as in this example:

```
n = "two"         ; this is a string
m = "two point zero two" ; this is also a string
a = n * m         ; we expect this step to fail
```

We'll talk more about string operations in [Chapter 6](#).

Substitution

In WinBatch, substitution allows you to insert the contents of a string variable, or of a numeric variable that can be treated as a string, into a statement before the statement line is parsed. To use substitution, you simply type the name of the string or numeric variable, enclosed within a pair of percent signs (%), into the statement where you want the variable inserted.

To include a single percent sign in a string or numeric variable used in substitution, enter a double percent sign (%%). The result will appear in the string as %.

The WIL language has a powerful substitution feature which inserts the contents of a string variable into a statement before the line is parsed. However, substitution should be used with caution.

Do not use variable substitution, unless absolutely necessary. Many problems can be created, simply by using variable substitution. For example, if you have a large string variable, and you attempt variable substitution, you will receive an error that the line is too long. This occurs because the contents of the string variable are substituted into the line before the line gets executed. No single line can exceed 2048 characters.

Introduction to Programming

It is a better idea, to build a string variable using the function `StrCat` or using the colon string concatenation operator.

```
MyName = "George"
NewStr = "Hello, My name is: " : MyName
Message( "Name Tag", NewStr )
```

If it is absolutely necessary to substitute the contents of a variable in the statement, simply put a percent-sign (%) on both sides of the variable name.

```
mycmd = "DirChange('c:\') " ;set mycmd to a command
%mycmd% ;execute the command
```

Longer Strings

While WinBatch does have a 2048 character limit on strings appearing in the source code, you can create longer string within a program by applying a small trick.

For example, suppose that you have several variables named `str1`, `str2`, `str3`, each of which contains a long string in the 1000 character range. These three strings can be joined together – concatenated – using the `StrCat` function as:

```
newStr = StrCat( str1, str2, str3 )
```

On the other hand, if you were to try to create `newStr` using substitution as:

```
newStr = "%str1% %str2% %str3%"
```

... or even ...

```
newStr = StrCat( "%str1%", "%str2%", "%str3%" )
```

... these operations would fail with the error "line exceeds more than 2048 characters".

For an analogy to aid memory, think of a variable as being like an envelope containing a multipage letter. While `StrCat` can copy the letters into a new envelope without looking at them, using substitution opens the envelope and exposes the contents and this is where the violation occurs.

Lists

Lists are string variables containing one or more delimited substrings. The delimiter character can be a space, comma, or tab character, but only one type of delimiter character can be used in any list (mixed delimiters are not supported). The primary use of a list variable is to supply a list of elements for display in a list box. The [ListTest.wbt](#) shows how to create a delimited list and display it to the user:

```
listFruits = "apple,pear,orange,banana,peach,apricot,plum"
sFruit = AskItemList( "Fruits", listFruits, ",", @SORTED, @SINGLE )
Message( "The selected fruit is:", sFruit )
```

Lists also can be parsed and used for other purposes. The variable `listFruits`, for example, could also be used very much like an array, except lists are one-based. For example, to reference the third element from the `listFruits` list:

```
sSelect = ItemExtract( n, listFruits, "," )
```

Here, if `n` were equal to 3, `ItemExtract` would return the third item from the list and `sSelect` would receive the string assignment “Orange”.

Remember that integer and float variables can also be treated as strings and, therefore, can be included in lists. Further, since a variable such as a list can contain a virtually limitless number of entries, lists offer a very convenient method of storing a large group of values.

Lists are used by several WinBatch functions. The `FileItemize` function, for example, returns a tab-delimited list of file names, and the `WinItemize` function returns a tab-delimited list containing the names of open windows (applications).

Keywords

Every programming language has certain keywords or reserved words that cannot be used except for specific purposes. Keywords have meanings and purposes that cannot be changed, modified, or assigned to new uses. You cannot use these words as variable or function names.

In WinBatch, the reserved keywords include the names of all functions, commands, and predefined constants. Examples include the terms `Average`, `About` and `@TRUE`, each of which is the name of a WinBatch function or predefined constant.

Summary

Now that you've been introduced to data types, constants, and variables, you should have some grasp of how data elements are handled in applications. Remember, however, that we are still introducing the basic vocabulary of programming, and that the types of manipulation shown thus far barely scratch the surface of our repertory. In this chapter, you've learned the basic nouns of the language. You are, at this stage, able to point at something and name it.

In our next step, your vocabulary will be expanded as we introduce operators and operations. These represent the basic verbs needed to perform actions.

CHAPTER 5 : COMPUTER VOCABULARY – PART II

SIMPLE VERBS – OPERATORS AND OPERATIONS

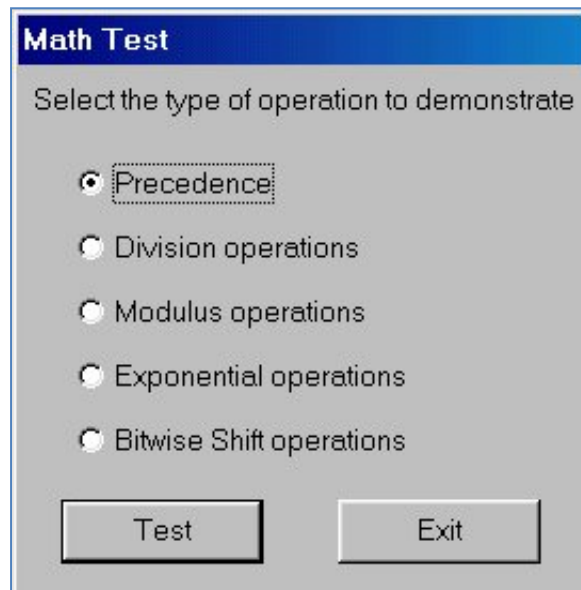
“The materials of action are variable but the use we make of them should be constant.” (Epictetus)

operator \`äp-(ë)-rät-er\ n. 3: a mathematical or logical symbol denoting an operation to be performed.

operation \`äp-(ë)-rä-shen\ n. 5: any of various mathematical or logical processes of deriving one expression from others according to a rule.

Operators are the basic verbs of the computer language. Without operators, we would have no means of instructing our silicon golems to perform even the simplest actions.

In the previous chapters, while introducing the basics of programming, you’ve seen examples of operators and operations. Here, we will fill in the missing information about what the operator symbols represent and the operations they perform. A number of the operators discussed in this chapter are demonstrated in the [MathTest.wbt](#). The opening dialog for [MathTest.wbt](#) appears below:



Each option shown here will demonstrate one or more examples of that type of operation. After each set of examples completes, the selection will automatically step to the next operation type.

Math Operators

Like other computer languages, WinBatch uses two types of operators:

- A *unary operator* is one that requires—operates on—only one object (or operand).
- A *binary operator* is one that operates on two objects.

In some cases, the distinction between unary and binary operators is unimportant; some operator symbols are used for both unary and binary operations. When the difference does matter, for the most part, the use of these operators follows common sense and does not necessarily require esoteric knowledge nor initiation into any secret society, the swearing of blood oaths, nor the wearing of elaborate headpieces with fur and horns.

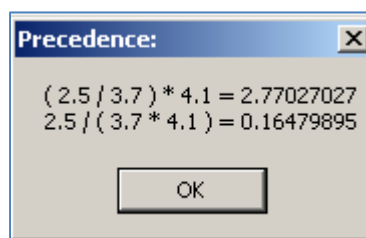
Grouping Operators ()

Parentheses are used to group or collect operations and operands, both visually and to ensure that the evaluation of operands and operators is carried out in a specific order. Expressions within parentheses are always evaluated first, before the expressions outside the parentheses are evaluated. For example, here are two formulas, identical except for the parentheses used to group the operations:

```
a = 2.5  
b = 3.7  
c = ( a / b ) * 4.1  
d = a / ( b * 4.1 )
```

After these operations are carried out, *c* and *d* will not contain the same values. Instead, the first operation (*c*) will report a result of 2.77027027, and the second (*d*) will return 0.164798945.

You can add parentheses even when they are not required. Use them to ensure that operations are carried out in the proper order or to test results to determine if they are being performed correctly. For an example of parenthetical grouping, refer to the [MathTest.wbt](#) demo or to the illustration following where two different sets of parentheses applied to the same figures produce quite different results.



The Assignment Operator (=)

A basic operation is the assignment operation denoted by the equal sign (=). You've already seen this operator repeatedly, in the example above and in previous chapters. When we say:

```
nApples = 3
```

we are assigning a value (3) to the variable `nApples`. In like fashion, the statement:

```
sReport = "This is a string"
```

assigns (copies) a sequence of characters into the memory space indicated by the variable name `sReport`.

The assignment operator is also used to store the results of other operations, as in this example:

```
fTotal = nItems * fItemCost
```

Here, the product of an operation—multiplying `fItemCost` by the number of items (`nItems`)—is stored in the variable `nTotal`.

At this point, you are probably thinking that there is no need to talk anymore about assignment operations. After all, you learned the operations and the notational format back in grade school, right? However, for programming purposes, assignment operations require a bit of additional explanation.

The assignment operator is one of the binary operators because it always requires two operands: a source operand (on the right of the equation) and a target operand (on the left). In the first two examples in this section, the source operands are obvious, but they are not readily apparent in the third example. There, the source operand is not `nItems` nor `fItemCost`. Instead, the source operand is the **product** (or result) of the operation expressed on the right of the equation.

For another example, an operation can be performed without the assignment operator, as in:

```
nItems * fItemCost
```

However, in this format, without the assignment operator to store the results of the operation, the information derived from the action is simply lost. In cases involving mathematical or logical operations, you can save the result of the calculation using the

Introduction to Programming

assignment operator or, optionally, use the result in some other fashion without an explicit assignment. This latter usage will be discussed later in this chapter, in the “Precedence and Evaluation Order” section.

In some languages, an operation without an assignment (such as `nItems * nItemCost`) would produce an error. In this respect, WinBatch is more forgiving, but the practical result is still as if the operation had not occurred.

For some operations, such as calls to functions, using the assignment operator to store the reported result is optional. In such cases, the requested action is carried out whether we query the result or ignore it.

A Difference in Format

One difference between programming and what you learned in school is the order in which formulas are written. In school, you were presented with problems stated as:

$$5 + 3 = _? _$$

Here, you were expected to write the answer to the right of the assignment operator. This convention came about because it suits the convenience of the six out of seven people who are right-handed.

However, in the world of computer programming, this syntax is reversed. For computers, the assignment operator functions right to left:

$$_? _ = 5 + 3$$

Thus, following this format, the variable receiving the result of an assignment must always precede the assignment operator.

The Addition and Subtraction Operators (+ and –)

The well-known addition (+) and subtraction (–) operators work as you would expect, but they also have more than one function as operators. Both can be used as either a unary or binary operator.

As a unary operator—an operator requiring only one operand—the subtraction symbol is called the *arithmetic negation* operator. This title simply means that preceding a variable with the subtraction symbol changes the sign of the quantity stored in the variable. Or,

more accurately, the value in the variable is not itself changed, but any operations involving the variable use that variable's negative value. For example, the expression:

```
fRefund = fItemCost * -nItems
```

uses the negative of the value in `nItems` in performing the subsequent calculation. (Of course, if `nItems` is already a negative value, the subsequent multiplication operation is performed using a positive integer.)

As a binary operator—an operator requiring only one operand—the subtraction symbol functions precisely as you learned in grade school to perform subtraction on both integer and floating-point values.

Remember that any mathematical operation involving a floating-point value as any one of the terms will produce a floating-point result. WinBatch's `Int` function can be used to convert a floating-point value to an integer expression by truncating the fractional portion of the argument.

The plus symbol used as a unary operator is called the *identity* operator and really doesn't do anything. For example, the expression:

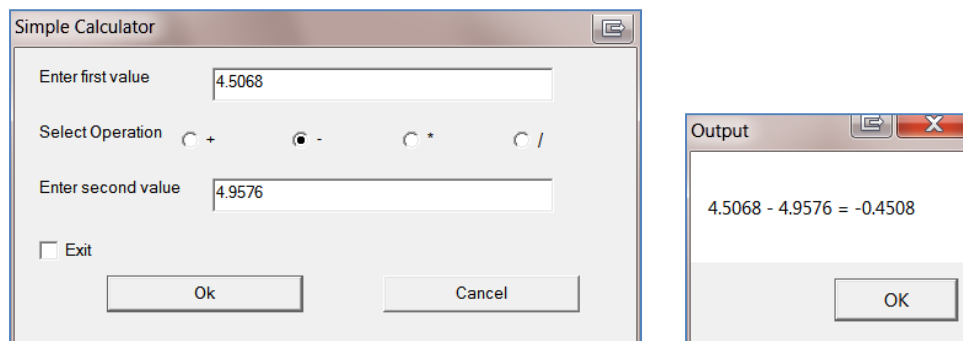
```
nTotal = +nItems
```

is no different than:

```
nTotal = nItems
```

As a binary operator, the plus symbol is used to perform addition on integer and floating-point values.

The demo program [SimpleCalculator.wbt](#) shows the results of simple math operations involving the addition and subtraction operators, as well as the multiplication and division operators. Below, you can see [SimpleCalculator.wbt](#) accepting instructions for a calculation which then shows the results of the operation in a message box.



The Multiplication and Division Operators (* and /)

Like the other operators we have discussed so far, the multiplication (*) and division (/) operators follow essentially the same rules that you learned in grade school.

The multiplication operator has been shown in a number of previous examples, such as:

```
fTotal = nItems * fItemCost
```

If either operand is a floating-point value, the product of the operation will also be a floating-point value.

In like fashion, the division operator could be used as:

```
fAverage = fItemCost / nItems
```

However, division operations are an area where you can easily make mistakes if you don't take the "golem principle" – which was introduced back in [Chapter 1](#) – into account.

When you perform division, sometimes the result should be a fraction. For example, when you divide three into two parts (3 / 2), you know that the result is one and a half (1.5 or 1½). But remember that the computer is literal-minded and not very smart. If you don't express the operation correctly, to the computer, three divided by two is one, not one and a half. For examples, look at the following operations:

OPERANDS	RESULT	NOTES
2 / 3	= 0	integer / integer = integer
3 / 2	= 1	integer / integer = integer
3.0 / 2	= 1.5	float / integer = float
2 / 3.0	= 0.66666667	integer / float = float
2.0 / 3.0	= 0.66666667	float / float = float
6.0 / 3.0	= 2.0	float / float = float

As you can see, if both operands are integers, the result is an integer, **even if this means the value of the operation is truncated.**

To solve this problem, you need to convert either the dividend or the divisor to a floating-point value to ensure that the product of the operation will be a float.

For example, to change the dividend temporarily from an integer to a float, the code could be written like this:

```
fResult = ( nDividend * 1.0 ) / nDivisor
```

The result of the operation will appear as a floating-point value, but with both the operands unchanged.

The Modulus Operator (mod)

The modulus operator (`mod`) performs remainder division; that is, the product of modulus division is the remainder following integer division. Here are two examples:

```
6 mod 3 = 0
```

```
7 mod 3 = 1
```

Because 3 divides into 6 evenly (twice) there is no remainder, and the modulus operation returns 0 (zero) as a result. But for 7 divided by 3, using integer (modulus) division, a remainder of 1 is returned.

The term integer division refers to the fact that the divisor is subtracted from the dividend an integer number of times until the remainder is less than the divisor. This remainder becomes the product of modulus division.

Modulus division can be performed on floating-point values as well as integers, thus:

```
7.53 mod 3.1 = 1.33
```

One convenient use for modulus division is to extract the fractional portion of a floating-point value. This can be accomplished as:

```
7.53 mod 1 = 0.53
```

An alternative approach would be to use the `Int` function to find the integer portion of a floating-point value and then subtract this from the floating-point value.

The Exponential Operator (**)

The exponential, or power, operator (`**`) raises the first operand to the power of the second (for `x ** n`, `x` is multiplied by itself `n` times). Here are two examples:

```
fResult = 2 ** 3 ; ( 2 * 2 * 2 or 23 = 8 )
```

```
fResult = 3 ** 2 ; ( 3 * 3 or 32 = 9 )
```

The exponential operator is not limited to integer values. It is perfectly valid to write:

```
fResult = 2.5 ** 3 ; ( 2.53 = 15.625 )
```

Introduction to Programming

```
fResult = 2.5 ** 3.9 ; ( 2.5 3.9 = 35.64232565 )
```


In like fashion, either operand can be a negative value:

```
fResult = -2.5 ** 3 ; ( -2.5 3 = -15.625 )  
fResult = 2.5 ** -3 ; ( 2.5 -3 = 0.064 )
```

The exponential operator is particularly useful for engineering and scientific applications.

Logical Operators

Logical operators return a result of `TRUE` or `FALSE` where `TRUE` is nominally one (1 or non-zero) and `FALSE` is always zero (0).

 In any Boolean (logical) operation, any non-zero value is recognized as `TRUE`. Thus `TRUE` can be `-5`, `-1.5`, `1`, `2.7`, `1234` or any value except zero.

The logical operators consist of logical AND (`&&`), logical OR (`||`), and logical NOT (`!`). The AND and OR operators are binary. The NOT operator is unary (it affects only one operand). The logical operators are used extensively by programmers to combine tests to make decisions (a topic discussed in [Chapter 8](#)).

Caution: Binary logical operators can be used only for integer variables. Do not apply these operators to floating-point values or strings.

The Logical AND Operator (`&&`)

The logical AND operator requires that both operands (*arguments*) be `TRUE` before a `TRUE` result is returned. Here is an example:

```
if ( a > b ) && ( c < d ) ...
```

In this example, the `if` condition will be satisfied only if both arguments—`a` is greater than `b` (`a > b`) and `c` is less than `d` (`c < d`)—are found to be true. If either (or both) arguments is not true, a `FALSE` result will be returned. The greater-than (`>`) and less-than (`<`) operators used in this example are discussed later in this chapter.

The Logical OR Operator (`||`)

Like the logical AND operator, the logical OR operator tests two operands, or arguments. The difference is that the OR operator returns `TRUE` if *either* operand tests `TRUE`. Here is an example:

```
if ( a > b ) || ( c < d ) ...
```

In this case, the `if` condition is satisfied when either `(a > b)` or `(c < d)` is found to be true. A `FALSE` result will be reported only if *both* arguments are not true.

The Logical NOT Operator (!)

The unary logical NOT operator returns a 0 (`@FALSE`) if the operand is nonzero (or `TRUE`) and returns 1 (`@TRUE`) if the operand is initially 0 (or `FALSE`).

The `@` (at-sign) character is used in WinBatch to identify pre-defined constants. Thus, `@true` is the same as 1 and `@false` is the equivalent of 0 – refer to [Predefined Constants in Chapter 4](#).

Here is an example:

```
If ! ( a == b ) then...
```

If `a` and `b` were equal or identical (`a == b`), the operand term within the parentheses would be `TRUE`, but using the NOT operator reverses the result to be evaluated as `FALSE`, and the condition would not be met. If `a` and `b` were unequal, the test would evaluate as `TRUE`, and the condition would be met. The equality operator (`==`) used in this example is discussed in the next section.

The statement `! (a == b)` could equally well be written `a != b`. The present construction is used simply to illustrate the `not` operator (`!`).

Relational Operators

Relational operators are used to test relationships and to determine if two items are equal (`==`) or unequal (`!=` or `<>`) or if one item is greater than (`>`), greater than or equal to (`>=`), less than (`<`), or less than or equal to (`<=`) another item. Relational operators are always binary, since a relationship requires two elements to be related in some fashion.

The equality and inequality operators apply to numeric operands as well as strings. The remaining relational operators apply only to numeric operands (integers or floating-point values).

The Equality and Inequality Operators (== and != or <>)

The equality operator returns a `TRUE` result if the two operands are equal or identical. Otherwise, it returns `FALSE`. The equality operator can be applied to all data types, including strings. When you use this operator with strings, remember that strings are treated as case-sensitive. Consider this example:

Introduction to Programming

```
s1 = "ThIs Is A sTrInG"  
s2 = "this is a string"  
if( s1 == s2 )...
```

Here, the test will confirm inequality because one string has mixed caps and the other is lowercase.

The inequality operator returns `TRUE` if the two operands are not equal or not identical and returns `FALSE` if they are. Like an equality test, an inequality test can also be applied to strings. Changing the last line in the example above to:

```
if( s1 <> s2 )...
```

or

```
if( s1 != s2 )...
```

returns a `TRUE` result.

The Greater-Than and Less-Than Operators (>, >=, <, and <=)

The greater-than and less-than operators work with numeric operands, as follows:

- The greater-than (>) operator reports `TRUE` if the left operand is larger than the one on the right.
- The greater-than-or-equal (>=) operand reports `TRUE` if the left operand is larger than the one on the right or if the two are equal.
- The less-than (<) reports `TRUE` if the left operand is smaller than the one on the right.
- The less-than-or-equal (<=) operators reports `TRUE` if the left operand is smaller than the one on the right or if the two are equal.

For example:

```
2 > 3 = @FALSE    obviously 2 is less than 3 so the result would be 0 or  
                FALSE  
3 > 2 = @TRUE      since 3 is greater than 2, the result would be 1 or TRUE  
3 >= 3 = @TRUE     since the two values are equal, the result is TRUE
```

Bitwise Operators

Bitwise operators are used to manipulate the series of ones and zeros that the computer employs to represent all types of data. Bitwise operations are normally applied only to integer values. In practice, bitwise operations (and thus, bitwise operators) are rarely used directly by high-level programmers.

Six bitwise operators are provided by WinBatch: the left-shift (<<), right-shift (>>), bitwise AND (&), bitwise OR (|), bitwise XOR or eXclusive OR (^), and bitwise NOT (~). All of these, except the bitwise NOT operator, are binary operators.

Words and Bits

In computer terms, a bit is the smallest unit of memory and can store either a 1 or a 0. Since bits are rather small, we find it more convenient to talk about words where a word is the computer's natural unit of storage. The size of a word can differ depending on the operating system and the language but, for our purposes, we can consider a word as being 32 bits in size.

When we perform bitwise operations, we are manipulating the individual bits within word values. While we will offer illustrations of bitwise operations – following – these will not be discussed in depth. Bitwise binary operations are best left to experienced programmers who, presumably, are familiar with binary and hexadecimal representation and the reasons and purposes for low-level manipulation of values.

Binary Numbers

A binary number is simply a notational format that matches the form in which numbers are stored in the computer – that is: in 1's and 0's. For example, the value 121 in decimal notation is 1 in the one's column, 2 in the ten's column and 1 in the hundred's column.

In binary notation, the information consists of only 1's or 0's whose value depends on the column position. But, instead of a one's column, a ten's column and a hundred's column, the columns are 1, 2, 4, 8, 16, 32, 64, etc., proceeding by powers of 2 (hence binary).

As examples:

1 = 0000 0001

2 = 0000 0010

3 = 0000 0011 (2 + 1)

5 = 0000 0101 (4 + 1)

121 = 0111 1001 (64 + 32 + 16 + 8 + 1)

(For convenience, binary numbers are often written in groups of four.)

Introduction to Programming

Binary numbers can also be used as flags where each position in a number – each bit – is a separate True/False flag (1=True, 0=False). That is, the number as a whole has no meaning but each flag (bit) can control some completely unrelated element.

As an example, suppose that we want to record whether each of a series of items are **a)** solid [0] or hollow [1], **b)** red [0] or green [1], **c)** soft [0] or hard [1], **d)** wet [0] or dry [1], **e)** light [0] or dark [1], **f)** fuzzy [0] or smooth [1] and **g)** vegetable [0] or mineral [1].

Now, assume that we have an item which is hollow [1], red [0], hard [1], wet [0], dark [1], smooth [1] and vegetable [0].

Instead of storing seven different facts about this item, we can store all of this information in a single value as:

hgfe dcba (the least significant bit comes last)

0011 0101 (binary) = $32 + 16 + 4 + 1 = 53$ (decimal)

Later, to find out if an item is smooth, for example, we can simply test the value stored in the flag variable by ANDing it with the value for smooth, thus:

nSmooth = 32

if nFlag & nSmooth then ...

In binary terms:

0010 0000 (nSmooth)

& 0011 0101 (nFlag)

= 0010 0000 since the result is not zero (TRUE), the item is smooth.

Alternately, if the result had been zero, the item would have been identified as fuzzy – see Bitwise AND, OR and XOR Operators.

The Left-Shift and Right-Shift Operators (<< and >>)

The left-shift and right-shift operators are used to shift an integer value's bits to the left or right a specified number of places. A shift operation manipulates the bits making up an integer value. Here is an example that uses the left-shift operator:

```
n = 9
nResult = n << 2
```


This produces a result of 36, just as if n had been multiplied by 4.

For an integer value of 9, the binary representation (the ones and zeros) would be:

```
0000 0000 0000 1001
```

Shifting this value left two places produces:

```
0000 0000 0010 0100
```

which, in decimal format, is 36.

When a left-shift operation occurs, the n leftmost bits are lost and n zeros are added to the right. Likewise, for a right-shift operation, the n rightmost bits are lost and the value is left-padded with zeros.

The [MathTest.wbt](#) demo (see [Appendix A](#)) includes two examples of bitwise shift operations.

The Bitwise AND, OR, and XOR Operators (&, |, and ^)

The bitwise AND operator combines two values to produce a result where the bits set are those that appear in both operands. Here is an example:

```
1001 0001 ( 145 )
0000 1001 ( & 9 )
0000 0001 ( = 1 )
```

The bitwise OR operator also combines values, but the result is a value where the bits set are those that appear in either operand, as in this example:

```
1001 0001 ( 145 )
0000 1001 ( | 9 )
1001 1001 ( = 153 )
```

The bitwise XOR operator, also known as the eXclusive OR, combines values but produces a result where the bits set are those that appear in either but not both operands, as in this example:

```
1001 0001 ( 145 )
0000 1001 ( ^ 9 )
1001 1000 ( = 152 )
```

The Bitwise NOT Operator (~)

The bitwise NOT operator is a unary operator. The NOT operator inverts the bits in the single operand it is applied to. Here is an example:

```
1001 1001                      ( = 153 )  
~1001 1001 becomes 0110 0110 ( = 102 )
```

The operand that is the object of the bitwise NOT is not actually changed by the operation; however, the way in which the operand is evaluated is changed.

Precedence and Evaluation Order

The order in which operations are performed can drastically affect the results, as demonstrated earlier in the chapter, in the “Grouping Operators” section. The order of operation performance is governed by *precedence*.

The precedence of the operators affects the evaluation of operands in expressions. Operands associated with higher-precedence operators are evaluated before the lower-precedence operators. The table below shows the precedence of the operators. Where operators have the same precedence, they are evaluated from left to right.

Operators and Precedence

Operator	Description
()	Parenthetical grouping
& *	Unary pointer operators
~ ! - +	Unary operators
**	Exponential (power of) operator
* / mod	Multiplication and division operators
+ -	Addition and subtraction operators
<< >>	Shift operators
:	Concatenation
< <= == >= > != <>	Relational operators
& ^	Bit-manipulation operators
&&	Logical operators
=	Assignment operator

There is another circumstance where precedence comes into play. When we write a statement as:

```
c = ( a / b ) * 4.1
```

...or...

```
d = a / ( b * 4.1 )
```

...we automatically assume that all of the operations to the right of the assignment operator (=) are performed before the assignment is made to the variable on the left. Rather obviously, this is also true when an assignment is made to copy the results returned by a function to a variable – that is, the function performs all of its operations before the assignment operation is handled.

There are circumstances, however, when we may not choose to use explicit assignments – that is, we may not wish to assign the results of an operation to a variable – but, instead will simply use the results of a calculation, operation or a function call directly in another operation. For example, if we wanted a random percentage, we could begin by calling the `random` function as:

```
n = Random(100)
; returns a value between 0 and 100
nPercent = n / 100
```

Here, we've assigned the result returned by the `random` function to the variable `n` and then used `n` in a further calculation. We could simplify this operation, however, and omit the variable `n` entirely by writing this instruction as:

```
nPercent = Random(100) / 100
```

In this example, the value returned by the `random` function is not assigned to any variable but is simply used as a value in the next stage of the calculation. The order of precedence in this statement is simple: the `random` function takes the highest precedence and is carried out first, the division operation (/) is evaluated next and the assignment operation (=) occurs last.

When several functions appear in a statement, the order of precedence begins at the left, proceeding to the right. For example, take the following statement which uses three theoretical functions:

```
nResult = ( function1( a ) * function2( b, c ) ) / function3( d )
```

Introduction to Programming

Here `function1` and `function2` would be evaluated in left to right order but the next step would be to evaluate the product of the results returned by `function1` and `function2`. Only after this was done – as directed by the parentheses – would `function3` be evaluated. Only then would the product of `function1` times `function2` would be divided by the value returned by `function3` before, last, the results would be assigned to `nResult`.

Note that the order of operations follows the same logical sequence which you learned in grade school – a circumstance which does not occur by accident. After all, this is a high level language intended for use – and understanding – by humans which makes it quite logical to have the order follow a human standard.

Comments

Strictly speaking, a program comment is a *non-operator*. A comment is a note or observation included for the benefit of the person who wrote the program or for other programmers.

A comment is denoted by a semicolon (;) which precedes the text of the comment. Everything following the comment mark (to the right of the semicolon) is ignored when the program is executed or compiled.

Here are some examples of comments:

```
; this line is a comment and will be ignored
nItems = 25                ; and this is also a comment
fPrice = 1.25
fTotal = nItems * fPrice   ; and a third comment
```

Note that any blank lines in the application code are also ignored.

Unary Operators (Variable Reference Operators)

Two unary reference operators – the ampersand & and the asterisk * – are used to refer to or reference a variable. For example:

```
pIndex = &Index    ;creates a pointer to the variable Index
```

and

```
*pIndex ;references (connects with) Index via the pointer.
```

Thus, `arrayVar[Index]` is the same as `arrayVar[*pIndex]` while the actual value of `Index` may be defined or modified somewhere else in the program. In effect, the unary operators function like a phone number; the phone number is the reference to someone somewhere else and can be used to make a connection to them without having to travel to meet them and without the remote party having to interrupt their own business.

Binary String Operations

There is also one binary string operator – the colon `:` – which is a concatenation operator. In effect, the concatenation operator joins two strings together. For example the instructions:

```
sFirst = "Mary had"
sSecond = "a little lamb"
sFinal = sFirst : sSecond
```

gives us a `sFinal` which now reads "Mary had a little lamb".

Summary

The operators described here are the basic verbs of computer programming. They represent the simplest actions that can be carried out by an application. Combined with the simple nouns (variables and constants) you learned in the previous chapter, you now have the vocabulary to tell the computer how you want basic operations performed and how to perform simple tests.

Now we can move onto operations that are a bit more complex. In the next chapter, we'll deal more extensively with string operations. In [Chapter 7](#), we'll dip into the programmer's toolbox and begin working with functions that supplement, extend, and complement the basic tools covered thus far.

As mentioned at the beginning of this chapter, many of the operators discussed here are demonstrated in the [MathTest.wbt](#) demo. At this point, you may want to spend a few minutes first playing with the demo and then altering it to observe new results.

CHAPTER 6 : COMPUTER VOCABULARY – PART III

STRINGS AND TEXT OPERATIONS

"... men ought not to investigate things from words but words from things; for that things are not made for the sake of words, but words for things." (Diogenes Laertius, *Myson*)

"...where a neat rivulet of text shall meander through a meadow of margin."
(Richard Brinsley Sheridan, *St. Patrick's Day*)

Strings were introduced in [Chapter 4](#) as one of the basic data types. Until we have voice synthesis (and holograms and a few other fancies), strings (text) are how we communicate with the people using our applications, and you can't get much more basic than that.

Granted, voice synthesis is a reality and not some future dream. However, it is still not a commonplace device for computer communications. And, if it does come into common use, we strongly suspect that strings will still be used, even if only to tell the "voices" what to say.

In this chapter, we explore how we can manipulate this type of data so that our applications can both understand (in a limited sense) the strings entered by users and present other strings as information.

Although extensive string functions are provided in WinBatch, these functions are relatively slow while the Array functions can be much faster.

String-Manipulation Functions

As mentioned in a previous chapter, WinBatch does not entirely differentiate between strings and numbers. A string containing only numeric characters can be treated as if it were an integer or a floating-point value. Likewise, when reporting the results of various mathematical operations, integers and floating-point values are treated as if they were strings. (Note that this free and easy transmutation between data types is not common; most computer languages require explicit conversions from numerical to textual representation.)

In addition to providing essentially automatic conversions—with some limits— between the basic data types, WinBatch also offers an excellent variety of string-manipulation functions. String functions can be thought of as falling into two categories: auxiliary functions and specialized functions.

The auxiliary string functions provide support for routine string operations, such as concatenating two strings into a single string (`StrCat`), finding the length of a string

Introduction to Programming

(`StrLen`), and performing a comparison between two strings (`StrCmp`). These auxiliary string functions are similar to those found in most languages and can be used to create more complex string functions.

Some languages such as Visual Basic have permitted the addition operator (+) to concatenate strings, as:

```
sReport = "Your name is " + sName.
```

WinBatch does not permit the addition operator for strings but does provide the concatenation operator (:) thus:

```
sReport = "Your name is " : sName.
```

Alternately, you can use the `StrCat` function to perform concatenation.

The specialized string functions perform very specific string tasks, giving the programmer the ability to quickly build complex and useful text-processing utilities. Examples of specialized string functions include `SubStr`, which extracts a portion of a string; `StrScan` and `ParseData`, which count the words in a string; and `StrReplace`, which searches a string for occurrences of a substring and replaces them.

Most of the string-manipulation functions are easily identified by the `Str` prefix in the function name.

String-Parsing Operations

In many cases, WinBatch applications are called with parameter arguments, as in the following cases:

- An application to perform operations on several files might be called with a list of files to be used. The files listed are parameters.
- An application might be called with parameters that provide instructions for what types of tasks should be performed.
- A single parameter might refer to a file that contains a further list of instructions (parameters).

In any of these cases, the immediate problem the application must solve is to separate these parameters into individual arguments.

For example, suppose our application is reading a text file that contains a series of instructions. The text file identifies other applications that we want to execute and also specifies arguments to either be passed to these applications or used within the main application for other purposes. The following sections describe four alternative ways to

handle this. In the examples, the file containing the instructions is named [SearchList.txt](#), and it consists of three lines:

```
MyApp1 Apples Oranges Pears
MyApp2 23 456.4 93.0 128.6 93.2 35.6 87.456 7.65
MyApp3 %n1% %n2% %n3%
```

The sample programs in [Appendix A-SearchTest.wbt](#), [SearchTest2.wbt](#), [SearchTest3.wbt](#) and [SearchTest4.wbt](#)-demonstrate using the `ParseData` function, a combination of the `StrScan` and `StrSub` functions, a combination of the `ItemCount` and `ItemExtract` functions and the `ArrayFileGet` function.

The ParseData Function

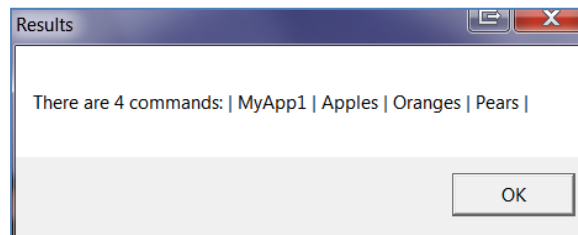
The [SearchTest.wbt](#) demonstrates using the `ParseData` function. As [SearchTest.wbt](#) reads each line of the [SearchList.txt](#) file (contents shown above), the `ParseData` function performs two tasks. First, `ParseData` counts the number of individual words in the line by looking for the spaces separating the words. Second, `ParseData` assigns each of the words found to a series of string variables named `param1`, `param2`, and so on. The number of arguments returned by `ParseData` is also stored in a variable named `param0`.

In the [SearchTest.wbt](#) example, the `ParseData` function is called as:

```
nCmds = ParseData( Line )
```

Here, `nCmds` receives the command count returned by `ParseData`, but the assignment of the substrings found to the parameter variables is automatic.

Once these elements have been separated, they could be used for a variety of purposes. `MyApp1` could use the parsed string to launch other applications with command-line arguments (arguments that are passed when an application is called from the command line, using the Run command on the Start menu, or from other applications). In the case of `MyApp2`, the arguments might be treated to some form of numerical process. `MyApp3` might substitute other variables for the arguments. In this case, however, the parsed string is simply reported as the number of arguments and a list:



The `ParseData` function is useful for parsing strings, but it has a few limitations:

Introduction to Programming

- These arguments are automatically assigned to a series of predefined variables.
- The only delimiter allowed to identify words is the space character.

If the [SearchTest.wbt](#) demo had itself been called with a series of arguments, these arguments would have been in the `param_x` variables and would have been overwritten, unless they had been saved under new names, when the first line was read from the file.

In short, the `ParseData` function is useful, but it is also rather specialized. There are more flexible ways of breaking string data into substrings, as described in the following sections.

The `ParseData` function is primarily intended to handle command-line parameters. However, in WinBatch, when an application is called with command-line arguments, once the application starts, these arguments are already present as the variables `param1` through `paramN`. Similarly, the variable `param0` contains an integer value reporting the number of arguments provided.

The StrScan and StrSub Functions

As an alternative to using the `ParseData` function, you can use a combination of the `StrScan` and `StrSub` functions. The `StrScan` function can find any of several delimiters (your choice), and the `StrSub` function can extract a substring.

The [SearchTest2.wbt](#) demo uses the `StrScan` and `StrSub` functions to parse the data in the sample [SearchList.txt](#) file. To begin, we need to set a starting point (which should not be zero), to set the count to zero, and to set a flag, named `bDone`, to false:

```
nPos1 = 1 ; set starting point
nCmnds = 0 ; zero the count
bDone = @FALSE ;and set an end flag
```

With this done, we can start a loop to look for delimiters in the string read from the file (loops are explained in [Chapter 8](#)):

```
While @TRUE
    nPos2 = StrScan( Line, ', : ', nPos1, @FWDSCAN ) ;checking for three
characters
```

When we call `StrScan`, we're supplying several pieces of information:

- The string to be searched
- The delimiters to find, which are a comma, a colon, and a space (finding any of these three characters will satisfy the search) in this example

- The position in the string where the search should begin, which is the first character in the string in this example
- The predefined constant `@fwdscan`, which tells `StrScan` to search from the first to the last (`@backscan` would reverse the search direction)

In WinBatch, using a start position of zero (0) in any search has a special meaning. For a forward search (`@FWDSCAN`), zero initiates the search at the beginning of the string. For a reverse search (`@BACKSCAN`), an argument of zero initiates the search at the end of the string. However, in both cases, the position reported will be the position from the start of the string.

Next, we check the results of the search. If we've hit the end of the line—that is, there are no delimiters remaining to be found—the value returned to `nPos2` will be zero. If we have come to the end of the line, we don't want to forget the last word (or phrase) in the string. We set `nPos2` to point to the end, and we also set a flag, `bDone`, to indicate that we're finished searching:

```
If( nPos2 == 0 )           ; we've hit the end of the line
    nPos2 = StrLen( line ) + 1    ; find end of line
    bDone = @TRUE              ; and set an end flag
Endif
```

Next, once we've actually found a position—either a delimiter or the end of the string—we can increment `nCmds`. Now that we have both a start and an end position, we can extract the substring from `Line`. However, since the `StrSub` function doesn't want an end point but only a start point and a length, the final argument passed to `StrSub` is a length derived from the two points:

```
nCmds = nCmds + 1
param%nCmds% = StrSub( line, nPos1, nPos2 - nPos1 )
```

Also, `nCmds` is used, as `param%nCmds%`, to copy each substring to a different element in a variable. Note that the `paramxxx` variable name was used here simply to match the format demonstrated in [SearchTest.wbt](#); we could have used any variable name.

The next step is to update our first search position, `nPos1`, to one position past `nPos2`. If we started the next loop at the position where the last delimiter was found, the program would find the same match over and over again, creating an infinite loop (another example of the mindless golem principle in action).

```
nPos1 = nPos2 + 1           ; set a new starting point
If bDone == @TRUE then Break ; reached end, exit loop
```

Introduction to Programming

EndWhile

Finally, if the `bDone` flag has been set, then we need to break out of the `while` loop and let the rest of the program proceed identically to the [SearchTest.wbt](#) demo.

Feel free to try this on your own and see what happens if you don't update the first search position to past where the last search item was found. In this specific situation, the loop will abort when the number of new variables is exhausted ... somewhere in the larger numbers or when memory is exhausted. Do not, however, count on such a convenient "out" from infinitely repeating loops.

The [SearchTest2.wbt](#) demo is more flexible than the [SearchTest.wbt](#) example for two reasons:

- It accepts a variety of delimiters.
- It allows us to assign the substrings located to whatever variables we choose, not simply to the predefined `paramx` variable series.

The ItemCount and ItemExtract Functions

The [SearchTest3.wbt](#) demo demonstrates using the `ItemCount` and `ItemExtract` functions to parse a string. The advantage of this approach is that it takes fewer lines of code.

The [SearchTest3.wbt](#) program requires only four lines of code to accomplish the same task that required fourteen lines in the [SearchTest2.wbt](#) demo.

In this example, the first instruction uses `ItemCount` and the space character as a delimiter. It searches the string to return the number of individual substrings found. Then, once the number of items is known, the `ItemExtract` function, using the same delimiter, is called within a loop to return each of the substrings individually:

```
nCmds = ItemCount( line, " " ) ; get the number of commands
For i = 1 to nCmds
    param%i% = ItemExtract( i, line, " " ) ; extract each substring
Next
```

The end result is precisely the same as what is accomplished by the [SearchTest.wbt](#) and [SearchTest2.wbt](#) programs. High-level functions such as `ItemCount` and `ItemExtract` can perform relatively complex tasks with very brief instructions. However, lower-level functions such as `StrScan` and `StrSub` offer greater flexibility and can be used to perform custom tasks not supported by higher-level functions.

Using the ArrayFileGet function

The [SearchTest4.wbt](#) program performs the same task but copies the text file into an array using the `ArrayFileGet` function which returns each line of the file as an array element. If you examine the code, you should notice how brief this code is when compared to the previous examples.

Of course, in this example, instead of parsing each line for individual words, the array retrieval allows us to simply display the retrieved line without breaking down the individual parameters.

Differences in the String-Parsing Techniques

There are several differences in the string-parsing techniques demonstrated in the four examples:

- [SearchTest.wbt](#), which uses `ParseData`, is limited to a maximum of nine items. All the others can parse any number of substrings (or list or other items) and can assign these to any variables desired.
- The only delimiter allowed in [SearchTest.wbt](#) is the space character. The rest can use any delimiter desired. [SearchTest2.wbt](#) has the extra advantage of allowing a mixed list of delimiters, not merely a single delimiter character; when multiple delimiters are provided, any one of these can indicate a break between items.
- Comparing [SearchTest2.wbt](#) and [SearchTest3.wbt](#), the latter is most notable for the brevity of the code required. ([SearchTest.wbt](#) and [SearchTest3.wbt](#) are approximately the same size although the former is less sophisticated functionally.)
- The process demonstrated in [SearchTest2.wbt](#) can be modified to serve a variety of different tasks as well as supporting a variety of different tests. This flexibility is not supported by the `ItemCount` and `ItemExtract` functions demonstrated in [SearchTest3.wbt](#).
- [SearchTest4.wbt](#) is the briefest and quickest operating code sample. Arrays are recommended when dealing with very large data sets.

Search-and-Replace Operations

Locating strings and finding strings and replacing them with other strings are common operations. The WinBatch functions that support these types of operations include `StrIndex`, `StrIndexNc`, and `StrReplace`.

The StrIndex and StrIndexNc Functions

You can use the `StrIndex` or `StrIndexNc` function to find matching strings. Unlike the `StrScan` function (discussed earlier in the chapter), which searches for one character or any one character in a set of characters, the `StrIndex` function searches only for an entire and complete substring.

For example, consider this code:

```
sSample = "The quick red fox jumped over the lazy brown dog"
nPos = StrIndex( sSample, "brown dog", 1, @FWDSCAN )
```

Introduction to Programming

In this example, `StrIndex` will return a value of 40, reporting that the substring "brown dog" was located beginning with the fortieth character in the string.

The `StrIndex` and `StrIndexNc` functions are called with the same arguments and operate in essentially the same fashion, with one significant difference: `StrIndex` searches for an exact match of uppercase and lowercase characters, but `StrIndexNc` is not case-sensitive. For example, if the sample string were searched for the substring "Brown Dog" rather than "brown dog", like this:

```
nPos = StrIndex( sSample, "Brown Dog", 1, @FWDSCAN )
```

the value returned to `nPos` would be 0, since no match would be found. However, if the search were performed using `StrIndexNc`, like this:

```
nPos = StrIndexNc( sSample, "Brown Dog", 1, @FWDSCAN )
```

`nPos` would contain a match position, because it ignores the differences between uppercase and lowercase characters.

Quite often, however, we're interested in locating all occurrences of the substring, not just one instance. For this purpose, we can use a `while` loop, [StrIndex.wbt](#):

```
sSample = "The quick red fox jumped over the lazy brown dog"
sTarget = "brown dog"
nPos = 0
While @TRUE
    nPos = StrIndex( sSample, sTarget, nPos, @FWDSCAN )
    If nPos == 0 Then Break ; time to quit
    Pause( "Found at position", nPos )
    nPos = nPos + 1 ; start one place further
EndWhile
exit
```

Here, the search begins at the first of the `sSample` string and then repeats each time, searching from one position further than the previous match. This one position offset is necessary to avoid an infinite loop; otherwise, the same match would simply be found over and over again, *ad infinitum*.

In WinBatch, a list is really not very different from a string and can be searched in much the same fashion as a string. For example, if we wanted to find all items in a list that contained the word `blue`, we could use the search routine shown in this section by incrementing a counter for each match. However, actually extracting each entry that

contains a match requires a little more sophistication. A suitable technique will be illustrated later in this chapter, when we discuss lists in more detail.

The StrReplace Function

Once the `StrIndex` or `StrIndexNc` function has been called to find an occurrence of a substring, the next step might be to replace the target (found) string with a new string. If we want to replace all occurrences of the target string without reservation, using the `StrReplace` function is straightforward. `StrReplace` is called as:

```
sStringRevised = StrReplace( sString, sOld, sNew )
```

And that's it—no fuss, no muss, no decisions, and no complications. All occurrences of the old substring are located and replaced by the new substring.

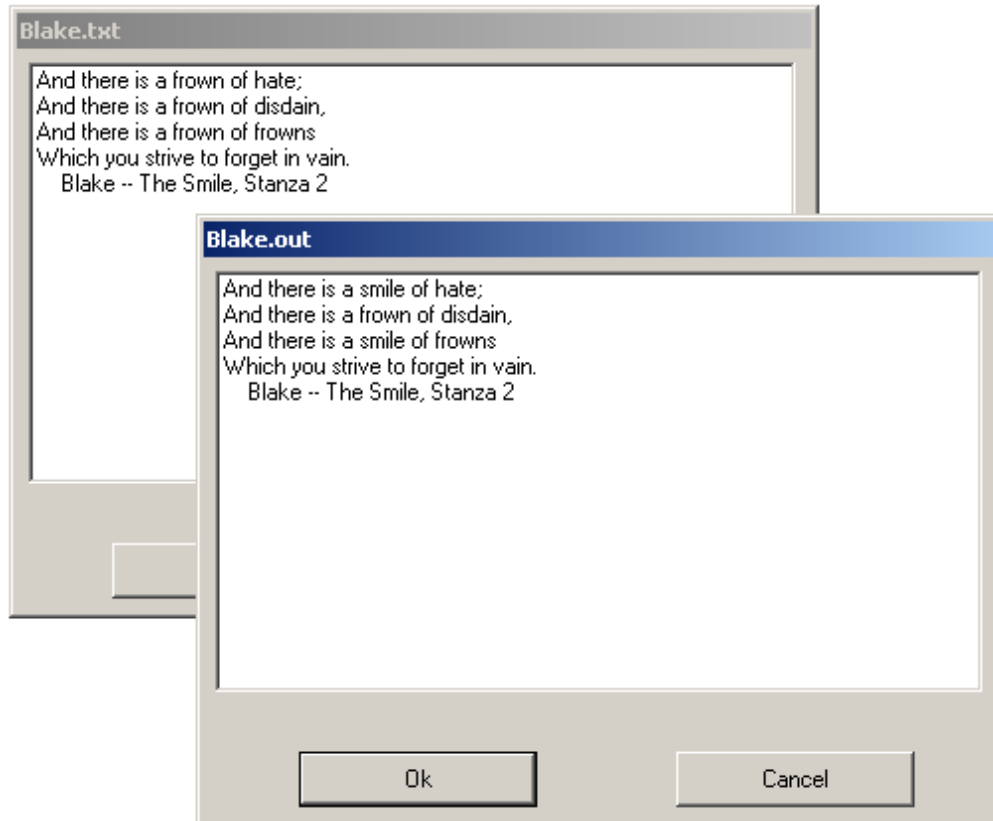
However, if we want to do something a little more sophisticated, such as deciding which occurrences to replace and which to leave alone, then we need to be a little trickier.

Selective Search and Replace

The [SearchReplace.wbt](#) program demonstrates a selective search-and-replace operation. The program reads from a text file ([Blake.txt](#)) containing an excerpt from Blake's *The Smile*, where one word is repeated four times in three lines. To show that the replacement is selective rather than global, as would happen using the `StrReplace` function, we change only the first and third occurrences of the string, leaving the second and fourth as they are in the original.

A simple text box displays the original file and then, following the search and replace operation, the text box shows the altered text. Both the before and after versions are shown in the following dialog.

Introduction to Programming



We begin by defining a target and a replacement string, getting the length of the target string (using `StrLen`) and, since we're going to alternate replacements, setting a counter:

```
sTarget = "frown"  
sReplace = "smile"  
nLen = StrLen( sTarget )  
nCount = 1
```

The heart of the operation begins using `StrIndexNc` to search for a single occurrence of the target string.

```
nPos = StrIndexNc( sLineIn, sTarget, nPos, @FWDSCAN )  
If nPos == 0 Then Break ; that's it, jump to next line  
If nCount mod 2 == 1
```

If no match is found, then the `break` statement jumps out of the `while` loop so that we can proceed with the next line. Otherwise, we continue the loop as long as any occurrence of the target remains to be found. The `nCount mod 2` test is provided to alternate whether the target is replaced or skipped.

We have start position for the target. The replacement operation begins by getting a second position (`nPos2`) for the end of the string and getting a length (`nLen2`) for the string remaining after the end of the target substring.

```
nPos2 = nPos + nLen
nLen2 = StrLen( sLineIn ) - nPos2 + 1
sTemp1 = StrSub( sLineIn, 1, nPos-1 )
```

Next, we use `StrSub` to copy the first part of the line—however much there is—into the variable `sTemp1`. This gives us everything preceding the target substring. Then we use `StrSub` again to copy everything following the target substring into a second variable, `sTemp2`.

```
sTemp2 = StrSub( sLineIn, nPos2, nLen2 )
```

Now that we have the first part of the string and the last, but not the original search target, we can use `StrCat` to reassemble a new string with the replacement string (`sReplace`) in the position previously occupied by the search target:

```
sLineOut = StrCat( sTemp1, sReplace, sTemp2 )
Endif
```

We’ve selectively replaced a single occurrence of the target string without replacing all occurrences. But we’re still inside a `while` loop, and the search isn’t finished. We increment the counter used for alternations. Then the important next step is to increment the initial search position so that the search can continue for the next match.

```
nCount = nCount + 1
nPos = nPos + 1 ; start the next search one place further
EndWhile
```

Granted, this process is not as easy as using the `StrReplace` function, but unlike `StrReplace`, it is selective and allows replacement of a specific substring. Most important, this search-and-replace operation could be used for any substring, containing one or more words and using any delimiter we choose.

As program routines go, this one is actually quite simple and relatively brief. It is also flexible and customizable.

String-Conversion Operations

In addition to the automatic conversions allowing strings to be treated as numeric values and vice versa, WinBatch provides a pair of conversion functions:

Introduction to Programming

- `StrLower` converts a string to all lowercase letters.
- `StrUpper` converts a string to all uppercase letters.

For example, using `StrLower`, the following example changes all uppercase characters in the string to lowercase:

```
sTemp = StrLower( "ThIs StRiNg WaS MiXeD cAsE" )
```

The result is `sTemp` rendered as "this string was mixed case".

Using `StrUpper`, like this:

```
sTemp = StrUpper( "ThIs StRiNg WaS MiXeD cAsE" )
```

yields "THIS STRING WAS MIXED CASE".

Initially, you may think that these sound like trivial functions, but there are occasions when a difference in case would be inconvenient or erroneous. In these situations, we can simply convert strings to full uppercase or full lowercase before performing any operations. This is a common requirement when making comparisons between strings, as described in the next section.

Other String Conversions

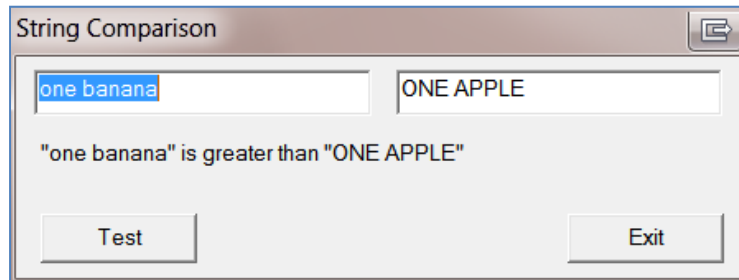
To handle international text displays (international alphabets) the Unicode alphabets offer a very wide variety of text fonts and provides access to these by setting code pages for each alphabet. To support these, WIL provides a comprehensive set of conversion functions as:

- `ChrHexToString(hex-string)` converts a hex string to a string.
- `ChrHexToUnicode(hex-string)` converts a hex string to a Unicode string.
- `ChrSetCodePage (code-page)` sets the current WIL code page.
- `ChrStringToHex(string)` converts a string to a Hex string.
- `ChrStringToUnicode(string)` converts an ANSI string to a Unicode string.
- `ChrUnicodeToHex(Unicode-string)` converts a Unicode string to a Hex string.
- `ChrUnicodeToString(Unicode-string)` converts a Unicode string to an ANSI string.

You may, of course, use the standard English/American alphabet without needing access to any of these conversion options. But, if you need Thai or Bengali or N’Ko ... well, it’s a wide, wide world out there.

String-Comparison Operations

For string comparisons, WinBatch provides the `StrCmp` function, which performs a case-insensitive string comparison. The [StrCmp.wbt](#) program demonstrates how `StrCmp` works, as shown below:



If the two strings were compared strictly according to the ASCII values of the characters, the lowercase string would always be less than the uppercase string, even if they were otherwise identical. Therefore, we can assume that the `StrCmp` function has performed a case conversion on both string arguments before performing a comparison. (Whether these strings were converted to uppercase or to lowercase is immaterial, because the result would be the same as long as they are both in the same case.)

The code used in the [StrCmp.wbt](#) demo is fairly simple:

```
nResult = StrCmp( sTemp1, sTemp2 )
Select nResult
    case -1                                ; less than
        sResult = " is less than "
        break
    case 0                                ; equal
        sResult = " is equal to "
        break
    case 1                                ; greater than
        sResult = " is greater than "
        break
EndSelect
sReport = "'" : sTemp1 : "'" : sResult : "'" : sTemp2 : "'"
```

`StrCmp` returns a -1 if the first argument is less than the second, a 0 (zero) if they are equal, or a 1 if the first is greater than the second.

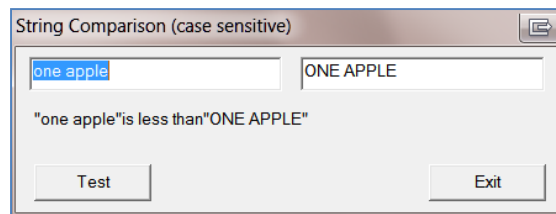
Unlike some languages, WinBatch quite readily accepts a negative value in a select/case statement. We'll talk about selection mechanisms in [Chapter 8](#).

While `StrCmp` is convenient and very useful, there also may be times when we want to make a string comparison on a case-sensitive basis. For example, if we request a password before permitting access to data, we might prefer to test for an exact match and not allow mistakes in capitalization to pass unchallenged. For this purpose, we have two choices:

- The `StriCmp` function performs the same operation as `StrCmp` but executes a case-dependent comparison.
- The `>`, `>=`, `==`, `!=`, `<=`, and `<` relational operators, introduced in [Chapter 5](#), can be used for string comparisons.

Remember: to the computer, everything is simply a number. Strings are only a convenient pretense for our benefit.

The [RelationalOperators.wbt](#) program demonstrates the use of the relational operators to perform a case-sensitive comparison. Here is the program reporting that "one apple" is less than "ONE APPLE":



When a string comparison is made, the lengths of the strings are ignored. Instead, the comparison begins with the first character of each string. If the first characters of each match, the next characters of each are compared, continuing until a mismatch is found. The rules for string matches made using the relational operators are:

- If every character matches exactly and the strings are the same length, the strings are reported as identical.
- If every character matches until the last character of one string is reached but the other string has remaining characters, the longer string will be reported as larger.
- If the mismatch is between two numeric characters, the string with the larger numeric character is reported as larger.
- When two case-mismatched alphabetic characters are found—one uppercase and one lowercase—the string with the uppercase character is reported as larger (even though the ASCII character code for *A* is 65, or 41h, while the character code for *a* is 97, or 61h). This is how the comparison of apples appears above.
- For mismatched nonalphabetic and nonnumeric characters, the ASCII character codes are used. Following the rules for alphabetic characters, the string with the **lower** character code is reported as the larger string. For example, "one apple#" "

is reported as less than "one apple\$" because the character code for # is 35 (23h), one less than the character code for \$ (36 or 24h).

The code for a case-sensitive comparison in the [RelationalOperators.wbt](#) program is:

```
While @TRUE
    ButtonPushed = Dialog("StringCmp")
    if sTemp1 == sTemp2 then sResult = " is equal to "
    if sTemp1 > sTemp2 then sResult = " is greater than "
    if sTemp1 < sTemp2 then sResult = " is less than "
    sReport = "'" : sTemp1 : "'" : sResult : "'" : sTemp2 : "'"
EndWhile
```

In many cases, the test you will be interested in is equality (==) to determine a match. A test for greater than (>) or less than (<) is useful for sorting a list.

Alternately, these comparisons could be performed using the `StriCmp` function.

Other String Operations

WinBatch provides several other useful string functions. These functions count characters in a string, fill strings, pad strings, truncate strings, and remove leading and trailing spaces from strings.

The StrCharCount Function

The `StrCharCount` function returns the number of character in a string. This function is useful when dealing with double-byte character sets, such as Kanji, because the `StrLen` function would report the length as if the characters were single-byte. For the standard European (English) character sets, the `StrLen` and `StrCharCount` functions will return the same value.

The StrFill Function

The `StrFill` function creates a string of a specific length and fills it with a specific character. If no fill character is specified, the string is filled with spaces. Alternatively, a string of characters can be specified. For example, `StrFill('ABC', 20)` produces the string "ABCABCABCABCABCABCAB".

The StrFix Functions

The `StrFix`, `StrFixLeft`, `StrFixChars`, and `StrFixCharsL` functions either pad or truncate existing strings to a fixed length. Strings requiring padding are filled with a specified character (or string) or with spaces. Padding or truncation can be applied from either the right or left.

The StrTrim Function

Also useful, the `StrTrim` function removes both leading and trailing spaces from a string. To the eye, leading – and especially trailing – spaces aren’t particularly apparent but there are many circumstances where space characters can cause undesired or unintended results. For example, if we were to run a comparison between a term entered by the user against a list of terms but the user’s entry had an (invisible to the user) trailing space, our comparison would not find a match even though, technically, there might be a matching entry.

For this reason and others, it is always helpful to trim excess spaces from strings before converting them to numbers, making comparisons or, not least, checking password entries.

Additional String Functions

Several additional string functions appear in WinBatch to offer additional convenience in string handling. These are:

- `StrClean` removes or replaces characters in a string.
- `StrCnt` counts the occurrences of a substring within a string.
- `StrInsert` inserts a new string into an existing string.
- `StrOverlay` overlays a new string onto an existing string.
- `StrTypeInfo` gets character-type information for a string, or information for a character-type.

Lists and List-Selection Operations

In WinBatch, lists are simply special cases of the string data type. WinBatch treats a list as a string where the list entries are separated by a single character. The `ItemExtract` function (introduced earlier) searches a string (a list) looking for the specified delimiter to return a substring.

A tab-delimited format is a common standard. Most spreadsheets, for example, both export and import data as tab-delimited lists; that is, as text files with the fields separated by tab characters. An example of a list (the tab-delimited string used by WinBatch) might appear as:

```
Apple»Pear»Orange»Plum»Banana»Grape»Pomello»Mango»Papaya»Lemon
```

Here, a list of fruits are separated by tab characters (») identifying (delimiting) the separate fields.

More commonly, a tab-delimited file consists of records with several columns per line. For example, a record may contain an item name, quantity, and price. Each field (column) in the record is separated by a tab character (»), and each row ends with a

carriage return/line feed (CRLF) character pair (denoted as `␣`). An exported file might look like this:

```
apples»345»1.25␣
pears»198»2.37␣
oranges»216»3.45␣
plums»57»2.45␣
bananas»623»1.34␣
grapes»16»4.57␣
pomellos»98»5.72␣
mangos»92»6.87␣
papayas»344»2.99␣
lemons»1234»1.54␣
```

Having a list format is only the first step. We also need a way to present the list. In a WinBatch dialog, the ListBox control is the convenient way to display a list for selection.

The tab-delimited file format that is an industry standard and the tab-delimited list format used by WinBatch are not completely compatible. This dichotomy is not a problem, but you should keep the differences in mind. For example, if the preceding example were read from a file, WinBatch would read these as individual lines, ending with CRLF pairs, which could then be reassembled into a list.

When we used the `ArrayFileGet` in [SearchTest4.wbt](#), this is what we saw there as a result; each line was read – breaking on CRLF pairs – into a separate array element.

Another alternative is the `FileGet` function which also reads an entire file, returning a string variable containing the file.

A ListBox control accepts and displays an associated tab-delimited list, allowing a selection to be made from the list. The selected item is highlighted and, when the dialog returns (closes), the associated list contains only the selected item.

The WinBatch ListBox control is similar, but not identical in operation, to the ListBox control used in other languages, for example, in Visual Basic or Visual C/C++.

List Initialization

The first step is to initialize the list. We could initialize a list using a tab-delimited string:

Introduction to Programming

```
itemList =  
"Apples":@TAB:"Pears":@TAB:"Oranges":@TAB:"Plums":@TAB:"Bananas"
```

However, more commonly we initialize a list by assigning an empty string to the list:

```
itemList = ""
```

Then we use the `ItemInsert` function to add entries to the empty list. The `ItemInsert` function is called as:

```
itemList = ItemInsert( newItem, index, itemList, delimiter )
```

First, notice that the list variable (`itemList`) serves as both a variable that receives the value returned by the `ItemInsert` function and as one of the function's arguments. As mentioned previously, a list is simply a string receiving special treatment; thus, the original list (string) is passed to the `ItemInsert` function, and the revised list (string) is returned after inserting the new entry.

The `newItem` argument is the entry to be added to the list. The `index` argument tells `ItemInsert` where the item should be inserted. The `index` argument can be used in several ways:

- An `index` of 0 places the new item at the beginning of the list.
- An `index` of -1 places the new item at the end of the list.
- Any numeric `index` places the new item *following* the numbered entry. For example, an `index` of 2 places the entry in the third list position (following the second entry).

The `delimiter` argument should always be specified as `@TAB` if you are planning to use the list with a `ListBox` control. If this is simply a list maintained for some other use, then you are free to specify a different delimiter—`@cr`, `@lf`, `|`, `&`, comma or any other single character you choose. If you are creating a list of lists, each level in the hierarchy of lists should use a different delimiter. The exported tab-delimited file example shown earlier (the apples, oranges, etc. file) is actually a list of lists where the lowest hierarchy is tab-delimited (`»`), while the lists themselves are delimited using a carriage return (`↵`).

List Creation

The [ListSelection.wbt](#) program demonstrates creating, displaying, and selecting from lists. It uses four lists: one associated with a displayed `ListBox` and three auxiliary lists to contain items associated with the selection list. The information used to populate the lists is read from a tab/CRLF-delimited file.

Before opening the data file, our first requirement is to initialize several variables, including the variables that will be used to contain the lists created from the data file:


```
fileIn = "Parts.lst" ; source file for data
listItem      = "" ; initialize all of the lists
listStkPN     = ""
listStkQuant  = ""
listStkLoc    = ""
listDisplay   = ""
```

The string variables are initialized by assigning them to empty strings. If a string variable is not initialized, an error will occur when that variable is passed as an argument to a function (for example, to the `ItemInsert` function).

Once the variables have been initialized, the next step is to open the data file and begin reading the data (file operations will be covered in detail in [Chapter 10](#)):

```
hFileIn = FileOpen( fileIn, "READ" ) ; open input file and get a handle
```

A file handle (`hFileIn`) is a numerical identifier assigned when the `FileOpen` function is called. This handle uniquely identifies the opened file – within the application – and is required for all subsequent file access functions as well as for closing the file. WinBatch applications may have up to five files open at one time.

```
While @TRUE
    sLineIn = FileRead( hFileIn )
    If sLineIn == "*EOF*" Then Break ; break out of while loop
    If sLineIn == "" Then Continue ; repeat while loop
```

Each `FileRead` operation reads from the open file until a line break (CRLF) is reached, so that each read operation retrieves one line of text of whatever length.

When the end of the file is reached, `FileRead` reports by returning the string `*EOF*`. We test for this condition, and break from the loop. We also test for blank lines in the data file, and `Continue` looping if this condition is met, so that we don't try to add empty lines to our lists.

Assuming that we have read a line of data, our next task is to parse this line into separate fields and to place these fields into lists:

```
sListItem      = ItemExtract( 1, sLineIn, @TAB ) ; extract item name
sListItem      = StrFix( sListItem, "", 100 ) ; pad with spaces
sListItem = StrCat( sListItem, nItemCount ) ; then add the index number
```

Introduction to Programming

A little special handling was used for the first data field. After calling `ItemExtract` to retrieve the data item, the `StrFix` function is used to pad the line with spaces to a length of 100 characters before adding our list index (`nItemCount`) to the end of the string. The result is a string that looks something like this:

```
Webley Defaminizer
```

```
1
```

When we display this list (as shown a bit later), `Webley Defaminizer` (whatever that may be) appears at the end of the list even though it happens to be the first item read from the data file. The reason is that the displayed list has been sorted alphabetically. However, the list index, which is 1 in this case, is both the order in which the items were read and an index that will be used to retrieve associated data elements from other lists.

The padding added to the string is to space the index value to the right where it will not appear on the display. We're using a trick to hide a bit of auxiliary data while still keeping it where we can find it later. The reasons for this bit of subterfuge will be shown in a moment.

The next step is to add the modified item to `listItem`:

```
listItem = ItemInsert( sListItem, 0, listItem, @TAB ) ; insert at first  
of list
```

Each new item read from the data file is being inserted at the beginning of this list, simply because that's a convenient location. For `listItem`, the order isn't immediately relevant since we're going to sort the results before displaying them.

For the next three lists, however, the order of the items inserted is relevant. We need to maintain these lists in the same order in which the items were read from the data file—in the same order as the index values that we added to the item names.

Because we can treat `sLineIn` as a tab-delimited list, we can also use the `ItemExtract` function to conveniently retrieve the second, third, and fourth items from each line:

```
sListItem = ItemExtract( 2, sLineIn, @TAB ) ; extract quantity  
listStkQuant = ItemInsert( sListItem, -1, listStkQuant, @TAB ) ; insert  
at end of list
```

```
sListItem = ItemExtract( 3, sLineIn, @TAB ) ; extract P/N  
listStkPN = ItemInsert( sListItem, -1, listStkPN, @TAB ) ; insert at  
end of list
```

```
sListItem = ItemExtract( 4, sLineIn, @TAB ) ; extract location
```

```
listStkLoc = ItemInsert( sListItem, -1, listStkLoc, @TAB ) ; insert at
end of list
```

As each data element is retrieved, it is added to the `listStkQuant`, `listStkPN`, and `listStkLoc` lists, with each successive item placed at the end of its respective list, thus maintaining the lists in proper order.

Next, the variable `nItemCount`, which is the list index value, is incremented before the `while` loop continues:

```
nItemCount = nItemCount + 1
EndWhile
```

Finally, once the end of the data file has been reached—the `*EOF*` flag has resulted in the `while` loop being ended—the file handle is used to close the file before proceeding further:

```
FileClose( hFileIn ) ; close the input file
```

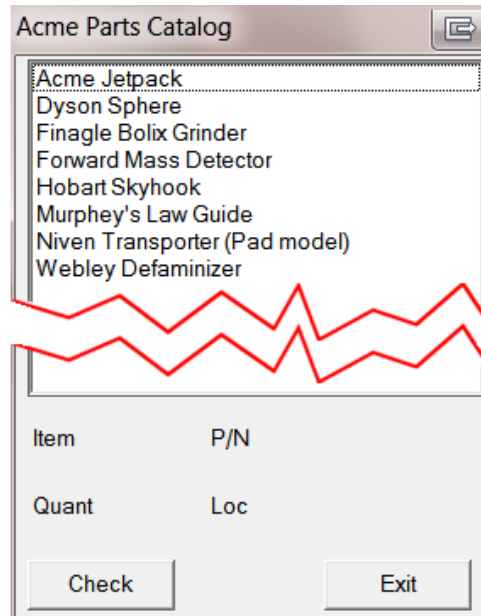
We close the file now because the file was only opened to read the data, so it isn't needed for any further input or output.

List Display

Our next step is to copy the `listItem` list, by sorting it with `ItemSort`, into the `listDisplay` variable. Since `listDisplay` is the variable member associated with the `ListBox` dialog control, the sorted list will be displayed automatically when the dialog is called, which is done next:

```
While @TRUE
    listDisplay = ItemSort( listItem, @TAB ) ; initialize listbox with
sorted list
    ButtonPushed=Dialog("Selection")
```

Here is the resulting dialog, with the sorted list ready for selection:



Notice that the index values, which are at the ends of strings too long to display, do not appear in the list. Also notice that `Webley Defaminizer`, which was the first item in the text file, appears last in the displayed list.

At the bottom, below the `ListBox`, four fields will display data associated with the list selection (after clicking the `Check` button). To accomplish this, we need the auxiliary data lists that were created at the same time as the displayed list.

List-Selection Handling

Our first step is to ensure that a selection was made from the list. We do this by testing for an empty string:

```
If listDisplay != "" ; is there a selection?
```

For a `File ListBox`, which is a file-selection list, there is an option to require a selection before a dialog can return. (The `File ListBox` dialog control is discussed in [Chapter 13](#).)

If no selection has been made, we simply ignore the reporting instructions and go back to displaying the dialog.

Once we're sure that a selection has been made, we start by separating the returned string into two values: the name of the selected item and the index value at the end of the string.

```
nLen = StrLen( listDisplay ) ; get entry length  
sIndex = StrSub( listDisplay, nLen - 10, -1 ) ; get rightmost chars
```

```

sIndex = StrTrim( sIndex ) ; trim for index value
sItemSelect = StrSub( listDisplay, 1, nLen - 10 ) ; drop rightmost
chars
sItemSelect = StrTrim( sItemSelect ) ; trim for item name

```

Here, we use the `StrTrim` function to remove both leading and trailing spaces from the two substrings. This is a convenient way to clean up and remove the padding used to arrange the display.

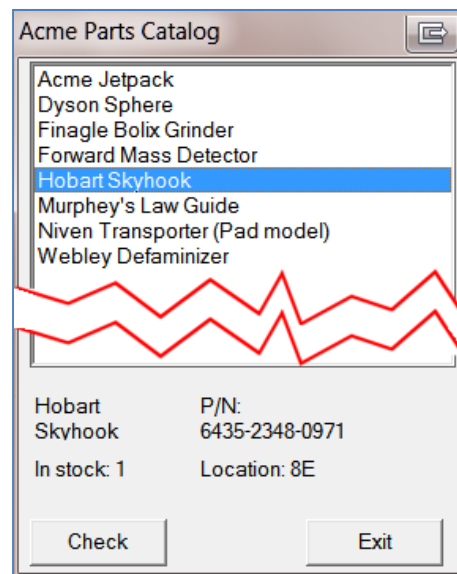
Now that the index value has been retrieved, we can use the `ItemExtract` function again to fetch the corresponding entry from the `listStkQuant`, `listStkPN`, and `listStkLoc` lists and to display each of these as part of the report as we loop back to displaying the dialog:

```

; get the rest of the particulars from the separate lists
sStkQuant = "In stock: ":ItemExtract( sIndex, listStkQuant, @TAB)
sPN = "P/N: ": ItemExtract( sIndex, listStkPN, @TAB )
sLoc = "Location: ": ItemExtract( sIndex, listStkLoc, @TAB)
EndIf
EndWhile
exit

```

Because we have no convenient method of rearranging the three auxiliary lists to match the sorted list, we use the index value to retrieve the associated data from each and display this data at the bottom of the dialog:



Lists of Lists

Using index values and several auxiliary lists, as shown in the previous example, isn't the only method of retrieving data associated with a list selection. The [ListSelection2.wbt](#) program produces the same results as the [ListSelection.wbt](#) program, but demonstrates different (and more compact) mechanisms for tracking the data.

Earlier, we mentioned that you can create a list of lists by using a different delimiter for each level in the hierarchy. In [ListSelection2.wbt](#), we create a WinBatch list where the sublists are tab-delimited (») and the lists are delimited by carriage returns (↵). We will use this list of lists in place of the several auxiliary lists employed in the [ListSelection.wbt](#) demo.

As before, we will begin by initializing blank lists. This time, there are only two of them:

```
fileIn = "Parts.lst" ; source file for data
listItem      = ""    ; initialize all of the lists
listStkData   = ""
```

The file-open operation hasn't changed and, again, we read one line at a time from the data file and then use `ItemExtract` to retrieve the item name. But, once we have the item name, we insert this piece of data into the `listItem` list. In this example, we simply insert each new item at the end of the list; we do not pad the entry or append an index.

```
sListItem = ItemExtract( 1, sLineIn, @TAB ) ; extract the item name
listItem = ItemInsert( sListItem, -1, listItem, @TAB ) ; insert at
end of list
```

Next, we insert the entire retrieved data line into the `listStkData` list. However, instead of using a tab-delimiter, which is already used in our sublist, we insert a carriage return as the delimiter between the sublists.

```
listStkData = ItemInsert( listStkData, -1, sLineIn, @CR ) ; insert at
end of list
```

Each item is inserted at the end of the list, maintaining the same order in both `listItem` and `listStkData`. At this point, we have two lists: one contains only the item names, and the second contains the item names and all associated data in a list of lists.

As before, the list of item names is copied to `listDisplay` as a sorted list:

```
listDisplay = ItemSort( listItem, @TAB ) ; initialize listbox with
sorted list
ButtonPushed = Dialog( "Selection" )
```

Next, as in the previous example, when the dialog returns, a test is used to determine if a selection was made before copying the selection to the `sItemSelect` variable.

```
If listDisplay != "" ; is there a selection?
sItemSelect = listDisplay ; copy to report display
```

This time, however, we don't have an index value to extract. We only have the list of lists to search for the data we want. To accomplish this, we need to use a `for` loop to temporarily extract each of the sublists as `listTemp`:

```
For i = 1 to ItemCount( listStkData, @CR )
    listTemp = ItemExtract( i, listStkData, @CR )
    If ItemLocate( sItemSelect, listTemp, @TAB ) == 1 Then Break
Next
```

Once we have extracted each sublist, `ItemLocate` is perfectly suited for a test to determine if our target matches the first field in the sublist. If `ItemLocate` returns a value of 1, indicating a match in the first field, we know that we've found the proper sublist, and a `break` statement takes us out of the loop.

If you're familiar with the `Itemxxxx` functions provided by WinBatch, your first thought might be to use the `ItemLocate` function, as `nIndex = ItemLocate(sItemSelect, listStkData, @CR)`. Unfortunately the `ItemLocate` function isn't appropriate since it tries to find a complete match between the substrings identified by the delimiter and the target string; it does not identify a partial match as a reportable result.

Then, with the appropriate sublist, using `ItemExtract` to retrieve the three subfields is easy. Just remember, however, that this time the subfields are in 2, 3, 4 order, rather than being the *n*th items in three separate lists.

```
; get the rest of the particulars from the sublist
sStkQuant = "In stock: " : ItemExtract( 2, listTemp, @TAB )
sPN = "P/N: " : ItemExtract( 3, listTemp, @TAB )
sLoc = "Location: " : ItemExtract( 4, listTemp, @TAB )
EndIf
EndWhile
exit
```

Introduction to Programming

And that's it. This is a more compact method of handling some complex data, not just in terms of the code required but also, and more important, in terms of memory usage. The padding used in the earlier example does require space in memory, and saving that is a fair tradeoff for a bit more complexity in programming.

List Item Removal

One more item list function deserves mention: `ItemRemove`. As its name suggests, the `ItemRemove` function simply removes a designated item entry (or sublist) from a list. `ItemRemove` is called as:

```
list = ItemRemove( nIndex, list, delimiter )
```

Another very appropriate method of create lists is found in the `Arr...` functions where, for example, the `ArrayFileGetCsv` function can read the entire file using a single command instead of opening the file and reading entries individually.

Passwords

When we discussed edit boxes in [Chapter 3](#), we noted that any edit box with a variable name beginning with the characters `PW` is automatically treated as a password entry box, which means that the characters typed are echoed by asterisks (*). This said, there really isn't much else involved in asking for a password, although you can use the `AskPassword` function for a convenient predefined dialog. The `AskPassword` function is called as:

```
pw_Password = AskPassword( "Security check", "Enter password" )
```

For a hacker, most password lists are not particularly secure. In fact, many applications do a poor job of "concealing" their password lists. Rather than storing passwords, a somewhat better method is to use the password entry to generate a numeric value or an encrypted key and to store this value. Later, when a password is entered for access, the same process is repeated, and the result is compared to the stored key. The passwords themselves are never stored and, therefore, cannot be quite so easily compromised. The [Password.wbt](#) program demonstrates a simple key-generation routine.

While often misused, the term hacker properly refers to anyone who is experienced at digging into computer programs and systems to find out how they work. Most good programmers are also hackers. The term cracker properly refers to those who have just enough programming talent to be dangerous without necessarily being good enough to find employment and who massage their puny egos by attacking other peoples' systems. In other words, hackers take things apart to find out how they work; crackers try to destroy what they lack the talents to understand.

Keyboard input

While string entries are the most common form of input, there are occasions when a simple keystroke would be sufficient. Furthermore, sometimes it's useful to be able to watch for a keystroke when another application is running.

The `WaitForKey` and `WaitForKeyEx` functions offer a conditional keyboard-input function that can wait (even in the background) for keys to be pressed and report which key was struck. The `WaitForKey` function takes the form:

```
nKey = WaitForKey( char1, char2, char3, char4, char5 )
```

If your application needs to monitor fewer keys, any of the five arguments can be specified as blanks.

Most keys are entered as a single character string in the form: "a"... "z", "1"... "0", or " " (space). However, some keys, such as "!", are not allowed. (If in doubt, experiment.) Also, `WaitForKey` ignores the state of the Shift key, so it can't differentiate between lowercase and uppercase; "A" and "a" are both allowed, but you cannot use both the uppercase and lowercase characters in the same instruction.

Function keys and some other special keys also can be used, as follows:

Key	Code with WaitForKey
Function keys (F1 through F12)	"{Fn}"
Enter key	"{enter}"
Insert key	"{insert}"
Escape key	"{escape}."
PageUp key	"{pgup}"
PageDown key	"{pgdn}"
Home key	"{home}"
End key	"{end}"

The arrow keys are not accepted in any form.

The `WaitForKey` function is demonstrated in the [WaitForKey.wbt](#) program.

Summary

Relatively speaking, we've spent more than a little time on strings and various aspects of string handling, simply because it is unlikely that you will do very much programming without strings in one form or another. In addition to building and parsing strings, we've

Introduction to Programming

also covered searching and replacing substrings for both case-sensitive and case-insensitive operations and changing the case of strings from mixed to full uppercase or full lowercase.

Since lists in WinBatch are simply a special case of strings, we've also gone into this topic at some length to cover such operations as building lists, sorting lists, retrieving list items, and searching lists. Also, for flexibility, we've shown how to create lists of lists.

Finally, to round out the topic, we've touched briefly on passwords as strings and, in closing, on a method of monitoring the keyboard for specific keystrokes.

Now that you have a fairly good tool set to work with, in [Chapter 7](#), we'll start working with somewhat more sophisticated operations. We will undertake larger tasks and link small procedures and even small WinBatch programs to produce something larger.

So, you've made it this far. Don't back out now—things are just beginning to get really interesting.

CHAPTER 7 : A TOOLKIT FOR OPERATIONS

FUNCTIONS AND SUBROUTINES

function – a function is a form of subroutine that returns a single value to the main program – The PC User's Pocket Dictionary

subroutine – (a.k.a. subprocedure) a related set of instructions that perform a single task, called by name from the main program. Commonly performed tasks are isolated into subroutines so that they can be used over and over by different parts of the same program – The PC User's Pocket Dictionary

WinBatch provides a variety of ready-to-use tools, in the form of predefined functions, which are the principal components used in building applications. When the WinBatch functions do not satisfy your programming needs, you can define your own subroutine (UDFs – i.e., User Defined Functions or Subroutines) or use external batch files. Another option is to call independent executable applications from your programs.

IMPORTANT TIP: In WinBatch Studio, if you click on a function name (shown in blue) and then hit Shift-F1, WinBatch Studio will open the help page for that function.

Functions

In previous chapters, a number of functions have already been introduced but, thus far, we have not discussed functions in general. A *function* is simply a predefined piece of code that may accept one or more arguments (parameters), performs a task and, optionally, returns a result value.

Functions have the form:

```
FunctionName( parameter1, parameter2, ... )
```

For example, the `StrCat` function is defined in the WIL (Windows Interface Language) Reference Help file as:

StrCat

Concatenates two or more strings.

Syntax:

```
StrCat( string1, string2, ..., stringN )
```

Parameters:

Introduction to Programming

(s) `string1`, etc. at least two strings you want to concatenate

Returns:

(s) concatenation of the entire list of input strings

In this case, the ellipsis (...) indicates that a variable number of arguments can be supplied. In most cases, however, a function requires a set number of parameters that must be supplied in the order specified.

The `:` concatenation operator can be used quite conveniently in place of the `StrCat` function.

In the help file, five conventions are used to denote the types of data used as parameters and as return values.

Convention	Data Type	Description
(a)	array	A collection of data elements as sets of data (see Chapter 4).
(f)	float	A fractional (decimal) numeric value (see Chapter 4).
(h)	huge number	A long decimal number string, which may represent a number too large to be converted to an integer. 'Huge number' is a special data type; a long decimal number string, which may represent a number too large to be converted to an integer. This value cannot be modified with standard arithmetic operations; it requires the use of the Huge Math extender. (For example, the function <code>FileSize</code> can be called with a flag to return a Huge number).
(i)	integer	A numeric value that must be a whole number; also used to pass or return Boolean (TRUE/FALSE) values (see Chapter 4).
(s)	string	A word, sentence, or array of characters; may also be a list or a variable name (see Chapter 6).

The WIL Reference Help file also provides brief examples of how functions are used. For the `StrCat` function, the example looks something like this:

Example:

```
user = AskLine("Login", "Your Name:", "", 0)
msg = StrCat("Hi, ", user)
Message("Login", msg)
```

The help file also offers advice and a description of how a function is used, as well as comments about any idiosyncrasies or special features that apply to the function. For instance, in the description of the `StrCat` function, you will find that the sample code could also be written using substitution:

```
sMsg = "Hi, %sUser%"
```

There is also a caution that substitution should be used only for simple, short cases.

The main point here is that the WIL Reference Help file is an excellent source of reference information. When you are in doubt about how to use a function, you should refer to the help file for the list of calling parameters, format, and other information.

User Defined Functions

Some languages, such as C/C++, allow programmers to create their own functions to perform tasks, including defining the calling parameters and the return value types. While previous versions of WinBatch did not provide a means of defining your own functions (except as sub-procedures), now WIL supports user-defined functions (UDF's).

There are two types of User Defined Functions (UDF's):

- `#DefineFunction` UDF's; variables are local to the UDF and cannot be referenced by the calling function or procedure.
- `#DefineSubroutine` UDF's; variables are global and accessible to the calling function or procedure.

A `#DefineFunction` UDF is defined as follows:

```
#DefineFunction funcname(param1, param2, param3,... param16)
    <...code...>
Return retval
#EndFunction
```

`#DefineFunction` and `#EndFunction` are the keywords indicating the beginning and end of the UDF.

`funcname` is a placeholder for the name of the function. The function name must begin with a letter, can contain letters, numbers, and underscores, and can be up to 30 characters long. You may not use a function name that is the same as the name of a WIL DLL function, but you may override a function name that's in an extender DLL.

WIL extender DLLs are special DLLs designed to extend the built-in function set of the WIL processor. These DLLs typically add functions not provided in the basic WIL set, such as network commands for particular networks (Novell, Windows for WorkGroups, LAN Manager and others), MAPI, TAPI, and other important Application Program Interface functions as may be defined by the various players in the computer industry from time to time. These DLLs may also include custom built function libraries either by the original authors, or by independent third party developers. (An Extender SDK is available). Custom extender DLLs may add nearly any sort of function to the WIL language, from the mundane network math or database extensions, to items that can control fancy peripherals, including laboratory or manufacturing equipment. See [Chapter 16](#) for more on Extender DLLs.

You may specify up to 16 optional parameters. `Param1 ... param16` are placeholders for your actual variable names that your UDF will receive when it is called.

Between the `#DefineFunction` and `#EndFunction` keywords is the code that will get executed when the UDF is called. It may contain a `Return` command followed by a value (or an expression that evaluates to a value), in which case the UDF will end and return this value. If you specify a `Return` command without a value, the UDF will return 0.

If a UDF does not contain a `Return` command, it will execute to the end of the UDF and return 0.

An `Exit` command in a UDF will cause the entire script to end, not just the UDF.

A UDF may be defined anywhere in a script, as long as it is defined prior to being used for the first time. A UDF may be defined or used in a separate script that is called with the `Call` command, as long as it is defined before it is used. You may not have nested UDF definitions (i.e., each `#DefineFunction` must be followed by an `#EndFunction` as a pair).

A `#DefineFunction` UDF will not have access to any variables in the main WIL script, other than the variables passed as `param1 ... param16`. Any variables set in a UDF will be destroyed when the UDF returns, other than the return value of the UDF. Any percent signs in the UDF code will be expanded at runtime (when the code is called), not at define time.

You may return a file handle or binary buffer or COM/OLE object from a UDF using the `Return` command. However, if you open one of these types of handles in your UDF and do not return it using the `Return` command, you are responsible for freeing it before the UDF returns (i.e., `FileClose` or `BinaryFree` or `{object} = ""`); otherwise, the object represented by the handle will become an "orphan" and will no longer be accessible and may not be automatically freed when the script exits.

Subroutines

The principal difference between a function and a subroutine is that the subroutine is not called with an argument list and does not return a value. Instead, a subroutine uses variables already defined within the program and returns results by setting values for one or more variables.

Subroutines offer two potential benefits.

- Subroutines can be used to organize an application by creating and identifying blocks of code devoted to specific tasks. Giving an application a structure in this fashion makes it more convenient to locate, observe, and debug particular tasks.
- Any task that will be used more than once, from different points in the program, can be placed in a subprocedure. This lets you avoid writing duplicated code and means that only one copy of the code needs to be debugged.

External versus Internal Subroutines

In WinBatch, we have a choice between using external subroutines written as external .wbt batch files and using internal subroutines called as `gosub` instructions. In choosing between these two forms, the external subroutine offers a few advantages including:

- isolation from the calling program – that is, the external subroutine does not share variable names with the calling application, thus avoiding possible conflicts.
- reusability – since the subroutines are contained in external files, once a subroutine has been created and debugged, it can be used with multiple applications.

The disadvantages parallel the advantages:

- isolation means that all arguments must be passed using the command-line parameter format.
- unlike a `gosub` subroutine, the external file containing the subroutine must be explicitly referenced when the subroutine is invoked.
- In addition, when compiling a script and its subroutines into an executable file, remember that all called subroutines must be compiled using the encrypted or encoded option of the compiler.

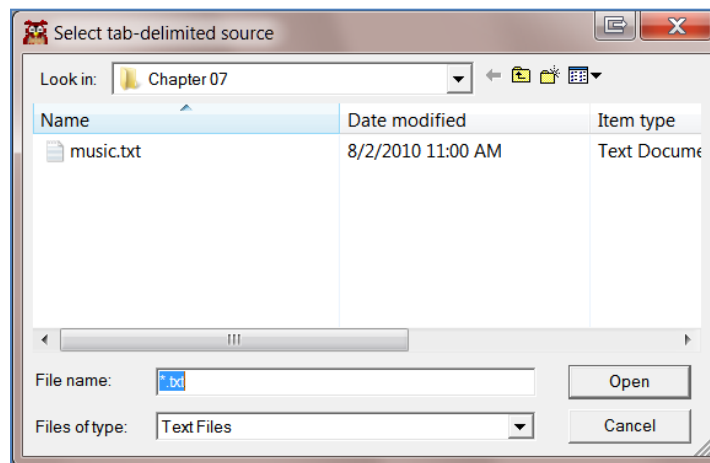
Which form – internal or external – you choose to employ is mostly a matter of circumstance and preference. If the routine is one which you expect to use frequently with different applications, then creating an external subroutine has the advantage of convenient reusability and may well be worth your while.

As an additional example, the [Mortgage.wbt](#) program in [Chapter 9](#) includes provisions to call an external subroutine – from [FormatCurrency.wbt](#) – or to call the same routine as a `gosub` instruction from `:Format_Dollar_String`. Both sets of instructions are in the [Mortgage.wbt](#) program. To test the alternatives, simply change the variable value for `bExternal` to either `@TRUE` or `@FALSE` and execute.

The general format of a subroutine is not complex. The application uses a `gosub` instruction that references a label identifying the subroutine. The subroutine terminates with a `return` statement, which sends execution back to the next statement following the `gosub` instruction.

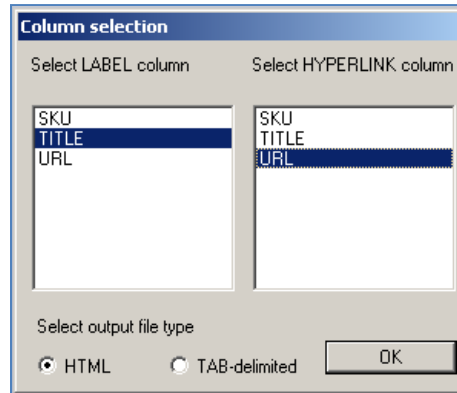
For an example, let's look at the [HyperLink.wbt](#) program which uses several subroutines. The [HyperLink.wbt](#) program is a utility that allows the user to create an HTML page with links from a large body of data supplied as a multicolumn spreadsheet.

The [HyperLink.wbt](#) utility uses two dialogs. The first is a file-selection dialog:



This dialog just shows a directory tree with the default file mask set to `*.txt` so that a tab-delimited text file can be loaded as the source. (Most spreadsheets can export data in a tab-delimited format, with tab characters representing the column breaks and carriage return/linefeed breaks identifying rows.)

Once a file has been selected, the second dialog begins by reading the first line of the tab-delimited file—the line containing the column headers—and presenting two item lists containing the individual headers. A selection from one list identifies the label column, and the second list identifies the hyperlink information.



Options are also provided to select either an HTML format (one containing only a list of hypertext links) or a tab-delimited format, where the original columns are retained except that the labels are reformatted as hyperlink references.

In the illustrations, the program is being used with a file listing of music selections that are available through the Internet. The tab-delimited source file is included with the program files, but it supplied for demonstration purposes only. It is quite likely that some of the links will not be valid, since links frequently change over time.

The program was written so that it could be employed in a generic fashion rather than being limited to a single data source and format. It has been employed in preparing a list of antique auto parts for publication on a web site and for parsing several other database-derived files for publication. You may, of course, modify the source file as desired and put it to such uses as you see fit.

The gosub Statements

The [HyperLink.wbt](#) program begins with a sequence of `gosub` statements executing subroutines:

```
DirChange( DirScript() )
```

```
sFileIn = "music.txt"
```

```
sFileOut = ""
```

```
gosub selectFile
```

```
gosub selectColumn
```

```
gosub testSelection
```

```
gosub processFile
```

```
exit
```

Introduction to Programming

The four `gosub` statements have self-explanatory labels, and each statement sends execution of the program to the label identifying the subroutine. For example, the entry point for the `selectFile` subroutine is identified by the label `:selectFile`:

```
;=====
;  Select the input (source) file
;=====
:selectFile
```

Notice that the label itself must be identified by a leading colon, although the `gosub` statement does not include the colon in the reference.

The three comment lines that precede the label provide a visual identifier, making it easy for the programmer to locate the subroutine and offering a brief amplification of the purpose and function of the subroutine. The comments and their placement are optional—they could follow the label or could be omitted entirely.

The Subroutines

The code appearing within the subroutine can be virtually anything desired. In the `selectFile` code, the subroutine simply invokes a predefined function: `AskFileName` as:

```
:selectFile
    sFileIn= AskFileName( "Select tab-delimited source", "", "Text
Files|*.txt", "*.txt", 1)
return
```

Each subroutine must terminate with a `return` statement. The `return` statement sends program execution back to the point immediately following the invocation of the `gosub` statement. In this example, the next statement in line is another `gosub`, but the program can resume execution at any point for any purpose. (as in the `testSelection` and `processFile` subroutines, described shortly).

Also notice that the variable `sFileIn`, which will return the name of the selected file, was initialized before the subroutine was called. In this case, initialization provides a mask used to limit the files displayed for selection. Without this initialization, the `FILELISTBOX` member of the dialog would default to showing all files. In effect, the `FILELISTBOX` member provides its own internal initialization.

In many other cases, however, variables must be initialized before being used. For instance, calling the `StrCat` function with an uninitialized string as an argument will cause WinBatch to declare a fault and interrupt execution. WinBatch rejects the uninitialized argument because it could contain anything—it could be pointing at some

location in memory that holds whatever values were left there by other operations—or it could contain nothing.

When in doubt, always initialize variables before use. If there are no default values to be assigned, float and integer variables should be initialized as 0; string (and list) variables should be initialized as an empty string ("").

When the `testSelection` subroutine is called, the application already has a number of variables that have been read by previous subroutines. The `testSelection` subroutine extracts specific information and queries the user for confirmation before proceeding:

```

;=====
;   Requests confirmation of the file and option selection
;=====
:testSelection
    nHyperLink = ItemLocate( listHyperlinkColumn, listColumns, @TAB )
    nLabel = ItemLocate( listLabelColumn, listColumns, @TAB )
    If( nLabel == 0 ) || ( nHyperLink == 0 ) Then exit
    GoSub formatLine
    sReport = 'The output format will appear in the format:
[':sLineOut:']. If this is correct, select "Yes" to continue.'
    If AskYesNo( "Question", sReport ) == @NO Then exit
return

```

Notice the call to another subroutine:

```
GoSub formatLine
```

After the `formatLine` subroutine is called and returns, execution simply resumes within the `testSelection` subroutine.

The `formatLine` subroutine is also called from the `processFile` subroutine, but this time, it is called from inside a `while` loop. In this case, the subroutine is called multiple times—each time the loop repeats—always returning execution to inside the `while` loop.

```

;=====
;   Processes the input file according to selections
;=====
:processFile
    sFileIn = FileFullName( sFileIn )
    sFileOut = StrFix( sFileIn, "", StrLen( sFileIn ) - 4 )

```

Introduction to Programming

```
If rbHypertext == 1 then
    sFileOut = StrCat( sFileOut, ".HTML" )
Else
    sFileOut = StrCat( sFileOut, ".LST" )
EndIf

hFileOut = FileOpen( sFileOut, "WRITE" ) ; open the output file and
get a handle

If rbHypertext == 1
    FileWrite( hFileOut, "<html>" )
    FileWrite( hFileOut, "<body>" )
EndIf

While @TRUE
    sLineIn = FileRead( hFileIn )
    If sLineIn == "*EOF*" Then break
    gosub formatLine
    If sRef != ""
        FileWrite( hFileOut, sLineOut )
    EndIf
EndWhile

If rbHypertext == 1
    FileWrite( hFileOut, "</body>" )
    FileWrite( hFileOut, "</html>" )
EndIf

FileClose( hFileIn ) ; close the input file
FileClose( hFileOut ) ; close the output file
Message( "Done", sFileOut )

return
```

Finally, the `formatLine` subroutine uses the information read from the file and the column selections specified in the `selectColumn` subroutine to format an output string.

```
;=====
;   Format the column fields for output
;=====

:formatLine
    sTemp = ""
    If rbHyperText == 2
```

```

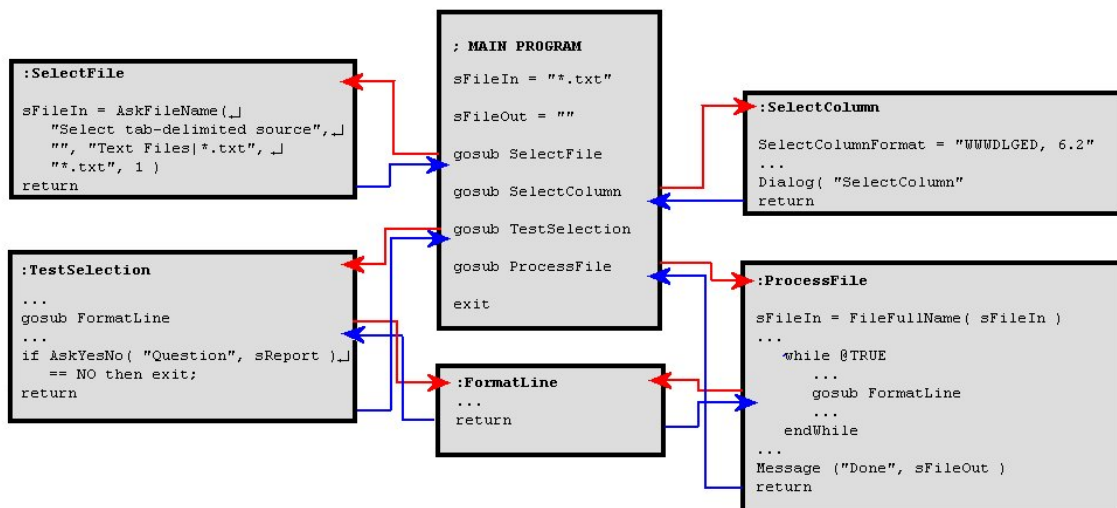
For i = 1 to ( Min( nLabel, nHyperlink ) - 1 )
    sTemp = StrCat( sTemp, ItemExtract( i, sLineIn, @TAB ), " " )
Next
EndIf

sRef = ItemExtract( nHyperlink, sLineIn, @TAB )
sName = ItemExtract( nLabel, sLineIn, @TAB )
If sRef != ""
    sLineOut = StrCat( sTemp, '<a href=', sRef, '>', sName,
'</a><br>' )
Endif
return

```

Subprocedure Execution

At this point, you may be feeling a little confused about what is happening and to whom. The following is a diagrammatic representation of the [HyperLink](#) program, showing how the various `gosub` instructions direct execution through different subroutines.



Execution using gosub instructions

The main program uses a sequence of four `gosub` statements, and both the `testSelection` and `processFile` subroutines make calls to the `formatLine` subprocedure. This use of `formatLine` demonstrates one of the strengths of subroutines: The same block of code can be employed more than once, from different locations within the program, without needing to repeat the code. In this example, there is only one `formatLine` subprocedure, but it can be called from anywhere within the program as needed.

You'll see subroutines used in later examples in this book both to avoid duplicating code and to organize programs.

External Batch Files

WinBatch also provides functionality allowing other .WBT programs to be called and to return results to the calling program.

The `Call` function accepts two parameters:

- The name of a .WBT program file to be invoked
- A list containing the names of variables to be passed as parameters

Here is an example where a program file called [GetData.wbt](#) is invoked with three arguments named in the quoted list:

```
Call("GetData.wbt", "'Parts List.lst' sList nCount")
```

The first argument is the name of a file that contains data to be used to create a list. The second argument is the name of the variable that will receive the generated list. The final argument is the name of the variable to receive the item count.

An external batch file is essentially a small utility program to be invoked by another .WBT application.

There is an important distinction here between the list of variable names and the variables. The parameter list consists of string arguments naming variables that belong to the calling application; these are simply labels and are not the variables themselves. Instead, under WinBatch, the named variables are globally available both to the calling .WBT application and to the .WBT application called.

A maximum of nine (9) arguments can be passed as command-line parameters.

In this fashion, the called application can make changes by assigning values to the variables named as parameters, thus passing the results of operations back to the calling application.

The `Run` function, allows other .WBT files to be launched as independent applications. Using `Run`, arguments can be passed as command-line parameters in the same fashion as with the `Call` function, but the two programs do not share variables. In other words, the application launched using `Run` cannot return values to the calling application.

The [ExternCall.wbt](#) program demonstrates using the `Call` function to execute two external batch files.

So, the first `Call` invocation is made to the [GetData.wbt](#) program:

```
nCount = 0  
sList = ""
```

```

DirChange( DirScript() )
; open the source file and return an array of entries
Call( 'GetData.wbt', '"Parts List.lst" sList nCount' )
Message( "List count", "Found ":nCount:" items in [":sList:"]" )

```

The program reports the number of elements found in the source file (`Parts List.lst`), followed by the generated list. If the file couldn't be opened because it wasn't found, the variable `nCount` is returned with a zero count, and the string `sList` contains a brief error message that is displayed.

Next, because the returned list is unsorted, [ExternCall.wbt](#) will invoke another .WBT file, [SortData.wbt](#), to put the data in alphanumeric order. This time, however, instead of a file name, the arguments passed are the name of the array and the delimiter used in the array:

```

Call( "SortData.wbt", "sList @TAB" )
Message( "Sort results", "Sorted ":nCount:" items as [":sList:"]" )
exit

```

Again, the results are reported on return from the call.

Let's take a look at the two external programs called by [ExternCall.wbt](#) to see how we pass arguments to routines invoked using the `Call` function and, within these external programs, how the parameter lists are recognized and used.

The First External Program

When the [GetData.wbt](#) program is called – the variable `param0` contains a count of the number of parameters received. We begin by testing to ensure that the appropriate number of arguments was passed when [GetData.wbt](#) was invoked. A series of tests is also applied to check the parameter types. Regardless of what we plan to use these variables for, `param1` is expected to be a file name, and `param2` and `param3` are each expected to be labels (strings). We also check that the filename passed as `Param1` exists.

```

If param0 < 3 ; insufficient arguments
    Message("Attention", "This script is not meant to use used alone.
    It is used by other scripts")
    exit
Endif

If IsNumber( param3 ) Then exit ; parameter 3 isn't a variable name
If IsNumber( param2 ) Then exit ; parameter 2 isn't a variable name
If IsNumber( param1 ) Then exit ; parameter 1 isn't a filename

```

Introduction to Programming

```
If FileExist( param1 ) == 0
    %param2% = "File Error"
    %param3% = 0
    return
Endif
```

Once we're relatively sure that the arguments are appropriate, the next step is to initialize a couple of local variables, then open the file to read the contents.

```
nIndex = 0 ; initialize a count index
sResult = ""
```

```
hFileIn = FileOpen( param1, "READ" ) ; open the input file and get a
handle
```

The `FileOpen` function expects a string argument with the file name. However, rather than a literal string to pass, `param1` contains the name of a variable containing the name of the file. To pass this name as an argument, instead of the usual string enclosed in quotes, we use the `param1` argument directly.

If you use the debugger to step through execution, you'll be able to watch the `param1` variable and see that it does, indeed, contain the appropriate file name. Using the debugger is discussed in [Chapter 15](#).

Once the file is opened, the rest of the process is relatively routine. The contents of the file are read. This file is a list of lists. The first item in each list is used to build the data that we intend to return as an item list.

```
While @TRUE
    sTemp = ""
    sLineIn = FileRead( hFileIn )
    If( sLineIn == "*EOF*" ) Then break
    nIndex = nIndex + 1
    sTemp = ItemExtract( 1, sLineIn, @TAB )
    sResult = StrCat( sResult, sTemp, @TAB )
EndWhile

FileClose( hFileIn ) ; close the input file
```


Once we're finished with the file, we need to transfer the item list and the item count from the local variables, which will not be accessible when [GetData.wbt](#) returns, to the variables identified by `param2` and `param3`. Remember that we don't access the variables themselves; instead, we get the names that [ExternCall.wbt](#) uses for the variables. The [GetData.wbt](#) program could be called by any WBT application, so we cannot be sure which variable names the calling application will use.

By using the `%substitution%` operation, we can assign the values in `sResult` and `nIndex` directly to the `sList` and `nCount` variables belonging to the [ExternCall.wbt](#) program. We do this by enclosing the names contained in `param2` and `param3` in percent signs:

```
%param2% = sResult ; assign the result string to param2
%param3% = nIndex ; assign the count to param3
Drop( sLineIn, nIndex, sResult, sTemp ) ; discard local variables
Return
```

Since we're using substitution instead of writing the variable names directly into the code, the uncertainty about which names the calling application will use is not a concern.

The `Drop` function, which is invoked before the [GetData.wbt](#) program returns, is not strictly essential. The `Drop` function simply removes the listed variables from memory, performing cleanup and saving some memory space.

When variables are declared (by appearing to the left of an equal sign, as in `nVar = value`), memory is allocated to hold each variable. These variables remain allocated until the `Drop` function is invoked or until WinBatch exits. Normally, we aren't too worried about clean up. Most of these demo programs are simple, and once they're finished, any variables used are de-allocated on closure. Here, a small provision for cleanup seems appropriate because we're writing what are essentially small utility programs to be invoked by another .WBT application.

As a general rule, applications should clean up after themselves by using the `Drop` function to remove variables from memory when they are no longer needed. It is possible for WinBatch applications (and other applications) to exhaust system resources unnecessarily.

The Second External Program

The second external program called from the [ExternCall.wbt](#) program is [SortData.wbt](#). This program relies on a standard function supplied by WinBatch.

Here, we have only two arguments passed as parameters: the string (item list) and the delimiter used in the list. To sort the list, we invoke the `ItemSort` function with the two arguments, accept the sorted list in a local variable (`sList`), and then assign `sList` to the original list argument:

```
If param0 < 2 Then exit
```

Introduction to Programming

```
If IsNumber( param1 ) Then exit      ; should be list of data
If IsNumber( param2 ) Then exit      ; should be char (delimiter)

sList = ItemSort( %param1%, %param2% )
return
```

This task would have been just as easy (or easier) to accomplish in the original program. The only reason for invoking [SortData.wbt](#) as a subprogram is to show a second example of using the `Call` function.

Executable Programs

While the `Call` function allows a WinBatch application to call other WinBatch programs interactively, the `Run` function allows WinBatch to launch other executable programs. Executable programs include .EXE, .COM, .PIF, and .BAT files, as well as data files associated with executable programs.

Like the `Call` function, the `Run` function accepts two parameters: the name of the executable program and, optionally, a list of command-line parameters. Its format is:

```
Run( program_name, parameters )
```

If a data file, such as a .DOC file, is executed using the `Run` command, any command-line parameters will be ignored.

If only the executable name is provided, without a drive/path specification, the `Run` function searches the current directory first, the Windows and Windows/System directories next, and the DOS path specification last. If the requested program is found, the `Run` function returns `@true`; if an error occurs, it displays an error.

Unlike DOS, where an error result could be returned from a launched application, once a Windows application has been launched using the `Run` function, there is no further communication with the .WBT program. Under DOS, batch programs sometimes relied on error codes returned by launched applications. In a multitasking environment such as Windows, however, launched applications are independent and there is no readily returned error code.

The [Run_Exe.wbt](#) program demonstrates running an executable program. It begins by displaying a file directory dialog with the file mask set to `"*.exe"`, allowing an executable file to be selected. The dialog also offers an edit box for the entry of command-line parameters. After the user makes a selection and, optionally, supplies parameters, the `Run` function is called to execute the chosen program.

This is the relevant portion of the code for [Run_Exe.wbt](#):

```

DirChange( DirScript() )

:loop
sFileRun = AskFileName( "Select executable application", DirWindows(2),
"EXE Files|*.exe", "notepad.exe", 1 )
sArgs = AskLine( "Enter arguments", "Enter arguments: (optional)", "" )
if Run( sFileRun, sArgs ) then goto loop
Message( "Error", "Can not run " : sFileRun )
goto loop
exit

```

The labeled subprocedure `loop` does not need a `gosub` instruction because it executes automatically. The final instruction `goto Loop` keeps this set of instructions executing until closed by clicking on the `CANCEL` button.

Summary

We began this chapter by introducing the concept of functions and how functions are called and used. We explained how parameter types and return types are identified (in the WinBatch documentation) and offered some suggestions for using the reference help file.

The next subject was how WinBatch allows you to create subroutines, which serve a similar purpose to custom functions. The `gosub` and `return` commands were introduced as elements used in calling and returning from subroutines. Subroutines are always a part of the same application file.

We then described how to call subroutines provided by external `.WBT` files, explaining how arguments and values are passed to external subroutines and how results can be returned.

Last, we discussed launching external applications, not as external subroutines but simply as independent programs.

In this chapter, we mentioned that one of the advantages of using subroutines is that they help add structure to your applications. The subject of structuring applications will be expanded on in upcoming chapters. More immediately, in [Chapter 8](#), we'll look at the principal methods of controlling operations within an application.

CHAPTER 8 : GOING WITH THE FLOW

CONTROLLING OPERATIONS

flow \`flō\ n. 3: a smooth, uninterrupted movement.

enumerate \i-`n(y)ü-me-rāt\ v. 2: to specify one after another.

One of the basic requirements in any programming language is a means of controlling the flow of execution within an application. We need to be able to direct an application to perform different tasks in response to user inputs, the results of previous operations, or the results of testing data values from a file or another source.

We have two basic choices for controlling the execution of operations: use a stop-and-ask approach or write decision mechanisms.

The stop-and-ask approach involves having the application stop and request instructions every time a choice is required. Although this is theoretically possible, it would be an extreme exercise in frustration, both for the user and the programmer.

The frustrations to the user should be obvious. Using a program that required you to make all of these decisions—in terms the computer could understand—would be orders of magnitude harder than simply performing the task without using a computer.

From the programmer's standpoint, the amount of code required to query the user at each decision point—to explain where the program had reached and ask what choice to make—would not be nearly as onerous as the job of writing the code to handle the selections. In almost every case, handling the user's selections would require more work than simply writing code to execute a logical decision without asking.

Our alternative, writing decision mechanisms, is really what programming is about. We use decision mechanisms to direct the program to perform the appropriate tasks at each juncture, causing the application to perform different task sequences according to the information available at each point.

Branching and Program Control Mechanisms

If everything in an application was a strictly linear process—open file A, read value B, close A, and write to C—its flow chart would look like this:



A linear execution

If we were able to diagram an application like this, there would be little reason for us to write a program to handle the job. After all, if an application was designed to perform

Introduction to Programming

exactly and precisely absolutely identical steps (with identical data) every time it was run, then why write the program in the first place?

Instead of a linear flow, we usually require *branching* execution, or sending the program to perform different sets of tasks. The practice of branching is one you should find yourself using frequently, because few practical programs are purely linear.

There are occasions when the 'very simple' is useful, but they are rare. For example, years ago, a very useful and simple utility required 14 lines in assembly code to ensure that, when the computer was booted, the NumLock key was turned off. That utility is now obsolete, since that function can now be found in the BIOS settings.

In contrast, virtually all applications depend on conditional instructions and on branching – often using `goto` and `gosub` instructions – to perform different sets of operations.

Goto and Gosub Branches

The `goto` and `gosub` instructions are similar in operation. Both instructions are called with a label and execution of the program branches to continue with the instructions immediately following the label. The label in the application program is simply a unique word which is preceded by a colon (:). For example:

```
goto label_a
exit

:label_a ;label serving as the target for a goto or gosub statement
{statements to execute}
```

Here the `goto` instruction branches to the point in the program identified as `:label_a` and execution simply continues from that point.

The `gosub` instruction functions in a similar fashion except that the `gosub` expects to reach a `return` statement which sends execution back to the point following the original `gosub`. For example:

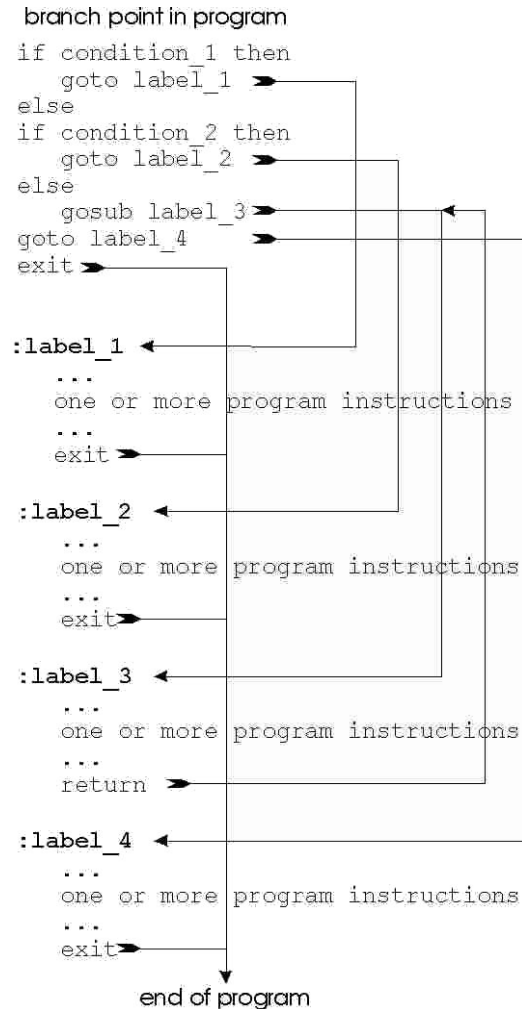
```
gosub label_a
    {further statements to execute}
exit

:label_a ; label serving as the target for a goto or gosub statement
    {statements to execute}
return
```

Here the `gosub` branches to the point following `:label_a`, executes the statements following the label and then – when the `return` statement is reached – returns to the instructions following the `gosub` statement.

Most often, a `gosub` – or a `goto` – statement would be reached after a conditional test – a decision to execute a subroutine for some purpose and to then return to the original portion of the program. This is not an absolute, of course, since both `gosub` and `goto` statements can be used simply to organize program code or to use a single subroutine multiple times by calling it from more than one point in a program. For examples, see the topic Subroutines and the [HyperLink.wbt](#) program in [Chapter 7](#).

The figure below shows a diagrammatic representation of a program where conditional statements use `goto`'s and `gosub`'s to branch to program labels to perform different subtasks.

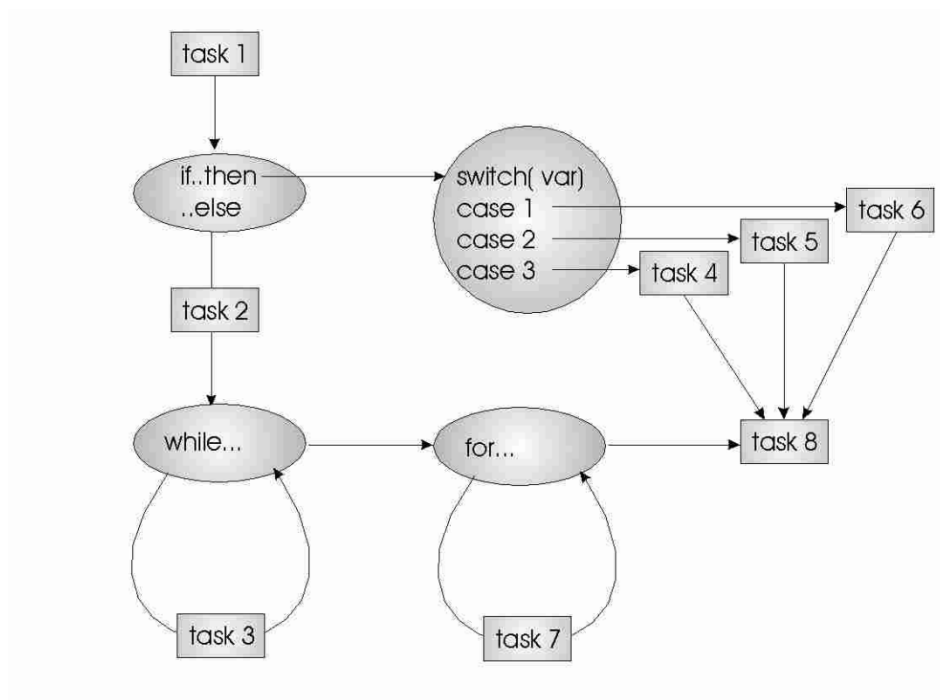


Branching using goto and gosub instructions

Notice that the subroutines following `:label_1`, `:label_2` and `:label_4` conclude with `exit` statements while `:label_3` – which is reached with a `gosub` rather than a `goto` branch – ends with a `return` statement allowing execution to resume – following the original `gosub` – with the `goto label_4` instruction. The illustrated example, of course, is only a hypothetical structure and there are several other mechanisms used for controlling execution.

Forms of controlled branching

Typically, we need our applications to execute complex operations, and this requires methods to provide control and to allow the application to make decisions. Basically, there are four basic types of control mechanisms exercised in computer programs: `if` statements, `select` statements, `for` loops, and `while` loops. These types of control are illustrated here:



Types of program flow control

Each of these types of flow-control mechanisms has a variety of subtypes and can be structured in several fashions.

If Decisions

There are five basic structures which a conditional `if` statement can take:

The first is a simple `if ... then` where a single statement is executed if the conditional is true thus:

```
if expression then statement
```

Notice that there is no `endif` needed to close the conditional execution because only one statement or instruction can be executed

The second is the structured `if ... endif` statement, thus:


```

if expression
    series
    of
    statements
endif

```

... which allows several instructions or a block of instructions to be executed before the block terminates.

Third, the `if.. then ... else ...` format allows either of two statements to be executed as:

```

if expression
    then statement
    else statement

```

Like the single statement `if ... then` format, there is no terminating `endif`.

Fourth is the structured `if ... else ... endif` statement, thus:

```

if expression
    series
    of
    statements
else
    series
    of
    statements
endif

```

... which provides two alternatives, each of which can be one or more statements.

Depending on the conditional test, only one of these blocks will be executed, the other will not.

Finally there is the complex `if...elseif...else...endif` format which provides two (or more) tests and two (or more) alternatives as:

```

if expression
    series
    of

```

Introduction to Programming

```
statements
elseif expression
    series
of
statements
else
    series
of
statements
endif
```

This type of `if` statement can take any of many forms and these can – potentially – become quite complex and, also potentially, quite confusing. Of course, the confusion belongs to the programmer, not the application because, to a computer, everything is quite simple; either a condition is met or it is not.

In each of these varied conditional statements, we begin with a statement of the form:

```
if (test condition is true) then (do this)
```

If the stated condition is found to be `TRUE`, then the defined task (or tasks) is executed. Alternatively, if the test condition is not evaluated as `TRUE`, then the task is not performed.

True or False

In developing your test conditions, you need to be aware of what constitutes a Boolean `TRUE` or `FALSE`. To the computer, a Boolean `TRUE` is simply a nonzero result, and a Boolean `FALSE` is a zero result. Thus, if we write a test condition as:

```
if ( a - b ) then ...
```

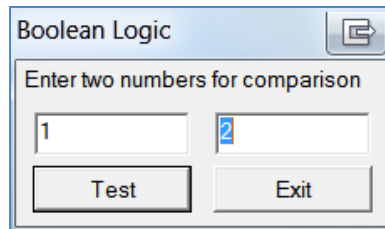
...where `a` equals 1 and `b` equals 5, the result is a `-4`, which is decidedly not zero; therefore, the test result is `TRUE`.

Conversely, if we write a test condition as:

```
if( a == b ) then ...
```

...the equality test (`==`) returns a zero result, correctly identifying the two arguments as unequal.

For a simple demonstration of Boolean `TRUE/FALSE`, run the [Logic.wbt](#) program. The [Logic.wbt](#) demo begins by asking for two numbers:



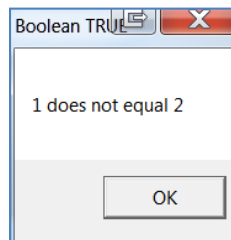
A Boolean comparison

The test used in [Logic.wbt](#) is:

```
If( nNumber1 - nNumber2 )
    Message( "Boolean TRUE", nNumber1 : " does not equal " : nNumber2 )
Else
    Message( "Boolean FALSE", nNumber1 : " equals " : nNumber2 )
Endif
exit
```

This particular format emphasizes the point that Boolean tests may not be completely intuitive. In this test, a `FALSE` result is recognized if the two values are the same, and a `TRUE` result is returned if they are different.

The program reports the evaluation coherently in a message box:



Reporting Boolean results

The point to keep in mind is that how the computer interprets `TRUE` and `FALSE`—as values of nonzero versus zero—is not necessarily the same as how you are accustomed to interpreting true and false.

Simple Tests

The simplest form of an `if` statement is:

Introduction to Programming

```
if( test ) then ...
```

This form can be shortened by omitting both the word `then` and the parentheses:

```
if test ...
```

In either variation, if the test condition yields a Boolean `TRUE`, the statement that follows is executed.

Test conditions can take many different forms. For example, you can use the comparison operators (described in [Chapter 5](#)) or WinBatch functions that report `TRUE` or `FALSE` results, such as `IsNumber` or `IsMenuChecked`.

In many cases, you will want to have more than one statement executed if a test condition is `TRUE`. For these cases, the format is:

```
if ( test ) then
    statement one
    statement two
    ...
    statement n
endIf
```

In this form, any number of statements can be controlled by the `if` statement the conditional list ending when the `endIf` statement is reached.

The stepped indentation is used to show statements that are executed as a group as, in this case, when the `if` statement is satisfied. This indentation is solely for the programmer's benefit and is not required by WinBatch or by other languages or compilers.

Another form of the `if` statement, executes an action when the test is `TRUE` and an alternate action when the test is `FALSE`. The general format is:

```
if( test ) then statement one
else statement two
```

Here, `statement one` is executed if the test is `TRUE`, and `statement two` is executed if the test is `FALSE`.

To execute multiple statements for `TRUE` and `FALSE` tests, use the form:

```

if ( test ) then
    statement one
    statement two
else
    statement three
    ...
    statement n
endif

```

Compound Tests

Quite often, you'll need test statements that take account of more than one factor within a single test. For example, we can write an `if` test as:

```

if ( test_1 && test_2 ) then ...

```

In this case, both `test_1` and `test_2`, either or both of which can be complex expressions, must evaluate as `TRUE` before the test passes.

As an example, suppose that our program will request a value in the range of 32 to 212, representing the temperature range in degrees Fahrenheit for water to remain fluid (at normal atmospheric pressures). We could write a simple test using the AND operator to ensure that the value entered was in the valid range:

```

if ( ( nTemp >= 32 ) && ( nTemp <= 212 ) ) ...

```

Now, if a value of 216 were entered by accident, the first conditional test would pass, because the value is greater than 32. However, the second test would fail, and the `if` statement would remain unsatisfied, which is exactly what is desired.

In a similar fashion, we can write a condition where either of two tests will satisfy the `if` statement:

```

if ( ( nTemp <= 32 ) || ( nTemp >= 212 ) ) ...

```

Using the OR operator, this condition says that a temperature less than 32 (freezing) or a temperature greater than 212 (steaming) is acceptable.

This form can be extended to include more tests, like this:

```

if( ( ( test1 && test2 ) || (test3 && test4 ) ) && ( test5 || test6 ) )
...

```

Introduction to Programming

Here, either `test5` or `test6` can be true, but the statement also requires one of the other pairs—`test1` and `test2` or `test3` and `test4`—to satisfy the conditions.

Alternatively, we might want to write a test where one of two conditions was required to be true but not both (called an XOR test). For this, we could use negation in a complex condition, thus:

```
if ( ( test1 || test2 ) && ! ( test1 && test2 ) ) ...
```

Here, we have two separate sets of conditions: the first, `(test1 || test2)`, is satisfied if either `test1` or `test2` is valid. The second, `! (test1 && test2)`, imposes negation and is evaluated as `TRUE` as long as either one or both test conditions are `FALSE`. The result is that the combination requires one of the test conditions to evaluate as `TRUE` but not both, making this an XOR test condition.

WinBatch supplies both the AND (`&&`) and OR (`||`) logical operations, but not an XOR (eXclusive OR) operator, as is supported in some other languages. This does not mean that an XOR test can't be written using WinBatch, just that it requires a complex statement to do so.

In like fashion, other compound test conditions can be constructed, such as a tertiary XOR where one of three conditions would pass but two or more would fail. Essentially, there are no limits to compound tests (although you do want to stop before you become completely confused).

Complex Tests

For more complex tests, you can use nested `if..then..else` statements, in this format:

```
if( condition_1 ) then
    statement_1
    if( condition_2 ) then
        statement_2
    else
        statement_3
    endif
    statement_4
else
    if( condition_3 ) then
        statement_5
    else
        statement_6
```

```

if( condition_4 ) then
    statement_7
else
    statement_8
endIf
endIf
statement_9
endif

```

Trick question: Can statements 1, 4, and 8 be executed under a single set of conditions? Which statements could be executed together? (See table below for the answer.)

The problem with creating complex nested `if..then..else` statements is that it can become difficult to say exactly what will happen under which conditions. The table below shows the `TRUE/FALSE` conditions required for each of the nine statements in the example to be executed. This type of tracing of the conditions is called a *truth table*.

Truth Table for a Complex Nested `if..then..else` Statement

C1	T			F				
	S1			C3	T	F		
	C2	T	F		S5	S6		
		S2	S3			C4	T	F
		S4					S7	S8
			S9					

Since this particular format – however accurate – may be a little difficult to read, the same information is shown below in an expanded format.

Expanded Truth Table

if Condition_1 is TRUE then Statement_1		if Condition 1 is FALSE then		
if Condition_2 is TRUE then Statement_2	if Condition_2 is FALSE then Statement_3	if Condition_3 is TRUE then Statement_5	if Condition_3 is FALSE then Statement_6	
			if Condition_4 is TRUE then Statement_7	if Condition_4 is FALSE then StatInt_8
Statement 4		Statement 9		

Nested If..Else..Endif Statements

Another variation of an `if..else..endif` structure nests the tests many layers deep. The [Select1.wbt](#) demo uses deeply nested `if` statements to make a selection from a list. In [Select1.wbt](#), the decision mechanism appears as:

```

If( CountryKey == "France" ) then
    If( sResponse == "Paris" ) then
        sMessage = "Correct"
    Else
        sMessage = "Wrong"
    EndIf
Else
    If( CountryKey == "Egypt" ) then
        If( sResponse == "Cairo" ) then
            sMessage = "Correct"
        Else
            sMessage = "Wrong"
        EndIf
    Else
        If( CountryKey == "Russia" ) then
            If( sResponse == "Moscow" ) then
                sMessage = "Correct"
            Else
                sMessage = "Wrong"
            EndIf
        Else

```



```

If( CountryKey == "Japan" ) then
    If( sResponse == "Tokyo" ) then
        sMessage = "Correct"
    Else
        sMessage = "Wrong"
    EndIf
Else
    If( CountryKey == "England" ) then
        If( sResponse == "London" ) then
            sMessage = "Correct"
        Else
            sMessage = "Wrong"
        EndIf
    EndIf
EndIf
EndIf
EndIf
EndIf

```

This is a functional method that serves its purpose of demonstrating a deeply nested `if..else..endif` structure. However, there are several other methods such as the `if..elseif..` conditional that are more compact, efficient, and elegant. The [Select2.wbt](#) demo shows this alternative solution.

As nested statements can get quite complex, it's a good idea to always write your open/close together first and then fill in the middle; i.e.,

```

If
    If
    Endif
Endif

```

... then fill in the code in the middle.

As you can see from the examples in this section, setting up the code for complex conditions to ensure that all possible results are tested can be a real pain. However, if you absolutely need controls with this degree of complexity, your only reasonable choice is construct them carefully. To guide you in programming complex conditions, create a truth table to see exactly which conditions and paths should be followed.

Switch/Case Decisions

Making selections within an application is a common requirement. The [Select1.wbt](#) and [Select2.wbt](#) demos both employ complex `if` statements to choose between five separate possibilities. In general, however, we prefer to have some simpler mechanism allowing a choice from two or more options. The `switch/case` mechanism (also called `select/case`) provides such a means of choosing from any list of options that can be enumerated.

The `switch` and `select` statements serve the same function in the same fashion. This choice of terminology is offered for compatibility (and familiarity) with other languages where one or the other term is supported.

A `switch/case` decision tree is a convenient mechanism for multiple-choice circumstances (where there are more than two selections involved). The `switch/case` decision tree takes the form:

```
switch test_value
  case value_1
    action statement set #1
    break
  case value_2
    action statement set #2
    break
  case value_3
    action statement set #3
    continue
  case value_4
    action statement set #4
    break
  case test_value
    default statement set
endswitch
```

The rules for a `switch/case` decision tree are:

- The test value used must have an integer value. Neither floating-point values nor strings are acceptable in a `switch/case` statement, although conditionals are allowed and this permits a "trick", thus:

```
string = 'dummy'
```

```

Switch @TRUE
    Case string == 'dummy'
        Pause('String = ',string)
        break
    Case string == 'hello'
        Pause('String = ',string)
        break
    Case string == 'world'
        Pause('String = ',string)
        break
Endswitch

```

The trick is that the conditional test evaluate to a numerical value (i.e., either `true` or `false` and only one will be `true`).

- The case values must also be integers and, except for the default case, ideally they should not be duplicates.

In most languages which support `switch/case` structures, duplicate case values are not allowed. In contrast, WinBatch does permit duplicates – see [Duplicate Case Statements](#) – but these should be used carefully.

- Each set of response statements may be terminated by either a `break` statement or a `continue` statement (although this isn't necessary, as explained in the "[Fall-Through Execution](#)" section).
 - A `break` statement sends execute to the end of the decision tree, which is the `endSwitch` statement.
 - A `continue` statement halts execution and resumes scanning the decision tree for a matching case statement.
- The `Switch/case` decision tree must end with an `endSwitch`, `endSelect`, `End Switch`, or `End Select` statement. (These four forms, like the `switch` and `select` statements, are interchangeable.)

As an example, the [Select3.wbt](#) program replaces the nested `if..else..endif` structure used in the [Select1.wbt](#) program with a `switch\case` statement.

Since `nItem` is an integer value, the `switch/case` statement makes it possible to select a corresponding response to test against:

```
Switch nItem
```

Introduction to Programming

```
case 1
    sCapital = "Paris"
    break
case 2
    sCapital = "Cairo"
    break
case 3
    sCapital = "Moscow"
    break
case 4
    sCapital = "Tokyo"
    break
case 5
    sCapital = "London"
    break
EndSwitch
```

Here, the `switch nItem` statement begins a search through the tree looking for a `case` statement where the value matches `nItem`. Once a match is found, the statement (or statements) following is executed until either a `break` or `continue` statement is reached.

This example uses only `break` instructions. Once the `case` instructions are completed, the decision tree resumes with the next instruction following the `endSwitch` statement. And, this time, there's only one test performed instead of five:

```
If( sResponse == sCapital )
    sMessage = "Correct"
Else
    sMessage = "Wrong"
EndIf

Message( "Your guess was:", sMessage )
```

You can apply the same type of solution to much more complex situations.

Fall-Through Execution

In a `switch/case` decision tree, it isn't absolutely necessary for each case to be terminated with a `break` or `continue` statement. If neither appears, once a matching `case` has been

found and execution has finished the instructions for that `case`, execution simply continues with the next `case` in the list.

To illustrate fall-through execution, the following table shows a fragment of a `switch/case` decision tree and the value of `t` at successive steps while `nItem` is set to 1 through 5 for the various cases.

If nItem equals	1	2	3	4	5
t = 0	t = 0	t = 0	t = 0	t = 0	t = 0
switch nItem	t = 0	t = 0	t = 0	t = 0	t = 0
case 5	*	*	*	*	t = 0
t = t + 11	*	*	*	*	t = 11
	*	*	*	*	t = 11
case 4	*	*	*	t = 0	t = 11
t = t + 22	*	*	*	t = 22	t = 33
	*	*	*	t = 22	t = 33
case 3	*	*	t = 0	t = 22	t = 33
t = t + 33	*	*	t = 33	t = 55	t = 66
	*	*	t = 33	t = 55	t = 66
case 2	*	t = 0	t = 33	t = 55	t = 66
t = t + 44	*	t = 44	t = 77	t = 99	t = 110
break	*	t = 44	t = 77	t = 99	t = 110
case 1	t = 0	*	*	*	*
t = t + 55	t = 55	*	*	*	*
break	t = 55	*	*	*	*
endSwitch	t = 55	t = 44	t = 77	t = 99	t = 110
Final value of t is	55	44	77	99	110

These are essentially the values you would observe if you were debugging the operation by stepping through the code. (The steps showing asterisks are not executed for those values of `nItem`.) You could use this same type of selection to perform different subsets of operational steps.

As another example, here is a `switch` structure that handles receipt of several different types of data:

```
switch DataType
  case CryptoData
    gosub DecodeData
  case RawData
    gosub ProcessData
  case ProcessedData
    gosub EvaluateData
    break
  case Outgoing
```

Introduction to Programming

```
        gosub EncryptData
        break
endswitch
```

If the `switch` is handling encrypted data, the process begins with the `DecodeData` subprocedure, followed by `ProcessData`, and ending with `EvaluateData`. On the other hand, if the incoming data is already processed, only the `EvaluateData` subroutine is called. For outgoing data, none of the preceding would be relevant, and operations would branch to the `EncryptData` routines.

The assumption here is that `CryptoData`, `RawData`, `ProcessedData`, and `Outgoing` are each integer values which have been given recognizable names. Using mnemonics rather than numbers helps to avoid confusion and prevent possible mistakes.

The same sequences could be accomplished with a different `switch` structure:

```
switch DataType
    case CryptoData
        gosub DecodeData
        gosub ProcessData
        gosub EvaluateData
        break
    case RawData
        gosub ProcessData
        gosub EvaluateData
        break
    case ProcessedData
        gosub EvaluateData
        break
    case Outgoing
        gosub EncryptData
        break
endswitch
```

Neither of these alternatives is "better" than the other; they are simply different options for structuring the code.

The two preceding examples also demonstrate branching operations. The `gosub` instructions send execution off to subroutines, which then return. (A `goto` instruction would simply branch execution to perform different tasks without the return.)

Duplicate Case Statements

As a general rule, `switch/case` structures are the same in any language: They accept an enumerated value, find a matching case, execute the instructions for that case, and then either leave the decision tree on a `break` statement or allow execution to "fall through," as explained in the previous section.

However, the `continue` option supported by WinBatch is a distinct difference, partially because the `switch/case` structures used in most languages do not permit duplicate values.

Consider the following fragment, assuming that `nItem` equals 2 when the `switch` begins:

```
switch nItem
  case 1
    sResult = "Apples"
    continue
  case 2
    sResult = "Oranges"
    continue
  case 3
    sResult = "Pears"
    continue
  case 2
    sResult = "Grapes"
    continue
  case 1
    Result = "Peaches"
    continue
endswitch
```

When the first `case 2` is reached, `sResult` is set to "Oranges", but since there is a `continue` statement, execution proceeds to reach the second `case 2`, where `sResult` is set to "Grapes".

If `break` instructions had been used rather than `continue` instructions, only the first `case 2` statement encountered would have been executed before leaving the `switch/case` decision tree.

Default Cases

The `switch/case` structures in most languages support a default `case` statement, normally as the last `case` statement. The default is executed if none of the other `case` statements match the `switch` selection.

WinBatch also supports a "default" `case`, but it does so in a slightly unusual fashion. In a WinBatch program, the default can appear anywhere in the list of cases and can appear more than once.

While most languages don't permit variables in `case` statements, WinBatch allows creating a default `case`—one that will always be true—by entering the same variable name in both the `switch` statement and one (or more) of the `case` statements.

An example of a default `case` statement appears in the following fragment:

```
switch nItem
    case 2
        gosub ProcessData
        break
    case 7
        gosub EvaluateData
        break
    case 4
        gosub FormatData
        break
    case nItem ; Default case
        gosub ReportData
endswitch
```

Here, if `nItem` has a value of 2, 4, or 7, there is a specific `case` statement. For all other values, the default `case` statement (`case nItem`) takes effect.

Case statements are not required to appear in sequential order.

As a variation, the `break` statements could be replaced with `continue` statements:

```
switch nItem
    case 2
        gosub ProcessData
        continue
```



```

case 7
    gosub EvaluateData
    continue
case 4
    gosub FormatData
    continue
case nItem
    gosub ReportData
endswitch

```

Then the three special cases would each trigger their specific handling, but all cases would trigger the default.

Loops

Like most programming languages, WinBatch supports two formal loop statements:

- The fixed `for` loop executes a set number of times and then stops.
- The `while` loop executes as long as a test condition remains valid or until some other operation causes the loop to exit.

For Loops

The `for` loop takes the form:

```

For( loop_var = start_value to end_value [by step_value] )
    ...statements to execute...
Next

```

When the `for` loop starts, the loop variable (`loop_var`) is set to the initial value (`start_value`) before starting to execute the instructions following the `for` statement. These instructions are performed in order until the `next` statement is reached.

When the `next` statement is reached, execution returns to the `for` statement, where the loop variable `loop_var` is incremented.

The increment (`by step_value`) can be any integer value—positive or negative. The default increment, if `by step_value` is not specified, is always one (1).

If, after incrementing (or decrementing if the step value is negative), the loop variable is greater than the end value, the loop exits, and the program continues with the first instruction following the `next` statement.

Introduction to Programming

Until the limit is passed, each repetition of the loop repeats the same set of instructions. However, this does not mean that the values of the variables or the data used in the repeated instructions have not changed.

When a negative increment is used to create a down-counting `for` loop, you must be careful to ensure that the start value is greater than the end value. If not, then the `for` loop will attempt to become an endless loop.

To demonstrate `for` loop operations, the [Prime.wbt](#) program uses two loops as it executes a search for prime numbers in the range 9 to 1001. (A prime number is defined as any integer that is evenly divisible only by 1 and by itself.)

The first loop in [Prime.wbt](#) sets up the range of integers to be tested. However, since all even numbers can be eliminated (2 is the only even prime), the loop is set to step by 2, leaving only the odd integers to be tested.

```
sList = ""  
For i = 9 to 1001 by 2
```

Using this loop, the test begins with values for `i` of 9, 11, 13, 15, ..., 1001. Inside the loop, a Boolean flag is set to `TRUE`:

```
bPrime = @TRUE  
For j = 3 to Sqrt( i )
```

The inner loop sets up divisors to test against `i` to determine if `i` is a prime; that is, to test if it is not evenly divisible by any of the divisors. However, this inner loop uses a start value of 3, because 1 isn't relevant (all numbers are divisible by 1) and all even numbers (divisible by 2) have already been eliminated.

Also, the largest divisor that there is any point in testing for any given `i` is the square root of `i`. While any integer `i` could have a factor larger than its own square root, it will have a corresponding factor that is smaller. Once the smaller factor is discovered, there is no need to test further.

Finally, the simplest test to perform is whether modulo division returns a remainder of zero (0). If so, then the integer being tested is not a prime number, the flag is set to `@FALSE`, and since there's no need to test further, a `break` statement is used to jump out of the inner `for` loop:

```
If( i mod j == 0 )  
    bPrime = @FALSE  
    break  
Endif
```

```

Next ; j
    if bPrime then sList = sList : i : " "
Next ; i
Message( "Primes found are:", sList )
exit

```

ForEach Loop

The `ForEach` loop is similar to the `For` loop, but it executes the statement block for each element in a COM/OLE collection, instead of a specified number of times.

What Is COM/OLE?

COM/OLE Automation is an industry standard that applications use to expose their COM objects to development tools, macro languages, and container applications that support COM/OLE Automation. For example, a spreadsheet application may expose a worksheet, chart, cell, or range of cells - all as different types of objects. A word processor might expose objects such as applications, paragraphs, sentences, bookmarks, or selections. When an application supports COM/OLE Automation, the objects it exposes can be accessed by WIL. You use WIL scripts to manipulate these objects by invoking methods (subroutines) on the objects, or by getting and setting the objects' properties (values). A full discussion on COM/OLE is beyond the scope of this book.

The `ForEach` loop takes the form:

```

ForEach elementvariable in collection
...
Next

```

For each iteration of the loop, WinBatch sets the variable *elementvariable* to one of the elements in *collection* and executes the statement block between the `ForEach` and `Next`. When all the elements in *collection* have been assigned to *elementvariable*, the `ForEach` loop terminates and control passes to the statement following the `Next` statement.

If *elementvariable* is not used before the loop, it will be created for you. If it is used before the loop, the previously existing value of the variable is lost when the `ForEach` statement executes for the first time.

The elements of *collection* can be of any data type, so the data assigned to *elementvariable* can be of any supported type including object references.

Introduction to Programming

After the loop terminates *elementvariable* contains the last element of the collection. If the elements of the collection are object references, you are responsible for releasing the last references with an assignment statement or the `ObjectClose` function.

To terminate a loop before the last element of the *collection* is assigned to *elementvariable* use a `break` statement.

`Goto` statements are not permitted inside of `ForEach...Next` loops and will cause WinBatch to generate an error message.

It is recommended that you not modify the *elementvariables* in a `ForEach...Next` loop. Any modification you make may affect only the local copy of the element and may not be reflected back into the collection. Attempts to modify an element may also generate an error message.

It is also recommended that you not alter the collection by adding, deleting, replacing, or reordering any elements. If you alter the collection after you have initiated a `ForEach...Next` loop, the *Collection* object becomes invalid, and the next attempt to access an element may result in an error message or other unexpected behavior.

The [ForEach.wbt](#) example shows a trick where you can iterate through single dimension arrays returned by object properties and methods with a `ForEach...Next` loop. However, you should not attempt to change array elements from within the loop.

For Loop Interruption

Although a `for` loop is nominally expected to execute a certain number of times, there are occasions when it may be preferable to interrupt the loop rather than allowing it to continue to completion.

In the [Prime.wbt](#) example, when the inner test loop finds that the integer being examined is not a prime number and there's no need to continue testing, a `break` statement causes the loop to terminate without waiting until the loop variable reaches the limit. This is a simple matter of efficiency, since only one confirmation is required.

There are other ways to manipulate the loop operation. For instance, instead of a conditional `break` statement within the loop, the loop variable could be increased within the loop, and this change would cause the loop to terminate sooner. On the other hand, if the loop variable were decreased within the loop, the loop might well continue, perhaps indefinitely.

Likewise, the end value for the loop can also be increased or decreased while the loop is executing with similar results, because the "current" end value is tested each time the loop cycles. In other words, each iteration of the loop begins with a new test to determine if the limit has been reached. If the limit is a variable that has been changed during prior iterations, the current value is what is used to end the loop.

However, although changing the loop variable or end value while inside a loop serves the purpose of changing the loop operation, this is not a recommended practice, because the

actual results might not be what you expected. Instead, when a more flexible loop is desired, the `while` loop is probably your better choice. With `while` loops, the controlling conditions tend to be more clearly apparent. The [Prime2.wbt](#) program demonstrates using a `while` loop in place of a `for` loop.

While Loops

Like a `for` loop, a `while` loop is used to repeat a series of one or more instructions, with the loop ending when the controlling condition is changed or when some other instruction causes the loop to exit. However, a difference is that a `for` loop is expected to execute a number of times according to preset limits; a `while` loop can be set to repeat indefinitely and to exit only when a desired task is completed. The exit condition and command can occur anywhere within the loop.

A `while` loop takes the form:

```
while( condition_is_true )
    ...execute these instructions...
endWhile
```

For example, the following code fragment depends on the variable `bTest` remaining `@TRUE` (that is, not equal to zero):

```
bTest = @TRUE
while( bTest )
    ..first set of instructions ...
    if( aData > bData ) then bTest = @FALSE
    ... second set of instructions ...
endWhile
```

But, somewhere within the loop, there is a test or an operation that changes the value in `bTest`, setting the variable to `@FALSE` (or zero), which is the trigger to exit the loop.

In this example, the test condition `bTest` was explicitly set to `@TRUE` before the loop was initiated. If `bTest` had not been initialized before the loop started, WinBatch would have halted operation on an error. However, under other circumstances—for example, if `bTest` were initially `@FALSE`—the loop would not execute at all.

Also notice that, in this fragment, the loop-breaking test condition is altered only in the middle of a series of other instructions. However, the loop condition is tested only at the start of each iteration. This means that the second set of instructions will continue to execute after the test condition has changed. (Whether these latter instructions should or shouldn't execute depends on the program routine and other provisions could be made to ensure, if necessary, that they did not.) Although this method works, there are other ways of breaking `while` loops that provide more direct control.

Introduction to Programming

While Loops Interruption

The [SearchReplace.wbt](#) demo (introduced in [Chapter 6](#)) contains a `while` loop that has no possible chance of terminating automatically:

```
while @TRUE
```

Here, since the predefined condition `@TRUE` is not going to change during execution of the loop (nor any other time), the `while` loop could be expected to continue indefinitely.

Within the loop, an indefinite continuation is exactly what's desired. It allows the file-read operation, which is reading one line at a time from a file of indefinite size, to proceed until the end of the file is reached:

```
sLineIn = FileRead( hFileIn );  
If sLineIn == "*EOF*" Then Break ; break out of while loop
```

When the end of the file is found (indicated by the string `"*EOF*"`), a `break` instruction carries the execution outside the loop, forcefully terminating what otherwise would attempt to continue *ad infinitum*.

```
...  
endWhile
```

Once the end of file has been reached, the remaining instructions are not followed. Instead, the `while` loop exits immediately, at the point desired. It does not wait for the current sequence to finish before the loop condition is tested.

Always remember that a `while` loop is potentially an infinite loop. Be sure that you do provide an exit.

Summary

Controlling the flow of an application is a little like setting up a string of dominoes for toppling. If you do everything right, the results are perfect; if not Well, the advantage in programming is that applications are easier to modify and retest than setting up a floor filled with scattered dominoes, and using the controlling mechanisms—`if`, `switch/case` (or `select/case`), `for`, `foreach` and `while`—are usually simpler than balancing rows of dominoes.

We began by showing the some variations of the `if` statement and explaining how computers recognize `@TRUE` and `@FALSE` as conditions. Simple, compound, and complex (nested `if..else..endif`) `if` forms were discussed.

While complex choices can be handled using `if` statements, the `switch/case` structure offers a method of selecting from a list of options or branches. We described the

`switch/case` decision map structure and its uses. Also, we explained how the WinBatch implementation of the `switch/case` structure is different from similar mechanisms used in other languages.

In the final sections, the `for` and `while` loop formats were introduced and demonstrated. We also covered methods of interrupting loops as alternatives to relying on automatic termination.

Next, in [Chapter 9](#), we'll look at the mathematics of computer programs—how to make the computer handle the math. After all, it's only a glorified adding machine, right?

CHAPTER 9 : IT'S ALL IN THE NUMBERS

MATHEMATICAL OPERATIONS

"Mathematics is that subject where we neither know what we are speaking about nor why" – Bertrand Russell

To some degree, Mr. Russell's remark is justifiable, since pure mathematics seems to have little connection with the reality. This actually may be the case initially. However, mathematicians frequently find that their most abstract theorems, their most fantastic excursions into the realm of numbers, and their purest thoughts on the subject of the totally imaginary all eventually come to roost in some form of practical application. For example, the mathematics of soap bubbles – surely a very 'theoretical' field – have found applications in routing theory used for networks (and, of course, the Internet) while a 'toy' of my own youth – ring theory – is integral to the operations carried out by computers.

With the advent of computers as the theorists' newest tool, variously allowing brute-force solutions to some mathematical problems but more often offering graphic (the pun is unavoidable) insights into the nature of numbers, operations that were once possible only under laborious effort can now be accomplished by our electronic savants in matters of seconds or minutes.

We will not venture into the more rarefied realms of number theory in this chapter, but we will look at many mathematical operations that are more immediately useful, especially in the world of business. While doing so, please consider for a moment just how difficult and laborious even these (relatively) simple operations would have been in the days before computers.

In this chapter, we'll look at the mathematical functions supplied by WinBatch, together with brief examples of how each function operates. This chapter also covers date and time functions. In the final section, we'll use these functions in an example of a practical business application.

Simple Numerical Manipulations

Since there is no ready nor rational standard for ranking mathematical operations in order of importance, the only remaining order is to discuss the operations in an alphabetical sequence. For this reason, we'll begin with `Abs`.

The Abs and Fabs Functions

The `Abs` function returns the absolute value of an integer. The `Fabs` function performs the same task for a floating-point value. When speaking of *absolute* value, we are simply ignoring whether a value is positive or negative and treating it as if it were positive.

Introduction to Programming

For example, you could use `Abs` like this:

```
year1 = 1954
year2 = 1999
years = Abs( year1 - year2 )
Message( "Years", "There are ":years:" years between ":year1:" and ":
year2 )
```

There are more complicated ways to perform a calculation of this type, for instance, we could:

1. Subtract the two values.
2. Ask if the result was less than zero.
3. If so, multiply the result by -1 to create a positive integer.

In fact, steps 2 and 3 are essentially what the `Abs` function does for us.

In like fashion, the `Fabs` function returns a positive floating-point value, thus:

```
f = -12.3456
Message( "Fabs( ":f:" ) = ", Fabs( f ) )
```

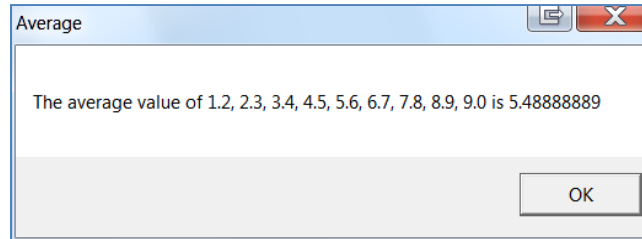
These may seem like rather trivial functions, but keep them in mind. Sooner or later, you'll find you need an easy way to convert to absolute values.

The Average Function

The `Average` function returns the mean average of a series of values. The list of values can be created as a string with the entries delimited by commas, and the individual values may be integers, floating-point numbers, or both.

```
fList = "1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9, 9.0"
fAvg = Average( %fList% )
Message( "Average", "The average value of ":fList:" is ":fAvg )
```

The `Average` function is demonstrated in the [Average.wbt](#) program. For the list of values shown above, it returns a result of 5.48888889:



The Ceiling and Floor Functions

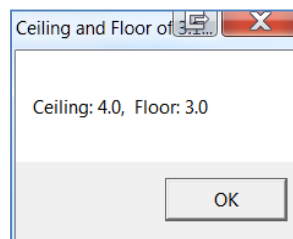
The `Ceiling` and `Floor` functions accept floating-point values and return results that are the floating-point value rounded to the nearest integer. Thus, the `Ceiling` function returns a floating-point value that is the smallest integer larger than or equal to the floating-point value. Similarly, the `Floor` function returns a floating-point value that is the largest integer smaller than or equal to the floating-point value.

Although the `Ceiling` and `Floor` functions return floating-point values, because the decimal portions of these values are truncated, the results are the floating-point equivalents of integers.

For example, the following code, from the [Floor_Ceiling.wbt](#) program accepts a number string and reports the `Ceiling` and `Floor` values for the number.

```
fVal = AskLine( "Ceiling/Floor", "Please enter a decimal number (ie.
1.234)", "3.1415" )
nCeiling = Ceiling( fVal )
nFloor = Floor( fVal )
Message( "Ceiling and Floor of ":fVal, "Ceiling: ":nCeiling:", Floor:
":nFloor )
Exit
```

The demo returns the `Ceiling` and `Floor` results:



Following are some examples of results returned by the `Floor` and `Ceiling` functions:

fVal	nCeiling	nFloor
------	----------	--------

Introduction to Programming

25.2	26.0	25.0
25.7	26.0	25.0
24.9	25.0	24.0
-14.3	-14.0	-15.0
13.0	13.0	13.0

Notice that a value of 13.0 results in identical floor and ceiling values.

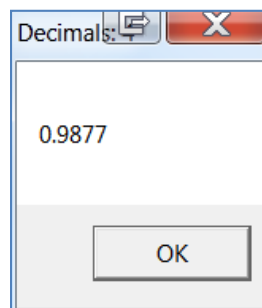
The Decimals Function

The `Decimals` function allows you to specify how many decimal places should be shown when displaying a floating-point value. When `Decimals` is called with a new decimal-point setting, the previous decimal-point setting is returned (but can simply be ignored). Alternatively, if `Decimals` is called with an argument of `-1`, the maximum number of decimal places are shown automatically.

The [Decimals.wbt](#) demo steps through a series of decimal settings using this code:

```
fVal = 0.9876543210
Decimals( -1 )
Message( "Decimals (full)", fVal )
For d = 0 to 10
    Decimals( d )
    Message( "Decimals: " : d, fVal )
Next
Exit
```

The result for a decimal setting of 4 is shown like this:



Notice that the value 0.987654321 has been rounded to 0.9877 for display purposes only. The actual floating-point value in the variable has not changed and will not change.

Also, while you can set any number of decimal places you desire (the [Decimals.wbt](#) demo goes up to ten places), a maximum of eight decimal places will be displayed. Or, if you prefer, any argument above 8 is treated as 8.

The Int Function

The `Int` function provides conversion from a floating-point value or a string to an integer value.

```
sVal = "4.9"
nVal = Int( sVal )
Message( sVal:" becomes", nVal )
```

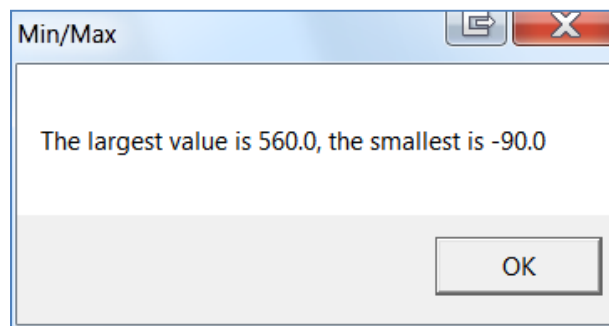
If the argument is a string, the string is first converted to a number value, if possible, and then rounded to the nearest integer. Thus, a string of "4.9" would be reported as an integer value of 5.

The Min and Max Functions

The `Min` and `Max` functions return the smaller or larger of a list of two or more arguments, respectively.

```
fMin = Min( -7.8, 1.2, 560, 0.34, 45, 6.7, 8.9, -2.3, -90 )
fMax = Max( -7.8, 1.2, 560, 0.34, 45, 6.7, 8.9, -2.3, -90 )
Message( "Min/Max", "The largest value is ":fMax:", the smallest is
":fMin )
```

The `Min` and `Max` functions are demonstrated in the [Min_Max.wbt](#) program. For the list of values shown above, it returns the values 560.0 and -90.0:



Number Testing

Since WinBatch allows strings to be treated as numbers—that is, to be easily converted to numbers—three functions are provided to conveniently test whether a variable is a

Introduction to Programming

number, a floating-point value, or an integer. You've seen these functions in examples in the previous chapters.

The `IsNumber` function simply reports whether the argument can be converted to a number, either a `Integer` or a floating-point value. `IsNumber` is called as:

```
if IsNumber( sVal ) then ...
```

The `IsFloat` and `IsInt` functions perform the same as `IsNumber` but report whether a variable can be treated as a floating-point value or as an integer, respectively.

The [TestNumber.wbt](#) program demonstrates a series of tests for each of these functions. The results are reported as:

Variable	IsNumber	IsFloat	IsInt
"This is not a number"	FALSE	FALSE	FALSE
"0.012345"	TRUE	TRUE	TRUE
"12345"	TRUE	TRUE	TRUE

As you can see, the `IsNumber`, `IsFloat`, and `IsInt` functions seem somewhat redundant (since there appears to be no particular differentiation between integers and floating-point values (of course, both are numbers)).

`IsNumber` and `IsFloat` are, in fact, exactly synonymous. `IsInt` can be useful for validating numbers obtained by user input or from other sources. Although WinBatch will convert floating-point arguments to integers for functions that expect integers, there is some rounding involved.

Pseudo-Random Numbers

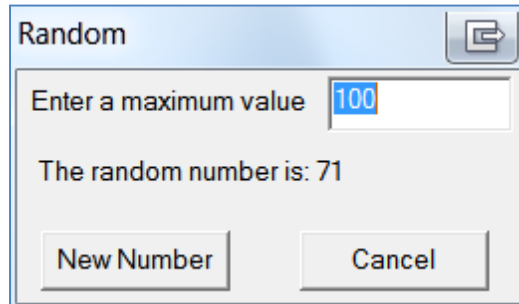
Random numbers have a variety of uses. Perhaps the most obvious applications are in games, but, entertainment aside, random numbers do have more serious purposes. For example, random numbers are often used to test processes, to generate initial and run conditions for simulations, to test financial or business-transaction processing, or simply to vary the order or sequence of events in a variety of applications. Randomization is an essential element in a great many computer processes and applications.

However, while randomization can be very desirable, we do not have any way of generating truly random numbers. Instead, we rely on mathematical processes that return *pseudo-random numbers*. Pseudo-random numbers are not in themselves truly random, but for all practical purposes, they can be considered random.

The WinBatch `Random` function is typical of the pseudo-random generator functions. It accepts a single maximum range parameter and returns an integer that is greater than or equal to zero and less than or equal to the specified maximum. Thus, if we call `Random (`

100), the returned value will be an integer between 0 and 100 inclusive, for a range of 101 values.

The [Random.wbt](#) program accepts a maximum range value and displays a generated pseudo-random result:



Quite often, we want to generate random values in a range other than from zero to a maximum, or we may want to generate random decimals. The `Random` function does not provide these variations directly, but both are easily obtained. For example, to generate a random value between 32 degrees Fahrenheit (the freezing point of water) and 212 degrees Fahrenheit (the boiling point of water), we simply determine that we need a range of 180 degrees to include both the freezing and boiling point temperatures. Then we could write our program code as:

```
nTemp = Random( 180 ) + 32
```

For a returned result of 0, `nTemp` would be 32; for a result of 180, `nTemp` would equal 212. All other results would fall somewhere between the two extremes.

Now suppose that we need a decimal fraction in the range 0.40 and 0.60 inclusive, and we want two decimal places in the value. This is also easily accomplished as:

```
fPercent = ( Random( 20 ) + 40 ) / 100.0
```

Remember that the `Random` function returns values with an even distribution within the specified range. If we need skewed results, we can write more complex distribution formulas as well. For example, suppose that we want to simulate the results of rolling two six-sided dice. For a single die, which should generate results in the range 1 through 6, we would use this formula:

```
nSingleDie = Random( 5 ) + 1
```

For two dice, the formula would be:

Introduction to Programming

```
nTwoDice = Random( 5 ) + Random( 5 ) + 2
```

This would return results in the range 2 through 12 but with a skewed distribution matching real (and presumably honest) dice.

In like fashion, other results and other skewed distributions can be easily created to suit a variety of requirements.

Large and Transcendental Numbers

When we need to deal with larger numbers or manipulations that are a bit more complex than simple arithmetic, we can use WinBatch's more powerful numerical functions. Following our alphabetical arrangement, we'll start with the `Exp` function.

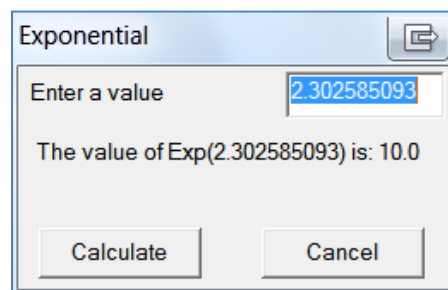
The Exp Function

The `Exp` function takes an exponent as an argument and returns the value of the natural log (e) raised to that value. The value of e is 2.71828183. Thus, e^1 equals 2.71828183, e^2 equals 7.3890561, and $e^{2.302585093}$ equals ~ 10 . Exponential expressions are frequently used to express large numbers in a compact format. (There are other, more important reasons for the use of exponential notation, which should be familiar to mathematicians, physicists and engineers, but these are rather beyond the scope of the present discussion.)

The `Exp` function is called as:

```
fVal = Exp( fExp )
```

The [Exponential.wbt](#) program demonstrates the `Exp` function, converting natural exponents to values, like this:



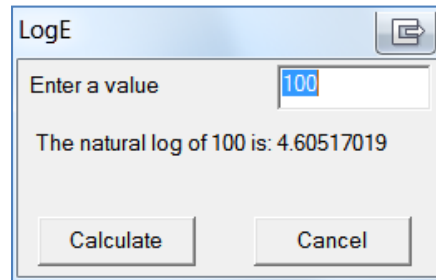
Related functions are `LogE` and `Log10`, discussed next. Also, see the discussion of the exponential operator (`**`) in [Chapter 5](#).

The LogE Function

The `LogE` function is used to calculate the natural logarithm for a value; that is, it calculates the exponential value of e that will produce the original value. The `LogE` function is called as:


```
fExp = LogE( fVal )
```

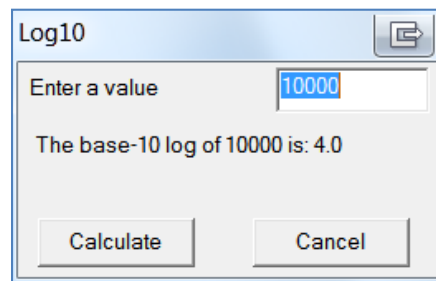
The [LogE.wbt](#) demo demonstrates the `LogE` function. Here, it is calculating the natural log of 100:



Both the `LogE` and `Log10` functions will return an error if they are called with a negative argument.

The Log10 Function

The `Log10` function performs the same task as `LogE`, except that the calculation is made on the base-10 logarithm instead of the natural log. The [Log10.wbt](#) demo demonstrates the `Log10` function. Here, the base-10 log of 10000 appears as 4.0 ($10^{4.0} = 10000$):



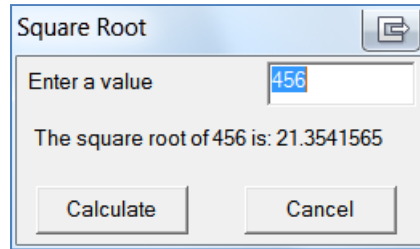
The Sqrt Function

The `Sqrt` function returns the square root of the argument as a floating-point value.

```
nVal = 456
fVal = Sqrt( nVal )
```

The [SquareRoot.wbt](#) demo calculates a square root, like this:

Introduction to Programming

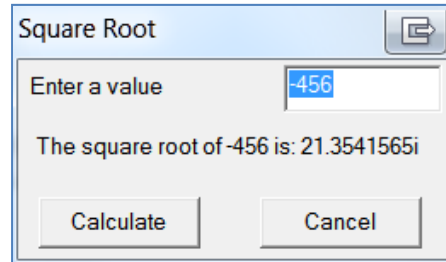


Be aware that calling `Sqrt` with a negative argument will result in an error report: *1599 FP Math: Function returned invalid floating point number (NaN)*

The easiest way to avoid having to handle invalid results like this is to prevent them in the first place. The [SquareRoot.wbt](#) program does this with the following code:

```
fSquareRoot = Sqrt( Fabs( fVal ) )
```

By using the `Fabs` function to ensure that `Sqrt` always receives a positive argument, the chance of a domain error is eliminated. However, since we want to be absolutely accurate (mathematically speaking), for the square root of a negative value, we append the letter *i* to the calculated result. The *i* (which stands for imaginary) is the standard notation for the square root of -1 . For example, the square root of -4 would be $2i$. Or, for -456 , we get the result following:



Trigonometric Operations

If you flunked trig in school, you may not be interested in the array of trigonometric functions provided by WinBatch. On the other hand, with the computer to do the dirty work for you, trigonometry has never been so easy.

Standard engineering practices dictate that angles are expressed in radians rather than degrees. To convert from degrees to radians, the formula is:

```
radians = @PI * degrees / 180
```

Conversely, to convert from radians to degrees, the formula is:

```
degrees = 180 * radians / @PI
```

In WinBatch, the formulas may also be written as:

```
radians = degrees * @Deg2Rad
```

```
degrees = radians * @Rad2Deg
```

...where the predefined constants @Deg2Rad and @Rad2Deg provide the necessary conversion values.

The demo programs for the trigonometric functions each accept angles in degrees, which are converted to radian values for calculation. After calculating angles in radians, the results are converted to degrees.

The Sin, Cos, and Tan Functions

The `Sin`, `Cos`, and `Tan` functions calculate the sine, cosine, and tangent values, respectively, for an angle expressed in radians. All three of these functions will accept arguments larger than 2π radians (more than 360°), but larger values may cause a loss of significance in the result or, in extreme cases, a significance error may occur. For this reason, calculations should always be made using smaller angles—less than 2π radians or 360° —since the sine, cosine, and tangent values remain the same at 270° , 630° , 990° , or 2430° , *ad infinitum*.

A significance error means that the result reported has lost all relationship to the expected calculation.

The functions are called as:

```
fSine = Sin( fRadians )
```

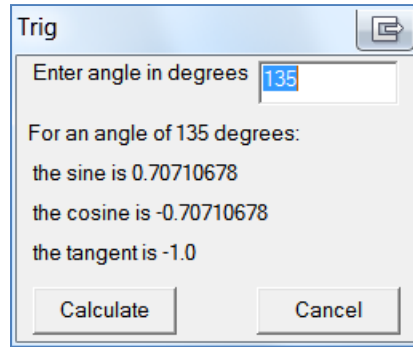
```
fCosine = Cos( fRadians )
```

```
fTangent = Tan( fRadians )
```

where `fRadians` is the angle in radians. As noted, with the `Sin`, `Cos`, and `Tan` functions, angles should be less than 2π radians.

The `Sin`, `Cos` and `Tan` functions are demonstrated in the [Trig.wbt](#) program. The demo calculates the sine, consinee, and tangent for the angle entered:

Introduction to Programming



For convenience and familiarity, the [Trig.wbt](#) program accepts an angle in degrees. Then it checks to ensure that the angle specified is within the range -360° and 360° , correcting the angle if it is outside this range:

```
While( nDegrees < -360 )  
    nD-grees = nDegrees + 360  
EndWhile  
While( nDegrees > 360 )  
    nDegrees = nDegrees - 360  
EndWhile
```

Second, the angle is converted to radians for the actual calculations, since each of these functions require arguments in radians:

```
fRadians = nDegrees * @DEG2RAD
```

Finally, after these checks and conversions, the actual calculations are made:

```
fSine = Sin( fRadians )  
fCosine = Cos( fRadians )  
fTangent = Tan( fRadians )
```

And here you have it—painless trigonometry!

The ASin, ACos, and ATan Functions

The names of the arcsine, arccosine, and arctangent operations come from the Latin phrase *arcus cuius sinus x est*. This literally means "the arc whose sine is x," which became abbreviated as arcsine. As you may gather from this derivation, the "arc" functions provide the inverse operations from the sine, cosine, and tangent conversions.

The ASin Function

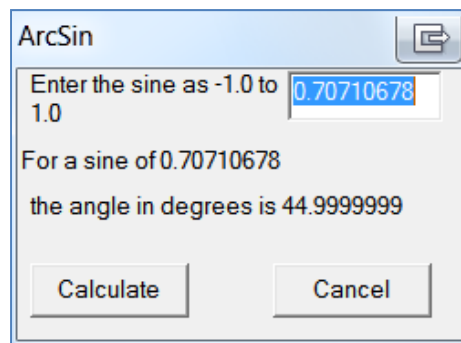
The `ASin` function accepts a sine value as an argument and returns the corresponding angle in radians. `ASin` is called as:

```
fRadians = ASin( fSine )
```

Because the sine values must fall in the range -1 to 1 , the [ArcSin.wbt](#) demo incorporates a simple test before calculating the angle:

```
if( Fabs( fSine ) > 1.0 ) then
    sReport = "value out of range"
```

In this fashion, the reported angle is always in the range $-\pi/2.. \pi/2$ radians (or -90° to 90°).

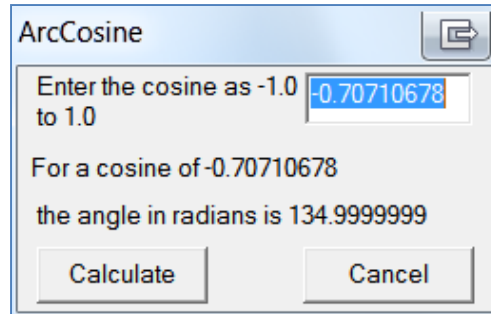


The ACos Function

The `ACos` function accepts a cosine value as an argument and returns the corresponding angle in radians. `ACos` is called as:

```
fRadians = ACos( fCosine )
```

Because the cosine values must fall in the same range as sine values (-1 to 1), the [ArcCosine.wbt](#) demo incorporates the same test as the [ArcSin.wbt](#) program. For the `ACos` function, however, the reported angle will always be in the range $0.. \pi$ radians (or 0° to 180°).



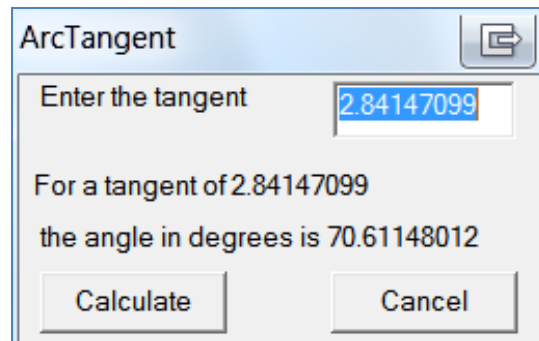
The ATan Function

The `ATan` function accepts a tangent value as an argument and returns the corresponding angle in radians. `ATan` is called as:

```
fRadians = ATan( fTangent )
```

The tangent of an angle of 0° or 180° is zero, but the `Tan` function will show an error at this value. Likewise, for angles of 90° and 270° , the tangent becomes infinite ($\pm\infty$).

The returned angle will be in the range $-\pi/2$ to $\pi/2$ radians (or -90° to 90°). However, since a value of 0 for `fTangent` will produce a domain error, the [ArcTangent.wbt](#) program includes a test requiring the `fTangent` argument to be nonzero.



Caveat: Trig Function Reconversion Discrepancies

In playing with the [Trig.wbt](#), [ArcSin.wbt](#), [ArcCosine.wbt](#), and [ArcTangent.wbt](#) programs, you may notice that the conversions to sine, cosine, and tangent values and the reconversions to degrees do not always agree completely. In some cases, the reconversion shows small errors in decimal values. In other cases, as with the tangent and arctangent calculations, an angle of 135° becomes -45°. This is perfectly normal.

In the case of decimal differences, these are simply the effects of working at the limits of accuracy of the system. For example, the difference between 45° and 44.9999999° is only 0.0000001°, a very small error. Granted, there are circumstances and applications where even this error is not acceptable, but when accuracy is that critical, a different language and different set of processes would normally be chosen.

As for angles of 135° becoming -45°, this is not an error since the trig functions do not differentiate a full 360° range. Instead, the angles of 90° and 180° each have the same sine value, 1.0000, and on reconversion, there is no way to differentiate between the angles. The same holds true for angles of 89° and 91°, where the sine is 0.99985, or for 269° and 271°, where the sine is -0.99985. In short, the arc functions always return angles in a limited range rather than attempting to differentiate between multiple angles with identical sine, cosine, or tangent values.

The Hyperbolic Functions: SinH, CosH and TanH

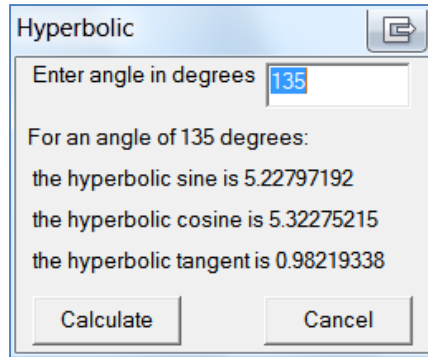
If you do not have an engineering or scientific background, a full explanation of the purpose of the hyperbolic trigonometry functions would be too lengthy and complex for this venue; if you do have such a background, the explanation would be unnecessary. Therefore, it should be sufficient to say that the `SinH`, `CosH` and `TanH` functions provide the *hyperbolic* sine, cosine, and tangent values for a given angle.

Like the `Sin`, `Cos`, and `Tan` functions, the `SinH`, `CosH`, and `TanH` functions accept angles expressed in radians and, for values greater than $\pm 2\pi$ radians ($\pm 360^\circ$) may return an error. With these cautions stated, the three hyperbolic functions perform in essentially the same manner as their counterparts discussed earlier.

These functions are called as:

```
fSineH = SinH( fRadians )
fCosineH = CosH( fRadians )
fTangentH = TanH( fRadians )
```

The three hyperbolic functions are demonstrated in the [HyperTrig.wbt](#) program. Enter an angle in degrees to put the functions to work:



Date and Time Operations

Although date and time functions may not sound like higher mathematics, the truth is that these actually involve rather complex operations. For example, minutes and seconds use base-60 notation, hours use base-12 and base-24, and months vary—some have 30 days, some have 31, and February may have 28 or 29 (which means a year may be 365 days or 366 in length).

The relevance and usefulness of date/time operations are unquestionable. Many applications, especially in business, can hardly function without access to clock or calendar information.

We could simply allow the user to supply date/time information as required. But since the computer has that information already, why not use it?

Date/Time Format

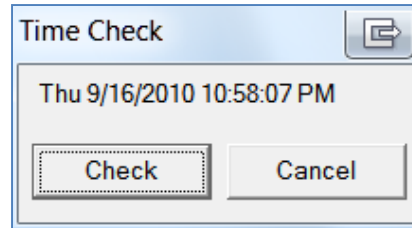
WinBatch uses a date/time format string, which is effectively a colon-delimited list, to represent date/time values in an easily manipulated format. The format for the string is "yyyy:mm:dd:hh:mm:ss". For example, noon on January 1, 2011, is represented as "2011:01:01:12:00:00". January 2, 2012, at 2:15 appears as "2012:01:02:02:15:00".

The TimeDate Function

In WinBatch, date/time operations start with the `TimeDate` function, which retrieves the current date and time from the system, supplying the information in a human-recognizable format. The `TimeDate` function is called as:

```
sDate = TimeDate()
```

The [TimeCheck.wbt](#) demo provides a simple example of checking the time and date:



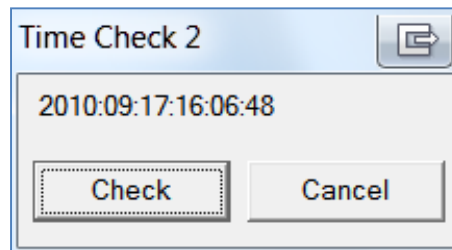
The `TimeDate` function relies on the local date/time format specification which is found in the registry.

An alternative, where we apply our own format to the display, is discussed later in the chapter, in the "Formatting a Date" section.

The TimeYmdHms Function

What is convenient for humans isn't always convenient for the computer (or, more accurately, is rarely convenient for the computer). Therefore, to retrieve date/time information in a form that can be used in calculations, the `TimeYmdHms` function returns a string formatted as "yyyy:mm:dd:hh:mm:ss".

The `TimeYmdHms` function is demonstrated in the [TimeCheck2.wbt](#) program, which checks the time and date in WinBatch format:



The TimeJulianDay Function and the Day of the Week

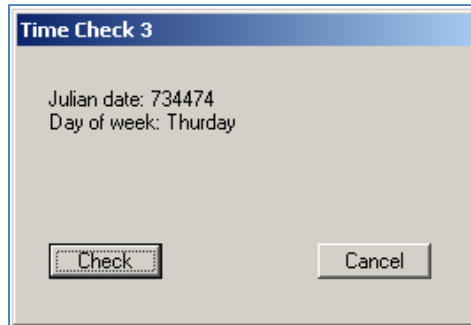
The `TimeJulianDay` function is called with the current date/time as a string in the `yyyy:mm:dd:hh:mm:ss` format (the format returned by the `TimeYmdHms` function) and reports the corresponding (modified business) Julian date as an integer value. (A Julian date is a number that represents the day and year.)

The `TimeJulianDay` function can be called as:

```
nJulianDate = TimeJulianDay( TimeYmdHms() )
```

The [TimeCheck3.wbt](#) program demonstrates getting the Julian date and day of the week:

Introduction to Programming



Because Julian dates are simple integer values rather than a combination of year, month, and day information, calculating elapsed periods between dates becomes a matter of subtracting one Julian date from another. WinBatch does not provide a special function for this purpose, but given a Julian date, the day-of-week calculation is relatively simple:

```
nJulianDay = TimeJulianDay( TimeYmdHms() )  
nDayCount = ( ( nJulianDay + 5 ) mod 7 )
```

Since the Julian date is dated from January 1, Year 0 (A.D.), which was a Friday, the calculation for the weekday requires an offset of 5, as shown. With this addition, modulo division by 7 returns the weekday, where 0 = Sunday, 1 = Monday, and so on.

```
sWeekDay = ItemExtract( nDayCount + 1, "Sunday Monday Tuesday Wednesday  
Thursday Friday Saturday", " " )
```

In order to extract a string with the name of the weekday, we add 1 to the numerical weekday (we can't extract the zeroth item from the list) before extracting the appropriate weekday.

The `TimeJulianDay` function uses the local system clock/calendar and does not take into account the local time zone offset when calculating the current business Julian date.

How the Julian Calendar Works

The Julian calendar was originally created by Joseph Justus Scaliger (1540–1609), an Italian-French philologist and historian who grew tired of reconciling dates in historical documents. These dates were commonly based on the reigns of different rules. Thus, one document might record an event as the twenty-seventh year of the rule of King Gregor, while another, from a different period or locality, would record dates since the birth of Queen Hypolita. In addition to the confusion in years, different calendars were also used in different areas and different eras.

To simplify cross-references, Scaliger created the Julian calendar (naming the calendar in honor of his father, Julius), where all dates were given as day dates since January 1 in the year 4713 B.C. This occasion was selected as the starting point because it was a time safely prior to any recorded history (thus avoiding negative dates). Also, it was the coincident point of the solar cycle (28 years), the Metonic cycle (19 solar years or 235 lunar months), and the Roman indiction of 15 years (decreed by the Emperor Constantine).

For a more convenient reference, the Julian day 2450000 began at noon on October 9, 1995, Universal Time (also called Zulu or Greenwich Mean Time).

Because Julian dates are so large, astronomers and others commonly rely on a modified Julian date (MJD), where the modified date equals the Julian date minus 2400000.5. The half-day in the conversion normalizes the MJD to start at midnight, as do the days recorded by the rest of the world's calendars. Thus, January 1, 1999, can be reckoned as Julian date 2341178.5 (midnight) or, using the modified Julian date, as 51215.

A second type of Julian calendar date, which we call the modified business Julian date, is also in common use. In this system, the date is reported as days since January 1, 0000. The WinBatch `TimeJulianDay` function relies on the modified business Julian calendar, which makes the Julian date for January 1, 1999, equal to 730121.

Now, isn't standardization wonderful? And exactly which Julian date do you mean: true, modified, or business?

The TimeJulToYmd Function

The `TimeJulToYmd` function provides a conversion from the (modified business) Julian date to the WinBatch "yyyy:mm:dd" date/time format. (The "hh:mm:ss" portion of the date/time format will always be "00:00:00".)

The `TimeJulToYmd` function is called as:

```
nJulian = 730131
dateYmd = TimeJulToYmd( nJulian )
```

Time-Difference Calculations

The `TimeAdd` and `TimeSubtract` functions add or subtract one date/time value from another. Both date/time values are expressed in the "yyyy:mm:dd:hh:mm:ss" format. Here is an example:

```
time_now = TimeYmdHms()
```

Introduction to Programming

```
time_ofs = "0000:00:00:01:30:00"    ; an hour and a half
time_later = TimeAdd( time_now, time_ofs )
time_earlier = TimeSubtract( time_now, time_ofs )
```

In other circumstances, it may be useful to determine the difference between two date/time values in days or in seconds. The `TimeDiffDays` function returns an integer value reporting the number of days between two date/time values. The `TimeDiffSecs` function accepts two date/time values and returns an integer value reporting the difference between the two in seconds.

Pause and Wait Functions

In addition to reporting and manipulating date/time values, WinBatch provides two functions that allow an application to pause and wait for a specified interval or until a specified time. Suspending operation for a specific interval or until a specific time can be useful for scheduling other applications and other tasks.

The `TimeDelay` function suspends execution of a program for a specified interval (1 to 3600 seconds or up to one hour). The `TimeDelay` function is called as:

```
nSeconds = Random( 1000 )
TimeDelay( nSeconds )
```

WinBatch allows `nSeconds` to be specified as a floating-point value (i.e., 2.5 or 3.7, etc). Negative values for `nSeconds` are also acceptable but are treated as simple positive integers.

For longer intervals, the `TimeWait` function suspends execution until a specified date/time has been reached. For example, assume that we want to suspend execution for an hour and a half—perhaps before launching some utility, or ringing an alarm, or popping up a reminder to check the turkey in the oven. To accomplish this hypothetical task, we can start by getting the current time as:

```
time_now = TimeYmdHms()
time_ofs = "0000:00:00:01:30:00"    ; an hour and a half
```

Then `time_ofs` is set to the delay interval, and the `TimeAdd` function is used to create a date/time an hour and a half later:

```
time_later = TimeAdd( time_now, time_ofs )
TimeWait( time_later )
```

Finally, the `TimeWait` function is called to suspend operation until the desired time is reached.

Mathematics in the Real World

Now that we've looked at the variety of mathematical and date/time functions available, it seems appropriate to show how these functions can be used in a practical application. The problem is choosing an application that is generally relevant, since it isn't practical to assume that the majority (or even a generous percentage) of the readers of this book are engineers or mathematically inclined. More specifically, we need an application that will be of interest to the majority of the audience.

On reflection, it seems reasonable to assume that virtually everyone has some experience with credit purchases, whether buying a vehicle, acquiring a house, or simply arranging a bank loan. Therefore, to provide a demonstration with a commonality of interest, the selected demo application is a simple mortgage calculator, named [Mortgage.wbt](#).

This demo is simple only in the sense that the calculations will not include sliding-interest scales, balloon payments, or fancy credit options. What will not be simple in this demo is that the code will include provisions for formatting dollar amounts and date information, as well as for checks for a variety of input formats.

The demo will accept three pieces of information: the loan principal, the interest rate, and the length (term) of the loan.

Loan Amount (\$)	% Interest (APR)	Term (months)
\$100,000.00	6.50%	360
1st Payment Date	Payment Amount	Total Interest
01 / 01 / 2011	\$632.07	\$127,544.49

Calculate Exit

Accepting Input Variants

A common mistake for novice programmers is to assume that input will follow some standard format. For example, assume that the loan amount you wish to enter is \$100,000.00. How should the entry be made in the Loan Amount field—as \$100,000.00, \$100000.00, 100,000, 100000, or some other variation?

In like fashion, should a 6½ percent interest rate be entered as 6.5%, 0.065, .065, or 6.5? Since the ½ character isn't conveniently supported from the keyboard, at least that format can be ignored.

Introduction to Programming

In contrast, most stock market applications do accept figures like $\frac{1}{2}$ and $\frac{3}{4}$ for trade orders and some reject decimal entries. Ergo, this format should not automatically be ruled out in all cases.

One option, which is too frequently selected, is to simply require a specific format and, whenever the proper format isn't used, to pop up a warning message explaining what is acceptable. The [Mortgage.wbt](#) program does display a message when an input field simply cannot be parsed, but this has been chosen as a last resort, not as a first option.

One way around the question of variants is to accept anything, reduce the input to the simplest possible format—a raw number—and then, for display, reformat the information in whatever style seems most intelligible.

In the [Mortgage.wbt](#) program, the first step is to treat the input for the dollar amount as a string. The `sPrincipal` variable is the variable associated with the edit box for the loan amount, but once the Calculate button has been selected, the string is copied to a second variable, `fPrincipal`, for further operations:

```
fPrincipal = sPrincipal
fPercentAPR = sPercent
nTerm = sTerm
sDate = TimeAdd( TimeYmdHms(), "0000:00:30:00:00:00" )
sStartDate = StrCat( ItemExtract( 2, sDate, ":" ), " / ", ItemExtract(
3, sDate, ":" ), " / ", ItemExtract( 1, sDate, ":" ) )
fPrincipal = StrTrim( fPrincipal ) ; trim all leading or trailing
blanks

If fPrincipal == ""
    Display( 10, "Entry error", "A loan amount is required" )
    continue ; loop_on_error
EndIf
```

Before doing anything else, the first check is to determine if there is an entry and, if not, to post an error message before looping back to displaying the dialog.

However, assuming that we have an entry, the next step is to decide if the entry can be treated directly as a number value:

```
If ! IsNumber( fPrincipal )
```

If the entry isn't one that can be treated directly as a number value, we need to look for a few common items that would cause a conflict:

```
fPrincipal = StrReplace( fPrincipal, "$", "" ) ; remove any $
```

```
fPrincipal = StrReplace( fPrincipal, ",", "" ) ; remove any commas
fPrincipal = StrTrim( fPrincipal ) ; trim a second time
```

By looking for and removing any dollar sign characters (\$) and commas, we have a pretty good chance of eliminating the most common elements that would prevent the string from being converted to a numerical value. And, just as a precaution, we'll call `StrTrim` again to remove any leading or trailing spaces that the previous operations might have uncovered.

The next step is to repeat the test to see if `fPrincipal` is now acceptable as a number. If not, we report the error and let the user try again:

```
If ! IsNumber( fPrincipal ) ; test a second time
    sPrincipal = ""
    Display( 10, "Entry error", "Principal amount entry is invalid" )
    Continue ; loop_on_error
EndIf
```

At this point, we've tried all of the easy format conversions, and it makes more sense for the user to reenter the information than to keep trying to discover what special form of entry was used.

When the information is acceptable and `fPrincipal` can be treated as number, the information is reformatted for display purposes before proceeding:

In [Mortgate.wbt](#) the `bExternal` variable can be modified to specify which type of operation to execute.

- If `bExternal` is set to `@TRUE` an external subroutine ([FormatCurrency.wbt](#)) will be called.
- If `bExternal` is set to `@FALSE` an internal subroutine will be called.

```
If bExternal ; use this code to call the external subroutine
    sPrincipal = fPrincipal
    Call( "FormatCurrency.wbt", "sPrincipal" )
Else ; use this code to call the internal subroutine
    sTempStr = fPrincipal
    GoSub Format_Dollar_string
    sPrincipal = sTempStr
EndIf
```

Introduction to Programming

The formatting provision of [FormatCurrency.wbt](#) and `Format_Dollar_String` will be discussed in a moment.

The loan rate information also requires testing similar to the tests done for the loan amount. Again, the first step is to check to see if there is an entry and display an error message if necessary:

```
fPercentAPR = StrTrim( fPercentAPR ) ; trim all leading or trailing
blanks
If fPercentAPR == ""
    Display( 10, "Entry error", "A loan rate is required" )
    Continue ; loop_on_error
EndIf
If ! IsNumber( fPercentAPR )
```

And, again, assuming that we have an entry, the next test is to determine if it can be treated as a number.

In this case, if the format isn't numeric, we can start by looking for a percent sign:

```
fPercentAPR = StrReplace( fPercentAPR, "%%", "" ) ; remove %% sign
fPercentAPR = StrTrim( fPercentAPR ) ; repeat trim
```

Notice that the `StrReplace` operation uses `"%%"` rather than `"%"`. This is necessary because in WinBatch, a single percent sign has a special meaning as the reference operator (as you've seen in many previous examples). As a single character, a percent sign in this context will cause an error.

After deleting any percent sign characters and a second trim for leading or trailing spaces, there's one more provision that needs attention.

One common way to enter an interest rate is in a decimal format, such as .18 for 18%. This particular format, however, presents a special problem, because WinBatch doesn't recognize a decimal number without a leading zero—0.18 is acceptable, but .18 is not. Therefore, as a precaution, a check is made for a leading decimal and, if one is found, a leading zero is added. After checking for a leading decimal and correcting the format, a second check is performed to decide if `fPercentAPR` can now be treated as a number. If not, the only remaining recourse is to tell the user to reenter the information:

```
If StrScan( fPercentAPR, ".", 1, @FWDSCAN ) == 1 ; check leading
decimal
    fPercentAPR = StrCat( "0", fPercentAPR ) ; add a leading zero
EndIf
```



```

If ! IsNumber( fPercentAPR ) ; check to ensure this is a number
    sPercent = ""
    Display( 10, "Entry error", "Percentage entry is invalid" )
    Continue ; loop_on_error
EndIf

```

If all goes well, at this point, the entry can be treated as a number. However, we still need to decide whether the entry was made as a percentage or a decimal; for example, was it in the form 18% or 0.18. The easy way is to assume that the percentage should fall in some reasonable range. In this case, a percentage less than 1% is unlikely, and a percentage greater than 99% is equally unreasonable. Therefore, if the raw value we have is greater than (or equal to) 1.0, we'll assume that the entry was made as a percentage value and convert this to the decimal format that we'll actually use in the calculations. Having made this decision, we'll reconvert the value to the format *nn%* for display purposes:

```

If fPercentAPR >= 1.0 ; must be percentage, not decimal
    fPercentAPR = fPercentAPR / 100.0
EndIf
sPercent = StrCat( ( fPercentAPR * 100.0 ), "%" ) ; reformat for
display

```

For the term of the loan, a similar set of tests is performed. This last case is the simplest of all, since we expect a period in months expressed as a simple integer value.

And, once we have the principal, interest rate, and the term of the loan, the next step is to do the calculations.

Doing the Math

The formula for calculating payments on a loan is actually fairly simple:

$$fPayment = \frac{fPrincipal * fRate}{1 - \left(\frac{1}{(1 + fRate)^{nTerm}} \right)}$$

In the formula, *fPrincipal* is the principal amount of the loan, *fRate* is the percentage rate per payment period (not annual or APR), and *nTerm* is the number of payments. The product of the formula, *fPayment*, is the payment amount required to repay the loan.

The process of calculating the loan payments is almost as simple as the formula and requires a mere three lines of code.

Introduction to Programming

The first step is to convert the annual percentage rate (APR) to the monthly interest rate needed in the formula:

```
fRate = fPercentAPR / 12.0 ; convert APR to monthly interest rate
```

The second step, even though the entire formula could be written in a single instruction, is to break the formula down, taking the term under the bar as the first factor to calculate:

```
fFactor = 1 - ( 1 / ( ( 1 + fRate ) ** nTerm ) )
```

Notice that this formula uses parentheses to group factors and to control the order of operations. Instead of using the parentheses, we could break this down into a half-dozen separate operations, but that would be adding potential complications and opportunities for error.

Breaking a complex formula down into parts is a recommended practice. By simplifying the instructions for the computer, the instructions become easier for the programmer to read and, therefore, less likely to contain errors. Of course, breaking a formula into too many parts could produce a new set of errors. Judgment and care are recommended.

We calculate the divisor term as `fFactor` and then use this variable in the payment calculation:

```
fPayment = ( fPrincipal * fRate ) / fFactor
```

The result, `fPayment`, is the monthly payment on the loan at the stated interest rate for the period specified.

One remaining task is to calculate the total interest that will be paid on the loan. The calculation for this is simple:

```
fInterest = ( fPayment * nTerm ) - fPrincipal
```

That's it. We just multiply the payment amount by the number of payments and subtract the principal. What remains is the cost of the loan in interest.

Here is an example of the results for a loan of \$100,000.00 at 6.5% for 360 months (30 years).

Loan Amount (\$)	% Interest (APR)	Term (months)
\$100,000.00	6.50%	360
1st Payment Date	Payment Amount	Total Interest
01 / 01 / 2011	\$632.07	\$127,544.49

Calculate Exit

So, to borrow a hundred grand for thirty years at 6½%, the cost is a relatively cheap twenty seven thousand, five hundred forty four and change ... not bad if you can get it.

Simply performing the calculations, however, is not the end of task. As you can see in the example, the date and amounts are all neatly formatted, making them easy to read. This didn't occur by accident.

Formatting Values

After reading the initial loan amount and calculating the payment and total interest costs, we use a internal subroutine named `Format_Dollar_String` to put this information into a standard currency format.

The `bExternal` must be set to `@FALSE` to call the internal subroutine

Since the only way we can create a truly independent subroutine in WinBatch is to use a call to an external batch file (as explained in [Chapter 7](#)), the `Format_Dollar_String` subroutine is actually a label reached by a `gosub` statement. But, since we want to call this subroutine to format several different values, before the `gosub` statement, we copy the value to be formatted to a variable used by the subroutine:

```
sTempStr = fPayment
GoSub Format_Dollar_string
sPaymentAmt = sTempStr
```

Then, after the `gosub` returns, we copy `sTempStr`, which is now a formatted string, to the display variable `sPaymentAmt`.

The actual process of formatting the currency string is considerably more complex than calculating the amount of the monthly payments.

Formatting Currency

The fact that formatting a string is harder than calculating loan payments should not be any great surprise. Pure numbers are easy. It's when we get into the human factors that the complications appear.

To start, even though the value in the `sTempStr` variable is presumably a floating-point value, we can still treat this as a string and, as such, we begin by looking for a decimal. If we find a decimal, the position of the decimal is all that we need. If not, we settle for the length of the string plus one for the position where the decimal should have been, and we add a decimal and two trailing zeros:

```
:Format_Dollar_String
    sTarget = ""
    nDecimal = StrIndex( sTempStr, ".", 1, @FWDSCAN )
    If( nDecimal )
        nLen = nDecimal
    Else
        nLen = StrLen( sTempStr ) + 1
        sTempStr = StrCat( sTempStr, ".00" ) ; add decimal places
    EndIf
```

Now, armed with a position, the next routine is to insert a comma three places before the decimal point (the standard format for making currency figures more readable). Of course, we also have to allow for the fact that this could be a figure with less than four digits; in which case, we don't need a comma inserted:

```
    If ( nLen > 4 ) ; have at least four digits
        sSubStr = StrSub( sTempStr, nLen - 3, -1 )
        sTempStr = StrSub( sTempStr, 1, nLen - 4 )
        sTempStr = StrCat( sTempStr, ",", sSubStr )
    Endif
```

Since we don't have a function for inserting a substring, we accomplish this task by copying the right portion of the string into one variable (`sSubStr`) and copying the left portion back to our original variable before using `StrCat` to recombine these with a comma inserted between them.

We also need to check for larger values—amounts of a million dollars or more—where the commas become even more important for clarity. This time, we don't need to worry about the decimal point because the string should already have one comma in place. So, instead, we look for the existing comma and then decide if the string is long enough—if the comma is far enough in the string—to warrant inserting another.

```

nLen = StrIndex( sTempStr, ",", 1, @FWDSCAN ) ; now see if more commas
are needed

While( nLen > 4 )
    sSubStr = StrSub( sTempStr, nLen - 3, -1 )
    sTempStr = StrSub( sTempStr, 1, nLen - 4 )
    sTempStr = StrCat( sTempStr, ",", sSubStr )
    nLen = StrIndex( sTempStr, ",", 1, @FWDSCAN )
EndWhile

```

If there are no commas already in place or if no more are required, the `while` loop will simply terminate without doing anything. If we find occasion for inserting a comma, the last action is to look for its position in the modified string and to continue to repeat the process as long as needed. After all, in today's world, some people might be trying to borrow multiple millions (or maybe just thinking about it).

Once we have the commas in place, the last step is to add a dollar sign at the start to produce a fully formatted currency string:

```

sTempStr = StrCat( "$", sTempStr ) ; add the leading dollar sign

Return

```

Formatting a currency string is only one special provision demonstrated in the [Mortgage.wbt](#) program. It also handles the formatting of the date display.

Formatting a Date

Along with the loan amount, interest, and term, the [Mortgage.wbt](#) program displays the date that the first payment will be due, assuming that it is one month from the date that the mortgage calculation was made.

The code for the date display begins by using the `TimeYmdHms` function to retrieve the date and the `TimeAdd` function to add one month to the date:

```

sDate = TimeAdd( TimeYmdHms(), "0000:01:00:00:00:00" )

```

The `TimeAdd` function is more sophisticated than you might expect. Suppose, instead of adding a month, we decided to add 30 days:

```

sDate = TimeAdd( TimeYmdHms(), "0000:00:30:00:00:00" )

```

If we started with a date of 2/7/2011, this format would give us a result of 3/9/2011, since February has only 28 days.

Introduction to Programming

Now we still need to format the date we calculated. WinBatch's `TimeDate` function returns a formatted date, but it has two shortcomings in this case:

- The available display format, which is set by the system preferences, isn't exactly what is desired here.
- This function only formats the current date and time, not a date one month or thirty days later.

Therefore, instead of using `TimeDate`, we use the date/time string `sDate` as a source to extract the day, month, and year and create our own format. Since `sDate` already has the information in a colon-delimited format, we can use the `ItemExtract` function to retrieve the month, day, and year, in that order, and use the `StrCat` function to produce a formatted date for display:

```
sStartDate = StrCat( ItemExtract( 2, sDate, ":" ), " / ", ItemExtract(
3, sDate, ":" ), " / ", ItemExtract( 1, sDate, ":" ) )
```

The result is a *month/day/year* formatted string. This is much more in keeping with standard (American) business usage than the default *year/month/date/hour/minute/second* format, as well as more useful for our purposes.

Summary

In this chapter, we introduced the math functions and briefly explained how they are used. Why the math functions are useful, however, is left more to your special interests and requirements, as appropriate.

The date and time functions, on the other hand, probably have wider appeal for the average user since date information seems to run rampant through our lives. Ergo, you've seen how to retrieve, manipulate and format date (and, by extension, time) information.

As a practical example, the [Mortgage.wbt](#) program has shown not only how to calculate mortgage rates but, probably more important, how to accept input in a variety of formats and how to format information for a useful presentation.

Next, in [Chapter 10](#), we'll look at file operations, including opening, reading, and writing files, as well as directory operations, file formats, and archiving and de-archiving files.

CHAPTER 10 : SHOE BOXES AND FILE CABINETS

DATA STORAGE AND FILE OPERATIONS

file – a named collection of data stored on disk, appearing to the user as a single entity. – The PC User's Pocket Dictionary

directory – in a hierarchical file system, a convenient way of organizing and grouping files and other directories on a disk. – The PC User's Pocket Dictionary

Our two main topics in this chapter are hard drive management and file management. Strictly speaking, "hard drive" management is a misnomer. What we're really talking about are drive operations that apply to CD-ROMs, DVDs, thumb drives, memory cards and other high-capacity removable drives, as well as any future devices that currently exist only on drawing boards and in the imagination. What these devices are isn't material, since we can feel assured that they will follow existing standards of organization and access. The operating system, device drivers, and programming language will handle the differences in hardware, leaving us to be concerned only with the logical organization of the information that the drives hold: the directories and files containing our data.

The importance of file management might be judged by the fact that WinBatch supplies 73 file functions plus an additional 18 directory and 7 disk drive functions, for a total of 98 operations supporting file and drive access. We're not going to cover all of these functions individually, but we will introduce the more important operations and, of course, show how they are used.

While we certainly trust that you're already aware of the nature of a file and of a directory, we hope you will forgive a brief refresher before we look into manipulating files and directories.

File and Directory Concepts

A computer file can contain various items, such as a program, a part of a program, a body of data, a graphic image, or a document created by the user. The actual file, as it is stored on the drive, may be *fragmented*, which means that it may be physically stored in fragments in many different locations on the drive. The operating system manages the task of locating all fragments on demand when the file is read and the task of finding space to write the material when a file is stored.

DOS stands for Disk Operating System, referring to its capabilities for managing disk (floppy or hard drive) file operations. Many versions of DOS have been used over the years, including HDOS, PC DOS, MS DOS, and others and, while the acronym DOS has largely disappeared, these data management tasks continue – in new forms – to be supported by Windows, Linux, Apple OS, etc.

The concepts of file directories and hierarchical file systems are relative latecomers to the PC world. The first disk operating systems supported files but had no concept of directories. At that time, disks were axiomatically "floppy" disks, and the usual capacity was 100 kilobytes (in actual use, more like 90 kilobytes). At that time, there was little need for directories on PCs.

As drive capacities grew, and particularly with the introduction of hard drives for PCs, flat file systems became hierarchical file systems and directories and subdirectories became the norm, just as they always had been under Unix and other mainframe systems (where random-access mass storage systems have a longer history).

A hierarchical file system begins with a root directory. The root directory has certain fixed characteristics imposed by the operating system, such as a limitation of 256 file entries (including subdirectories). In contrast, no such limitations are imposed on subdirectories.

Fortunately, programmers do not need to know the low-level details of how the file system operates in order to create applications. The operating system itself—DOS, Windows 7, Linux, or whatever—handles much of the scut work involved in file access. WinBatch (or any other high-level language) provides functions for handling file and directory management, which make file operations virtually painless.

Even though the details are taken care of for you, a general understanding of how files and directories function is still recommended for the programmer, because understanding what is happening makes using the provided functions easier. Or, in other words, simply knowing how to use a tool isn't half as good as understanding how a tool works. Ergo, in the following sections, as well as learning how to use files and directories, you will also learn something of the inside structures and how the functions work.

Hard Drive Management

A common requirement for a program is to get file name and directory information. We'll begin with a custom utility that uses several WinBatch functions to retrieve this information, then we'll look at an alternative approach. Finally, we'll look at some of the other useful directory functions.

A Utility for Directory Operations

We introduced a drive/directory/file operation back in [Chapter 6](#), when we needed to open a file to perform a search/replace operation on the contents. Several of the examples discussed in previous chapters involve drive/directory/file operations. We didn't go into detail about these operations earlier, focusing instead on the topic at hand. Now we are

ready to concentrate on the mechanisms for retrieving file and directory information. At that time, the operation was given a minimal description, with the promise that the subject and operations would be covered in more detail in this chapter.

One of the earlier examples presented in [Chapter 3](#), named [FileListBox.wbt](#), demonstrates displaying a file-selection dialog. Now we'll recreate a similar utility but this time we'll make it a callable function, named [CallFileList.wbt](#). Once we've created the [CallFileList.wbt](#) utility, the utility and selection dialog can be used by other applications without needing to rewrite the code each time.

Planning the Utility

We could create the [CallFileList.wbt](#) utility on a hit-or-miss basis and then modify it later as needed, but it's far more reasonable to take moment first to decide what the utility should do. The need for design planning was introduced in [Chapter 1](#), where we discussed the principles of designing applications (granted, thus far, there hasn't been a great deal of need for applying the principles). According to those principles, we should determine how the utility will be called, what presentation the utility will use, and what information the utility will return.

File selection is a feature needed by a multitude of applications, but how we select a file can vary greatly. Different applications have different needs, and our [CallFileList.wbt](#) utility should be able to accommodate the basic variations.

The two main items that may be wanted as available options before selecting a file are a file mask and a starting drive/directory. Let's begin with these as minimal criteria.

File Specification

A *file mask* is a mask limiting the displayed files to those that match a specification, either for the file name or the file extension. File masks can be as simple as: *.* to show all files, *.txt to display only files with the text format extension, A*.* to show only file names beginning with the letter *a*, or variations on these.

Since this will be the argument wanted most often, we'll specify that the first parameter passed to [CallFileList.wbt](#) should always be the file-specification string. Furthermore, because we expect to return a file name, we will also require at least one argument is supplied: either a file name or a file mask. Without a supplied argument when [CallFileList.wbt](#) is invoked, we simply have no means of returning the selection of file names, which is the purpose of the utility function.

Wildcards in File Specifications

The asterisk character (*) is a wildcard that will be matched by any characters and any length. For example, a specification of F*.* matches `FileList.wbt`, `FileList.wbt.backup`, `Format-Number.wbt`, `FormatNumber.wbt.backup`, `Free Disk Space.-WBT`, `"Free Disk Space.wbt.autosave`, and `Free Disk Space.wbt.backup`.

Another wildcard is the question mark character (?), which represents any single character. For example, given a list of file names including `File_2_notes.txt`, `File_3_remarks.doc`, `File index list.txt`, and `Free Disk Space.wbt` and a file specification of `File_?_*.*`, the list of matching files consists of `File_2_notes.txt` and `File_3_remarks.doc` only. Here, the two asterisks in the file specification, `*.*`, can match both `...notes.txt` and `...remarks.doc`. The `File_?_...` portion of the specification limits the match to `File_2_...` and `"File_3_...`

Multiple question marks will match any sequence of characters on a one-for-one basis. Thus, a file specification of `????.txt` matches `Note.txt` and `List.txt`, but does not match `Note3.txt` or `MyList.txt`.

Drive/Directory Specification

A drive/directory specification is our second option. Since such a specification won't always be needed, we'll make this argument optional. If it is supplied, it will be used; if not, the default will be the current directory (the directory where the utility is called).

The drive/directory specification can be any valid drive or combined drive/directory specification. Thus, the user can indicate `D:` to specify the root directory on drive D: or `D:\Program Files\Work Files` to select a specific directory on the drive.

Any valid directory specification is accepted. For example, if the current directory is `J:\WinBatch\Chapter 10`, a specification of `..\Chapter 07` moves from the current directory up one level and then back down to the new directory.

Exploring the CallFileList Utility

The [CallFileList.wbt](#) utility is similar to the [FileListBox.wbt](#) program introduced in [Chapter 3](#) but this version requires either one or two parameters: at a minimum, the default files specification and, optionally, the directory specification. To test (and demonstrate) the [CallFileList.wbt](#) utility, a second application, called [DirTest.wbt](#), allows entering a file and directory specification, then calls [CallFileList.wbt](#) and, finally, reports the selection.

In order to allow [CallFileList.wbt](#) to be invoked with a choice of parameters, we begin by checking to see how many parameters were supplied:

```
Switch param0
  case 2
    selectDir   = %param2%           ; two parameters supplied
                                           ; second must be initial directory

  case 1
    selectFile  = %param1%           ; first parameter is file spec
    break

  case 0
                                           ; must have one parameter
```

```

        exit
    EndSwitch

```

If two parameters were supplied, the second argument must be the directory specification, and this is copied to the `selectDir` variable.

If one parameter is supplied (or if the previous `case` falls through), the first parameter (required) is copied to the `selectFile` variable. At this point, a `break` statement moves execution out of the `switch/case` structure.

Finally, if no parameters were supplied, the [CallFileList.wbt](#) utility simply exits, since continuing without being able to return a file selection would be pointless.

Next, we have two tests for the `selectDir` variable:

```

If selectDir != ""
    If DirExist (selectDir) == @FALSE
        Message( "Drive or Directory Error", selectDir : " was not a
valid drive/directory specification" )
        return
    Endif
    DirChange( selectDir )
EndIf

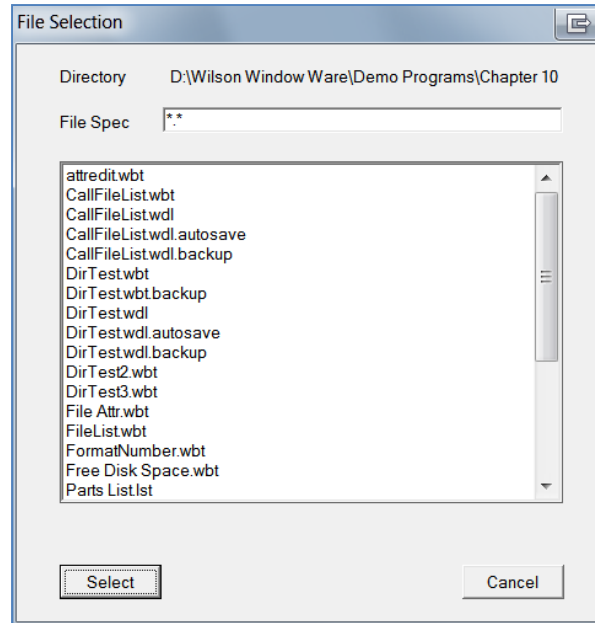
```

See also the `DirExist` function for another testing option.

The first test is to see if `selectDir` is an entry; that is, to check that the variable isn't empty. The second test is to see if the directory specification is a valid directory. If the directory doesn't exist (or if directory does exist but the specification is incorrect), the program displays an error message and returns.

If the directory specification is valid, then the `DirChange` function has already changed to the requested drive/directory. The requested directory is now the current (active) directory, and we're ready to call the dialog to display the directory tree and files. The figure below shows the dialog presented by the [CallFileList.wbt](#) utility.

Introduction to Programming



Here, the Directory entry at the top of the dialog shows the current drive and path. The file list shows, in order, all files matching the file-specification mask, the root of the current directory ([. .]), and (not visible) a list of the drives available on the system ([-a-] through [-j-]). Selecting one of the other drives or a subdirectory redisplay the file list for the new drive/path specification. Changing the drive or directory specification in the file-list dialog also automatically changes the current directory:

```
While @TRUE
    If Dialog( "FileSelect" ) == @FALSE then return
    ; if a valid file was not selected, redisplay the dialog
    If FileExist( selectFile ) then break
Endwhile
```

In contrast, selecting a specific file requires clicking on the Select button to return. Once the dialog has returned, the `selectFile` specification is tested to ensure that the file is valid. While it may seem reasonable to assume that the selected file name is valid, assumptions tend to become mistakes; the test is convenient insurance. If the `selectFile` specification is valid, then we can break out of the `while` loop.

The `FileExist` function simply verifies that the file path and file name are valid to make sure that a path and file with the given specification do exist. This does not say anything about what the file is or what the file contains, only that it can be found.

Before returning the file selection, we have one more task: checking again to see if a drive/path specification was passed as a parameter:

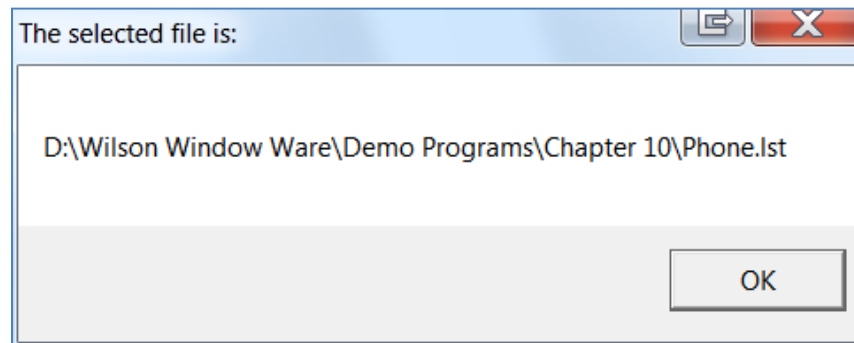
```

If param0 > 1 then %param2% = DirGet()           ; optional parameter
%param1% = selectFile                           ; always return this value
Return

```

If a drive/path specification was supplied, then we want to return the current directory, using `DirGet`, as `param2`. (The selected file is always returned as `param1`.)

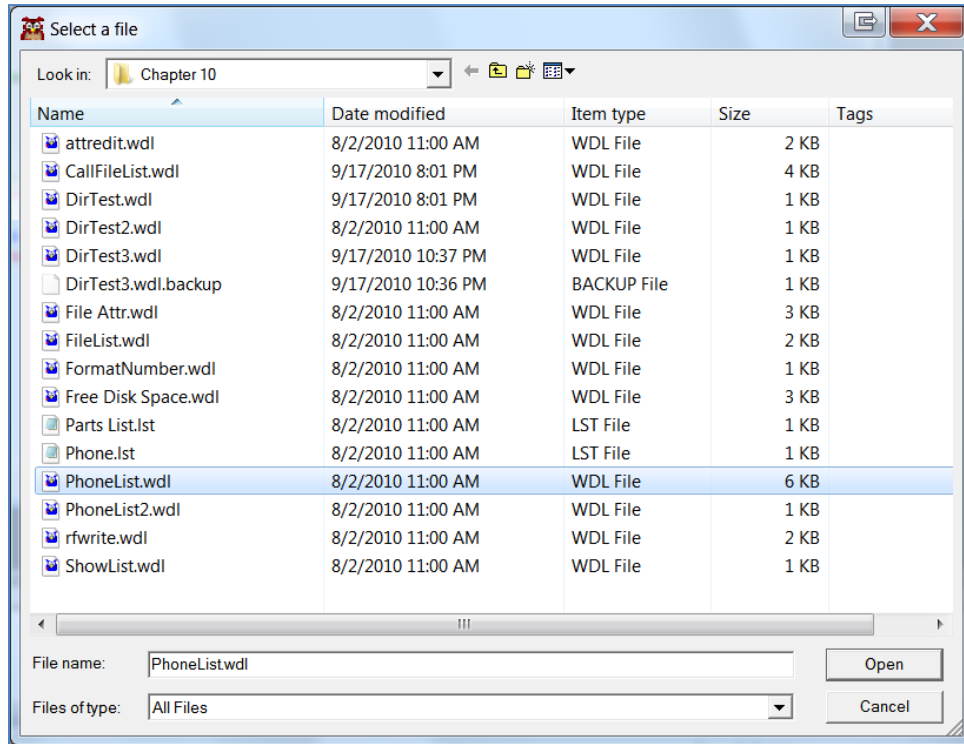
On return, the [DirTest.wbt](#) program displays the reported selection:



The [CallFileList.wbt](#) utility is only one method of getting file name and directory information. However, examining it has shown you how the file and directory operations work from the inside, so to speak, giving you some idea of what is going on behind the scenes. Now let's look at a better way to perform the same tasks.

The Windows Common File Dialog

Because of the importance of file access to all types of operations and applications, all recent versions of Windows provide a common file dialog to manage file selection. In WinBatch, the Common File Dialog shown below is invoked using the `AskFileName` function.



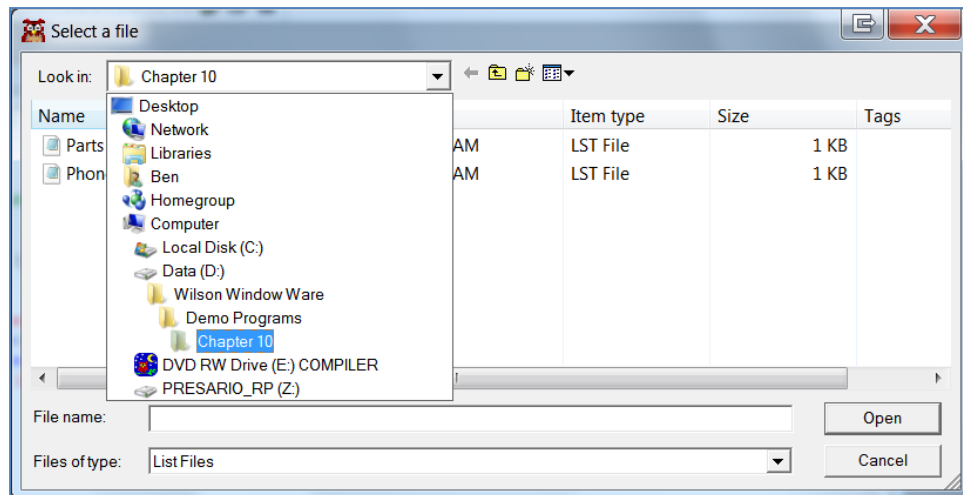
The Common File Dialog is much more elaborate than the file-selection dialog we created in the [CallFileList.wbt](#) utility (shown earlier). Using the Common File Dialog, we have a complete file-selection utility, which we can customize in several ways, including changing the dialog title, the types of files shown, how the file list is displayed, as well as choosing a variety of other options.

The Common File Dialog offers a standard file-selection utility that generally is used by all Windows applications. The Common File Dialog provided by each version of Windows may differ in appearance, but the principal features and the parameters used to call the Common File Dialog remain constant. By providing this degree of standardization, the users are not required to learn a new file-selection process for each application.

Although you should be familiar with Common File Dialog—since virtually every Windows application makes use of this facility—let's explore the dialog's operations and features from a programmer's point of view.


Features of the Common File Dialog


Beginning at the top of the dialog, on the toolbar, the caption "Look in:" labels a pull-down list box, as shown below. This list offers a drive/directory tree with icons for drives and drive types and for folders (or directories). The structure of drive/directory/subdirectory is shown by indentations in the list, and any branch can be expanded to show subdirectories (if any) or collapsed to hide directories and subdirectories.




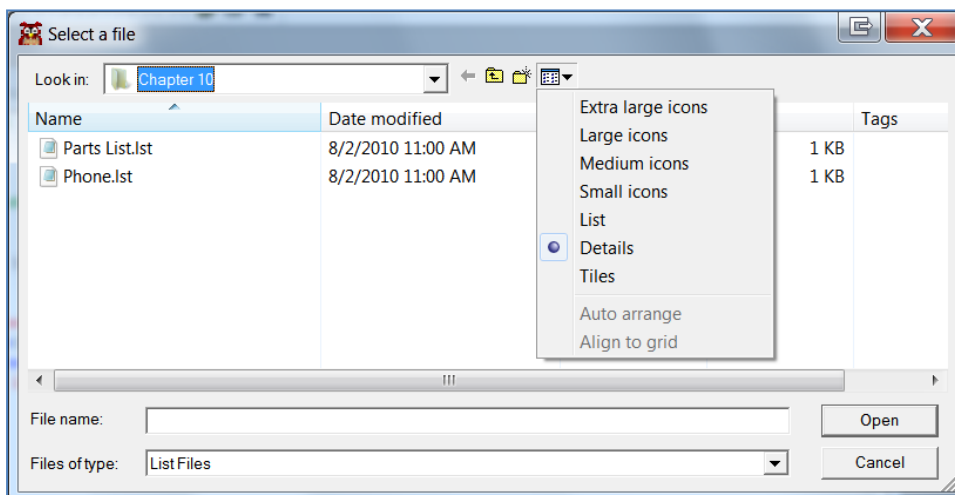
Selecting from the directory tree

In the illustration, the open folder icon shows the current display is on drive D: in the Wilson Window Ware / Demo Programs / Chapter 10 subdirectory. The window behind the pull-down list shows the list files (and any subdirectories) in this directory.

To the right of the pull-down list, the  icon is an Up-One-Level button. Clicking on the icon will step the display up to the root of the current directory.

The next icon  is the New Folder button, which allows you to create a new directory or subdirectory in the currently selected directory or drive.

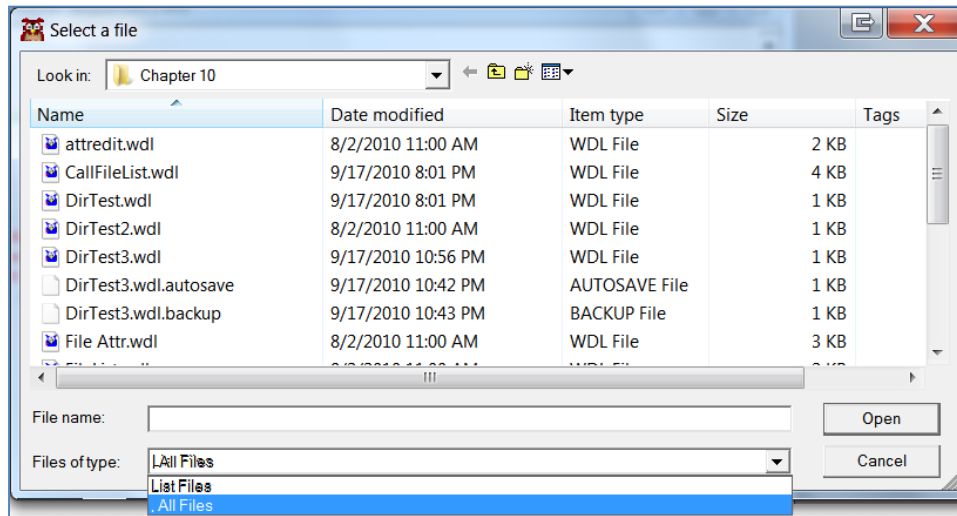
Last, the paired buttons  at the right end of the toolbar allow you to switch between list formats.



More important than the toolbar buttons and the display options is the fact that the Common File Dialog allows the program calling this display to specify which file types

Introduction to Programming

will be displayed initially and to provide a type-selection list. At the bottom of the dialog, a pull-down list allows the user to select a file type, as shown following.



The Files of type list is created by the application—in other words, the programmer—calling the Common File Dialog. Each entry in the list consists of two elements:

- The file type label, which appears in the pull-down list
- A file-specification mask, which is not shown but which is used to select the files displayed (the file selection is the same process discussed earlier in the section about the [CallFileList.wbt](#) utility)

Invoking the Common File Dialog

The [DirTest2.wbt](#) demo program shows how the `AskFileName` function is used to invoke the Common File Dialog:

```
sDirDrive = "" ; default is always the current directory
sFileTypes = "All Files|*.*|WIL files|*.wbt;*.mnu|Text files|*.txt|"
sFileSpec = AskFileName( "Select a file", sDirDrive, sFileTypes, "", 1 )
Message( "The selected file is:", sFileSpec )
exit
```

The `AskFileName` function is called with the five parameters shown below and explained in the following sections.

AskFileName Parameters

Variable Type	Variable Name	Comment
String	Label	The title that will appear on the Common File Dialog when it is displayed
String	Directory	The initial drive and directory specification; i.e., where the file display will begin
String	Filetypes	A bar-delimited list providing the file-type labels and the file-specification masks
String	Default filename	May be a default file name or a file mask
Integer	Flag	0 selects the File Save dialog style, 1 selects the File Open dialog style, 2 selects the File Open dialog style allowing multiple files to be selected, 3 selects the File Save style without a "Replace" confirmation (i.e., an automatic save when overwriting an existing file.)

The Label Parameter

The label parameter is simply any string you want displayed at the top of the Common File Dialog. In the [DirTest2.wbt](#) example, the label is "Select a file", but this label can be changed to accommodate an application's needs. For example, you might want to say something like "Pick a data file for processing" OR "Select or enter a name for saving data". Essentially, what the dialog caption says is up to you, but brevity is appropriate.

The Directory Parameter

The initial directory parameter is optional. In the [DirTest2.wbt](#) example, it is passed as an empty string, meaning the dialog opens in the current (active) directory. You may specify any drive/directory desired, either absolute, as in "C:\MyDir\MySubDir", or relative, as in "..\SubDir2". The rules are the same as for the DOS CD (CHDIR) command.

Before specifying an absolute or a relative directory, use the `DirExist` function to ensure that the specification is valid. Then, if the directory does not exist, pass the argument as an empty string.

The Filetypes Parameter

The file-types list argument (the pull-down list in the preceding illustration) consists of a string composed of description and mask substrings using the vertical bar character (|) as a delimiter. The file-type masks can be any DOS wildcard file mask. The format for this string is:

```
"description|mask|...|description|mask|description|mask|"
```

Introduction to Programming

There are two requirements:

- Entries must appear as pairs.
- The string must end with the vertical bar character.

In the [DirTest2.wbt](#) example, the file-types argument appears as:

```
sFileTypes = "All Files|*.*|WIL files|*.WBT;*.mnu|Text files|*.txt|"
```

This can be broken down as follows:

All files	*.*
WIL files	*.WBT; *.mnu
Text files	*.txt

Notice that the entry for WIL files has two file-mask specifications separated by a semicolon. There is no limit on the number of file masks. When the corresponding file-type label is selected, the dialog will display all files matching any of the specified masks.

The provided list is not sorted, and the default (initial) file type displayed will always be the first item in the list.

The Default Filename Parameter

The default file name or default file-mask specification is an optional parameter. If supplied, this file name or mask will appear in the "File name:" field of the dialog and will act as a mask overriding the default file-type mask. If the entry is a default file name, it can be accepted by the user clicking on the Save or Open button (depending on the type of operation chosen).

The Flag Parameter

The flag parameter is an integer argument where a value of 0 selects the File Save dialog style and a value of 1 selects the File Open dialog. A value of 2 also selects File Open but allows one or more files to be selected. Last an argument of 3 selects File Save but with an automatic overwrite if there is an existing file of the same name.

Returning a File Name

After the user makes a valid selection (enters a valid file name in the File name: field or selects a valid file from the displayed names) and clicks the Open or Save button, the Common File Dialog closes, and the `AskFileName` function returns a string containing the fully qualified file name (a complete drive/directory/file name/extension). At this point, it is up to the application to decide what to do with the returned file name—whether to open a file or to do something else with the information.

In the [DirTest2.wbt](#) program, the only use made of the file name is to display a message box reporting the selection. Later in this chapter and in following chapters, we'll use the

Common File Dialog for more realistic purposes. Before we do this, we have a few additional directory functions to discuss.

DIRECTORY INFORMATION FUNCTIONS

Along with the functions you've seen demonstrated so far, WinBatch provides other useful drive management functions. These fall into the categories of long file names versus short file names, default directory information, and drive information.

Converting Long and Short File Names

The change from the old 8.3 (*filename.ext*) file name format to the long file name format has been a tremendous relief to virtually everyone. However, there are still occasions when the new long file names require conversion to the old format or vice versa.

WinBatch provides two functions for this purpose:

- The `FileNameLong` function accepts a string argument containing a fully qualified file name, with or without a drive and path specification, and returns a string with the complete path/file name in the long format.
- The `FileNameShort` function performs the same way but returns the file name and any directory path names in the short (8.3) format.

The secret is that no conversion is actually being performed. These functions operate by querying the file system directly to return either the long or short forms.

Even though you are probably accustomed to seeing the long file name format when you open a directory, the file system actually maintains two names for each file and directory: one in the long format and a second name, generated from the first, in the short format. The bitmap in your Windows directory, for example, which appears as `Carved Stone.bmp`, also has a short name, `carved~1.bmp`, stored in the subdirectory file table.

Locating Default Directories

Throughout this chapter, we've mentioned the "current" or "active" directory. There's nothing esoteric about this; the current directory is simply the default directory that an application will access. Until a different directory has been specified, the current directory normally will be the directory where the application was launched.

In addition to the `DirGet` function used to return the current drive/directory specification (demonstrated in the [CallFileList.wbt](#) utility, discussed earlier in the chapter), there are three other functions that return drive/directory specifications: `DirHome`, `DirWindows` and `DirScript`.

The `DirHome` function is a specialized function returning the directory where the WIL interpreter's executable files are located. This information is actually stored in the system registry when WinBatch is installed. The usefulness of this data is a little uncertain since, to be perfectly honest, we can't think of many reasons for needing this data while executing an application.

Introduction to Programming

The `DirWindows` function is similar to `DirHome` but returns either the specification for the Windows directory or the `Windows\System` directory, either of which may occasionally be required by applications. For example, suppose that you've written a WinBatch utility that is used to install another WinBatch application or applications. The programs you are installing, however, may need to have one (or more) .DLLs, such as `WBD__44I.DLL`, copied as part of the installation. Although .DLLs can be located in the same directory as the application, ideally these – and any Extender DLLs – should be installed in the same directory as the executable file (.EXE).

To find the Windows directory, `DirWindows` is called as:

```
sWindowsDirectory = DirWindows( 0 )
```

To find the `Windows\System` directory, the call is:

```
sSystemDirectory = DirWindows( 1 )
```

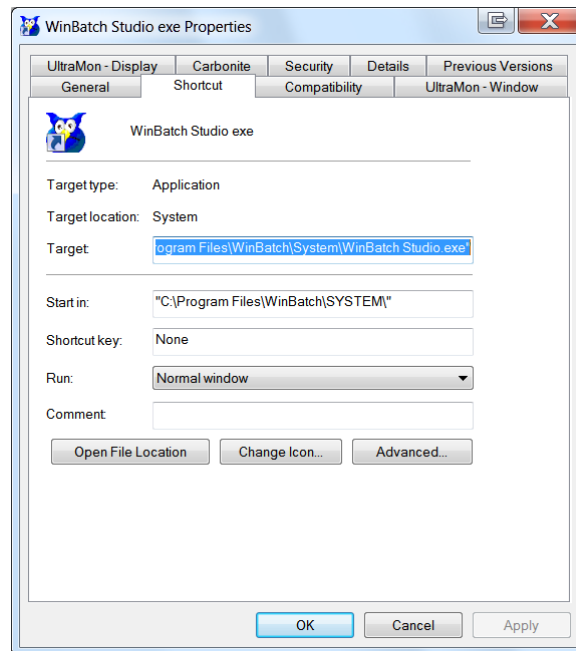
In both examples, the return value is a drive/path specification with the full directory information.

The `DirScript` function is a convenient method of finding the directory where the program script is located and you'll have seen this in several of the demo scripts in the form: `DirChange(DirScript()) ...` where the desired result is to make the current (active) directory the same as the directory where the script is located. This is especially critical if you are using the `Call` function to invoke another (external) script or executable.

If the current WinBatch script is a child program that was called with the `Call()` function, this function will return the full directory path (without the filename) of the main (calling) program.

Redefining the Default Directory

When an application is installed as a desktop icon or as a Start menu item, the application is assigned properties that include a startup (or Start in) directory. By default, this will be the same directory where the application is located, but you can change the application's startup directory. For example, right-click on your desktop WinBatch icon (you do have WinBatch installed by now, don't you?) and select the Properties option from the pop-up, you'll see the dialog for an application shortcut:



You can change the Start in property to specify any directory desired and, when WinBatch is opened, the specified directory will be the current directory used by the WinBatch. This specification does not affect any other application and does not prevent you from changing the directory at any time.

Getting Drive Information

For some applications, we may want to find out about both local and network drives, not for information about what files are located where, but to discover which drives are available and where we can find free space. WinBatch offers a half-dozen `Disk...` functions. Three of these—`DiskScan`, `DiskSize`, and `DiskFree` — are illustrated in the [Free Disk Space.wbt](#) program. The four remaining — `DiskExist`, `DiskInfo`, `DiskVolInfo` and `LogDisk` — are discussed in the Windows Interface Language Help File.

The DiskScan Function

The `DiskScan` function is called to find out which drives are available. There are six types of drives that we can request information about: unused drive IDs, removable (floppy) drives, local fixed drives (hard drives), remote (network) drives, CD-ROM drives, and RamDisk drives. The following shows the flag values used to request each drive type.

Drive Types used with DiskScan

Flag	Binary	Return value
0	0000 0000	List of unused disk IDs
1	0000 0001	Removable (floppy, ZIP, JAZ, etc.) drives
2	0000 0010	Local (fixed) hard drives
4	0000 0100	Remote (network) drives
8	0000 1000	CD-ROM (32 bit)
16	0001 0000	RamDisk (32 bit)
32	0010 0000	Persistent non-connected drives
64	0100 0000	USB buss disk drives (W2K and later only)

Notice that the flag values are all powers of 2 rather than sequential. This allows two or more flags to be used at the same time—what is essentially a binary OR operation (as described in [Chapter 5](#)). However, don't worry about remembering how to OR two values. All that's really required is to add the flag values as base₁₀ numbers and to use the sum.

Therefore, to check both local and remote drives, adding the flags 2 and 4 yields 6 (0000 0110). In the [Free Disk Space.wbt](#) demo, we call `DiskScan` as:

```
sDrives = DiskScan( 6 ) ; 6 = 4 + 2 = Network and Local Drives
```

This returns a list of both local fixed drives and network drives (if any are available) as a string with the format "A: C: D: E: F: G: H: I: J:".

With a list of the available drives, the next steps are setup and preparation:

```
sDrives = DiskScan( 6 ) ; 6 = 4 + 2 = Network and Local Drives
nMax = StrLen( sDrives )
...
nDrive = 1
TotalSize = 0
TotalFree = 0
DriveReport = "Drive":@TAB:"   Total":@TAB:"   Free":@TAB:"%%
Free":@CRLF:@CRLF
```

The DiskSize and DiskFree Functions

Once everything is ready, a loop is initiated to extract the individual drive letters from `sDrives` and then to call `DiskSize` and `DiskFree` to retrieve information on each drive. `DiskSize` and `DiskFree` report in bytes, which is more detail than we're likely to want, so we begin by converting the size values to kilobytes:

```
While @TRUE
    NextDrive = StrSub( sDrives, nDrive, 1 )
    nSize = DiskSize( NextDrive ) / 1024      ; convert to kilobytes
    nFree = DiskFree( NextDrive ) / 1024
```

For gigabyte-plus drives, even kilobytes can be awkward. The next step is to decide how large a drive we're dealing with—based on the total size of the drive, not the free space—and decide whether we want to report kilobytes or megabytes:

```
If nSize > 10240
    sUnit = " Mb"
    nSize = nSize / 1024.0 ; convert both to Megabytes
    nFree = nFree / 1024.0
Else
    sUnit = " Kb"
EndIf
```

Depending on the units used to report, we also want to have a corresponding label as either "Mb" or "Kb".

As we get the information from each individual file, we also want to track both the total free space and the total drive sizes:

```
TotalSize = TotalSize + nSize
sSize = Int( nSize )
Call( "FormatNumber.wbt", "sSize" )
sSize = StrFixCharsL( sSize, " ", 16 )

TotalFree = TotalFree + nFree
sFree = Int( nFree )
Call( "FormatNumber.wbt", "sFree" )
sFree = StrFixCharsL( sFree, " ", 16 )
```

Introduction to Programming

The [FormatNumber.wbt](#) utility that we call here is similar to the [FormatCurrency.wbt](#) utility introduced in [Chapter 9](#) (see the subroutine `Format_Dollar_String`) to add commas, decimal places, and a dollar sign to large numerical strings to make them more readable. The principal differences in [FormatNumber.wbt](#) are that no dollar sign is added and decimal places are optional.

Next, since we want to report free space as a percentage, which is more readable than simple size information, we begin with a test to ensure that we won't encounter a divide-by-zero situation (which can happen when checking a CD drive that does not have a disk in place):

```
If nSize > 0
    sPercent = Int( ( nFree / 1.0 ) / ( nSize * 1.0 ) * 100 )
    sPercent = StrCat( sPercent, "%" )
    sPercent = StrFixCharsL( sPercent, " ", 8 )
Else
    sPercent = ""
EndIf
```

For purposes of calculation, both `nFree` and `nSize` are multiplied by 1.0. This ensures that the calculations are made as floating-point rather than integer operations (as discussed in [Chapter 9](#)), even though we multiply the product by 100 and then reconvert to an integer for display.

The remainder of the loop is mostly a matter of formatting the information retrieved and calculated for eventual display:

```
BoxText( "Checking " : NextDrive : ":" )
DriveReport = StrCat( DriveReport, NextDrive, ":" )
DriveReport = StrCat( DriveReport, @TAB, sSize, sUnit )
DriveReport = StrCat( DriveReport, @TAB, sFree, sUnit )
DriveReport = StrCat( DriveReport, @TAB, sPercent, @CRLF )
nDrive = nDrive + 3 ; each entry is 3 bytes long
If nDrive > nMax then break
EndWhile
```

Notice that the `nDrive` variable, which is used to show the position of the drive identifier letter, is incremented by 3, not by 1. Remember that each drive specification in the list consists of three characters: the drive letter, a colon, and a separating space. (The `ItemExtract` function could also be used, and you are invited to try it, but the resulting code is somewhat more cumbersome than is actually necessary.)

Once we've stepped through the drives, all that remains is to format the information on the totals and to present the information.

```

DriveReport = StrCat( DriveReport, @CRLF, sUnit, "ytes" )

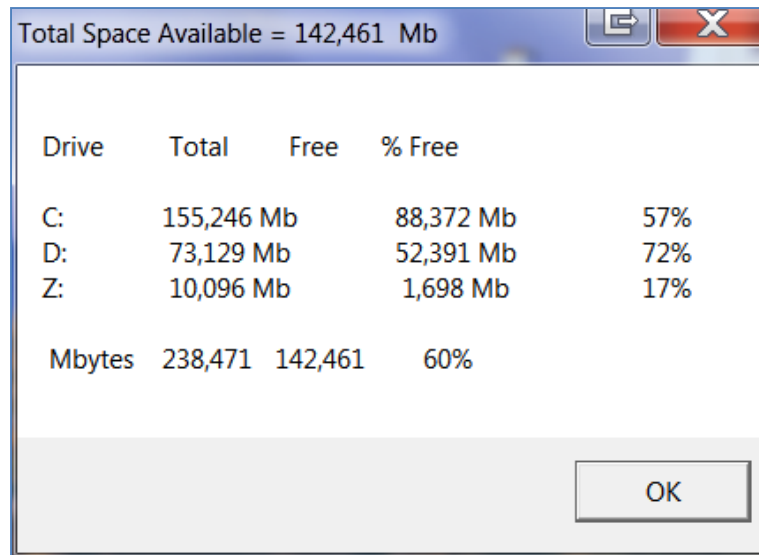
sTotalSize = Int( TotalSize )
Call( "FormatNumber.wbt", "sTotalSize" )
sTotalSize = StrFixLeft( sTotalSize, " ", 16)
n = StrLen( sTotalSize )
DriveReport = StrCat( DriveReport, @TAB, sTotalSize )

sTotalFree = Int( TotalFree )
Call( "FormatNumber.wbt", "sTotalFree" )
sTotalFree = StrFixCharsL( sTotalFree, " ", 16 )
DriveReport = StrCat( DriveReport, @TAB, sTotalFree )

sPercent = Int( ( TotalFree / 1.0 ) / ( TotalSize * 1.0 ) * 100 )
sPercent = StrFixCharsL( sPercent, " ", 8 )
DriveReport = StrCat( DriveReport, @TAB, sPercent, "%%" )

BoxShut()
TotalFree = Int( TotalFree )
Call( "FormatNumber.wbt", "TotalFree" )
Message( "Total Space Available = ":TotalFree:" ":sUnit, DriveReport )
Drop( TotalSize, TotalFree, DriveReport, Drives, NextDrive )
exit

```



The screenshot shows a window titled 'Total Space Available = 142,461 Mb'. Inside, there is a table with four columns: 'Drive', 'Total', 'Free', and '% Free'. The table lists data for drives C, D, and Z. At the bottom, there is a summary row for 'Mbytes' and an 'OK' button.

Drive	Total	Free	% Free
C:	155,246 Mb	88,372 Mb	57%
D:	73,129 Mb	52,391 Mb	72%
Z:	10,096 Mb	1,698 Mb	17%
Mbytes	238,471	142,461	60%

A sample report generated by the Free Disk Space demo.

Huge Numbers

With terabyte (and larger) drives available, both the `DiskSize` and `DiskFree` functions may easily need to handle values larger than 2 Gb. By default, results larger than 2 Gb are returned as floating point numbers but both functions now accept a flag specifying a 'Huge number'.

'Huge numbers' – as discussed previously – are numbers too large to be converted to integer values. Instead, a 'Huge number' is a special data type – a long decimal number string – which can represent a number too large to be converted to an integer. 'Huge numbers' cannot be modified using the standard arithmetic operations but must use the Huge Math extender.

File Management

File operations have been introduced in several of the previous demo programs. For example, we used file operations in the string-operation demonstrations simply because a reliable source of string material was needed. Conversely, if we had tried to introduce files prior to string operations, strings would have been needed to illustrate files. In short, it is very difficult to write sample applications that use only one or two operations and even more difficult to avoid using functions or operations until we've reached the appropriate point to cover them in detail. (On the other hand, if matters were that overly simple in the first place, there would hardly be any need for you to read this book.)

Here, we'll discuss the various file functions provided by WinBatch. However, before getting into the usual methods of accessing files, we'll talk about one special function for accessing files containing lists: the `AskFileText` function.

A Shortcut for Lists

The `AskFileText` function opens and reads a file—ideally a CRLF-delimited text file—and displays the contents as a list. The display can be presented as a sorted or unsorted list, and list selection may be limited to one item or multiple items. Following selection, `AskFileText` returns the selection as a tab-delimited list.

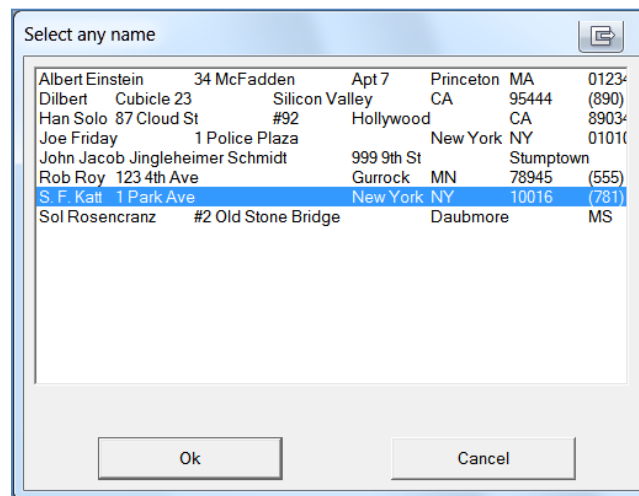
`AskFileText` is called as:

```
AskFileText(title, filename, sort_mode, select_mode,
selection_required)
```

The five parameters are specified as follows:

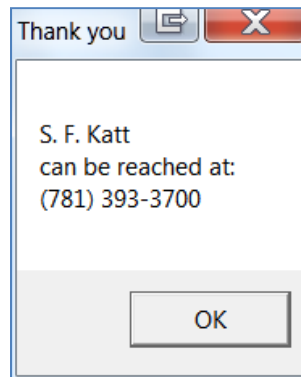
- The `title` parameter is a string setting the caption title.
- The `filename` parameter identifies the file to be read.
- The `sort_mode` parameter can be either `@SORTED` or `@UNSORTED`.
- The `select_mode` parameter may be specified as `@SINGLE` or `@MULTIPLE` or `@EXTENDED`. The `@SINGLE` flag limits selection to a single item; `@MULTIPLE` allows selection of one or more items; `@EXTENDED` allows selection of multiple items by extending selection using the shift key or the mouse.
- If the `selection_required` parameter (optional) is set to `@TRUE`, the OK button will be grayed out until at least one item has been selected. The parameter defaults to `@FALSE`.

The [ShowList.wbt](#) program calls `AskFileText` with a reference to the source file [Phone.lst](#) producing the display shown here:



It would be nice if there were a method to arrange this data in columns – short of using a fixed width font and calculating string lengths – but, unfortunately, there is no ready columnar facility available

What is done with the selection after `AskFileText` returns depends on the needs of your application. This particular example is brief and uses a message box to report the results:



The practical use for the `AskFileText` function, however, is to allow the user to make a selection from a list maintained in an external file. For example, suppose that we have an application used to book rooms in a motel. Since there is a terminal in every room (obviously, a Silicon Valley motel), once the maid has finished cleaning the room, the maid clicks on a network utility to tell the desk that the room is ready, or maybe to report an error or some problem. In any case, a list of ready-to-rent rooms is maintained as a file. Then, when the desk clerk is deciding which rooms are available, `AskFileText` queries the ready file and pops up the list for selection. All the clerk has to do to enter the room number (and maybe price information and other details) in the registration is to select a ready room with a double-click. Granted, this is a fairly trivial suggestion, but it illustrates the principle. You can probably think of your own uses. So, let's get on to the more conventional file operations.

File-Operation Functions

Access to a file—whether text, data, records, or something else—always begins in the same way: you first need to open the file. Once the file is open, data can be read from the file or written to the file (depending on the mode the file is opened with). Then, once you're finished with the file, the file needs to be closed.

And that's the short version, with three simple steps: open, use, close. However, that's not quite all of the story (but you expected that, didn't you?).

Before we make things too complicated, however, there are four file access functions which, in many cases, can simplify your file operations. These are `FileGet` and `FileGetW` and `FilePut` and `FilePutW`.

The `FileGet` and `FileGetW` functions each are called with a fully qualified filename and return the contents of the file as a string. The difference between the two is in the `...W` identifier because the `FileGet` function returns a simple string using 16-bit characters and the `FileGetW` (for Wide) converts a file into a Unicode string variable (using 16- or 32-bit characters).

In like fashion, the `FilePut` and `FilePutW` functions write a conventional string (8-bit characters) or a Unicode string (16- or 32-bit characters) to a file.

ANSI versus Unicode

The ANSI character set uses one byte per character to support 256 character which encompass the complete English alphabet, a set of special control characters and a set of primitive graphic characters. The ANSI character set, however, is quite inadequate to provide support for even a small fraction of the world's languages and alphabets.

The original Unicode alphabet using a 16-bit character set provided support for 65,536 characters, enough to provide support for quite a few additional alphabets ... but still not adequate for the entire world.

Consequently, the current 32-bit Unicode has a repertoire of more than 107,000 characters covering 90 scripts or alphabets ... and continues to evolve.

Now, having introduced the easy methods, let's go back to doing file operations the hard way ... because there may be circumstances where none of the preceding four functions are suitable for the tasks you need to accomplish.

Handling File I/O

I/O is the abbreviation for input/output. The term file I/O is almost a misnomer, since I/O seems to imply that both input and output are simultaneous. The truth is that you can either open a file to read the contents or open a file to write material to the file, but not both simultaneously. And, when you open a file to write new material, there's a second choice: the file can be truncated on opening (emptied and any existing contents discarded) or the file can be opened for appending (to write new material to the end of the file).

Thus, how the file will be used must be determined before you call the `FileOpen` function (but you can always close a file and reopen it again). After calling `FileOpen`, you can use the `FileRead`, `FileWrite`, or `FileClose` function.

The specified file must be a text file containing lines of text terminated by the carriage return/line feed (CRLF) characters. Within this limitation, the text contained in the file may be plain text, delimited data or numbers, or whatever. For files that contain binary data, you must use other functions, as discussed briefly in the "Binary File Operations" section later in this chapter.

The FileOpen Function

The `FileOpen` function is used to open standard ASCII/ANSI (only) files for reading, writing or appending. `FileOpen` is called with two parameters:

- The file name parameter can be any qualified file specification: either a name and extension or the file name and extension with a full drive path specification or a relative drive path specification.
- The mode parameter must be either "READ", "WRITE", or "APPEND", depending on the desired mode of operation.

In response, `FileOpen` returns a handle to the file, which is an integer value that temporarily identifies the file. The value returned will be zero if the file cannot be opened. The returned file handle will remain valid until a `FileClose` operation is executed.

WinBatch allows a maximum of 128 files to be opened at any one time.

In the [PhoneList.wbt](#) program, the first file access is performed as:

```
hFile = FileOpen( sDataFile, "READ" )
```

The `hFile` variable is used in subsequent operations until the file is closed. After that point, another `FileOpen` is required before any further access—a read, a write, or an append operation—can be performed.

The FileRead Function

The `FileRead` function is used to return (read) material from a file. `FileRead` is called with one parameter: the file handle supplied by a `FileOpen` operation. It returns a string of data from the file, ending when a CRLF character is read or when the end of the file (EOF) is reached.

Each subsequent `FileRead` operation reads another line of text, from the point where the previous read concluded, until the end of the file is found. When a `FileRead` operation comes to the end of a file, the returned string will be "*EOF*". In the [PhoneList.wbt](#) demo, the contents of the phone list data file are read as:

```
While @TRUE
    sLineIn = FileRead( hFile )
    If( sLineIn == "*EOF*" ) then break
    listPhone = ItemInsert( sLineIn, -1, listPhone, @CR )
    nCount = nCount + 1
EndWhile
FileClose( hFile ) ; Close the input file
```

When `FileRead` returns `"*EOF"`, the `break` instruction breaks us out of the loop and takes us to the `FileClose` operation. Since we're finished reading the file, there's no point in keeping the file open. It's always safest to close a file and reopen it later than to try to keep the file opened indefinitely.

Once we've read the data file, the next step is to parse the material and display it for use (operations similar to the ones you've seen in previous examples):

When the user selects a name from the list (left), [PhoneList.wbt](#) displays the full record for the selection in the fields at the right. The user can also click the Clear button to clear the current display, enter a new name and contact information, then click the Save New button to save the entry as a new addition. When an entry is saved, a check is made against the existing entries and, if a match is found, the old entry is deleted before the new record is written.

Notice that in the [PhoneList.wbt](#) demo, no provision appears in the dialog to simply delete an entry. This is left as an exercise for the reader. Here's one hint: the mechanisms are already in place ... all you need to do is figure out how to call them.

The FileWrite Function

As a general rule, any time you want to make a change to a file, the original file is erased (or renamed as a backup), and the file is rewritten with the new or changed material. In the [PhoneList.wbt](#) program, we dispense with keeping a backup version of the data. To make a change to the file, such as deleting an existing record, we will open the [Phone.lst](#) file for writing, an operation which truncates (erases) the existing file.

After deleting the old record from `listPhone`, which is done in memory, the file is opened as follows (see the `:Delete_Entry` subroutine in [PhoneList.wbt](#)):

```
:Delete_Entry
    listPhone = ItemRemove( nSelect, listPhone, @CR )
    hFile = FileOpen( sDataFile, "WRITE" )
```

Introduction to Programming

```
For i = 1 to ItemCount( listPhone, @CR )
    sTemp = ItemExtract( i, listPhone, @CR )
    FileWrite( hFile, sTemp )
Next
FileClose( hFile ) ; close the input file
return
```

As each line is extracted from `listPhone`, it is written to the file using `FileWrite`, which automatically adds its own CRLF character pair to the end of the line.

We can use the `FileWrite` function in another fashion as well. If we open the output file using the "APPEND" mode, the existing file is not truncated, but the file pointer—the location within the file where material will be written—is set to the end of the file so that the new material is added to the present data (see the `:Add_Entry` subroutine in [PhoneList.wbt](#)):

```
:Add_Entry
    hFile = FileOpen( sDataFile, "APPEND" )
    FileWrite( hFile, sNewEntry )
    FileClose( hFile ) ; close the input file
    listPhone = ItemInsert( sNewEntry, -1, listPhone, @CR )
    listPhone = ItemSort( listPhone, @CR )
return
```

The FileClose Function

The final standard operation is to close the file, as shown in all the previous file-operation examples. The `FileClose` function requires only one parameter: the file handle returned by the `FileOpen` function.

Remember that once `FileClose` has been invoked, the file handle used as a parameter is no longer valid and will produce an error if used further.

Manipulating Files

WinBatch includes several operations for manipulating existing files. These include the `FileAppend`, `FileMove`, `FileCompare`, `FileCopy`, and `FileDelete` functions.

The FileAppend Function

The `FileAppend` function should not be confused with opening a file in "APPEND" mode and using the `FileWrite` function to add material to the file. Rather, the `FileAppend` function is used to combine one or more files into a single file. The original source files are not altered by the `FileAppend` operation. `FileAppend` is called with two parameters:

- The source list parameter is a tab-delimited list containing one or more file names (which may include wildcard specifications).
- The destination is a file name where the concatenated data should be written. If the destination file exists, the material will be added to the end of the file; if the destination does not exist, it will be created.

The FileCopy Function

The `FileCopy` function is used to copy one or more files to a new location and, optionally, with a new file name. `FileCopy` is called as:

```
FileCopy( source_list, destination, warning_flag )
```

The three parameters are specified as follows:

- The source list parameter is a tab-delimited list containing one or more file names (which may include wildcard specifications).
- The destination parameter can be either a drive/path specification or a file name or file mask to rename the files.
- The warning flag parameter is either `@TRUE` if a warning is requested before any existing file(s) are overwritten or `@FALSE` for no warning.

The FileMove Function

The `FileMove` function operates in the same fashion as `FileCopy` except that the original files are deleted. `FileMove` is called as:

```
FileMove( source_list, destination, warning_flag )
```

Note that these parameters are the same as those used by `FileCopy`.

In fact, the `FileMove` function performs two different tasks depending on the source and destination locations. If the move is between locations on a single drive, the actual operation does not require copying the files but only changing the directory information. If the move is between locations of different drives, the files are copied to the destination and then deleted from the source.

The FileRename Function

The `FileRename` function allows renaming a file or a group of files. It is called as:

```
FileRename( source_list, destination )
```

The two parameters are specified as follows:

Introduction to Programming

- The source list parameter is a tab-delimited list containing one or more file names (which may include wildcard specifications).
- The destination parameter may also include a wildcard specification and operates in the same fashion as the DOS RENAME command.

The `FileRename` function, unlike the `FileMove` function, cannot reassign the file location.

The FileCompare Function

The `FileCompare` function is called with the names of two files and compares the file contents, returning an integer value to report the results. `FileCompare` is called as:

```
nResult = FileCompare( filename1, filename2 )
```

`FileCompare` returns the following result codes:

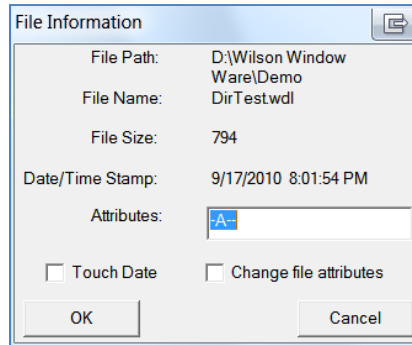
- 0 The contents of the two files are identical
- 1 The files are the same size but the contents are different and the first file is newer
- 1 The files are the same size but the contents are different and the second file is newer
- 2 The files are different in size and contents and the first file is newer
- 2 The files are different in size and contents and the second file is newer
- 3 The second file cannot be found
- 3 The first file cannot be found
- 4 Neither file can be found

The FileDelete Function

The `FileDelete` function is used to delete one (or more) files. `FileDelete` is called with a tab-delimited list of files to be deleted. The list may include any valid (DOS) wildcard specifications.

Additional File Functions

In addition to the file functions already discussed, WinBatch provides a variety of specialized functions to parse file names, locate files, handle file attributes, retrieve file size information, and a few miscellaneous operations. The `FilePath`, `FileAttrGet`, `FileAttrSet`, `FileTimeGet`, and `FileTimeTouch` functions are demonstrated in the [FileAttr.wbt](#) program



The file functions are listed with brief descriptions.

Specialized File Functions

Function	Description
Drive/Path/Name/Extension	
FileExtension	Returns the extension from a file name
FileFullName	Returns a fully qualified file name including the drive and path specification
FilePath	Returns the path portion of a fully qualified file name
FileRoot	Returns the root portion of a file name (removes the file extension)
File I/O Operations	
FileGet / FileGetW	Returns the contents of a file as a string
FilePut / FilePutW	Writes a string to a file
ArrayFileGet	Converts a file to a one-dimensional array
ArrayFilePut	Writes a one-dimensional array to a file
ArrayFileGetCsv	Converts a comma separated value (CSV) file to a two-dimensional array.
ArrayFilePutCsv	Writes a two-dimensional array to a comma separated value (CSV) file.
Locating Files	
FileLocate	Locates a file in the current directory or along the DOS path
FileItemize	Returns a tab-delimited list of files matching a wildcard

Introduction to Programming

	specification
File Attributes and Size	
FileAttrGet	Retrieves the attribute flags for a file
FileAttrSet	Sets attribute flags for a file
FileSize	Returns the total size for a file (or group of files)
Date/Time	
FileTimeCode	Returns a file timestamp in a machine-readable format (a 32-bit integer)
FileTimeGet	Returns a file timestamp in a human-readable format (date and time)
FileTimeSet	Sets the timestamp for one or more files
FileTimeTouch	Sets the timestamp for one or more files to the current time
FileYmdHms	Returns a file timestamp in the <i>ymdhms</i> date/time format
Miscellaneous	
FileVerInfo	Returns a version resource string (.EXE or .DLL files only)
FileMapName	Provides a new file name, such as a .BAK file, based on the original file name.

Binary File Operations

All of the file operations discussed so far involve text files (ASCII/ANSI standard files), but not all files are text-based. For example, neither image files nor sound (.WAV) files can be treated as text files. Also, applications often create files using specialized formats that contain nontext material. Collectively, nontext-based files of all types are referred to as *binary files*.

The binary file and binary data functions—beginning with `BinaryAlloc` and ranging through `BinaryIndex` and `BinaryRead` to end with `BinaryXor`—include 44 separate operations.

Be warned that binary operations require the program to carefully and correctly manage memory allocation and deallocation for all binary objects and operations. This single factor probably causes more problems than any other aspect of programming.

This caveat offered, here is a simple example of a binary file operation ([Binary.wbt](#)):

```
; This example edits the Config.sys file
```

```

; by adding a new line to the bottom of the file.

fs = FileSize("C:\CONFIG.SYS")
; Allocate a buffer the size of your file + 100 bytes.
binbuf = BinaryAlloc(fs+100)
If binbuf == 0
    Message("Error", "BinaryAlloc Failed")
Else
    ; Read the file into the buffer.
    BinaryRead(binbuf, "C:\CONFIG.SYS")
    ; Append a line to the end of the file in buffer.
    BinaryPokeStr(binbuf, fs, "DEVICE=C:\FLOOGLE.SYS%@crlf%")
    ; Write modified file back to the file from the buffer.
    BinaryWrite(binbuf, "C:\CONFIG.SYS")
    binbuf = BinaryFree(binbuf)
EndIf
Message("BinaryAlloc", "Done.")
exit

```

Further details on these functions can be found in the WIL Help files. Binary file operations are not an introductory topic and can easily require a book (or a major portion of one) of their own for coverage.

Summary

In this chapter, we've covered a variety of disk and file operations, beginning by showing you how to manage (and identify) the system drives and, most important, how to read and walk through directory information using a custom directory utility.

Then, after showing you something of the inside workings of directory and file access, you were introduced to the easy way: using the `AskFileName` function to display the Common File Dialog. And, no, this was not a sadistic approach, showing you the hard way first. The reasoning was that it can be very useful to understand how things work from the ground up before saying "hey, we also have an elevator."

Then, having shown you how to select directories and files from any application and after introducing a few of the more specialized directory functions, the next topic was getting extended information from drives including type, capacity, and free space.

The next topic introduced was file management. The most used functions in file access are reading and writing, which are demonstrated in the [PhoneList.wbt](#) program. We also discussed a wide variety of other file access and management functions.

Introduction to Programming

We concluded with a brief mention of the advanced binary file operations.

In [Chapter 11](#), we're going to move on to a new topic: building windows and managing application windows

CHAPTER 11 : WINDOWS AND GUI OPERATIONS

PAINTS, PENS, AND WINDOW BOXES

graphical user interface – Abbreviated GUI, pronounced "gooey." A graphics-based user interface that allows users to select files, programs, or commands by pointing to pictorial representations on the screen rather than typing long, complex commands from a command prompt.

window – In a graphical user interface, a rectangular portion of the screen that acts as a viewing area for application programs.

Categorizing a window as an element of a graphical user interface (GUI) is something of a misnomer, since it was perfectly possible (and practical) to use windows in a DOS (text) environment. Furthermore, before the advent of GUI-based operating systems such as Windows, many text applications used windows in much the same fashion as current applications: to present lists, directories, or other material for selection; to pop up messages; or to display graphical information in various forms.

For our present purposes, however, we will be using windows (also called *boxes*) in the Windows GUI environment. The WinBatch functions used to present a graphical interface are identified as `Box` functions.

The word windows (lowercase) should not be confused with Windows (uppercase), which is Microsoft's trademark for its version of a GUI. And, even though WinBatch is a language for Windows™, the term window remains operating-system independent and applies to any GUI operating system or even to a non-GUI, text-based display.

Creating a Window

Back in [Chapter 2](#), we introduced the basics of writing an application by creating a simple program titled [Hello World.wbt](#). Now, for our graphics window version, we'll create a new program with the title [Hello Windows.wbt](#).

We begin the window creation (after the color-value definitions, which are explained later in this chapter) with a set of window identifiers and one button identifier:

```
;=====
; Window identifiers
;=====
mainID = 1      ; requires IDs less than 9
drawID = 2
noteID = 3
```

Introduction to Programming

```
;=====
; Button identifiers
;=====
bExit  = 1
```

The window identifiers require values less than 9. There is also a limit of eight windows, including the top-level or main window that an application can open at any one time. The top-level window always has an identifier value of 1.

We could simply use the values rather than the identifiers, but the identifiers are much easier to remember.

The next step is to provide a generic initialization:

```
;=====
; Generic Initialization
;   allows windows to exit without warning (1)
;                                   + quiet termination (4)
;=====
IntControl( 12, 5, 0, 0, 0 )
```

The `IntControl` function is actually a large group of special-purpose *Internal Control* functions. The first parameter, which is 12 in this case, identifies the control to invoke. The subsequent parameters vary depending on the control used. In this case, the `IntControl` function is used to instruct WIL (or its parent application) how to handle termination of the program by the user. The second argument, 5, is actually a combination of two values (1+4), which include one Exit Windows Group code (1 allows Windows to be exited without warning) and one Terminate Group code (4 allows quiet termination). The `IntControl` functions are discussed further in [Chapter 13](#).

Following this minimal initialization, the next step is to create the top-level window:

```
;=====
; Creates the top-level Window
;=====
BoxesUp( "100, 100, 900, 900", @NORMAL )
TimeDelay(1)
```

The `BoxesUp` function is called with two parameters. The first argument is a list containing coordinates that set the window's position on the screen—left, top, right, and

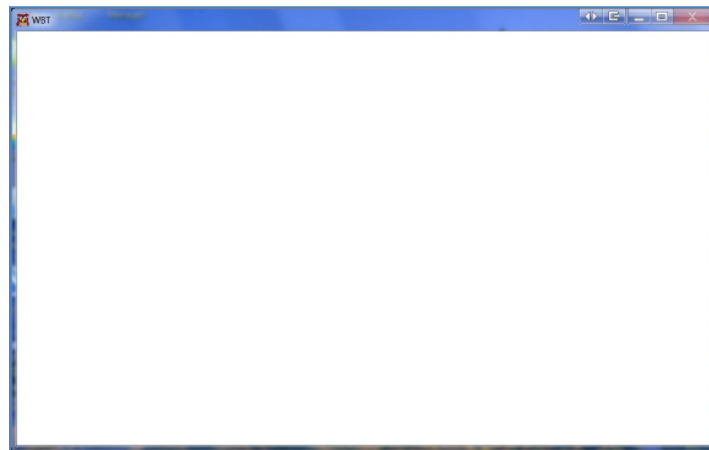
bottom—and establish the window's size (800x800). The second argument is the show mode, which can be one of the following:

@NORMAL	The window is displayed at the size and position specified.
@ICON	The window appears minimized (as an icon).
@ZOOMED	The window is maximized (zoomed) to the full-screen size.
@HIDDEN	The window does not appear on the screen even though the application continues to function.

Although we did not specify a window identifier, this initial, top-level window always has a window ID of 1, corresponding to the `mainID` defined at the beginning of the program.

💣 If you are running a system with a small display, such as 640x480, you'll want to change the coordinates in the [Hello Windows.wbt](#) program to fit your system's display.

The [Hello Windows.wbt](#) program uses a series of `TimeDelay` functions to pause execution after each step to allow you to see the results after each set of instructions. The initial window created by `BoxesUp` is shown below.



An initial window display

As you can see, this appears as a blank window, consisting of a frame (the window outline), the window title bar, and the window-control buttons. However, there is actually quite a lot of application provided by a single line of code. The window can be resized (either by using the minimize or maximize buttons or by dragging the frame), repositioned (by clicking and dragging the title bar), and terminated (using the close button).

To liven up our blank window, next we add a shaded background:

Introduction to Programming

```
BoxColor( mainID, BLACK, 7 ) ; third param sets shaded background
```

The `BoxColor` function's first parameter, `mainID`, is the window identifier. The second parameter, `BLACK`, sets the normal background color for the window. In this case, however, the third parameter, `7`, overrides the background color setting by specifying a wash, or gradient, background color.

The argument `BLACK` is one of the colors we have predefined as an RGB (red-green-blue) value. Color values and color specifications will be discussed in a moment.

The wash color specification can be any of eight values:

0	No wash (uses background color specified)
1	Red
2	Green
3	Yellow
4	Blue
5	Magenta
6	Cyan
7	White

Simply specifying a background color—whether wash or solid—is only part of the process of painting the window. Or, more accurately, setting a color does nothing to paint the window. The next requirement is to actually perform the paint operation using the `BoxDrawRect` function:

```
BoxDrawRect( mainID, "0, 0, 1000, 1000", 2 ) ; size in logical units
```

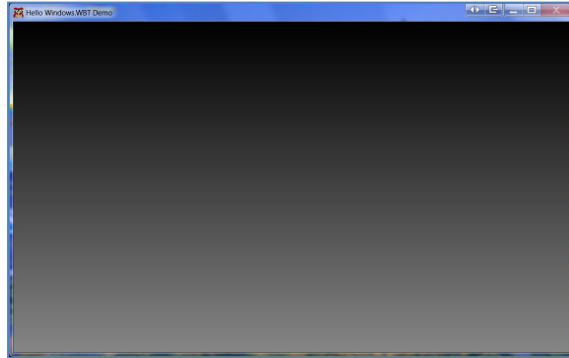
The `BoxDrawRect` function is called with three arguments: the window ID, the coordinates to paint, and a style parameter. The coordinate argument, `"0, 0, 1000, 1000"`, specifies the entire rectangle (window coordinates are discussed in the next section). The style argument `2` calls for a filled rectangle without a border. Four rectangle styles are supported:

0	Border only, unfilled
1	Filled rectangle with border
2	Filled rectangle without border

3	Transparent circle/rectangle with border.
---	---

If a border is selected, the border is drawn using the color specified by the `BoxPen` function.

Since a wash color setting was selected in the `BoxColor` function, the wash background runs from black at the top of the window to the color selected by the wash code at the bottom, as shown below.



Filling the window with a wash background

Colors and Color Codes

Colors, when working with colored lights such as CRT (cathode-ray tube) or LCD (liquid crystal display) panels generate, are composed of combinations of three primary colors: red, green, and blue. (Printed colors, which rely on dyes absorbing portions of the white light, produce colors by combining the complementary tints cyan, magenta, and yellow and adding black to produce darker hues—the CMYK color system.)

If we combine red and green in equal intensities, we produce yellow. Likewise, combining red and blue produces purple (magenta); combining blue and green produces cyan. Combining red, blue, and green gives us white. The absence of all three produces black. All other shades are created by varying the proportions of these three colors.

To specify a particular color using the RGB color system, the intensity of each component is identified as an eight-bit value in the range 0 through 255, creating an RGB color triplet. In WinBatch, these color specifications are written as comma-delimited strings with the form "`rrr,ggg,bbb`". In the [Hello Windows.wbt](#) program (and in the derived demo programs), a series of color specifications have been defined as:

```
;===== gray scale =====
BLACK      = " 0, 0, 0"
```

Introduction to Programming

```
DKGRAY   = " 64,  64,  64"
GRAY     = "128, 128, 128"
LTGRAY   = "192, 192, 192"
WHITE    = "255, 255, 255"
```

These first five color specifications range from black to white as a rudimentary gray-scale. Since the red, green, and blue levels for each are equal, each of these "colors" is simply an increasing level of white light.

The next six colors give us a dark palette:

```
;===== dark colors =====
DKBLUE   = "  0,   0, 128"
DKGREEN  = "  0, 160,   0"
DKRED    = "128,   0,   0"
DKCYAN   = "  0, 128, 128"
DKMAGENTA = "128,   0, 128"
BROWN    = "128, 128,   0"
```


And the final six colors offer a lighter palette:

```
;===== light colors =====
BLUE     = "  0,   0, 255"
GREEN    = "  0, 255,   0"
RED      = "255,   0,   0"
CYAN     = "  0, 255, 255"
MAGENTA  = "255,   0, 255"
YELLOW   = "255, 255,   0"
```

We will vary these definitions in later demos, offering a better brown, for example, by adding a little blue. In other demos, instead of relying on a predefined palette, we will define color specifications as they are used (but still as 24-bit RGB color triplets).

Window Coordinates

When the `BoxesUp` function is called to create the top-level window, or when `BoxNew` is called to create a child window, the new window created has a logical size, which is 1000x1000 units by default. This is **not** the screen size and does not affect the window size or position. Instead, this is the *virtual size inside the window frame*; that is, this 1000x1000 space is the graphical environment where drawing operations will occur, and all subsequent operations within the window will use this coordinate system.

 Operations within a window *do not use* desktop coordinates.

Within each window, the default origin point—the 0,0 coordinate—is at the upper-left corner of the area inside the window frame (not the frame or title bar). The 1000,1000 coordinate identifies the lower-right corner.

You can use the `BoxMapMode` function to switch between the default coordinate mapping system and a screen-based (display-size-based) system. Details on the `BoxMapMode` function can be found in the online documentation.

Labeling the Window

Once we've filled the window with a wash color, our next step is to supply a custom caption for the window using the `BoxCaption` function:

```
BoxCaption( mainID, "Hello Windows.wbt Demo" ) ; window caption
```

The `BoxCaption` function requires two parameters. As with most of the `Box` functions, the first argument is the window identifier, and the second is a string supplying the desired caption. And, if you'll look back at the windows in the first and second figures preceding, you'll see that the original (default) caption has been replaced as specified.

If you run the [Progress.wbt](#) demo, you'll see this caption change regularly to report more than just a title.

Windows within Windows

Even with a wash color background, our window is still pretty simple. Our next embellishments will be to create a box (window) within the main window, fill the box with a contrasting background, give it an outer and inner border for a three-dimensional (embossed) effect, and run a banner headline in the box.

The first step is to define the position and size of the banner box:

```
rectNote = "100, 100, 900, 340" ; set the size of the banner box
```

Remember that we're working in a logical display space—the client drawing area inside the parent window—which is 1000x1000 logical units in size, regardless of the window's actual screen size. Thus, the coordinates provided create a border of 100 logical units on the right, left, and top while making the window 240 units tall.

Introduction to Programming

This new window, however, will have its own logical coordinates and, within the window, will also be 1000x1000 logical units.

Next, having set the size, we call the `BoxNew` function to create the window:

```
BoxNew( noteID, rectNote, 1 ) ; create the box
```

To create the initial, top-level window, the `BoxesUp` function was called; however, to create child windows, the `BoxNew` function is needed. This function is called with a window identifier (`noteID`), the coordinates (`rectNote`), and a style parameter (1).

The child window will not have minimize, maximize, or close buttons but, depending on the style argument, may have a border and/or a caption bar. The accepted styles are:

0	Neither border nor caption
1	Border only
2	Both border and caption

In this case, even though the style parameter calls for a border, `BoxPen` has not been called to create a pen color and style, so no border will appear. But since we plan to draw a custom border for a 3-D style, it really doesn't matter at this point.

Before drawing the borders, we need to fill in the background:

```
BoxColor( noteID, LTGRAY, 0 ) ; background is Light Gray, no gradient
BoxDrawRect( noteID, "", 2 ) ; fill banner box with background color
```

This time, no wash (gradient) style was specified, so the `BoxDrawRect` function simply fills the box with light gray.

Then, after filling in the background, the next step is to set up to draw the outline around the entire box, beginning by defining a pen width and coordinates for the lines:

```
penWidthA = 20 ; note that all units are
line1A = " 0, 0, 1000, 0" ; logical units relative to
line2A = "1000, 1000, 1000, 0" ; the notebox which (by default)
line3A = " 0, 1000, 1000, 1000" ; has a logical size of
line4A = " 0, 0, 0, 1000" ; 1000 x 1000 units
```

Although these coordinates take the same form as rectangular coordinates, for a line, you should think of these as two x,y pairs identifying the beginning and end positions. Thus, `line1A` specifies a beginning position at 0,0 (left, top) and an end position at 1000,0 (right, top), forming a line along the top of the box (window). In like fashion, `line2A`

runs along the right side (from bottom to top), `line3A` runs along the bottom (from left to right), and `line4A` runs along the left (top to bottom).

Finally, after defining the coordinates, a white pen is selected to draw the top and left borders; then a gray pen is selected for the right and bottom:

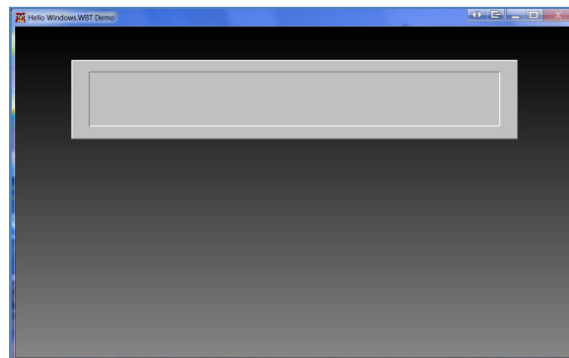
```
BoxPen( noteID, WHITE, penWidthA ) ; line color top and left
BoxDrawLine( noteID, line1A )      ; top
BoxDrawLine( noteID, line4A )      ; left
BoxPen( noteID, GRAY, penWidthA ) ; line color bottom and right
BoxDrawLine( noteID, line2A )      ; right
BoxDrawLine( noteID, line3A )      ; bottom
```

The inner outline is defined and drawn in the same fashion, except that the coordinates are moved in and the gray and white pens reversed:

```
penWidthB = 10
line1B = " 40, 150, 960, 150" ; top
line2B = " 960, 840, 960, 150" ; right
line3B = " 40, 840, 960, 840" ; bottom
line4B = " 40, 150, 40, 840" ; left

BoxPen( noteID, WHITE, penWidthB )
BoxDrawLine( noteID, line2B )      ; right
BoxDrawLine( noteID, line3B )      ; bottom
BoxPen( noteID, GRAY, penWidthB )
BoxDrawLine( noteID, line1B )      ; top
BoxDrawLine( noteID, line4B )      ; left
```

At this point, the result should look something like the image shown following.



Adding an embossed banner window

Displaying Text

The `BoxTextFont` function sets the font that will be used with the current window and controls all the text displayed in this window (but not other windows) until another font selection is made.

Now that we've embossed the window with two sets of borders, it's time to put a text message in the window. For this task, we start by defining a font height (font size) as 400 logical units, or four-tenths of the window height (1000 logical units). Then we call the `BoxTextFont` function:

```
noteHeight    = 400
BoxTextFont( noteID, "Arial", noteHeight, 170, 0 ) ; set headline font
```

The `BoxTextFont` function is called with the window identifier (`noteID`), the name of the typeface desired (`Arial`), the font height (relative to the window), a style flag, and, optionally, a pitch/family argument (since we've named the typeface).

The default font height is 100, or one-tenth of the window size.

Font Styles

The font style argument consists of three sets of values, which are added together to create a single argument. The possible values are:

0	Default	
1–99	Weight	
	40	Normal
	70	Bold
100	Italics	
1000	Underlined	

For example, the combined value 170 specifies **bold** (70) and *italic* (100) for a ***bold-italic*** font. In like fashion, a value of 1170 would produce ***bold-italic with underlining***.

Variations on weight beyond normal and bold may or may not be supported by your version of Windows. Experimentation is recommended.

Pitch and Family

When a typeface has been specified using the font parameter, the pitch and typeface family arguments are ignored and do not override the font selection. The pitch and family specifications can be OR'd (using the | operator) or simply added together. However, only one pitch and one family identifier should be used.

Pitch can be identified as:

0	Default
1	Fixed pitch (fixed-character width, such as Courier)
2	Variable pitch (most fonts, including Times-Roman, Arial, etc.)

The font family – the typeface group – can be selected from:

0	Default
16	Roman (Times-Roman, Century Schoolbook, etc.)
32	Swiss (Arial, Helvetica, Swiss, etc.)
48	Modern (Pica, Elite, Courier, etc.)
64	Script
80	Decorative (Old English, etc.)

The variety of fonts selectable using the pitch and family flags are demonstrated in the [Text Fonts.wbt](#) program, discussed later in this chapter.

To examine the fonts installed on your system, open the Control Panel and click on the Fonts folder, then click on any of the listed fonts to open a display showing a sample of the typeface. (Unfortunately, the font family and pitch are not identified.)

Displaying the Message

Now that a font (typeface) and size specification have been set, there are two minor tasks left before displaying the message: selecting a text color and setting a rectangle for the message display. To select a text color, `BoxTextColor` is called with one of the predefined color (RGB) specifications, and the selection becomes the default text color for the window. Then we call `BoxDrawText` to display the text:

```
rectNoteText = " 70, 200, 950, 800"
BoxTextColor( noteID, RED )
; creates the headline text - this line can be copied
; anywhere in the program where the headline needs to be changed
```

Introduction to Programming

```
BoxDrawText( noteID, rectNoteText, "Hello Windows", 1, 4 )
```

The `BoxDrawText` function is called with the window identifier, the bounding rectangle, the text string to display, and erase and alignment flags.

The bounding rectangle does not set boundaries for where the text should appear but does supply boundaries for alignment and for word wrap. If the font size is too large or if the text is too long, the displayed text may extend beyond the boundary rectangle but will not extend beyond the current window.

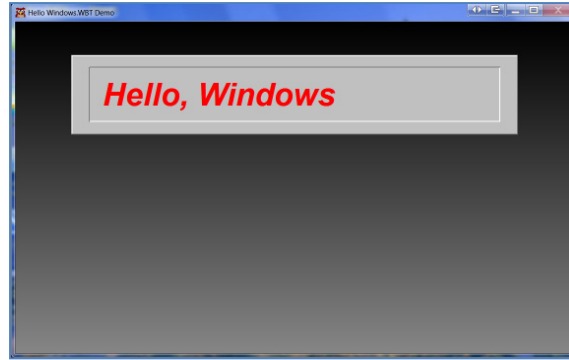
The erase flag can be either `@TRUE` (or 1) if the background should be cleared before writing the text or `@FALSE` (or 0) if any background information should be left untouched. In this example and in most cases, the erase flag is `@TRUE` so that everything within the bounding rectangle (except the background color) is erased before writing the new text. (In the [Text Fonts.wbt](#) demo, discussed later, the erase flag is set to `@FALSE` so that the background information remains as is while the new text is displayed.)

The alignment argument determines how the text is aligned relative to the bounding rectangle. The alignment argument is a bit flag, which means that the individual flags can be OR'd together (or added) to combine two or more settings. Alignment flag values can be:

0	Left justified
1	Centered horizontally
2	Right justified
4	Centered vertically within the bounding rectangle
8	Bottom justified (single line only)
16	Long lines wrapped (broken to fit the width of the bounding rectangle)
32	Font adjusted to fill the width of the boundary rectangle (single line only)
64	Right-justify text by adding space between words
128	Clip (truncate) text if it doesn't fit within specified rectangle

This `BoxDrawText` instruction can be copied anywhere in the program where the banner text needs to change. Because of the window identifier, new text can be entered without redefining the window, text color, font styles, or other information.

After displaying the banner information, our window looks like the one shown following.

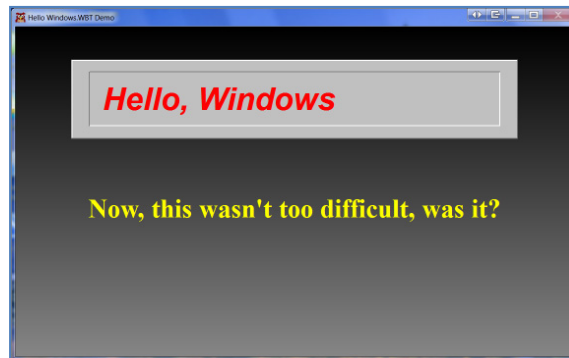


Displaying the banner text

Now that we've displayed the banner in its own box, we also want to display a message in the main window. We've set the font and color information for the child window (identified as `noteID`). Now we need to do the same for the main window (`mainID`):

```
BoxTextFont( mainID, "Times", 80, 80, 0 | 0 ) ; initial font
BoxTextColor( mainID, YELLOW ) ; initial font color
BoxDrawText( mainID, "10, 500, 990, 600", "Now, this wasn't too
difficult, was it?", 0, 1 | 4 )
```

Here, the bounding rectangle is set across the center of the main window, and the horizontal and vertical centering flags are specified so that the displayed message will be centered in the main window, as shown below.



Adding a message in the main window

Adding Buttons

Before we move on, there's one more feature that we want to add to this window: an Exit button. Adding a button is quite easy. All that's required is a call to the `BoxButtonDraw` function:

Introduction to Programming

```
BoxButtonDraw( mainID, bExit, "E&xit", "750, 820, 900, 890" )
```

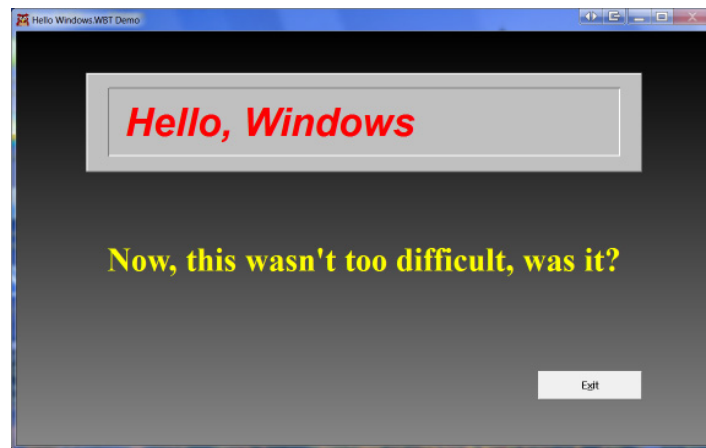
Here, the first parameter, `mainID`, identifies the window where the button will appear. The final parameter, as rectangular coordinates, provides the button's position and size. The third argument provides the button's text. Including the ampersand (&) makes the following character the hotkey for the button. In this example, pressing `Ctrl+X` will perform the same function as clicking on the button.

The second argument, `bExit`, requires a bit of explanation. The second argument is the button identifier; when more than one button is displayed, this identifies which button was pressed. In this example, `bExit` was defined with a value of 1. In later examples, we'll have several buttons. It is best to ensure that groups of buttons begin with 1 and that they are numbered sequentially.

Since we only have only one button in this example, all that we need to do is call `BoxButtonWait`, which simply waits for a button to be pressed. Then we can exit.

```
BoxButtonWait()  
exit
```

Finished, the [Hello Windows.wbt](#) program appears thus:



Adding an Exit button

A Quick Review

While this concludes our discussion of the [Hello Windows.wbt](#) program, we still have a variety of other `BOX` functions to discuss. Thus far, you have seen how to create a top-level window (box) and a child window, choose colors, draw simple lines, create a simple text message, and add a button to the window.

These are, of course, all important basics, but these are also only a few of the operations we can do with windows. Next, we'll take another look at colors. Then we'll explore

various drawing and mouse operations before taking a further shot at what we can do with fonts.

More About Colors

In the [Hello Windows.wbt](#) demo, a palette of colors was defined, although these colors were not really used. Now, in the [Colors.wbt](#) demo, we'll modify this original palette slightly (to improve the brown entry) and display these as color blocks. At the same time, we'll use multiple window buttons to select different sets of colors for display.

For another example of multiple buttons, see the [Buttons.wbt](#) program.

As a first step, we'll redefine the palette entries as `COLORnn` and we'll change the brown color definition from `128,128,0` (a dark gold) to `128,96,48`, which produces something closer to a realistic brown. The complete palette definition appears as:

```

;===== gray scale =====
;          -R-  -G-  -B-
COLOR1  = "  0,   0,   0" ; Black
COLOR2  = " 64,  64,  64" ; Dark Gray
COLOR3  = "128, 128, 128" ; Gray
COLOR4  = "192, 192, 192" ; Light Gray
COLOR5  = "236, 236, 236" ; Off-White
COLOR6  = "255, 255, 255" ; White

;===== dark colors =====
;          -R-  -G-  -B-
COLOR7  = "  0,   0, 128" ; Dark Blue
COLOR8  = "  0, 160,   0" ; Dark Green
COLOR9  = "128,   0,   0" ; Dark Red
COLOR10 = "  0, 128, 128" ; Dark Cyan
COLOR11 = "128,   0, 128" ; Dark Magenta
COLOR12 = "128,  96,  48" ; Brown

;===== light colors =====
;          -R-  -G-  -B-
COLOR13 = "  0,   0, 255" ; Blue
COLOR14 = "  0, 255,   0" ; Green
COLOR15 = "255,   0,   0" ; Red

```

Introduction to Programming

```
COLOR16 = " 0, 255, 255" ; Cyan
COLOR17 = "255, 0, 255" ; Magenta
COLOR18 = "255, 255, 0" ; Yellow
```

These are grouped at gray-scale, dark colors, and light colors because the intent is to display six of these at a time.

We'll also create a list with the names of the colors:

```
listColors = "Black,Dark Gray,Gray,Light Gray,Off-White,White,Dark
Blue,Dark Green,Dark Red,Dark Cyan,Dark
Magenta,Brown,Blue,Green,Red,Cyan,Magenta,Yellow"
```

We also need a window identifier (for convenience) and six rectangle areas to paint:

```
mainID = 1 ; requires IDs less than 9
rectColor1 = " 50, 100, 330, 400"
rectColor2 = "360, 100, 640, 400"
rectColor3 = "670, 100, 950, 400"
rectColor4 = " 50, 450, 330, 750"
rectColor5 = "360, 450, 640, 750"
rectColor6 = "670, 450, 950, 750"
```

The color rectangles are defined as two rows of three positioned in the top portion of the window.

Last, we want four identifiers for the buttons: three to select palettes and one for the Exit button.

```
bExit = 1
bGrays = 2
bDark = 3
bLight = 4
```

In theory, we could assign any button identifiers desired, but a set of sequential values is much easier to handle and, in some cases, appears to function more cleanly.

With these provisions, we're ready to create the window and start operations, thus:

```
IntControl( 12, 5, 0, 0, 0 )
BoxesUp( "100, 100, 900, 900", @NORMAL )
nOfs = 0
```

```

While @TRUE
    BoxDataClear( mainID, "TOP")
    BoxColor( mainID, COLOR14, 4 ) ; third param sets shaded background
    BoxCaption( mainID, "Colors.wbt Demo" ) ; window caption
    BoxDrawRect( mainID, " 0, 0, 1000, 1000", 2 ) ; size logical units

```

This part of the code is essentially the same as what was demonstrated in the [Hello Windows.wbt](#) program. Here, however, instead of a single Exit button, we're going to create four buttons:

```

BoxButtonDraw( mainID, bGrays, "Gray Scale", "100, 820, 250, 890" )
BoxButtonDraw( mainID, bDark, "Dark Colors", "275, 820, 425, 890" )
BoxButtonDraw( mainID, bLight, "Light Colors", "450, 820, 600, 890" )
BoxButtonDraw( mainID, bExit, "E&xit", "750, 820, 900, 890" )

```

The buttons are positioned across the bottom of the window.

The next step is to select a font (to label the color blocks) and to decide what text color will show best. Since the `nOfs` variable is used to select which group of colors will be displayed, we can also use this to select the appropriate text color:

```

If nOfs == 12 then textColor = 1
If nOfs == 6 then textColor = 6
BoxTextFont( mainID, "Times", 40, 40, 0 | 0 ) ; initial font info
For i = 1 to 6
    nColor = i + nOfs ; box color
    If nOfs == 0 then textColor = 7 - i ; text color for gray scale
    label = ItemExtract( nColor, listColors, "," )
    label = StrCat( @CRLF, " ", label, @CRLF, @TAB, COLOR%nColor% )

    BoxColor( mainID, COLOR%nColor%, 0 ) ; background color
    BoxPen( mainID, COLOR%textColor%, 5 ) ; outline color
    BoxDrawRect( mainID, rectColor%i%, 1 ) ; fill banner box with
background color
    BoxTextColor( mainID, COLOR%textColor% )
    BoxDrawText( mainID, rectColor%i%, label, 0, 0 )
Next

```

If the color palette displayed will be the dark colors, the white palette entry (`COLOR6`) provides the best contrast. Likewise, for the light colors, the black palette entry (`COLOR1`) is optimum.

Introduction to Programming

For the gray-scale colors, however, neither white nor black will show against all of the colors, so our choice is to use the inverse of the background color as the text color. This produces white text against a black background, off-white against dark gray, and so forth until we reach black against a white background. Since this requires the text color to change for each rectangle, this part is done within the same `for` loop used to draw the color rectangles. Also, we want to list the appropriate text label and to show the RGB color specification for each.

After setting up the caption information, drawing the color rectangle with an outline in the same color as is used for the text and writing the caption are fairly simple tasks:

Finally, having drawn the rectangles to show the colors, it is time to wait for a button to be pressed (clicked):

```
iBox = 0
BoxButtonWait()
```

While we can use the `BoxButtonWait` function to wait for a button event, we also need a means to decide which button was pressed. Once a button has been pressed, we poll the various buttons to inquire which button generated the event:

```
While iBox == 0
    For x = 1 to 4 ; sequential buttons required
        if BoxButtonStat( mainID, x ) then iBox = x
    Next
EndWhile
```

The use of a `for` loop to poll the buttons imposes the need for the button identifiers to be sequential. If the buttons queried in the routine preceding are not sequentially numbered, an error will occur when the `BoxButtonStat` function is called with a value of `x` that does not have a corresponding button defined.

Next, assuming that `iBox` actually identifies one of the buttons, a simple `switch` statement gives us the appropriate response for each:

```
If iBox
    Switch iBox
        case bExit
            exit
        break
```

If the Exit button is pressed, then the program simply exits:

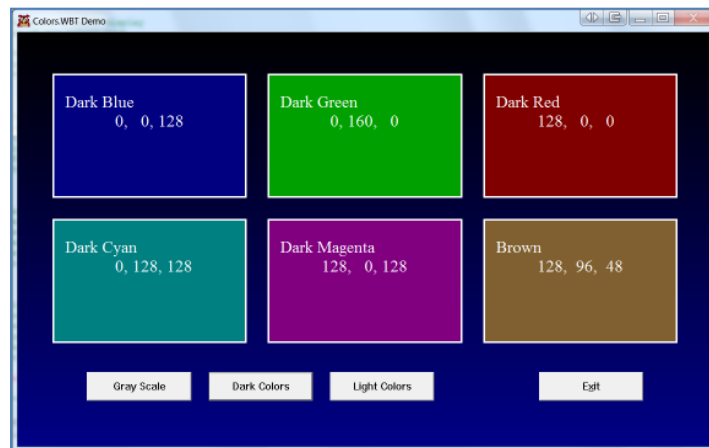

```

    case bGrays
        nOfs = 0
        break
    case bDark
        nOfs = 6
        break
    case bLight
        nOfs = 12
        break
    EndSwitch
Endif
EndWhile
exit

```

Pressing any of the other three buttons sets the `nOfs` variable as appropriate for the selected set of colors.

The figure below shows the [Colors.wbt](#) demo with the dark color palette selected.



Displaying a set of colors

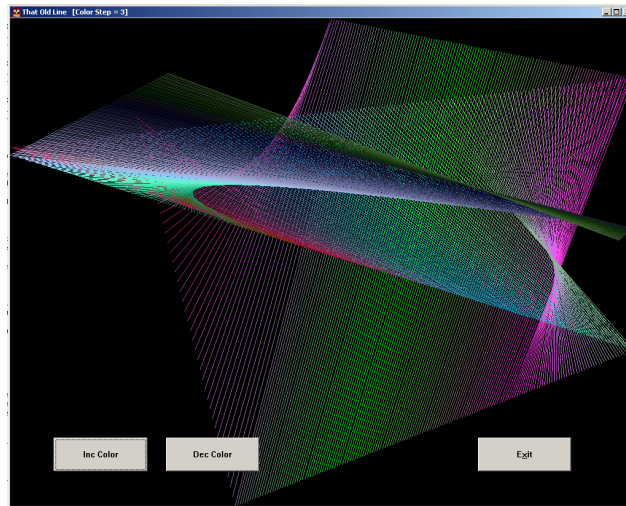
Drawing in a Window

The [Colors.wbt](#) program shows a set of predefined colors. However, most modern video cards (and most contemporary systems) support sufficiently high color resolutions that we can freely assign any color values desired with a reasonable expectation of the color being correctly rendered for display.

When an application calls for a color that is not supported by the system (by the video card and the selected resolution), the system uses the RGB color value to map the request to the nearest supported color.

Introduction to Programming

In the [Lines.wbt](#) demo, RGB color specifications are generated on-the-fly to produce color fans sweeping across the screen (see figure following). The individual lines are drawn between start and end points, which are incremented for each step, reversing direction when a point reaches the edge of the window. At the same time, the color value is also changed incrementally for each step to produce lines that differ from their neighbors but only slightly.



Drawing color fans

The keys to the color display are found in three subroutines. First, the `:RANDOM_COLOR` subroutine is used to initialize the color used before falling through to the `:CHECK_COLOR` subroutine:

```
:RANDOM_COLOR
    rVal = Int( Random(255) )
    gVal = Int( Random(255) )
    bVal = Int( Random(255) )
```

As you can see, the `:RANDOM_COLOR` subroutine simply generates an initial RGB value (as `rVal`, `gVal`, and `bVal`).

The `:CHECK_COLOR` routine, which follows, modifies the initial RGB value by setting the lowest of the three values to 0, setting the highest to 255, and leaving the median value unchanged. (Functionally, `:CHECK_COLOR` is simply part of the `:RANDOM_COLOR` subroutine rather than a separate subroutine.)

```
:CHECK_COLOR
    q = Min( rVal, gVal, bVal )
    If q == rVal then rVal = 0
    If q == gVal then gVal = 0
    If q == bVal then bVal = 0
```

```

q = Max( rVal, gVal, bVal )
If q == rVal then rVal = 255
If q == gVal then gVal = 255
If q == bVal then bVal = 255
return

```

Functionally, this provision sets the RGB value to something approximating a primary or complementary hue and avoiding blacks or grays. This is only the initial color setting, however. We've also set three step values—dRed, dGreen, and dBlue—which are used in the :STEP_COLOR subroutine to adjust the color for each successive line:

```

:STEP_COLOR
  If( rVal >= 245 ) then dRed = -nColorStep
  If( rVal <= 10 )  then dRed = nColorStep
  rVal = rVal + dRed

  If( gVal >= 245 ) then dGreen = -nColorStep
  If( gVal <= 10 )  then dGreen = nColorStep
  gVal = gVal + dGreen

  If( bVal >= 245 ) then dBlue = -nColorStep
  If( bVal <= 10 )  then dBlue = nColorStep
  bVal = bVal + dBlue
return

```

The Inc Color and Dec Color buttons are used to increment and decrement nColorStep, changing the rate at which the colors shift.

The :STEP_COLOR subroutine performs two tasks; in addition to incrementing or decrementing each of the RGB values, the subroutine also tests to ensure that the RGB values remain within the 0 to 255 range.

The third color-manipulation routine, :RANDOM_HUE, is called periodically to adjust how colors change by reversing the step increment for one of the three color values, while leaving the other two unchanged:

```

:RANDOM_HUE
  switch Random(3)
  case 1
    If( dRed == nColorStep )

```

Introduction to Programming

```
        dRed = -nColorStep
    else
        dRed = nColorStep
    EndIf
    break
case 2
    If( dGreen == nColorStep )
        dGreen = -nColorStep
    else
        dGreen = nColorStep
    EndIf
    break
case 3
    If( dBlue == nColorStep )
        dBlue = -nColorStep
    else
        dBlue = nColorStep
    EndIf
    break
Endswitch
return
```

These three subroutines provide shifting colors without the need of a predefined palette. Similar routines can be used in any application.

Feel free to experiment with the color routines in the [Lines.wbt](#) demo and observe the effects on the display.

An Alternative to BoxButtonWait

In the demos we've discussed so far, the `BoxButtonWait` function is used to pause until a button is clicked. In the [Lines.wbt](#) demo, we do not want to wait for a button event; we want other things happening in the intervening time. Therefore, a different approach is required.

In [Lines.wbt](#), a `while` loop provides the heart of the operation. Within the `while` loop, a series of tests are repeated each time around to query the button status individually.

```
While @TRUE
    If BoxButtonStat( drawID, bExit ) == 1 then
```

```

...
    break
EndIf

If BoxButtonStat( drawID, bColorUp ) == 1 then
    ...
EndIf

If BoxButtonStat( drawID, bColorDn ) == 1 then
    ...
EndIf

BoxDataClear( drawID, "NULL" )

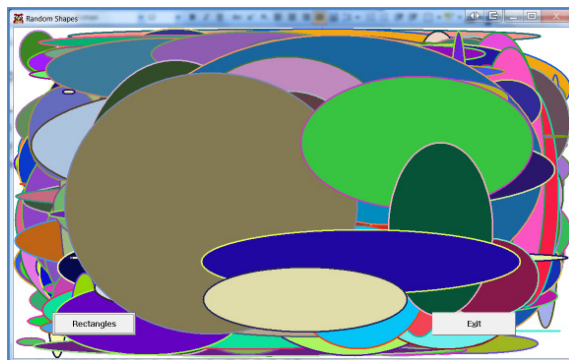
...
EndWhile

```

As a collateral benefit, since a `for` loop is not being used to query the button status, this part of the code is not going to complain if the buttons are not numbered sequentially.

The BoxDrawCircle and BoxDrawRect Functions

In addition to the `BoxDrawLine` function, demonstrated in the [Lines.wbt](#) program and other window examples, WinBatch also offers the `BoxDrawCircle` and `BoxDrawRect` functions. These are demonstrated in the [Shapes.wbt](#) program as shown here.



Drawing shapes

The [Shapes.wbt](#) program uses many of the same routines discussed in the previous demo examples, including random color settings for each shape. The outline color for each

Introduction to Programming

shape (set using `BoxPen`) is simply the inverted RGB value used for the fill color, an easy way of providing some contrast (if not always aesthetically pleasing).

In common with many other programming languages, both the circle and rectangle shapes are defined by corner coordinates. Thus, `BoxDrawRect` is called as:

```
BoxDrawRect( drawID, "%boxLf%,%boxTop%,%boxRt%,%boxBtm%", 1 )
```

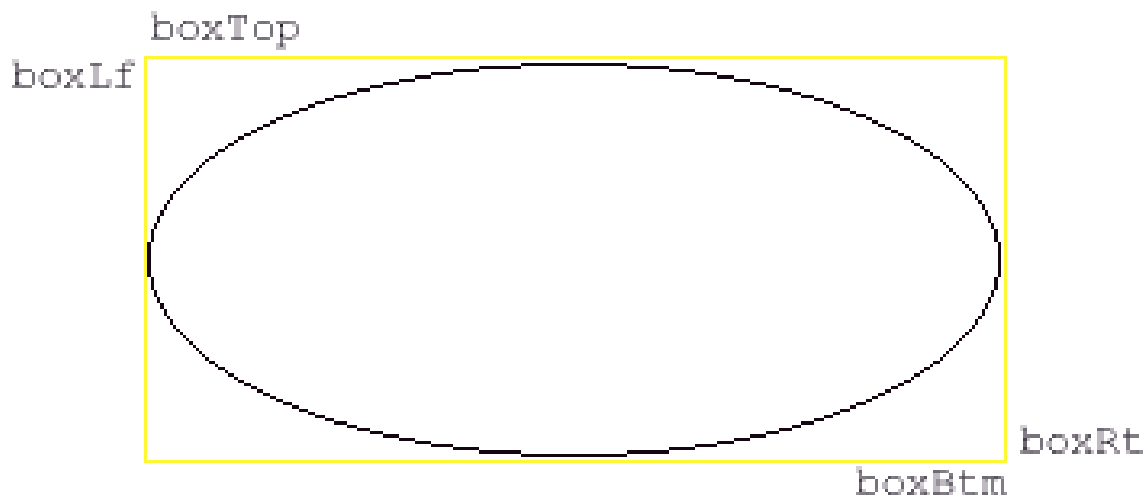
The `boxLf` and `boxTop` values set the upper-left corner, and the `boxRt` and `boxBtm` values set the lower-right corner position.

The `BoxDrawCircle` function is called in the same fashion, as:

```
BoxDrawCircle( drawID, "%boxLf%,%boxTop%,%boxRt%,%boxBtm%", 1 )
```

The difference is that this time a circle, or an oval ellipse, is drawn inside the bounding rectangle (but, of course, without drawing the rectangle).

An example of a bounding rectangle for an ellipse is shown below:



Both `BoxDrawCircle` and `BoxDrawRect` accept a style parameter, which can be:

0	Empty (no fill) with a border
1	Filled with a border
2	Filled with no border
3	Transparent with a border

Defining shapes by corner coordinates

It might seem that defining shapes by corner coordinates is rather unimaginative, however corner coordinates are easily calculated and easily controlled. Some early graphics libraries did use more flexible arrangements – including the ability to define circles and ellipses in terms of loci and radii, allowing the major and minor axes to lie along any angle – but these drawing functions proved to be too complicated for most purposes (and many programmers). As a result, the reliance on simple rectangular frames with x/y orientations has become the *de facto* standard even though packages such as CAD/CAM and some of the more elaborate drawing programs do continue to support more flexible arrangements.

Drawing Stack Management

A requirement in standard Windows applications is for the program to be prepared and ready to redraw the entire window or any portion of the window at any time. This requirement demands a certain structure in programming conventional applications and adds a degree of complexity for the programmer.

In WinBatch, however, the programmer is shielded from the details and demands of a dynamic redraw operation. Toward this end, WinBatch maintains a small database of the `Box` commands requested by the application and refers to this list when Windows requests a portion of the window to be redrawn. As a general rule, this data remains transparent to the programmer and does not require any special provisions.

In some cases, however, the database of instructions must be actively managed by the programmer to ensure that the application does not exceed the maximum limits of the database. If these limits are reached, the application will terminate with a "Box Stack Exceeded" error.

Each window belonging to an application has a separate stack and maintains a separate list of drawing commands.

The stack limit is around 150 commands, but asking the program (and the programmer) to keep track of how many commands were in the stack would be an onerous and unnecessary requirement. Instead, rather than simply allowing an unexpected termination to occur, there are provisions to clear the stack when the database becomes too full. This can be done using the `BoxDataTag` and `BoxDataClear` functions along with the `LastError` function. While it is possible to simply dump the entire stack, we may not want to clear the entire database; instead, we may want to reserve some operations (and objects) for an automatic redraw operation.

Partial Clearing

In practice, we can usually assume that there are objects—controls or whatever—that we do not want to change, while others are changing constantly. The ideal is to draw the

Introduction to Programming

fixed, unchanging objects first and then to place a tag, using `BoxDataTag`, in the data stack:

```
BoxDataTag( WindowID, "NULL" )
```

The name used for the tag— "NULL" in this case—can be anything you like. Acceptable names include "George", "Oranges", "lemons", and "tag123". (The tag "TOP" is automatically placed at the top of the data stack and can be used at any time to clear the entire stack.) The `windowID`, of course, must identify the window where the drawing operations are taking place.



While multiple tags are permitted, they are not advised.

Next, after assigning the tag, the application should proceed to draw the changing objects.

The `BoxUpdates` function is called with a window identifier and an update flag. Supported update flags are:

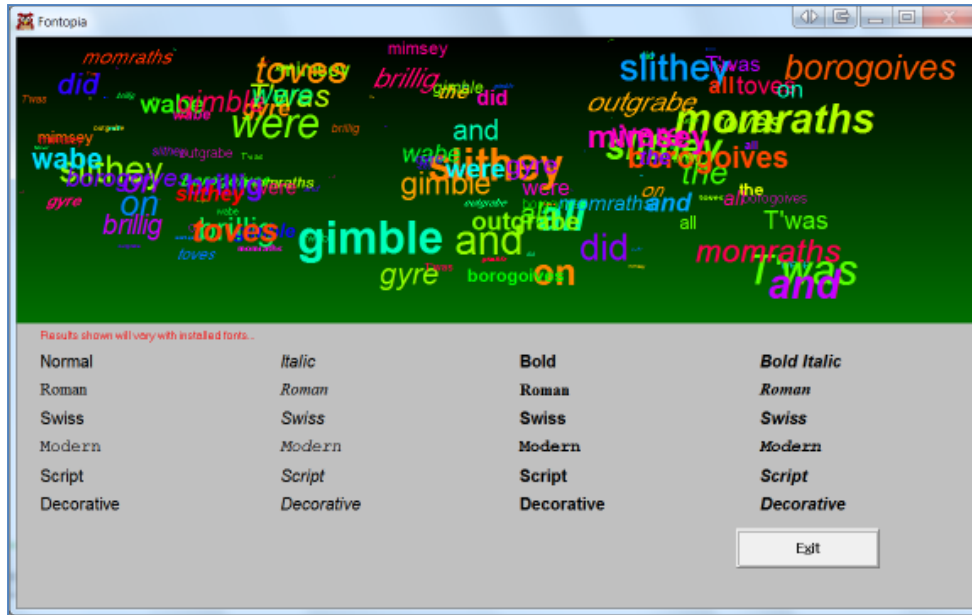
0	Suppress screen updates
1	Enable updates (default setting)
2	Catch up on updates
3	Redraw the entire box

The `BoxUpdates` function is rarely required.

When the stack containing previous drawing instructions is emptied, this does not mean that the existing screen image is erased, only that these instructions will not be repeated if a screen redraw operation is required.

Formatting Text In Windows

The sample program [Text Fonts.wbt](#), shown in below, demonstrates how fonts and typeface families are selected. In the top pane of the window, words from a list (courtesy of Lewis Carroll) are displayed in a variety of fonts, colors, and sizes. In the lower pane, five font families are shown in normal, italic, bold, and bold italic.



The Text Fonts.wbt demo

In this instance, we don't appear to have appropriate typefaces for the Script or Decorative font families

Notice that the Decorative row in the Normal and Italic columns shows a quite different typeface than what you see in the Bold and Bold Italic columns. This occurs because, instead of requesting a specific typeface, the font selection specifies a font family with particular characteristics. In the Decorative category, one typeface was more appropriate for normal weight, while a second typeface was more appropriate for a bold font.

Displaying fancy fonts is interesting and attractive, but we also want to be able to do something more with fonts (or, at least, with text) than create a colorful display. To demonstrate a useful application, we'll improve on the [ShowList.wbt](#) program presented in [Chapter 10](#). The [ShowList.wbt](#) reads the contents of the [Phone.Lst](#) file and displays the contents in a list box. This display, however, is rather ragged—there was no ready method of aligning the material in columns and the results were less than aesthetically appealing.

In contrast, using the `BoxDrawText` function, we have the option of placing material on the screen however we like. The [PhoneListBox.wbt](#) program, shown in following, demonstrates not only how text can be arranged neatly, but also how the mouse operations (discussed earlier) can be used to select an item from a formatted list.



A formatted display with selection

Here, the selected item is shown by changing the display color for the item. Also, while WinBatch does not provide a generic means of detecting a double-click event, the [PhoneListBox.wbt](#) program watches for multiple mouse clicks within a short interval, using a repeated mouse button event as the signal to make a selection from the list.

Thus, the first mouse click is taken as a position and the logical coordinates reported are used, by matching the vertical position against the list display positions, to determine which item in the list was selected.

When the list was created, each entry was positioned within specific vertical coordinates, as shown in the following fragment from the :DISPLAYLIST subroutine:

```
For i = 1 to ItemCount( listPhone, sDelimiter )
    If i == nSelect
        BoxTextColor( drawID, GREEN )
    Else
        BoxTextColor( drawID, WHITE )
    EndIf

    sTemp      = ItemExtract( i, listPhone, sDelimiter )
    sName      = ItemExtract( 1, sTemp, @TAB )
    sPhone     = ItemExtract( 7, sTemp, @TAB )
    yTop = Int( nRowHeight * ( i - 1 ) )
    yBottom = yTop + nFontHeight
    BoxDrawText( drawID, "10, %yTop%, 490, %yBottom%", sName, 0, 0 )
    BoxDrawText( drawID, "500, %yTop%, 990, %yBottom%", sPhone, 0, 0 )
Next
```

Next, when a mouse button is clicked, the mouse position is retrieved, and then the vertical position is extracted from the mouse position:

```
sPosition = MouseInfo( 6 ) ; get position
nRow = ItemExtract( 2, sPosition, " " )
```

```
nSelect = Int( nRow / nRowHeight ) + 1
```

Finally, the `nRowHeight` information is converted back to a row count to tell us which entry was selected from the list.

Then, once a match has been calculated, the list is redrawn to highlight the selected item before the program waits for a repeated mouse click to confirm the selection and display the results.

This example is rather rough and dirty, and several aspects could be improved. For example, when waiting for a second mouse click, the new position should be checked against the original to find out if the mouse has moved to select a different item. Also, while the application is simply recording mouse events, there is no assurance that the mouse events actually occurred within the application window. These changes, as well as a variety of other possible improvements, are left as an exercise for the reader.

For more examples of graphics operations, take a look at the Box Drawing Demo.wbt, provided with the WinBatch install in the [Samples] subdirectory.

Finally, don't be afraid to poke around in the code and make changes—it's the only way you'll really learn to program. And it's not like a mistake is going to have flames coming out of your computer ... Well, maybe from your monitor ... if you're really good, that is.

Summary

In this chapter, we showed you how to use the WinBatch `Box` functions to create windowed applications and windows within applications, how to use colors, how to draw objects, and how to present text information in a variety of fonts.

Since this is an introductory programming book, we've covered all of the basics but have not explored any of these topics in great depth. Still, with the information provided, along with the online documentation and a little imagination, you should be able to create virtually any program desired.

CHAPTER 12 : MOUSING AROUND

GETTING AWAY FROM THE KEYBOARD

mouse (computer) – a pointing device which functions by detecting and reporting two-dimensional movements relative to its supporting surface or movements of a trackball without surface movements. The mouse's motion is normally used to control the motion of a cursor on a display, providing fine control – and an intuitive input – for a graphical user interface.

mouse movements – with the technical reverence typical of programmers (and other geeks) the movement of a mouse has always been reported in mickies, with the mouse resolution (how fine a movement can be recognized) referred to as mickies-per-inch. The speed of movement, of course, is mickies-per-second.

Mouse Operations in Windows

As in most Windows applications, WinBatch programs handle most of the common mouse operations without any special provisions on the part of the programmer. For example, you don't need to track the mouse position and watch for a mouse button click to then determine where the event occurred and whether this event involved clicking on a button object.

In other circumstances where the application does need to track the mouse, WinBatch supplies six `Mouse` functions: `MouseClick`, `MouseClickBtn`, `MouseCoords`, `MouseInfo`, `MouseMove` and `MousePlay`.

Forcing Mouse Operations

Of the six `Mouse` functions supplied, four allow WinBatch applications to make the mouse appear to perform an action.

The `MouseMove` function moves the mouse pointer to a specified position within a window. `MouseMove` is called as:

```
MouseMove(xPos, yPos, parent_window_name, child_window_name)
```

The parent and child window names are optional arguments. If the parent window name is specified, the `xPos` and `yPos` coordinates are treated as logical coordinates relative to the named window and based on a virtual 1000x1000 screen. Likewise, if the child window name is specified, the coordinates are relative to this named window. If both the child and parent window arguments are blank strings (""), `xPos` and `yPos` are treated as the position in desktop (absolute screen) coordinates.

Introduction to Programming

The `MouseClicked` function simply generates a mouse-button-click message at the current mouse position. The `MouseClicked` function is called with two arguments as:

```
MouseClicked( click_type, modifier )
```

The click type argument can be:

@LCLICK	Left mouse button click
@RCLICK	Right mouse button click
@MCLICK	Middle mouse button click
@LDBLCLICK	Left mouse button double-click
@RDBLCLICK	Right mouse button double-click
@MDBLCLICK	Middle mouse button double-click

The modifier argument is optional (set to 0 if not needed) but allows key combinations to be added to the mouse button click. Modifier values are:

@SHIFT	Shift key held down
@CTRL	Ctrl key held down
@LBUTTON	Left mouse button held down
@RBUTTON	Right mouse button held down
@MBUTTON	Middle mouse button held down

The `MouseClickedBtn` function permits emulating a mouse click on a pushbutton, radiobutton, or checkbox. The `MouseClickedBtn` function is called as:

```
MouseClickedBtn( parent_window_name, child_window_name, button_text )
```

The `button_text` argument identifies the control by the text displayed on the control. If the control or button is located in a top-level window, only the parent window name is required, and the child window name is passed as an empty string ("").

Finally, the `MousePlay` function performs full range of mouse associated activities. You can simulate drag-and-drop operations by specifying the drop location in the "X-Y-coordinates" parameter and adding the `@MPLAYLBUTTON` constant to the "buttons" parameter. This tells `MousePlay` to move the mouse to the position x,y with the left mouse button down.

You can perform a mouse button click at a specific location by using one of the button click values in the "buttons" parameter.

`MousePlay` can move the mouse cursor relative to the upper left-hand corner of a window. To do this simply place a window name in the "parent window" parameter and optionally in the "child window" parameter. If you give `MousePlay` a child window name you must give it a parent name as well. When window names are present, `MousePlay` considers the upper left-hand corner of the parent or child window to be 0,0. It is, therefore, possible to give it a negative x or y value to move the mouse cursor to the left or above the window. If the window you specify is minimized, `MousePlay` will use its last un-minimized size for calculating mouse position.

`MousePlay` also accepts the WIL constants `@SHIFT` and `@CTRL` in the "buttons" parameter. You can combine these constants with a mouse button constant using the bitwise OR (`|`) operator to duplicate holding down the Shift or Control key while performing a mouse drag-and-drop or button click.

You can use the "delay" parameter to control the amount of time `MousePlay` takes to perform an action. Sometimes it is necessary to slow down the mouse so that Windows will properly recognize the action. You may also want to slow things down to better track events or to just give your mouse activity a natural appearance. This parameter expects values in seconds, and only recognizes the first three digits to the right of the decimal point.

Tracking the Mouse

The remaining mouse operations functions, `MouseCoords` and `MouseInfo`, are probably the most useful. These functions allow you to track where the mouse is and to discover the actual status of the mouse buttons.

The `MouseCoords` function is called as:

```
MouseCoords( parent-windowname, child-windowname)
```

The `MouseCoords` function returns coordinates of the mouse within a window. If "parent-windowname" specifies a top-level window and "child-windowname" is a blank string, the specified X-Y coordinates are relative to "parent-windowname".

If "parent-windowname" specifies a top-level window and "child-windowname" specifies a child window of "parent-windowname", the specified X-Y coordinates are relative to "child-windowname".

If "parent-windowname" and "child-windowname" are both blank strings, the specified X-Y coordinates are relative to the Windows desktop.

All coordinates are based on a virtual 1000 x 1000 screen.

The `MouseInfo` function is called as:

Introduction to Programming

`MouseInfo(request_number)`

The `MouseInfo` function returns a string containing the information identified by the `request_number` argument. Request number values are:

0	Returns the name of the window where the mouse is positioned
1	Returns the top-level (parent) window where the mouse is positioned
2	Returns the mouse position in logical coordinates (based on a 1000x1000 virtual screen)
3	Returns the mouse position in absolute screen coordinates (pixels)
4	Returns the mouse button status reported as a bitmask value. See table below.
5	Returns the mouse position relative to the client area of the window where the mouse is positioned, reported in virtual screen units (based on a 1000x1000 virtual screen)
6	Returns the mouse position relative to the client area of the window where the mouse is positioned, reported in virtual client units (based on a 1000x1000 virtual screen)
7	Mouse coordinates relative to the bounding rectangle of the window under the cursor, in virtual (1000x1000) screen units.
8	Synchronous status of mouse buttons, as a bitmask. This is like request #4, except #8 returns the asynchronous (current) state of the buttons, whereas #4 returns the state at the time the function was called.
9	Window ID of top level parent window under mouse.
10	Similar to request 0 except that the request can also return the window name of static, hidden and disabled child windows that have a caption. Examples of static windows include WIL Dialog VARYTEXT, STATICTEXT and PICTURE controls.

The mouse button status is reported as a bitmask value, which may contain a combination of individual flags. The bitmask values are shown following.

Mouse Button Status Bitmask Values

Binary	Decimal	Comments
000	0	No mouse buttons down

001	1	Right mouse button down
010	2	Middle mouse button down
011	3	Right and middle buttons down
100	4	Left mouse button down
101	5	Left and right buttons down
110	6	Left and middle buttons down
111	7	Left, middle, and right buttons down

The operations of the `MouseInfo` function are demonstrated in the `:DO_DRAW` subprocedure in the [Freehand.wbt](#) program. The [Freehand.wbt](#) program is a very simple drawing program that responds to the right and left mouse buttons. Holding the left (primary) mouse button down draws a line as you move the mouse.

The right (secondary) mouse button is a little more sophisticated. The first time the right mouse button is pressed, Freehand records the position as the start position for a line (using the `savePoint` variable). Then, when the right mouse button is pressed again, a straight line is drawn between the start position and the current position, with the current position becoming the new start position.

On entering the `:DO_DRAW` subprocedure, we begin with a little setup, including creating a workspace (`drawID`), selecting a drawing color, creating an Exit button, and setting a tag in the stack.

```
:DO_DRAW
    nPenResult = 0
    lastPoint = "-1,-1"
    savePoint = "-1,-1"
    BoxCaption( mainID, "Freehand" )
    BoxNew( drawID, "0, 0, 1000, 1000", 0 )
    BoxPen( drawID, COLOR%nColor%, 1 )
    BoxButtonDraw( drawID, bExit, "E&xit", "750, 860, 900, 930" )
    BoxDataTag( drawID, "NULL" )
```

To track mouse movements, the `:DO_DRAW` subroutine uses a loop to call the `MouseInfo` function until a mouse button event is read. At this point, we don't care which mouse button was pressed, only that one button was clicked.

```
Exclusive( @ON )
While @TRUE
    BoxDataClear( drawID, "NULL" )
```

Introduction to Programming

```
nButton = 0
nUp = 0

While nButton == 0 ; loop while waiting for mouse event
    nButton = MouseInfo( 4 )
    nUp = nUp + 1
    if nUp == 10 then lastPoint = "-1,-1"
EndWhile
```

The `nUp` variable in this loop is used to reset the `lastPoint` variable, which stores a starting point for a line, after the mouse button has been released for an arbitrary interval. This is so that we can start a new line, drawing freehand at a new location on the screen without automatically joining the old and new lines.

Next, after a mouse button has been clicked, we need to determine which button by checking the bitmask flags in `nButton`. We test the left mouse button first using the flag value 04 (or 4). If we find that the left mouse button was pressed, then a query calling `MouseInfo(6)` returns the mouse position in logical coordinates:

```
If( nButton & 04 ) ; left button is down
    Point = MouseInfo( 6 )
```

However, the value returned by the `MouseInfo` function has one small flaw: the x and y values are there but they're separated by a space. The `BoxDrawLine` function, which will be called next, needs the coordinates in a comma-delimited format. Fortunately, the `StrReplace` function can replace the delimiting space character with a comma:

```
Point = StrReplace( Point, " ", "," )
```

After ensuring that we have valid coordinates in the `lastPoint` variable, we draw a line from `lastPoint` to `Point`. After drawing the line, we update both the `lastPoint` and `savePoint` variables for future use.

```
If lastPoint != "-1,-1" then BoxDrawLine( drawID,
"%lastPoint%, %Point%" )
    lastPoint = Point
    savePoint = Point
EndIf
```

If the right mouse button was pressed rather than the left, the operation is similar except that the line is drawn from the `savePoint` location to the current mouse position:

```

If( nButton & 01 )
    Point = MouseInfo( 6 )
    Point = StrReplace( Point, " ", ", " )
    If savePoint != "-1,-1" then BoxDrawLine( drawID,
"%savePoint%, %Point%" )
    savePoint = Point
EndIf

```

Note that the `savePoint` variable, unlike the `lastPoint` variable, is not cleared (reset to `-1,-1`) in the wait loop. For drawing a line from point to point, there is an expected delay between mouse events, and clearing the start point for the line would be counterproductive.

Finally, there's also a test to decide if the Exit button was clicked and to clean up and exit when this occurs:

```

If BoxButtonStat( drawID, bExit ) == 1
    BoxButtonKill( drawID, bExit )
    break
EndIf
Endwhile
BoxDestroy( drawID )
return

```

Quite frankly, this isn't much of a drawing program. It has only one pen (red) and two drawing operations. However, it does serve its purpose in demonstrating how the mouse can be used in an application by recognizing mouse button events and mouse position information.

Summary

While mouse operations are integral in almost all contemporary applications, they are also largely transparent to the programmer. Since transparency has its limits, however, we've shown you the basics of writing your own mouse-oriented procedures, if only in the most basic forms.

Next, in [Chapter 13](#), we'll look at a few advanced methods to allow you to "poke inside the box" for those occasions when the available tools just aren't quite enough.

CHAPTER 13 : POKING INSIDE THE BOX

THE IntControl FUNCTIONS

control *n b*: a mechanism used to regulate or guide the operation of a machine, apparatus or system, *c*: a personality or spirit believed to actuate the utterances or performances of a spiritualist medium.

The `IntControl` (or *Internal Control*) functions are a large group of functions that provide the programmer with access to a variety of internal settings, ranging from controlling list box operations, to sending Windows event messages, to rebooting the system.

The chances are very good that you will not need any of these `IntControl` functions. In fact, we've used only one in the demo programs accompanying this book—`IntControl 12`. However, when you find that you need something beyond the usual, the `IntControl` functions can provide the solution you require.

All of the `IntControl` functions are invoked as:

```
IntControl( nn, p1, p2, p3, p4 )
```

The `nn` argument is an integer identifying the specific function desired, and `p1` through `p4` are parameters appropriate to each function. Some `IntControl` functions require no arguments, some require one, and a few require two or more arguments.

In this chapter, the `IntControl` functions have been grouped into rough categories according to their use: internal control test, general purpose, application control, and miscellaneous.

The Internal Control Test Function

`IntControl` function 1 is simply a test for the internal control function.

```
IntControl( 1, p1, p2, p3, p4 )
```

The [SelfTest.wbt](#) sample reports the `p1`, `p2` and `p3`, `p4` arguments in a pair of message boxes:



General-Purpose Functions

Quite a number of the `IntControl` functions are devoted to general-purpose objectives, including the following:

- Retrieving the handles from existing application windows
- Selecting system fonts
- Determining how file list boxes display files
- Determining whether single or multiple selections are permitted from list boxes

Window Interactions and Window Handles

Several functions are provided to support lower-level Windows (system) interactions:

- The window handle of the current parent window is returned by `IntControl` function 20. This function is similar to the `DllHwnd` function and requires no arguments.

```
IntControl( 20, 0, 0, 0, 0 )
```

- The window handle of a named window is returned by `IntControl` function 21, where `p1` contains a partial window name to identify the window desired.

```
IntControl( 21, p1, 0, 0, 0 )
```

- Windows IDs for all Windows Explorer windows are returned as a tab-delimited list by `IntControl` function 31.

```
IntControl( 31, p1, 0, 0, 0 )
```

- The class name for a specified window handle is returned by `IntControl` function 44, where the `p1` argument supplies the window handle.

```
IntControl( 44, p1, 0, 0, 0 )
```

The window handle can be obtained using the `DllHwnd` function or other `IntControl` functions. A blank string will be returned if the window does not exist.

System Font Selection

The system font used in list boxes is set using `IntControl` function 28.

```
IntControl( 28, p1, 0, 0, 0 )
```

The `p1` argument can be:

0	Proportional font (default)
1	Fixed-width font
2	GUI font

This function returns the current font type as 0 (proportional font) or 1 (fixed-width font).

File Operations

`IntControl` functions can perform delayed file move operations and set the file-sharing mode.

File Moves

A delayed file move is performed by `IntControl` function 30.

```
IntControl( 30, p1, p2, 0, 0 )
```

The `p1` argument identifies the source file (no wildcards are permitted), and the `p2` argument provides the destination. Since the files are not moved until the system is restarted, this function can be used to move or replace system files.

The destination argument may be a file name (matching the source file), a directory name, or an empty string. When an empty string is used, the source file will be deleted.

Return values are 1 for success, 2 if a `FileMove` operation was performed, or 0 on failure.

File-Sharing Mode

The file-sharing mode for file reads is set by `IntControl` function 39.

```
IntControl( 39, p1, 0, 0, 0 )
```

The `p1` parameter sets the share mode, as follows:

-1	No change to current share settings (default)
0	File sharing is not allowed
1	Allows other application to open files for read access
2	Allows other application to open files for write access

Introduction to Programming

3	Allows other application to open files for read and write access
---	--

The file sharing mode for file writes is set by `IntControl` function 40.

```
IntControl( 40, p1, 0, 0, 0 )
```

The `p1` parameter is the same as for `IntControl` function 39.

File List Box Settings

Settings for the file list box display are controlled using `IntControl` functions 4 and 5.

File list box return requirements are set using `IntControl` function 4.

```
IntControl( 4, p1, 0, 0, 0 )
```

Values for `p1` are:

0	The file list box is permitted to return a directory name only or may return nothing at all
1	The file list box is required to return a file name selection (default)

The display and processing of system and hidden files are set by `IntControl` function 5.

```
IntControl( 5, p1, 0, 0, 0 )
```

Values for `p1` are:

0	System and hidden files are not listed (default)
1	System and hidden files are listed and can be selected

The default file delimiter is set by `IntControl` function 29, which returns the previous file delimiter.

```
IntControl( 29, p1, 0, 0, 0 )
```

The `p1` argument should be a string specifying the new delimiter. Alternatively, if `p1` is an empty string, the current file delimiter is returned but no change is made.

General List Box Settings

Single or multiple selection requirements from a list box control (in a dialog) are set using `IntControl` function 33.

```
IntControl( 33, p1, 0, 0, 0 )
```

The `p1` argument can be:

0	Permits a single selection only
1	Allows multiple selections (default)

Application Control Functions

Another area where the `IntControl` functions are used is in providing controls for both the application itself and for external applications, as well as some for the operating system. These functions allow you to perform the following tasks:

- Generate Windows message events
- Control some WinBatch functions
- Restart Windows
- Close programs
- Get error messages
- Set `CreateProcess` flags
- Access memory addresses
- Set timing and waits for input

Some of these functions are mentioned only with brief explanations, and you'll need to refer to other sources for details. Or, to phrase it bluntly, some of these functions are not intended for use by novices, while others are easier to invoke. For example, if you're planning on invoking the `SendMessage` or `PostMessage` events, it would be a good idea to have some familiarity with the topic before doing so. Check Windows programming references for more information about Windows events and messages.

Windows Messages

A Windows `SendMessage` event is generated by `IntControl` function 22.

```
IntControl( 22, p1, p2, p3, p4 )
```

The four parameters required are:

Introduction to Programming

p1	The handle for the destination window (where the message is sent)
p2	The message ID (in decimal format)
p3	A wParam value
p4	A character string, which is copied to a GMEM_LOWER buffer while a pointer to the string is passed as the lParam argument in the message (the buffer is released when the SendMessage function returns)

The contents of the *wParam* and *lParam* arguments depend on the message ID. Refer to a Windows programming reference for details.

A Windows `PostMessage` event is generated by `IntControl` function 23, following the same rules as `SendMessage` event generation (22).

```
IntControl( 23, p1, p2, p3, p4 )
```

SendMessageA / SendMessageW

More conveniently, WinBatch now provides the `SendMessageA` and `SendMessageW` functions.

`SendMessageA` issues a Windows `SendMessage` event with the `lParam` parameter as an ANSI string. The format is:

```
SendMessageA( window-id/handle, message-id, wParam, lParam )
```

...where the parameters are:

- (i) `windows-id/handle` Window ID or handle message is sent to
- (i) `message-id` Message ID (decimal number)
- (i) `wParam` Message specific information
- (s) `lParam` The ANSI character string to be displayed

The `SendMessageW` function is the same as `SendMessageA` with the exception that the `lParam` parameter is presumed to be a Unicode (wide character) message for display rather than ANSI.

WinBatch Control

`IntControl` functions provide access to WinBatch exit codes, server status, icon states, WinBatch program file names, and program exit options.

Exit Code

The exit code returned by WinBatch's `WinMain` (entry point) function is set by `IntControl` function 1000.

```
IntControl( 1000, p1, 0, 0, 0 )
```

The `p1` argument contains the new exit code value. The old exit code is returned by the function.

Icon States

The icon display state is set by `IntControl` function 1002.

```
IntControl( 1002, p1, 0, 0, 0 )
```

The display state of the WinBatch icon is set for the duration of the script as instructed by the `p1` argument. Values for `p1` are:

-1	No change, returns current setting
0	The WinBatch icon is hidden
1	The WinBatch icon is displayed (default)

The "openable" state for the WinBatch icon is set by `IntControl` function 1003.

```
IntControl( 1003, p1, 0, 0, 0 )
```

The `p1` argument sets the openable flag for the WinBatch icon. If the flag is `ON` (default), then the WinBatch icon can be opened (restored) to a normal window by clicking on the icon. If the flag is `OFF`, the window cannot be opened by any means. Values for `p1` are:

-1	No change, return current setting
0	The WinBatch icon cannot be opened
1	The WinBatch icon can be opened (default)

Introduction to Programming

Program File Names

The file name of the current WinBatch program is returned by `IntControl` function 1004.

```
IntControl( 1004, 0, 0, 0, 0 )
```

If the current application is a child program called with the `Call` function, the name of the main (calling) program will be returned.

WIL Termination Codes

WinBatch program exit options are set by `IntControl` function 12 (introduced in [Chapter 11](#) as part of the generic initialization for a WinBatch window program).

```
IntControl( 12, p1, p2, 0, 0 )
```

The `p1` argument consists of a combination of the Exit Windows option and a Terminate Group option. The `p2` argument is a string to be displayed when an exit is not permitted.

The Exit Windows option can be any one of the following:

0	A pop-up message box allows the user the option to terminate or to continue
1	Applications are allowed to exit without warning
2	Applications are not allowed to exit (if <code>p2</code> is not "" or 0, the message provided will be displayed)
3	Reserved value, do not use

The Terminate Group option can be any one of the following:

0	Provide a notification message when the program is terminated by the user
4	Allow a quiet termination (no message)
8	Refuse termination

The Exit Windows and Terminate Group options are combined by adding the two values.

Windows Restarting

Several `IntControl` functions allow restarting and warm rebooting (as if the Ctrl+Alt+Del key combination was pressed) of Windows systems.

Windows System Restart

You can restart a Windows system using `IntControl` function 66, just as if Windows had exited to DOS and then restarted.

```
IntControl( 66, 0, p2, 0, 0 )
```

You can execute a warm reboot using `IntControl` function 67.

```
IntControl( 67, 0, p2, 0, 0 )
```

A complete shutdown, including an automatic power-off is performed using `IntControl` function 68.

```
IntControl( 68, 0, p2, p3, 0 )
```

Application Closing

WinBatch programs can be closed by other WinBatch applications using `IntControl` function 47.

```
IntControl( 47, p1, 0, 0, 0 )
```

The `p1` argument supplies the full or partial window name for the WinBatch program window to close. The function returns `@TRUE` if successful or `@FALSE` on failure.

DOS applications can be closed by a WinBatch application using `IntControl` function 48. (This function is not supported under Windows NT.)

```
IntControl( 48, p1, 0, 0, 0 )
```

The `p1` argument supplies the full or partial window name for the DOS program window to close. The function returns `@TRUE` if successful or `@FALSE` on failure. `IntControl` 48 has been superceded with the function `TerminateApp`.

Error Messages

An error message string is returned by `IntControl` function 34, corresponding to the WIL error identified by the `p1` parameter.

Introduction to Programming

```
IntControl( 34, p1, p2, 0, 0 )
```

The Go to Web Page button appearing in WIL error boxes can be displayed or hidden using `IntControl` function 50.

```
IntControl( 50, p1, p2, 0, 0 )
```

The `p1` argument determines whether the button is included when a WIL error box is created. Settings for `p1` are:

-1	No change, return current setting
0	Remove the Go to Web Page button from error boxes
1	Add the Go to Web Page button to error boxes (default)

When the Go to Web Page button is pressed, the Web browser is launched and opens the the url specified by the `p2` argument.

Error Handling

There are two major methods to trap errors. The more powerful method is to use `IntControl` 73 to set an error handler to capture all types of errors. Where as, the older `ErrorMode` can only be used to capture minor errors.

```
IntControl(73, p1, p2, p3, 0)
```

The `p1` argument lets you specify what should happen when the next error occurs in the script.

-1	Don't change (just return current setting)
0	Normal error processing (default)
1	Goto the label :WBERRORHANDLER
2	Gosub the label :WBERRORHANDLER
3	Call the UDF specified by <code>p3</code>

This is a one-time flag, which gets reset to 0 (default) when an error occurs.

If you simply want to suppress all displayed errors then you might want to use `IntControl` 38. `IntControl` 38 can tell the script to exit quietly upon error.

CreateProcess Flags

Flags for the `CreateProcess` operation can be set using `IntControl` function 51.

```
IntControl( 51, p1, 0, 0, 0 )
```

The `CreateProcess` function is used to determine the type of process and the process priority when an application is launched. Refer to the online documentation for more information.

Memory Access

The contents of a memory address are returned using `IntControl` function 32.

```
IntControl( 32, p1, p2, p3, p4 )
```

The `p1` argument specifies the memory address.

The `p2` argument specifies the type of data to be retrieved. The `p2` argument should be one of the following:

"BYTE"	Returns byte data
"WORD"	Returns word data
"LONG"	Returns a long integer

The `p3` argument specifies to either read or write from memory. Argument `p4` specifies the value to be read written to the memory location. It must be a single character or integer, corresponding to "data-type".

A pointer to a binary buffer is returned by `IntControl` function 42.

```
IntControl( 42, p1, 0, 0, 0 )
```

The `p1` argument specifies the binary buffer (see the `BinaryAlloc` function). The address returned will be within the memory space of the WIL Interpreter.

Input Timing and Waits

You can use `IntControl` functions to set options for timeouts and the `SendKey` function.

Introduction to Programming

Timeouts

A wait (timeout) until a target application is waiting for user input is initiated by `IntControl` function 36.

```
IntControl( 36, p1, p2, 0, 0 )
```

The `p1` argument specifies a window name associated with (identifying) the target application. The `p2` argument is a timeout (in milliseconds) or `-1` for no timeout.

This function waits until the process that created the specified window has paused and is waiting for user input (with no input pending) or until the specified timeout interval has elapsed. The function returns `@TRUE` if it has waited successfully or `@FALSE` if a timeout has occurred or if it was unable to initiate a wait.

`IntControl` 36 has been superseded with the function `WinWaitReady`.

The window retry timeout interval is set by `IntControl` function 46.

```
IntControl( 46, p1, 0, 0, 0 )
```

The `p1` argument can be:

-1	No change, return current setting
0	No retries
nn	Number of seconds to wait (default = 9)

WIL functions that accept a window title as a parameter (except for the `WinExist` function) will wait for the specified interval for the window to exist.

The SendKey Function

The `SendKey` function can be slowed using `IntControl` function 35.

```
IntControl( 35, p1, p2, 0, 0 )
```

The `p1` argument is the amount of time (in milliseconds) to delay between keystrokes. The function returns the previous delay setting. The default delay is 0 milliseconds (no delay).

The `p2` argument is the amount of time to delay in `MouseClick` and `MousePlay` between pressing and releasing the mouse button for a single click (not a double-click), in milliseconds. Default is 0 (no delay).

The `WaitForInputIdle` function can be enabled or disabled using `IntControl` function 43.


```
IntControl( 43, p1, 0, 0, 0 )
```

This function controls whether the `SendKey`, `SendKeyTo`, and `SendKeysChild` functions attempt to wait until the active application is ready to accept input before sending keystrokes. The function returns the previous setting.

Miscellaneous Operations

The `IntControl` functions discussed here simply cannot be conveniently categorized. One of the functions allows you to reassess the current language. Another function is for adding system menus to dialogs and list boxes.

Language Control

`IntControl` function 26 initiates a reassessment of the language currently used and makes any changes required to the language strings used by the WIL Interpreter. Function 26 requires no parameters and is normally invoked when an application starts.

```
IntControl( 26, 0, 0, 0, 0 )
```

System Menus

System menus can be added to WIL pop-up windows using `IntControl` function 49.

```
IntControl( 49, p1, p2, 0, 0 )
```

The `p1` argument specifies which pop-up windows will have system menus. Values for `p1` are:

-1	No change, return current setting
0	No system menus (default)
1	Add a system menu to dialogs created using the WIL Dialog Editor
2	Add a system menu to list boxes (those created using <code>AskItemList</code> , <code>AskFileText</code> or related functions)
3	Add a system menu to both dialogs and item list boxes

The `p2` argument specifies the value that a dialog box (created using the `Dialog` function) will return if the user closes the dialog without pressing one of the pushbuttons in the dialog (e.g., by pressing <Alt-F4>, or by clicking on the "Close" icon in the title bar). If a dialog returns 0, processing will be transferred to the label in the script marked `":Cancel"`.

Summary

The `IntControl` functions offer an opportunity for the programmer to poke around inside the box and perform tasks that are not supplied by conventional WinBatch functions. These are not mainstays in your programming toolbox, but when needed, they are useful and usable.

Next, in [Chapter 14](#), we'll take a look – briefly – at Dynamic Dialogs; i.e., dialogs which are constructed "on the fly" and, therefore, can be configured by the application as desired.

CHAPTER 14 : DYNAMIC DIALOGS, MENUS, CALLBACKS

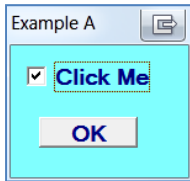
MAKING DYNAMIC DIALOGS

WinBatch supports two types of dialog: static or dynamic. Both are customarily drawn using the WinBatch Dialog Editor but they differ in how they function.

For a static dialog, the user fills in fields, checks various items or makes assorted selections before pressing a Submit (or Ok or whatever) button. At that point, the dialog exits and the WinBatch code continues to execute.

The shortcoming of a static dialog, however, is that outside of a few standard, primitive responses, aside from accepting input, the dialog has no ability to interact with the user. That is, the only choices presented to the user are those which have been defined in the original dialog.

Using dynamic dialogs, however, you have the potential to use your own user defined functions or subroutines to perform a variety of functions while the user is still interacting with the dialog. For example, a dynamic dialog might perform such tasks as validating a password, calculating costs for an order based on the user's selections, enabling or disabling options based on prior selections or any of a multitude of other tasks.



To show how dynamic dialogs work, we'll begin with a very simple dialog which contains a check box and one button as shown at left. The dialog was generated by the WinBatch Dialog Editor and produced the WinBatch code shown following (see also [Exercise A.wbt](#), etc. for the source code for this and subsequent examples.)

```
EXAFormat=`WWWDLGED,6.2`

EXACaption=`Exercise A`
EXAX=9999 ; -01
EXAY=9999 ; -01
EXAWidth=060
EXAHeight=045
EXANumControls=002
EXAProcedure=`DEFAULT`
EXAFont=`Microsoft Sans Serif|7373|70|34`
EXATextColor=`0|0|128`
EXABackground=`DEFAULT,128|255|255`
EXAConfig=0
```

Introduction to Programming

```
EXA001=`005,005,046,010,CHECKBOX,"CheckBox_1",MyCheckBox, "Click  
Me",1,1,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
EXA002=`009,023,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,2,DEF  
AULT,DEFAULT,DEFAULT,DEFAULT`  
ButtonPushed=Dialog("EXA")
```

To begin, we'll take a look at the basic dialog code with a rundown of the code structure.

By default, the dialog editor supplies the name MyDialog but this can be – and should be – changed to something more appropriate. After all, how many MyDialogs can you keep track of?

First, EXA is the name, specified in the dialog editor, of this dialog and is short for Exercise A and all of the dialog variables in this example begin with EXA (required).

EXAFormat	Used to tell WinBatch what kind of dialogs we are dealing with. These are the version 6.2 dialogs.
EXACaption	Simply specifies the caption or title of the dialog box.
EXAX	Specifies the position for the upper-left corner of the dialog relative to the left edge of the screen. A value of –1 specifies the dialog should be centered.
EXAY	Specifies the position for the upper-left corner of the dialog relative to the top edge of the screen. A value of –1 specifies the dialog should be centered.
EXAWidth	Specifies the width of the dialog.
EXAHeight	Specifies the height of the dialog.
EXANumControls	Reports the number of controls in the dialog.
EXAProcedure	At this point, EXAProcedure is set to 'DEFAULT' meaning that there is no desired procedure for this dialog. This variable will be discussed later in this chapter since the dialog procedure is at the heart of this topic.
EXAFont	Defines a default font for the dialog. Optionally, individual controls may have their own font specifications.
EXATextColor	Defines the default text color for the dialog. Again, individual controls may use a different text color.
EXABackground	Defines a background to use for the dialog; may be either a BMP file or a color specification.

EXAConfig	<p>This controls how the Dialog Editor creates the WIL Dialog command in your template. If the variable is given a value of one (1), the editor will create a return variable name that includes the dialog name (<dlg-variable>). For example, if your dialog name is EXA, the dialog statement would be:</p> <pre>EXAButtonPushed = Dialog("EXA", 1)</pre> <p>Without this setting the dialog statement would be:</p> <pre>ButtonPushed = Dialog("EXA", 1)</pre> <p>If the variable is given a value of two (2), the Dialog Editor will place a zero (0) in the optional second parameter of the Dialog statement when it creates your dialog template. This optional parameter tells WinBatch to ignore the Dialog statement when it processes a script containing the dialog template. The Dialog Editor, on the other hand, will not ignore the dialog statement so that you can reload the template into the editor at any time. If you do not specify this configuration option, the Dialog Editor will create a dialog statement with the second parameter set to one (1). This tells WinBatch to process the Dialog statement by loading and displaying the dialog template.</p> <p>You can combine the values 1 and 2 using the binary OR() operator to enable both features.</p>
EXA001, EXA002, etc.	<p>The numbered variables define each of the controls in the dialog and each begins with the dialog name ('EXA') followed sequentially by number. Since they are difficult to code by hand, the Dialog Editor is the preferred means of building a dialog. You will need to be able to tell what control each variable is defining to get your Dynamic Dialogs to work and, usually, a quick inspection of the data on the line will tell you this.</p>
ButtonPushed= Dialog("EXA")	<p>This is not actually a dialog definition variable, but included for completeness. This is the (default) instruction which actually tells WinBatch to display the dialog on the screen. This line may be replaced (and usually will be replaced) with other instructions beyond a simple display.</p>

Where the Dialog Editor provides automatic code generation for a static dialog, we're now going to look at how a static dialog can be transformed into a dynamic dialog.

Adding Dialog Procedure code

Basically, to do Dynamic Dialogs, you have to add code, either in a #DefineFunction or a #DefineSubroutine block to instruct WinBatch what to do. Although it is easily possible to write all the Dialog Procedure by hand, it gets exceedingly tedious very quickly. And, furthermore, most Dialog Procedure code has the same basic structure as

Introduction to Programming

any other Dialog Procedure code. In addition, most of the Dialog control functions have dozens of hard to remember request codes that give no really readable hint as to what is going on when examining the code.

To simplify matters, an automatic Dialog Procedure code generator has been added to WinBatch Studio and this will both define a whole bunch of useful constants (many, many more than you will likely need) and also generate a useful block of template code based on your dialog.

To have WinBatch Studio generate the Dialog Procedure template code, you simply highlight your dialog code, top to bottom, starting with the line similar to...

```
EXAFormat='WDDLGED, 6.2'
```

...all the way down to and including the...

```
ButtonPushed=Dialog("EXA")
```

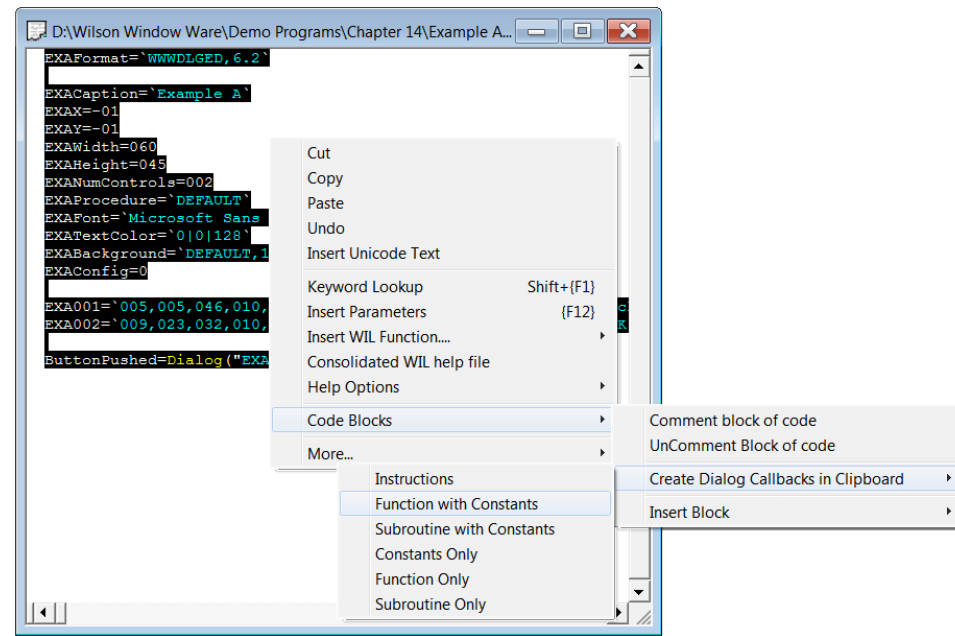
line.

Next – see illustration following – right-click the WinBatch edit window to get the WinBatch popup menu to appear. From the menu, select:

Code Blocks

- Create Dialog Callbacks in Clipboard
- Function with Constants

Generating code may take a few seconds. Once done, paste the generated code (it's now in your Windows Clipboard) into your script, usually preceding your original dialog definition code.



Creating a Dialog Procedure code from a Dialog Definition block

Once you have created the dialog procedure and pasted the generated code into your script, the next step is to locate the line similar to ...

```
EXAProcedure='EXACallbackProc'
```

... towards the bottom of the pasted code (generally in a block marked in red) and move this line into the dialog definition code created by the Dialog Editor where you will replace the original line ...

```
EXAProcedure='DEFAULT'
```

Next, you can, in the generated code, (optionally) delete the explanatory lines which tell you want to do in this particular operation ([highlighted block in following code sample Exercise B.wbt.](#))

Okay, with this done, you are now ready to make your dialog callback actually do something. To accomplish this, you will need to uncomment desired lines in the generated code and add your own code to accomplish the intended task; a process which will be covered later in this chapter.

First, however, let's look at what has happened and what we have.

```
;=====
;=====
;=====
#DefineSubRoutine InitDialogConstants()
```

Introduction to Programming

```
;DialogprocOptions Constants
MSG_INIT=0 ; The one-time initialization
MSG_TIMER=1 ; Timer event
MSG_BUTTONPUSHED=2 ; Pushbutton or Picturebutton
MSG_RADIOPUSHED=3 ; Radiobutton clicked
MSG_CHECKBOX=4 ; Checkbox clicked
MSG_EDITBOX=5 ; Editbox or Multilinebox
MSG_FILESELECT=6 ; Filelistbox
MSG_ITEMSELECT=7 ; Itembox
MSG_COMBOCHANGE=8 ; Combobox/Droplistbox
MSG_CALENDAR=9 ; Calendar date change
MSG_SPINNER=10 ; Spinner number change
MSG_CLOSEVIA49=11 ; Close clicked (Enabled via DialogProcOptions
1002
MSG_FILEBOXDOUBLECLICK=12 ; Get double-click message on a
FileListBox
MSG_ITEMBOXDOUBLECLICK=13 ; Get double-click message on an ItemBox
MSG_COMEVENT=14 ; COMCONTROL Event notification from DialogObject
(NOT DialogProcOptions)
MSG_MENUITEM=15 ; MenuItem selected
MSG_MENUITEMINIT=16 ; MenuItem initialized
MSG_RESIZE=17 ; Dialog resized

DPO_DISABLESTATE=1000 ; codes -1=GetSetting 0=EnableDialog
1=DisableDialog
DPO_CHANGEBACKGROUND=1001 ; -1=Get Current otherwise bitmap or color
string
DPO_CHANGESYSMENU=1002 ; -1=Get Current 0=none 1=close 2=close/min
3=close/max 4=close/min/max
DPO_CHANGETITLE=1003 ; Set/Get Dialog Title - (-1 to get)
DPO_GETNAME=1004 ; Returns the name associated with a control's
number.
DPO_GETNUMBER=1005 ; Returns the number associated with a control's
name.
DPO_GETCLIENTAREA=1007 ; Returns a space delimited list of the width
and height of the client area.

;DialogControlState Constants
DCSTATE_SETFOCUS=1 ; Give Control Focus
DCSTATE_QUERYSTYLE=2 ; Query control's style
```



```

DCSTATE_ADDSTYLE=3 ; Add control style
DCSTATE_REMOVESTYLE=4 ; Remove control style
DCSTATE_GETFOCUS=5 ; Get control that has focus
DCSTATE_MOVEMOUSEOVER=6 ; Move the mouse over the control


DCSTYLE_DEFAULT=0 ; Set Default Style
DCSTYLE_INVISIBLE=1 ; Set Control Invisible
DCSTYLE_DISABLED=2 ; Set Control Disabled
DCSTYLE_NOUSERDATA=4 ; Note: Setable via DialogControlState function
ONLY SPINNER control only
DCSTYLE_READONLY=8 ; Sets control to read-only (user cannot type in
data) EDITBOX MULTILINEBOX SPINNER
DCSTYLE_PASSWORD=16 ; Sets 'password mode' where only '*'s are
displayed EDITBOX
DCSTYLE_DEFAULTBUTTON=32 ; Sets a button as the default button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_DIGITONLY=64 ; Set edit box to accept digits only EDITMOX
MULTILINEBOX
DCSTYLE_FLAT=128 ; Makes a 'flat' hyperlink-looking button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_NOADJUST=256 ; Turns off auto-height adjustment ITEMBOX
FILELISTBOX
DCSTYLE_TEXTCENTER=512 ; Center text in control VARYTEXT STATICTEXT
DCSTYLE_TEXTRIGHT=1024 ; Flush-Right text in control VARYTEXT
STATICTEXT
DCSTYLE_NOSELCLURLEFT=2048 ; No selection, cursor left EDITBOX
MULTILINEBOX
DCSTYLE_NOSELCLURRIGHT=4096 ; No selection, cursor right EDITBOX
MULTILINEBOX
DCSTYLE_SHIELD=8192 ; Display Security Shield icon on button
(Vista/7 and newer) PUSHBUTTON PICTUREBUTTON
DCSTYLE_MENUCHECK=32768 ; Adds a check mark to the left of a menu
item MENUITEM
DCSTYLE_MENURADIO=65536 ; Adds a radio button like dot graphic to
the left of a menu item MENUITEM
DCSTYLE_MENUSEP=131072 ; Separator bar graphic MENUITEM
DCSTYLE_MENUBREAK=262144 ; column break MENUBAR


;DialogControlSet / DialogControlGet Constants
DC_CHECKBOX=1 ; CHECKBOX
DC_RADIOBUTTON=2 ; RADIOBUTTON
DC_EDITBOX=3 ; EDITBOX MULTILINEBOX

```

Introduction to Programming

```
DC_TITLE=4 ; PICTURE RADIOBUTTON CHECKBOX PICTUREBUTTON VARYTEXT
STATICTEXT GROUPBOX PUSHBUTTON MENUITEM

DC_ITEMBOXCONTENTS=5 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXSELECT=6 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_CALENDAR=7 ; CALENDAR
DC_SPINNER=8 ; SPINNER
DC_MULTITABSTOPS=9 ; MULTILINEBOX
DC_ITEMSCROLLPOS=10 ; ITEMBOX FILELISTBOX
DC_BACKGROUNDCOLOR=11 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT
GROUPBOX PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX
MULTILINEBOX

DC_PICTUREBITMAP=12 ; PICTURE PICTUREBUTTON
DC_TEXTCOLOR=13 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT GROUPBOX
PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX MULTILINEBOX
DC_ITEMBOXADD=14 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXREMOVE=15 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_RADIOVALUE=16 ; RADIOBUTTON
DC_POSITION=17 ; ALL CONTROLS (Except MENUBAR and MENUITEM)
DC_MENUNAMES=18 ; ALL CONTROLS
DC_HANDLE=19 ; ALL CONTROLS (Except MENUBAR and MENUITEM)

;DialogObject constants
DLGOBJECT_ADDEVENT=1 ; Call dialog callback when the specified event
occurs
DLGOBJECT_STOPEVENT=2 ; Stop calling dialog callback when an event
previously requested with
DLGOBJECT_GETOBJECT=3 ; Return an object references to the specified
control
DLGOBJECT_GETPICTURE=4 ; Create and return an object reference to a
picture object

;Return code constants
RET_DO_CANCEL=0 ; Cancels dialog
RET_DO_DEFAULT= -1 ; Continue with default processing for control
RET_DO_NOT_EXIT= -2 ; Do not exit the dialog
return
#EndSubroutine
;=====
;=====
;=====
```

```

#DefineFunction
EXBCallbackProc (EXB_Handle, EXB_Message, EXB_Name, EXB_EventInfo,
EXB_ChangeInfo)

    InitDialogConstants() ; Initialize Dialog Constants
    ON_EQUAL = @TRUE ; Initialize variable ON_EQUAL
    Switch EXB_Message ; Switch based on Dialog Message type
        Case MSG_INIT ; Standard Initialization message
;           DialogProcOptions (EXB_Handle, MSG_TIMER, 1000)
;           DialogProcOptions (EXB_Handle, MSG_BUTTONPUSHED, @TRUE)
;           DialogProcOptions (EXB_Handle, MSG_CHECKBOX, @TRUE)
        Return (RET_DO_DEFAULT)

;       case MSG_BUTTONPUSHED ; ID "PushButton_OK"  PushButton_OK
;           return (RET_DO_DEFAULT)

;       case MSG_CHECKBOX ; ID "CheckBox_1"  CheckBox_1 MyCheckBox
;           return (RET_DO_DEFAULT)

    EndSwitch ; EXB_Message
    Return (RET_DO_DEFAULT)
#EndFunction ; End of Dialog Callback EXBCallbackProc

;=====
;=====
;=====

"<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---"
"<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---"
"<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---"
"REMEMBER UPDATE EXAProcedure VARIABLE AS BELOW AND DELETE THESE LINES"
EXBProcedure=`EXBCallbackProc`
"<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---"
"<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---"
"<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---"

EXBFormat=`WWWDLGED, 6.2`

```

Introduction to Programming

```
EXBCaption=`Example B`
EXBX=9999 ; -01
EXBY=9999 ; -01
EXBWidth=060
EXBHeight=045
EXBNumControls=002
EXBProcedure=`EXBCallbackProc`; was EXAProcedure=`DEFAULT`
EXBFont=`Microsoft Sans Serif|7373|70|34`
EXBTextColor=`0|0|128`
EXBBackground=`DEFAULT,128|255|255`
EXBConfig=0

EXB001=`005,005,046,010,CHECKBOX,"CheckBox_1",MyCheckBox,"Click
Me",1,1,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EXB002=`009,023,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,2,DEF
AULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed=Dialog("EXB")

Message("CheckBox Value on Dialog Exit is", MyCheckbox)
exit
```

After this point, in our text examples, we'll omit a lot of the preceding code and only show portions of the script relevant to the topic. The entire code will, of course, be found in the program examples.

Before moving on, if you'll refer to the code preceding, you'll find that it begins with a large block – `#DefineSubroutine InitDialogConstants` – which defines a number of constants used by various dialog control functions. This allows you to use the constants – whose labels hopefully convey meaning to us – rather than relying on numbers ... which are clear to the computer but cryptic to carbon-based life forms.

Yes, we are relying on the computer to "translate" our labels – constants – into numbers for operations but we're already doing exactly the same thing with our procedure and function names ... and that's what computers are for; handling numbers at our convenience.

Some of you will also have noticed – hopefully – that the 'EXA's in the first example have now become 'EXB's. This change (and the corresponding dialog name change)

serves two purposes: first, to make the examples easy to differentiate and, second, so that we can have different examples as source files for you to refer to.

Following the constant definitions, you'll find the `#DefineFunction EXBCallbackProc` which is the template for your dialog procedure code.

And, following this (surrounded by "`<X>--^!---`" lines) is the single modified version of the original dialog code where the EXBProcedure, unlike the EXAProcedure variable in the first Exercise, has been set to `EXBCallbackProc`, the name of the `#DefineFunction` preceding. This is the instruction which links the dialog and your custom code and instructs WinBatch to begin processing the dialog callback procedure.

Functions vs Subroutines

The dialog procedure, `EXBCallBackProc` in this case, is a special case of either a `#DefineFunction` or a `#DefineSubroutine`. The difference between these; i.e., a function versus a subroutine is simple.

For a function, all variables are "pure local". That is, the variables used within the function are invisible to the rest of the program and the only way to get information into a `#DefineFunction` is via the passed parameters, and the only way to get information out of it is via the return value.

Note: some obscure exceptions do apply.

In contrast, for a subroutine, all variables are "global". All variables in the subroutine are visible and accessible the rest of your program. This allows easy import and export of information from your `#DefineSubroutine` from and to the rest of your script. The downside is that it is easy to mistakenly stomp on information in your main script that you did not intend to.

As before, some obscure exceptions apply.

In particular, the `#DefineFunction` (or `#DefineSubroutine`) statement must have five parameters. e.g.

```
#DefineFunction EXBCallbackProc( EXB_Handle, EXB_Message, EXB_Name,
EXB_EventInfo, EXB_ChangeInfo)
```

The parameters are

EXB_Handle	A special number that refers uniquely to this specific dialog and will be used later in subsequent calls to various dialog functions. (Not used in Exercise B.)
EXB_Message	A dlgmessage number to allow the dialog procedure to figure out

Introduction to Programming

	what is going on and why it was called. By default, the dialog procedure will be called once, with a <code>dlgmessage</code> value of zero. As shown in a later example, passing a non-zero <code>dlgmessage</code> allows requesting a specific service from the dialog.
<code>EXB_Name</code>	The name of the control requesting a specific event; i.e. identifying a pushbutton, checkbox or radio button selected and directing the dialog procedure to respond accordingly.
<code>EXB_EventInfo</code>	The event information object (valid only when <code>DialogObject</code> generates event-code = 14)
<code>EXB_ChangeInfo</code>	On <code>RESIZE</code> returns a space delimited list of dialog-units: in which the items represent the delta (change in width and height) of the dialog that resulted from resizing activity and the client height and width that represent the internally maintained size of the dialog's client area. {delta_width} {delta_height} {client_width} {client_height}

The dialog support functions make use of quite a few different numbers to represent different things, and it can get quite confusing if you hard code all the numbers, as most people will quickly forget what they mean. So we start off on the right foot here by first defining a number of constants by calling the `InitDialogConstants` subroutine.

```
InitDialogConstants() ; Initialize Dialog Constants
```

Note that since `InitDialogConstants` is defined as a subroutine, the variables the constants are assigned to are accessible by the code that called the subroutine (as opposed to a function, where the variables are hidden from the calling code).

Next is the "Switch" statement in the Dialog Procedure.

The switch statement is the heart and soul of a Dialog Procedure. There is one case for each different type of message that it can receive. Initially, the only message enabled by default is the `MSG_INIT` initialization message.

When the Dialog Procedure starts up, by virtue of the `EXBProcedure` variable in the Dialog Definition statements discussed earlier, the Dialog Procedure gets one chance to tell WinBatch what it wants. Just before the Dialog is displayed to the user, the Dialog Procedure is called with a `EXB_Message` value of `MSG_INIT`, which is the signal that the Dialog procedure should perform its initialization steps. Generally the initializations steps consist of telling WinBatch, via various Dialog support functions (discussed later) what other messages your Dialog procedure wishes to receive.

The Switch statement produced by the Winbatch Studio code looks like:

```
Switch EXB_Message ; Switch based on Dialog Message type
    Case MSG_INIT ; Standard Initialization message
;         DialogProcOptions(EXB_Handle,MSG_TIMER,1000)
```

```

;         DialogProcOptions (EXB_Handle,MSG_BUTTONPUSHED,@TRUE)
;         DialogProcOptions (EXB_Handle,MSG_CHECKBOX,@TRUE)
Return (RET_DO_DEFAULT)

;         case MSG_BUTTONPUSHED ; ID "PushButton_OK"  PushButton_OK
;         return (RET_DO_DEFAULT)

;         case MSG_CHECKBOX ; ID "CheckBox_1"  CheckBox_1 MyCheckBox
;         return (RET_DO_DEFAULT)

EndSwitch ; EXB_Message

```

Please note that most of the generated code is **commented out** and that the only active code looks like this:

```

Switch EXB_Message ; Switch based on Dialog Message type
Case MSG_INIT ; Standard Initialization message
Return (RET_DO_DEFAULT)
EndSwitch ; EXB_Message

```

In effect, all that happens is that the `MSG_INIT` command is captured and `RET_DO_RETURN` is returned, instructing WinBatch to do nothing except for the default processing. This will not, however, always be the case and, in further examples parts of the existing code will be uncommented and new code will be added.

But, before going too far, we should look at the possible return values. Any time a dialog procedure exits (closes), it is expected to return a value to tell WinBatch what's supposed to happen next.

The permitted return values and their actions are:

Any positive integer e.g. 1, 2, 3, etc.	Close the dialog, returning the specified value as the return value of the dialog.
<code>RET_DO_CANCEL = 0</code>	Cancel the dialog, i.e., do normal <code>CANCEL</code> processing.
<code>RET_DO_DEFAULT = -1</code>	Do default processing to close the dialog
<code>RET_DO_NOT_EXIT= -2</code>	Do default processing but <i>do not</i> close.

At this point, [Exercise B.wbt](#) has taken the step of hooking up a dialog procedure and, within the procedure, an initialization case which, in this example, actually does nothing. Now, however, all of the major pieces are in place and the next step will be to flesh out the initialization process.

Introduction to Programming

Exercise_C

The next step is to add functionality to the switch code so that the dialog is able to respond – appropriately – to the activity which we're interested in monitoring. While simple, for this Exercise, the objective is to have the user confirm clicking on the checkbox. And, to accomplish this task, notice is required when there's any activity in a checkbox and the code in [Exercise B.wbt](#) (now to become [Exercise C.wbt](#)) must be modified.

First, three previously commented-out lines in the `EXC_CallbackProc` are uncommented to become active (shown **highlighted** here):

```
#DefineFunction
EXC_CallbackProc (EXC_Handle, EXC_Message, EXC_Name, EXC_EventInfo, EXC_ChangeInfo)

    InitDialogConstants() ; Initialize Dialog Constants
    ON_EQUAL = @TRUE ; Initialize variable ON_EQUAL
    Switch EXC_Message ; Switch based on Dialog Message type
        Case MSG_INIT ; Standard Initialization message
;           DialogProcOptions (EXC_Handle, MSG_TIMER, 1000)
;           DialogProcOptions (EXC_Handle, MSG_BUTTONPUSHED, @TRUE)
           DialogProcOptions (EXC_Handle, MSG_CHECKBOX, @TRUE)
        Return (RET_DO_DEFAULT)
```

In the `MSG_INIT` code, the change above tells WinBatch that we do want to know about `CheckBox` events.

```
;           case MSG_BUTTONPUSHED ; ID "PushButton_OK" PushButton_OK
;           return (RET_DO_DEFAULT)

           Case MSG_CHECKBOX ; ID "CheckBox_1" CheckBox_1 MyCheckBox
           Return (RET_DO_DEFAULT)
```

And these two lines have been uncommented so that `MSG_CHECKBOX` messages (events) are reported even though the return value is still `-1 (RET_DO_DEFAULT)`, calling for default processing on closing the dialog.

```
EndSwitch ; EXC_Message
Return (RET_DO_DEFAULT)
```



```
#EndFunction ; End of Dialog Callback EXCCallbackProc
```

There's one further provision, following the dialog execution (i.e., after the dialog closes):

```
Message( "CheckBox Value on Dialog Exit is", MyCheckbox )
```

Strictly speaking, this last instruction has nothing to do with processing the dialog but is here to let you see what the exit value returned by the checkbox is: i.e., 1 (TRUE) if the checkbox was checked or 0 (FALSE) if unchecked.

Exercise_D

For the next step ([Exercise D.wbt](#)), we'll add code which does something at least vaguely useful. And, as with EXC, the changes for EXD are fairly brief, appear only in the EXDCallbackProc function in the `Case MSG_CHECKBOX` response and, following, again, are shown **highlighted**.

```
#DefineFunction
EXDCallbackProc (EXD_Handle, EXD_Message, EXD_Name, EXD_EventInfo, EXD_ChangeInfo)

    InitDialogConstants() ; Initialize Dialog Constants
    ON_EQUAL = @TRUE ; Initialize variable ON_EQUAL
    Switch EXD_Message ; Switch based on Dialog Message type
        Case MSG_INIT ; Standard Initialization message
;           DialogProcOptions (EXD_Handle, MSG_TIMER, 1000)
;           DialogProcOptions (EXD_Handle, MSG_BUTTONPUSHED, @TRUE)
           DialogProcOptions (EXD_Handle, MSG_CHECKBOX, @TRUE)
           Return (RET_DO_DEFAULT)

;       case MSG_BUTTONPUSHED ; ID "PushButton_OK" PushButton_OK
;           return (RET_DO_DEFAULT)

        Case MSG_CHECKBOX ; ID "CheckBox_1" CheckBox_1 MyCheckBox
           flag=AskYesNo("Example D", "Do you really want to change the
value of this checkbox?")
           If flag==@NO
               ;Set it back to what it was
               cbval=DialogControlGet (EXD_Handle, "CheckBox_1", DC_CHECKBOX)
; get new state
               DialogControlSet (EXD_Handle, "CheckBox_1", DC_CHECKBOX, !cbval)
; put back opposite state
           EndIf
```

Introduction to Programming

```
Return (RET_DO_DEFAULT)

EndSwitch ; EXD_Message
Return (RET_DO_DEFAULT)
#EndFunction ; End of Dialog Callback EXDCallbackProc
```

In Case MSG_INIT, the line:

```
DialogProcOptions ( EXD_Handle, MSG_CHECKBOX, @TRUE )
```

... ensured that any change to any checkbox in the dialog would cause this code to execute. i.e., either checking or unchecking the checkbox would invoke this provision with the `AskYesNo` function called to request confirmation.

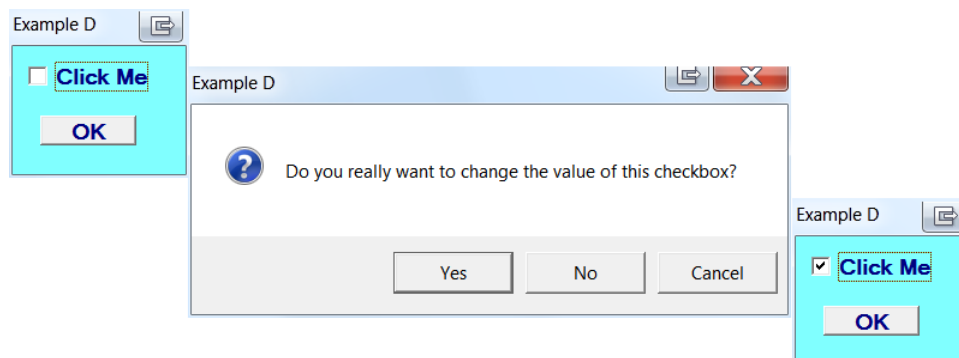
Note: in this example however, there will never be an option to uncheck the checkbox because checking the box – whether answered "Yes" or "No – already causes the dialog to return.

If the user responds "No", the `DialogControlGet` function is called with `EXD_Handle`, `EXD_ID ("CheckBox_1")` and the constant `DC_CHECKBOX` (defined in `InitDialogConstants` with a value of 1) and returns the present value of the checkbox (i.e., `@true` if checked or `@false` if not) as `cbval`.

Next, the `DialogControlSet` function is called – with the same three initial parameters plus a final argument: `!cbval` to set the new value of the checkbox. The exclamation mark (!) is the logical NOT operator and changes an `@true` to `@false` and vice versa so the value (setting) of the checkbox is cleared if set or set if previously cleared.

IMPORTANT TIP: In WinBatch Studio, if you click on a function name (shown in blue) and then hit Shift-F1, WinBatch Studio will open the help page for that function.

At this point, [Exercise D.wbt](#) is ready to run and you should see results like this:



Exercise_E

As a final exercise, [Exercise E.wbt](#) provides an example of how two (or more) checkboxes can be treated separately.

First, we've declared two different checkboxes (and also notice that `EXEEnumControls` has changed).

```
EXEFormat=`WWDLGED,6.2`

EXECaption=`Example E`
EXEX=9999 ; -01
EXEY=9999 ; -01
EXEWidth=068
EXEHeight=057
EXEEnumControls=003
EXEProcedure=`EXECallbackProc`
EXEFont=`Microsoft Sans Serif|7373|70|34`
EXETextColor=`0|0|128`
EXEBackground=`DEFAULT,128|255|255`
EXEConfig=0

EXE001=`005,005,056,010,CHECKBOX,"CheckBox_1",MyCheckBox,"&Confirm
Me",1,1,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EXE002=`005,023,056,010,CHECKBOX,"CheckBox_2",MyOtherCheckBox,"&But not
me",1,2,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EXE003=`015,039,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"&OK",1,3,DE
FAULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed=Dialog("EXE")

Message("CheckBox Values
are",StrCat("EXE001=",MyCheckBox,@CRLF,"EXE002=",MyOtherCheckBox))
exit
```

And we've changed the message reporting the results to show two different values.

Next, as with the previous examples, in Example E, the heart of the operation appears in the `MSG_CHECKBOX` case in the dialog procedure switch statement:

```
Case MSG_CHECKBOX
    Switch ON_EQUAL
```

Introduction to Programming

```
Case EXE_Name == "CheckBox_1" ; ID "CheckBox_1"  CheckBox_1
MyCheckBox

    flag=AskYesNo("Example D","Do you really want to change the value
of this checkbox?")

    If flag==@NO

        ;Set it back to what it was

        cbval=DialogControlGet(EXE_Handle,"CheckBox_1",DC_CHECKBOX) ;
get new state

        DialogControlSet(EXE_Handle,"CheckBox_1",DC_CHECKBOX,!cbval) ;
put back opposite state

    EndIf

    Return(RET_DO_DEFAULT)

Case EXE_Name == "CheckBox_2" ; ID "CheckBox_2"  CheckBox_2
MyOtherCheckBox

    Return(RET_DO_DEFAULT)

EndSwitch ; EXE_Name

Return(RET_DO_DEFAULT)
```

Here we have the `Switch EXE_Message` structure (familiar from previous examples) enclosing another, nested switch statement as: `Switch ON_EQUAL Case EXE_Name ==...` This inner switch statement allows differentiation between the checkboxes.

You should also notice that the switch – `ON_EQUAL` – is performed on a test `EXE_Name == ...` Since only one case will match, this allows a switch to occur where a simple:

```
Switch EXE_Name
    Case "CheckBox_1" ...
    Case "CheckBox_2" ...
```

... structure would fail (since a string can not be used as a case statement).

Exercise_F

A bonus example – see [Exercise F.wbt](#) – shows another nested `switch ... case ... switch ... case ...` format. Play with the code file and see what changes you can make, how they work and what effects you can get. Remember, the best way to learn – and understand – something is simply to get your hands dirty by doing something.

Summary

Dynamic dialogs offer a very useful method of extending static dialogs for much greater functionality by allowing dialogs to respond – interactively – to the user's actions. How you use these is an open question, limited only by your imagination ... and your skill in defining the responses ... as well as your skill with our next topic.

Thus far, you've learned a lot about how to build applications; now – in [Chapter 15](#) – it's time to learn how to debug your WinBatch applications. Don't skip that chapter. There isn't a programmer alive—professional or novice—who doesn't rely on debugging techniques and tools to make his or her programs function. For a programmer, debugging is just as routine a process as is filling a coffee cup (or grabbing a can of cola or whatever your tippie of preference happens to be).

CHAPTER 15 : WHEN THINGS GO WRONG*

DEBUGGING APPLICATIONS

Find out the cause of this effect,
Or, rather say, this the cause of this defect,
For this effect defective comes by cause.

(William Shakespeare, *Hamlet*)

debugging – The process of finding, locating, and removing logical or syntactical errors from a computer program. This can range from simply checking the results of calculations to locating obscure errors in logic that only occur under very specific conditions. (*The PC User's Pocket Dictionary*)

Debugging applications is simply a fact of life. Everyone does it, and those who claim they do not are simply as bad liars as they are programmers. Debugging is something to be ashamed of only if it is done badly.

Back in [Chapter 1](#), "The "Golem Principle," we stressed an important fact: **Computers do not think!** Using "The Sorcerer's Apprentice" as an analogy, we explained that the only thing that a computer can do is to follow instructions. A computer can accomplish its tasks very rapidly, efficiently, and patiently. But the computer does not take any initiative; it does not possess even a modicum of common sense. A computer will do something totally stupid **if this is what it has been instructed to do!**

The problem really is that the instructions we give to computers are composed of small details combined to form complex operations. Since the languages we use to create computer programs are not natural to us (you can't really carry on a conversation in a computer language), the fact that these instructions do not always perform in precisely the intended fashion shouldn't be any surprise.

And, when something fails, the fault is not the computer's. The fault is that the instructions we have provided are not sufficient to the task. All that the computer has done is to perform precisely as it was requested. Thus, the process of debugging is a matter of discovering where our instructions were inadequate and, equally important, why they were inadequate.

And, of course, along with debugging, there's also a little matter of correcting the problems once they've been found.

* ... and, rest assured, they will!

Learning to Debug

This chapter is organized as a series of exercises. The exercises are provided as programs where the titles for these programs take the form `Debug nn .wbt` and nn is the exercise number.

These exercises are not intended as tests; you will not be graded on your successful or unsuccessful solutions to the problems incorporated in each. Instead, these exercises are intended to show you some of the problems that can be found, help you learn how to identify the problems, and suggest how to correct the problems.

There will be a test, but in this classroom, the test is given by reality, not by your teacher. Therefore, the test comes later, when you create your own applications. And your grade is simply whether your applications perform or fail and, of course, how well they perform.

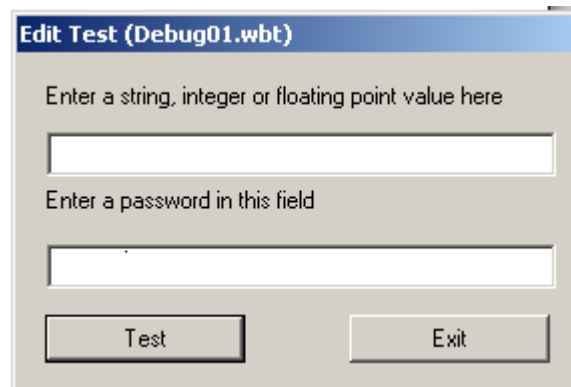
A solution is given for each of the exercises but, rather than looking ahead at the answers (since you can only cheat yourself by doing so), begin by trying to work through the exercise programs on your own and answering three questions:

- What's the problem?
- Why does the problem occur?
- How can the problem be corrected?

Debugging in an IDE

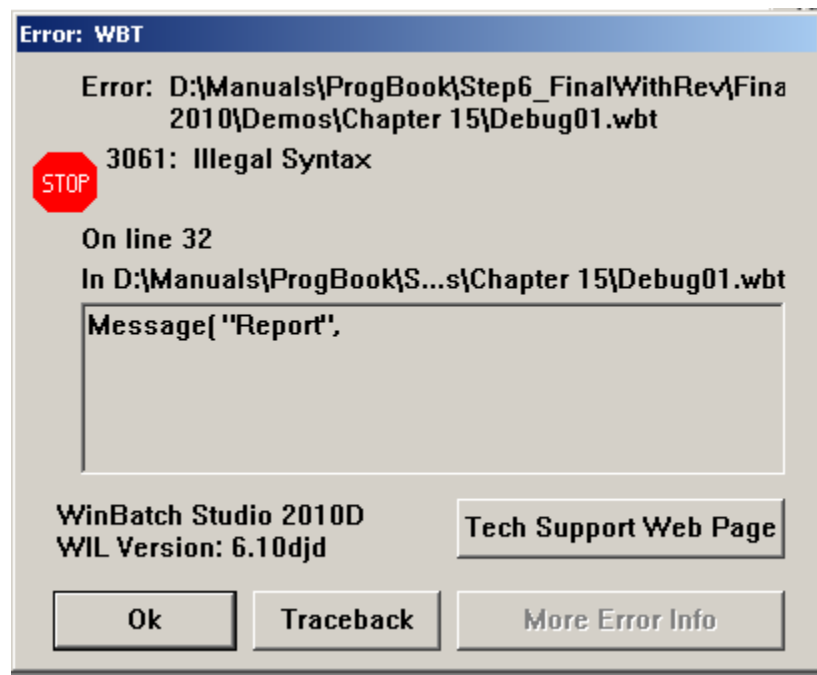
With WinBatch, as with most modern programming tools, the programming environment itself provides tools for debugging. These tools consist of several elements, including features that allow the programmer to execute a program while watching execution through the source code, stop execution at specific locations, and examine the contents of variables during execution.

For an example, let's begin with [Debug01.wbt](#). When we run the program in debug mode (`Ctrl-F7`), this dialog box appears:



Here, you're asked to enter a value in one field and to enter a password in the second. Presumably, you'll click on the Test button to proceed.

But, after clicking on Test, instead of a result report, you see:



Illegal Syntax? Cheeze-it, fellows! The code police are after you!

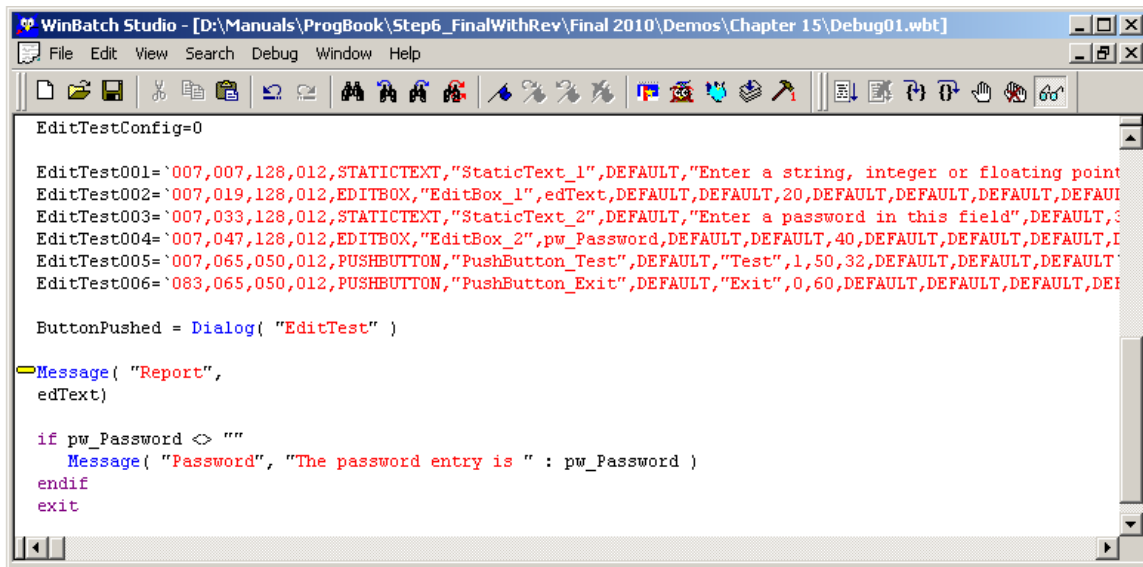
The terminology "Illegal operation," which features in a great many error messages, has caused many worried users to wonder what laws they had inadvertently broken.

In this case, the message is trying to tell you that the WIL Interpreter has encountered an instruction that does not follow acceptable (legal) rules of syntax, as a courtesy, the message is also reporting exactly which instruction: `Message("Report", ...`. For an experienced programmer, this may be a pretty clear explanation.

Except maybe you aren't an experience programmer and this isn't clear. So, what do you do? Go to the Tech Support Web Page?

That's a little drastic for this type of error. If you simply click Ok in the message box, WinBatch will take you to the exact point in the source code where the error occurred:

Introduction to Programming



A screenshot of the WinBatch Studio interface. The title bar reads "WinBatch Studio - [D:\Manuals\ProgBook\Step6_FinalWithRev\Final 2010\Demos\Chapter 15\Debug01.wbt]". The menu bar includes File, Edit, View, Search, Debug, Window, and Help. The toolbar contains various icons for file operations, editing, and debugging. The main text area displays a script with the following content:

```
EditTestConfig=0


EditTest001=`007,007,128,012,STATICTEXT,"StaticText_1",DEFAULT,"Enter a string, integer or floating point
EditTest002=`007,019,128,012,EDITBOX,"EditBox_1",edText,DEFAULT,DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAUI
EditTest003=`007,033,128,012,STATICTEXT,"StaticText_2",DEFAULT,"Enter a password in this field",DEFAULT,3
EditTest004=`007,047,128,012,EDITBOX,"EditBox_2",pw_Password,DEFAULT,DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,I
EditTest005=`007,065,050,012,PUSHBUTTON,"PushButton_Test",DEFAULT,"Test",1,50,32,DEFAULT,DEFAULT,DEFAULT`
EditTest006=`083,065,050,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit",0,60,DEFAULT,DEFAULT,DEFAULT,DEI

ButtonPushed = Dialog( "EditTest" )

Message( "Report",
edText)

if pw_Password <> ""
    Message( "Password", "The password entry is " : pw_Password )
endif
exit
```

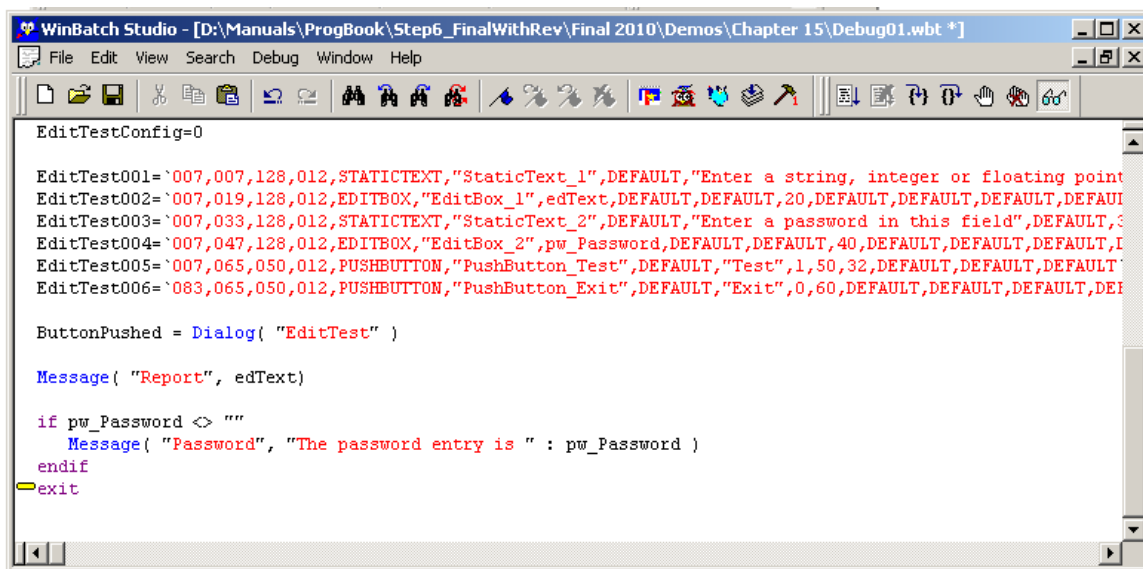
A yellow blip (marker) is positioned to the left of the `Message("Report", edText)` line, indicating a syntax error.

Now you can see a bit more of what's happening. Because this is an interactive debugger, the yellow blip () at the left is a marker showing exactly where in the code the error occurred. Where older debuggers provided a list of line numbers where errors occurred, an IDE takes you directly to where to look for the mistake.

Since you don't have to go hunting for the problem, does anything about the instruction strike you as an error?

Do you need a hint? Or did you remember the WinBatch restriction (which doesn't apply in many languages) requiring command instructions to appear on a single line?

So, how about correcting this error so that it reads:



A screenshot of the WinBatch Studio interface, similar to the previous one, but with the script corrected. The title bar and menu bar are the same. The main text area displays the following script:

```
EditTestConfig=0

EditTest001=`007,007,128,012,STATICTEXT,"StaticText_1",DEFAULT,"Enter a string, integer or floating point
EditTest002=`007,019,128,012,EDITBOX,"EditBox_1",edText,DEFAULT,DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAUI
EditTest003=`007,033,128,012,STATICTEXT,"StaticText_2",DEFAULT,"Enter a password in this field",DEFAULT,3
EditTest004=`007,047,128,012,EDITBOX,"EditBox_2",pw_Password,DEFAULT,DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,I
EditTest005=`007,065,050,012,PUSHBUTTON,"PushButton_Test",DEFAULT,"Test",1,50,32,DEFAULT,DEFAULT,DEFAULT`
EditTest006=`083,065,050,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit",0,60,DEFAULT,DEFAULT,DEFAULT,DEI

ButtonPushed = Dialog( "EditTest" )

Message( "Report", edText)

if pw_Password <> ""
    Message( "Password", "The password entry is " : pw_Password )
endif
exit
```

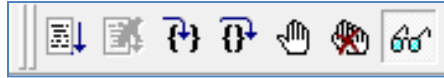
The yellow blip is no longer present, indicating that the error has been resolved.


Next, after doing so, run [Debug01.wbt](#) again...problem solved.

Debugging Tools

Debugging tools were introduced back in [Chapter 2](#) during your tour of the WinBatch Studio IDE. These are the tools you will need to debug not only the sample applications, but your applications as well.

The debug tools appear in the IDE toolbar:



Beginning at the left, the first tool is the Go button . Clicking on the Go button initiates execution of the application under development. If you are using the compiler version of WinBatch, the `Go` button ensures that execution takes place in debug mode.

The differences between debug and normal execution are several. Where normal execution simply runs until the application exits (or until a serious error occurs), debug execution permits you to do the following:


- Interrupt execution at preset breakpoints.
- Terminate execution on errors.
- Watch the values stored in variables.
- Change values stored in variables during execution.
- Step between breakpoints.
- Step through instructions individually.
- Step into subroutines.

Of course, if there are no errors and if no breakpoints have been set, the application will execute in debug mode exactly the same as in normal execution—that is, it will run until normal termination.

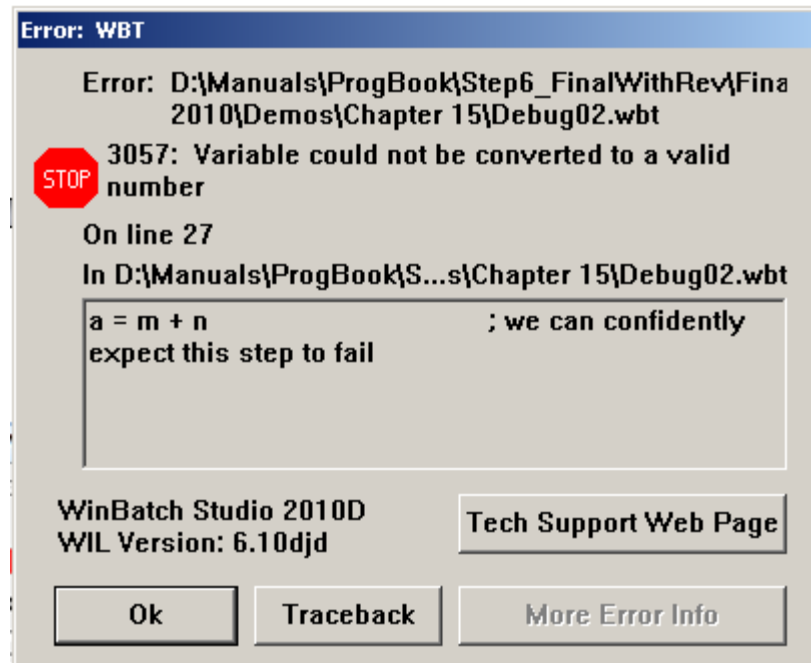
Debugging during Execution

Perhaps the easiest way to learn to debug is simply to debug an application. For an example, let's look at the demo program [Debug02.wbt](#).

This demo program is actually the same as the [VariTest.wbt](#) program used in [Chapter 4](#). Since it contains a deliberate operation error—as opposed to an error in instructions—it is an excellent example of how the debugger can interrupt execution when it encounters an error.

First, open WinBatch Studio. Next, load the [Debug02.wbt](#) program. And, finally, click on the Go button  to execute the program.

The first four steps in [Debug02.wbt](#) will execute normally. The fifth step, however, will not. Before it can be reached, an operational error will halt execution, and you'll see this message:



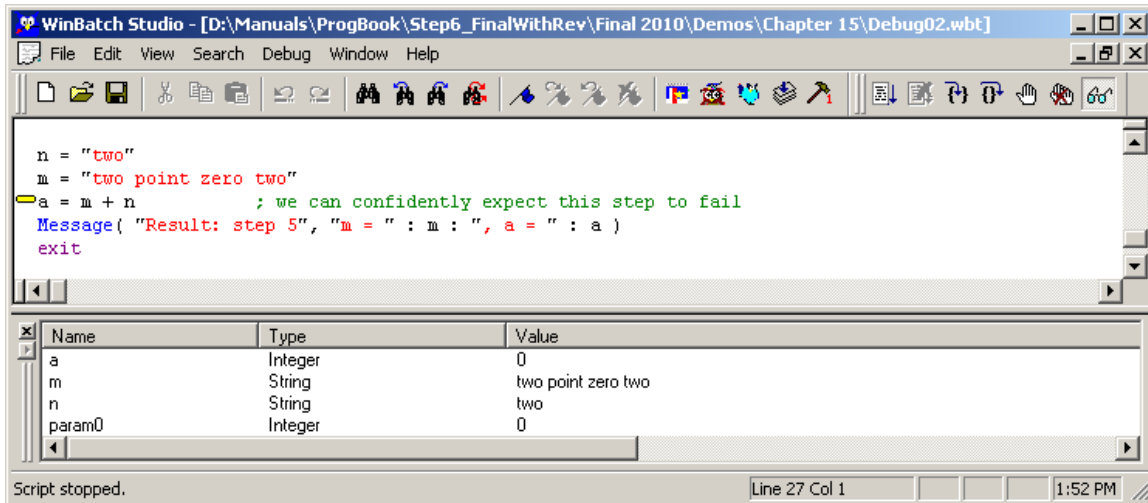
As the code remarks (in the comment), this step is expected to fail. The point isn't the failure per se but to discover why this step is invalid.

Notice the grayed-out `More Error Info` button in the image above? Not all error messages have the `More Error Info` button activated. It is only activated when there is extended error information which – usually – contains a Windows System level error.

Sure, there's a blatant clue right there in the title bar: "Variable could not be converted to a valid number." But what variable? And why not? And how are we going to find out?

Well, the last question is the easiest to answer ... and also the first step in finding the answers to the previous two questions. Therefore, the first step is simply to click on the `Ok` button.

When we do, WinBatch Studio takes us to the editor:








Here the yellow blip [] shows the instruction where execution was interrupted, and the bottom pane (the Watch window) shows the current values for the `a`, `m`, and `n` variables. (The remaining variable, `param0`, can be ignored since this was the number of arguments passed to the program when it started—that is, none.)

Notice that `a` is zero at this point because the addition operation failed. The real clue, however, lies in the `m` and `n` variables, both of which are string arguments. Since WinBatch doesn't support an addition operator (+) for strings ... well, doesn't this look like a probable cause?

Yes, in fact, this is precisely the cause. Since WinBatch doesn't know how to add two strings, execution has been interrupted on an error.

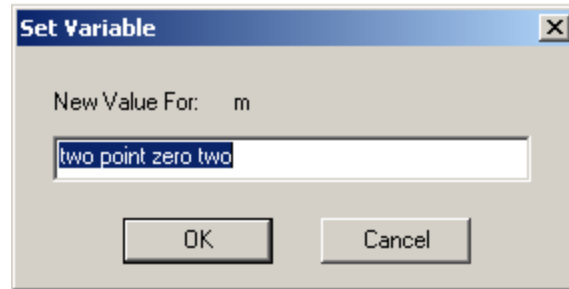
But we're not finished yet.


First, with the text cursor positioned on the line where the failure has occurred, click on the Breakpoint button . This action will insert a breakpoint (identified either as  or  at the left column) in the code. On the next execution, the program will stop at this point but will do so before an error occurs.

In addition to setting a breakpoint, the Breakpoint button  also toggles a breakpoint setting. By positioning the cursor on a line where a breakpoint is set and then clicking on the button, the existing breakpoint is cleared. The Clear All Breakpoints button  will remove *all* breakpoints from the program code.

Next, run the program again and proceed until the breakpoint is reached. This time, the WinBatch Studio editor will look the same as before (with the addition of the breakpoint marker), except that no error report has been issued because the instruction causing the error has not been executed yet.

Now, in the Watch window, double-click on the `m` variable. This will bring up the Set Variable dialog:



If the `Watch` window is not visible in WinBatch Studio, click on the Watch button  at the right of the debug toolbar to open the window.

In the Set Variable dialog, change the value in the edit box to a numerical value, such as 2.3, and then click on OK. Do the same for the `n` variable.


Finally, click on the Go button again. The program will resume execution from the breakpoint. This time, the program will run without errors, because the contents of the variables have been changed to values acceptable for an addition operation.



Using the `Watch` window and the `Set Variable` dialog is particularly useful for setting values in variables to test extreme conditions during execution and to test whether error traps and error correction provisions perform as expected.

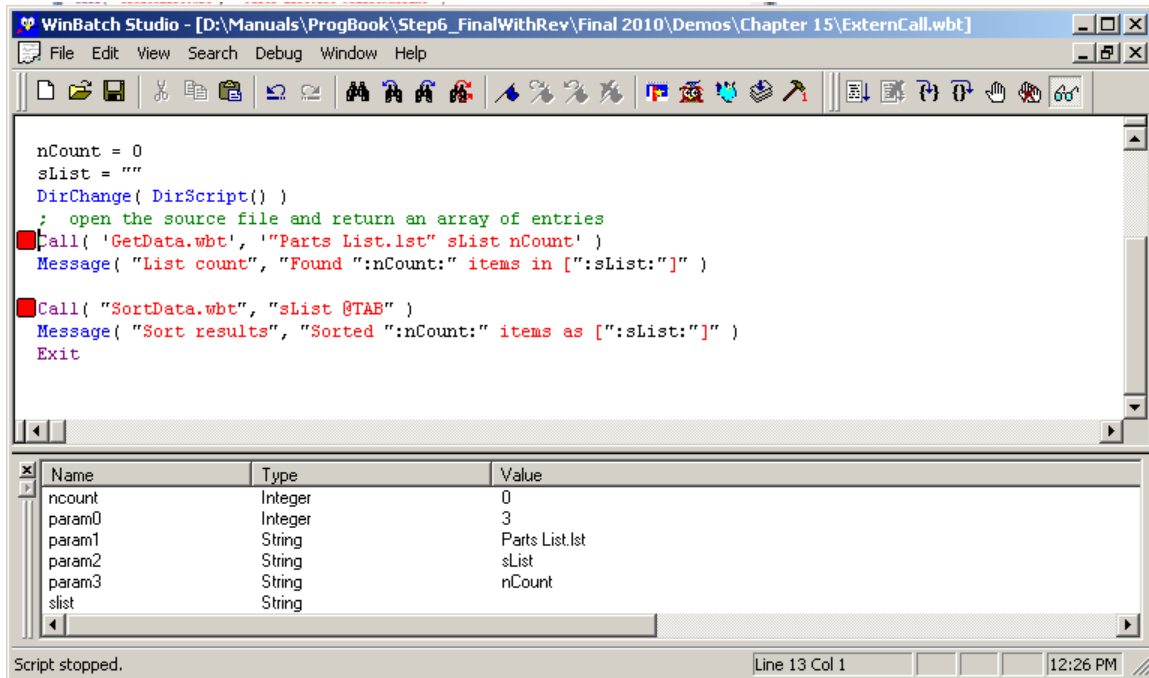
Tracing Execution Step by Step


Quite often, the important question isn't where an error became fatal but how the error condition or erroneous data came to exist in the first place. For this purpose, instead of relying on a single breakpoint to halt execution at the error point, we set a breakpoint prior to the error and then watch what is happening as we proceed in step-by-step execution.

To control execution on a step-by-step basis (one instruction at a time), WinBatch Studio provides two controls: Step Into and Step Over.


The Step Over button  executes the next instruction, but if the instruction is a `gosub`, `call`, execution does not trace into the subroutine or external program.

In contrast, the Step Into button  traces execution by stepping into subroutines and external programs. For an example of how each of these work, load the [ExternCall.wbt](#) program and then use the Breakpoint button  to set breakpoints in the program as shown:



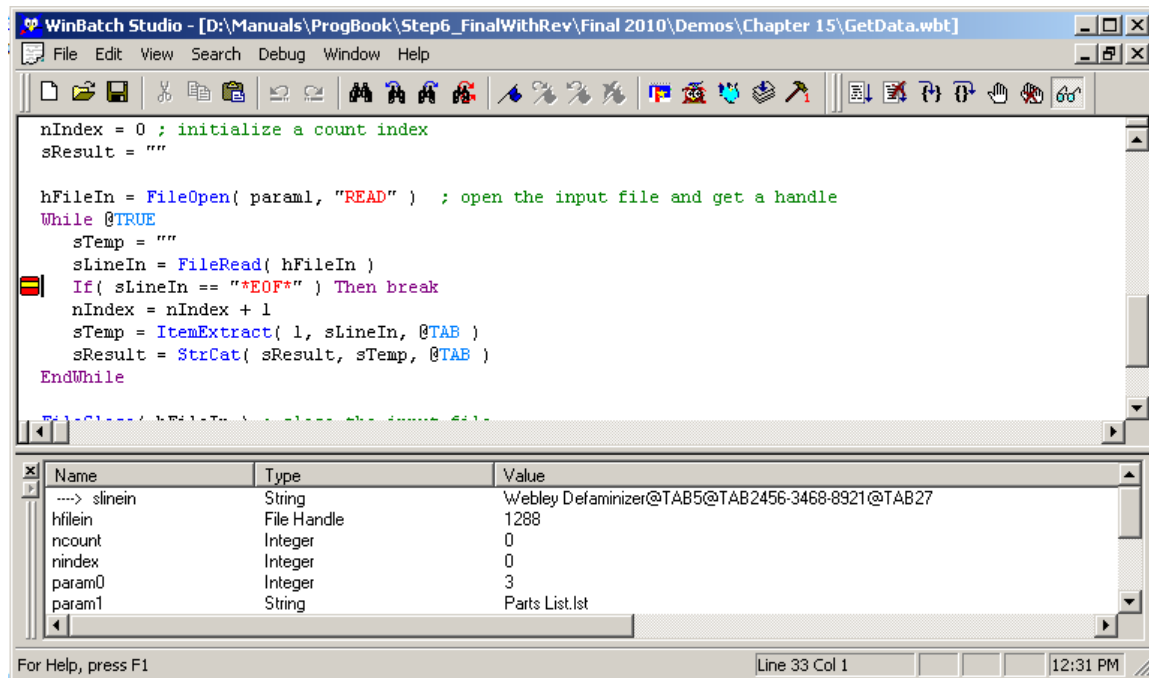
Now execute the program. Then, when execution halts at the first breakpoint, click once on the Step Into button . This will step into the external [GetData.wbt](#) program.

At this point, either the Step Into or the Step Over button will walk you through execution one step at a time. As you step through the program, take note of the variables in the Watch window and how the values shown change as instructions are executed.

If you want to speed up the debugging process simply click the Go button .

For example, in the following image, the program has traced execution from the [ExternCall.wbt](#) program into the [GetData.wbt](#) program and is now paused after reading a line of information from the external data file.

Introduction to Programming



In the Watch window, the `slinein` variable shows the line that was just read from the file.

The Watch window always shows variable names in full lower-case irrespective of how these names appear in the source file.

By placing a breakpoint within the `while` loop where the list file is being read, we have a convenient way to watch what is happening while the file is open. Then, instead of using the Step Into or Step Over buttons to walk through execution, we can simply click on the Go button each time to cycle through the loop and back to the same point.

In many cases, this can be exactly the type of procedure we'll use to find out why an error occurred. By watching the data—the contents of the variables—on each loop, we can get a sense of how an error happens and what provisions might be needed to ensure that the error doesn't happen in the future.

Terminating Execution

While debugging an application, we do not necessarily need to allow the program to terminate normally. And, sometimes, if the bug is bad enough, it may not even be possible for the program to end normally.

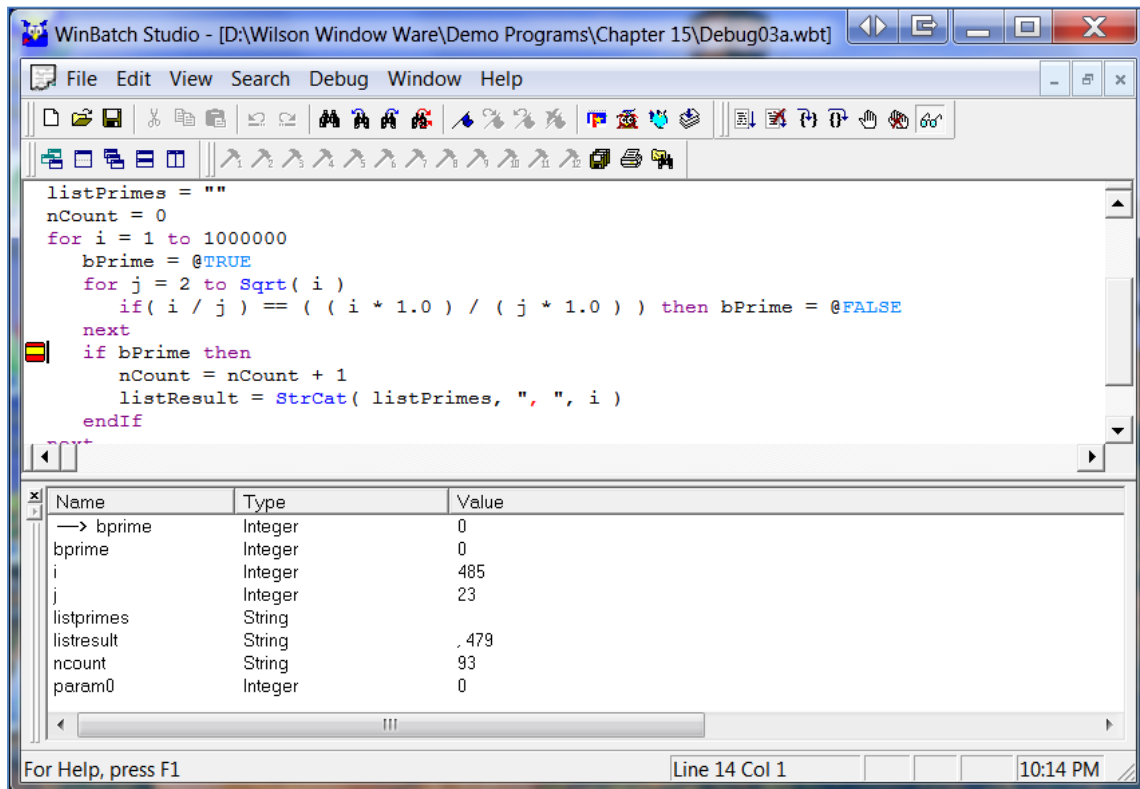
As an example, let's look at the [Debug03a.wbt](#) program, which executes a simple search for prime numbers. Of course, like the other demo programs discussed in this chapter, [Debug03a.wbt](#) has a couple of shortcomings.

One is that it is slow—deliberately so, to tell the truth. A second is that the program contains a serious error. And, if these weren't enough, there's an inherent problem to boot.

Because the program is slow, we're not likely to see the serious error until a long time after the program begins execution. In fact, the problem won't be visible until it finishes, and, even then, the cause of the error may not be immediately obvious.

But the point is that we don't need to wait forever to examine what is happening within a program. Instead, by inserting a breakpoint during execution, rather than before starting, we can pause the program and take a look at what's going on.

Here, we have a snapshot of the program during execution:



In this snapshot, we can see a number of things. From the Watch window, we can see that the program has tested integers up to 485, that we've found 93 prime numbers thus far, and, if you're sharp, you should see the error, too.

This is not an artificial error. This error came about purely by accident while your author was trying to decide how to best illustrate an error. But accident or not, it's a valid error and a good illustration of the value of debugging.


Have you spotted it yet?

If not, then look at the value of the listPrimes variable, which was intended to store a list of the prime numbers found. Except that it's empty, right? And what's this listResult variable which contains a very short string?

Initially, the name listResult was going to be used as the list storage variable ... until I decided to change it to listPrimes as a more representative label. The error is that I failed to change all occurrences.

Introduction to Programming

So now we've avoided waiting – potentially for hours – until the program was finished in order to find out that there was a serious flaw. But, having recognized the flaw, do we need to wait for completion before repairing the error?

No, we can click on the Stop Debugging button  to bring execution to an immediate halt.

The Stop Debugging button will interrupt the program at any point. It isn't necessary to use a breakpoint to pause execution first.

When you click on the Stop Debugging button, a dialog will ask if you want to stop the script. So, if you clicked on the button by accident, you can choose No to resume execution.

At this point, you know about the basic tools for debugging an application. But there's more to debugging than simply interrupting execution and examining variables.

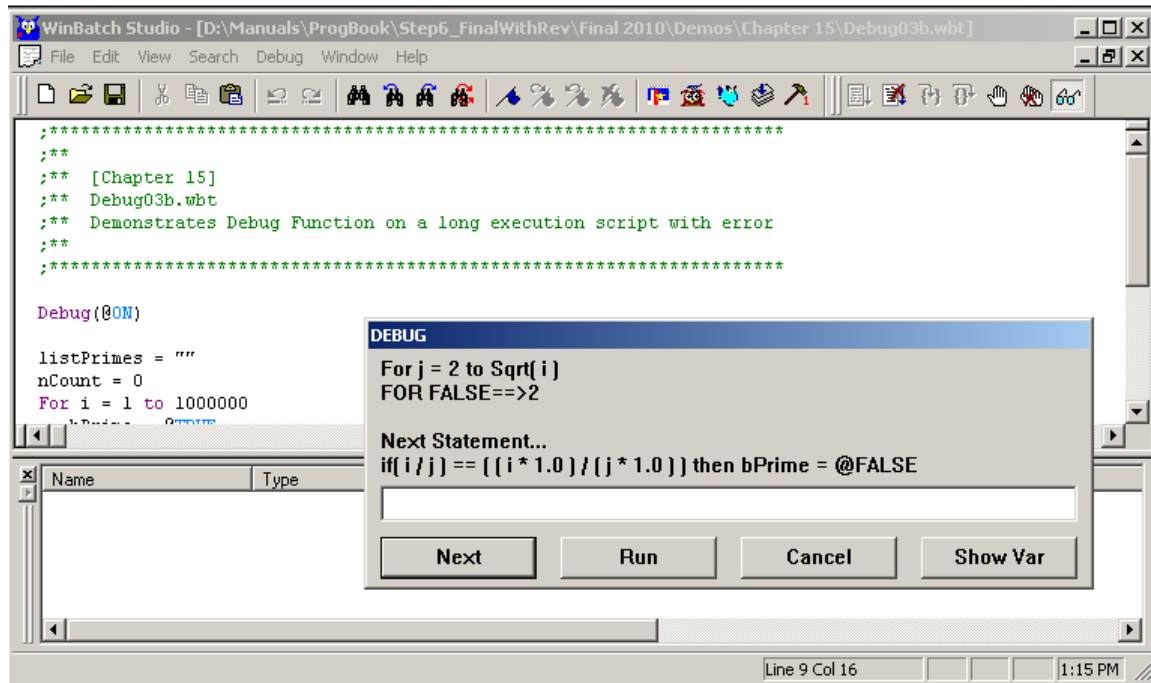
Debugging Options

WinBatch has a number of useful debugging utilities which can make troubleshooting easier. Before proceeding with debugging operations, give these utilities consideration and then think about how you can use them – as alternatives – in the following examples.

Debug

When Debug is initialized, a dialog box which controls the execution of each statement is displayed. Debug works line by line through the script, displaying the current statement, its value and the following statement. The script will also be executed in conjunction with the display of statements. Initialize Debug by adding `Debug (1)` or `Debug (@ON)` to a specific point in your script.

As an example, using [Debug03b.wbt](#), add the command `Debug (@ON)` at the start of the program. Now, when you execute the program, the Debug dialog will appear (see following) and you can use the `Next` button to step through the program until you reach an interesting point in the execution.



Now, when you're ready, enter a variable name – such as `listprimes` – in the edit window and then click the `Show Var` button and you will see the current value of the requested variable.

Okay, now it's your turn to perform your own debug operation so go back to the [Debug01.wbt](#) or [Debug02.wbt](#) examples and try using the Debug operation ... and try to decide where or how to trigger the Debug execution for the best operation.

Tip: think about using a Boolean IF test to trigger Debug at an appropriate point.

DebugTrace

DebugTrace will create a file showing each line executed and the return value of the line. It will consume considerable processing time and is generally reserved for particularly hard-to-debug problems. DebugTrace will append the debug data to the end of the file you specify. You may want to delete the file, if it already exists.

DebugTrace

Controls and customizes the WIL debugging trace logs.

Syntax:

```
DebugTrace ( request-code [ ,parm1 [, parm2 ]] )
```

Parameters:

(i) request-code: numeric code indicating desired action (see below)

Introduction to Programming

(s/i) parm1: [optional] depends on requestcode

(s/i) parm2: [optional] depends on requestcode

Returns:

(i) DebugTrace mode state: 0 = Off 1 = Statement by statement Tracing enabled.

To debug into a 'Called' WinBatch script, User Defined Function or User Defined Subroutine, make sure to add the corresponding debug command, to the 'called script', User Defined Function or User Defined Subroutine.

IntControl (71, p1, p2, 0, 0) can be used to dump WIL and extender function tables to the debug log file.

Request codes:

Note: For requests which take a file name, if "filename" is "*DEBUGDATA*" then debug output will be written to the system debugger using the OutputDebugString Windows API (see WIL `DebugData` function).

Request codes fall into one of three categories:

Modes	These control the DebugTrace line by line logging.
Mode Option Flags	These control the log file names, formatting, and content.
Immediate Action Codes	These cause additional data to be immediately written to the Debug Trace log file.

Modes:

Request code	Meaning	Parm1
-1	Returns previous trace mode	
0 (@OFF)	Stops statement by statement debug tracing.	
1 (@ON)	Starts or resumes statement by statement debug tracing.	Optional filename to set or change the current debug trace log file. If no previous debug trace file has been specified, then a filename is required. Output filename may be specified by a previous @ON, 10, 100, or 101

		requestcode.
10	Same as 1 (@ON) with the addition that if the output file exists, one attempt will be made to delete the file before continuing.	
22	<p>Allow <code>DebugTrace</code> continuation (inherit the debug mode from the caller).</p> <p>By default, when the code enters a UDF, UDS or Call'ed script file, statement by statement debugging is suppressed until the script returns.</p> <p>Adding <code>DebugTrace(22)</code> to a UDF, UDS, or called script will resume statement by statement debugging *IF* it was active on entry.</p>	

Mode Option Flags:

Request code	Meaning	Parm1
100	Specify output file name for subsequent debug trace logging.	Required filename parameter
101	Same as 100 with the addition that if the output file exists, one attempt will be made to delete the file before continuing.	
102	<p>Dump WIL variable table to the debug trace log file when a terminal error occurs.</p> <p>(Replaces option 2 in version 2006A and older)</p>	<p>0 - Do NOT dump variable table to debug trade log file. (default)</p> <p>1 - Dump WIL variable table to the debug trace log file if a terminal error occurs.</p>
103	Dump internal debug data for each keyword lookup.	<p>0 - Do NOT dump debug data for keyword lookups. (default)</p> <p>1 - Dump debug data for keyword lookups.</p>
104	Suppress statement timestamp information.	<p>0 - Include statement timing information (default).</p> <p>1 - Suppress statement timing information.</p> <p>note: This option is useful when creating multiple debug trace log files that you plan to compare with one another using a file-compare type program. As statement timing will vary slightly, this option allows statement timing information to be skipped to avoid non-relevant differences in the generated log files.</p>

Introduction to Programming

Immediate Action Codes:

Note: `DebugTrace (@OFF)` does not suppress the execution of these codes.

In the case of where a code would output information to the debug trace log file, if the file is defined, then an attempt will be made to write data to it.

Request code	Meaning	Parm1
200	Reset. Turns off statement by statement tracing, and resets all the Mode Option Flags to the default values	Optionally set new debug trace file name. If there is no optional parm parameter, or it is set to a null string, the output filename parameter will be cleared.
203	Write a string to the debug trace log file.	Specifies a string to be written to the debug trace log file.
204	Write a string, surrounded by "\$" signs to the debug trace log file.	Specifies a string to be written to the debug trace log file.
205	Dump WIL and loaded extender function tables to the debug trace log file.	
206	Dump WIL variable table to the debug trace log file. Includes values for simple variable types. Note: Only the first part of long strings will be dumped. Complex variable types (Arrays, Binary Buffers, etc) will not have variables dumped.	
207	Dump stack info to the debug trace log file.	
208	Dump machine information block.	
277	Dump formatted internal debugging information to the debug trace log file. Much of the same information as provided by <code>IntControl 77</code> is available.	Missing or "" - Display select internal information Numeric value - Displays more detailed information regarding a specific <code>IntControl 77</code> request.

Further, to debug into a 'Called' WinBatch script, User Defined Function or User Defined Subroutine, make sure to add the corresponding `DebugTrace` command, to the 'called script', User Defined Function or User Defined Subroutine.

Here is the [Debug03c.wbt](#) program – using the `DebugTrace` function – for you to play with.

```
listPrimes = ""
```

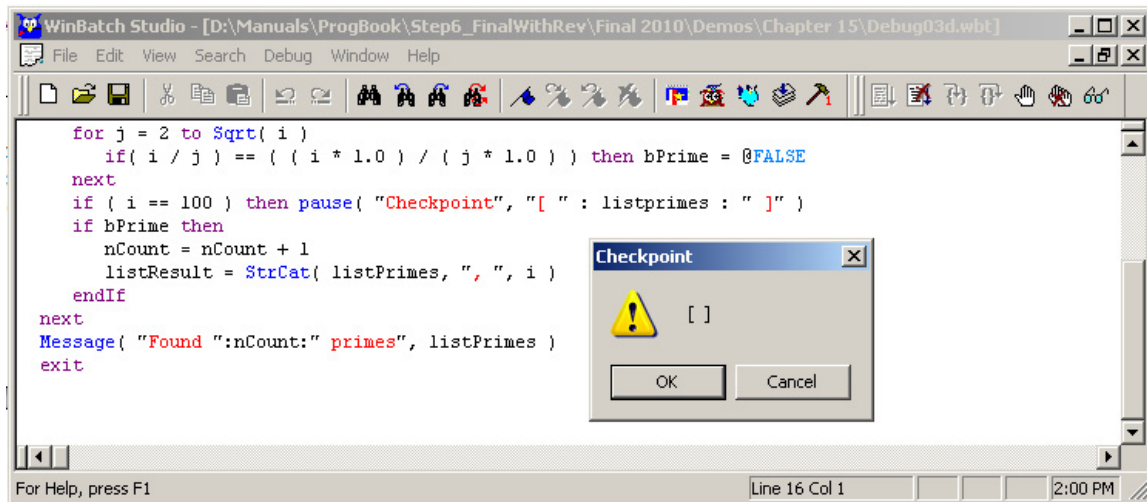
```

nCount = 0
For i = 1 to 1000000
    bPrime = @TRUE
    For j = 2 to Sqrt( i )
        if( i / j ) == ( ( i * 1.0 ) / ( j * 1.0 ) ) then bPrime = @FALSE
    Next
    If bPrime then
        DebugTrace( @ON, "TRACE.TXT" )
        nCount = nCount + 1
        listResult = StrCat( listPrimes, ", ", i )
        DebugTrace( 203, listprimes ) ; variable info as string
    Else
        DebugTrace( @OFF )
    EndIf
Next
Message( "Found ":nCount:" primes", listPrimes )
exit

```

Message or Pause Statements

Properly placed Message or Pause statements can be used sporadically through out the script to display values of variables. Here's an example – version [Debug03d.wbt](#) – where we've set a trigger for the 100th iteration of the primary loop.



The fact that our checkpoint report shows `listprimes` as an empty string ("[]") is another indication of what's going wrong. The `Pause` function could report more than one variable or we could simply exchange the reported variables until we find the information desired; i.e., the error point.

DebugData

DebugData writes data via the Windows OutputDebugString function to the default destination. The function is generally only useful if you have the proper tools and hardware to debug Windows applications. In general, for standard retail Windows, the default destination is COM1. The Windows SDK provides tools (DBWIN) to allow you to capture the debug data to an alternate device or to a special window.

For users without sophisticated (and expensive) debugging tools, the `WIL Debug`, `DebugTrace`, `Pause` or the `Message` function work incredibly well.

Debugging as a Process

The process of debugging an application can take several forms. In one sense, debugging is an art form; successful artists in the field can command excellent salaries.

For those who are not debug artists, however, there are a series of relatively simple rules that will locate the majority of the possible problems.

Debug Step 1: Run the Program

The first step in debugging is simply to run the program and to watch for anything unexpected.

This probably sounds easy enough, and chances are, you will catch the majority of the possible errors. For one thing, quite a few errors will catch themselves when the program halts on encountering them.

Also, those errors in the execution, rather than in the code, tend to be quite obvious. Well, at least they're obvious in the sense that something is blatantly wrong, even if it's less easy to determine exactly where and precisely why. Still, the obvious offers a starting point, and the rest of the job is simply patience and observation. And, of course, testing.

Debug Step 2: Test the Elements

The second step in debugging is to ensure that all possible elements have been tested. Earlier, in the [Debug02.wbt](#) example, you were challenged to find the error by testing it during execution, not by looking for it in the code. Were you successful?

That wasn't that difficult of an exercise.

The point is that a proper debugging operation must exercise every element of a program. However tedious, all elements—including all cases, conditions, value types, and possible operations—must be tested.

Testing by Destruction

This could be called the "Dennis the Menace" approach, since it's modeled on the old truism that the best way to find out if something can be broken is to give to a kid to play with.

For debugging a program, if you have kids handy, then use them. They are the ideal instruments of torture, they have the impatience to thoroughly trash something, and they are blessed with unfettered imaginations. But, if you don't have a kid available, ask your spouse, parent, secretary, boss, or someone from the mailroom.

The primary qualification for your testers is that they know nothing about your program. If they also know nothing at all about programming, this is a bonus.

To begin:

Tell them—*briefly*—what you hope the program will do. (*If you say too much, you'll lose their value as testers by supplying them with your own expectations and preconceptions.*)

Ask them to:

- Make a list of anything that seems awkward.
- If an error occurs, note what they were doing.

Walk away. If you can't bring yourself to leave, stay silent and don't interfere.

When an error occurs:

- Do not under any circumstances tell your tester what it was that he or she should have done.
- Do make a note of what the tester did do and think about how to correct the program so that action will not cause an error again.
- Listen carefully. This is likely to be the best feedback you will have. And it is happening at the best of times—before the program's flaws become obvious to everyone.
- Say thank you, and if the error is less than fatal, ask your tester to continue.

In this fashion, since your tester doesn't know your expectations and has no preconceptions about what the program expects and is ready to accept, the code and design will get the best possible workout.

Remember, "the fault, dear Brutus, lies not in our stars but in ourselves ..." or, anyway, in our programs.

Debug Step 3: Watch Branches, Tests, and Variable Tests

Make a list of everywhere in the program that tests are performed on variables or input information or where branches occur. Then, set breakpoints to halt execution at these locations, and run through the program again. But this time, use the `Set Variable` feature in the `Watch` window to ensure that all possible branches, tests, and variable types have been executed. And, of course, make sure that they work appropriately.

We can't guarantee that these steps will catch every possible error—the Gods of computers, as well as those of men, frown on perfection. However, they should help you trap and eliminate almost all bugs.

Unavoidable Bugs

Not a very reassuring thought, perhaps, but sometimes there are bugs that are not the responsibility of the programmer. And, worse yet, these can also be the most difficult to identify.

The difficulty arises because programmers, particularly experienced programmers, have an acquired (and justified) mindset that tells them two things:

- The bug is going to be the result of some error made by the programmer.
- The bug can be identified and corrected using conventional and familiar practices.

However, when neither of these conditions apply — because the bug is in the programming language used, in some external library, or in the system itself — the bug becomes much more difficult to correct, even when it is easily recognizable.

Sheer complexity and the mass of code ensures that all computer languages contain some bugs. Fortunately, these are usually obscure and rarely encountered. Unfortunately, when these are encountered, they are often tenacious and difficult to deal with.

Summary

The biggest secret to debugging is simple: Keep an open mind and observe what is actually happening. Don't assume anything. Or, as Sherlock would phrase it, "It is a prime mistake to theorize in the absence of data."

The process and practice of debugging is not something that can be taught easily or quickly. It depends on your familiarity with the language being used (10 percent), your experience in programming (10 percent) and, for the remaining 80 percent, pure sweat, work, and attention to detail.

In this chapter, you walked through a couple of simple examples as a prelude, and then you received a guided tour of a much uglier example. The principles, however, in all of these have been the same: Watch and observe what is actually occurring and relate the observations to what was expected to happen but did not.

Of course, 99 percent of the time, what you will encounter will be simple errors. Typos and simple syntax errors. These are the kind of things that demand corrections but don't really require debugging.

Still, every now and then, the big one will bite you. When it does, the practices discussed and illustrated here will be the path to your solution.

But enough about problems. In [Chapter 16](#), to conclude this book, we'll look at methods of expanding WinBatch's capabilities by using the WinBatch Language Extenders.

CHAPTER 16 : WINBATCH EXTENDERS

ADDING EXTENDED AND CUSTOM FUNCTIONS

dynamic link library – Abbreviated DLL. A program module that contains executable code and data that can be used by an application program, or even by other DLLs, in performing a specific task.

In the previous chapters, you've seen that WinBatch provides a broad choice of functionality for a wide variety of purposes. However, it's possible that you may require additional capabilities for particular applications. To provide even more functions, WinBatch offers a series of language extenders, as well as the ability to call external dynamic link libraries (DLLs).

Custom extender DLLs may add nearly any sort of function to the WIL language, from the mundane network, math, or database extensions to items that can control fancy peripherals, including laboratory or manufacturing equipment. We'll talk about custom DLLs briefly at the end of the chapter.

As "language extenders," WinBatch provides 32-bit versions of a number of DLLs. We'll describe the many of the extenders later in the chapter and begin with the more general-purpose WILX extender.

The WILX Language Extender

To use the WILX extender, your WinBatch application needs to include the following `AddExtender` instruction:

```
AddExtender ( "wilx44i.dll" )
```

The `AddExtender` function returns `@TRUE` if the function succeeds or `error` on failure.

For each extender (DLL) used, only one call to the `AddExtender` function is required. These function calls normally appear at the beginning of the program. Also, the `AddExtender` function must be executed before attempting to use any functions in the extender library.

The WILX language extender is a potpourri of functions. Here, we provide brief descriptions of these functions to offer you some idea of the available options and services. For more details on any of these functions, refer to the documentation in the extender specific help files.

Introduction to Programming

Getting Library Information

The `xExtenderInfo` function returns simple information about the library:

```
xExtenderInfo( request_# )
```

WinBatch Studio's syntax color highlighting makes all WIL Extender functions **Pink**. Whereas, all standard WIL functions are colored **Blue**.

The request number argument can be:

0	Returns the version number for the .DLL
1	Returns the number of functions supported
2	Returns the number of constants recognized

Of these, the use of the version information is obvious, but the remaining two reports may appear more curious than functional. However, the information may be important because of WinBatch's limitations on additional items.

A single extender can up handle up to 500 functions. It is only limited by WIL which allows you to load a total of 500 extender functions or 10 Extenders, whichever comes first.

When creating your own custom extenders, the documentation for the functions added through DLLs should be supplied in a separate help file accompanying the extender DLL. Information about all of the WinBatch extenders is provided as online documentation(.CHM help files). For example, the WILX functions and their arguments are available in the online documentation found in the WILX.CHM file.

The WIL Extenders provided by Wilson WindowWare are also included in the ConsolidatedWIL.CHM. Wilson WindowWare offers several different help files depending on what is installed on the system. The Consolidated WIL Help file acts as a single resource for many of the various help files. It combines multiple .CHM files at run time, allowing them to all be consolidated into a single Help system. So, as you install some new Wilson WindowWare product (i.e. WIL Extenders) you will see them show up in the table of contents of this help file.

This help file can be accessed from the Windows Start menu (Start | Programs | WinBatch) or from WinBatch Studio's context menu by clicking the right mouse button anywhere within an open file.

Converting between Numeric Systems

While WinBatch functions accept only decimal arguments, WILX offers two functions for performing conversions between numeric systems.

The `xBaseConvert` function performs conversions between different numeric bases:

```
xBaseConvert( value, from_base, to_base )
```

The `xHex` function converts from hexadecimal to decimal:

```
xHex( hex_val )
```

Accessing Drives

Three WILX functions provide extended disk/CD access.

The `xDiskLabelGet` function returns the volume label of the specified drive:

```
xDiskLabelGet( drive )
```

This function is useful for identifying a CD or ensuring that the correct CD is in the drive. It also can be used with other removable media.

The `xDriveReady` function checks whether the drive is ready:

```
xDriveReady( drive )
```

For example, you might use it to check whether a CD is ready for access.

Finally, you can use the `xEjectMedia` function:

```
xEjectMedia( drive )
```

This function is useful for ejecting media from CD (it's not applicable to hard drives or floppies).

Accessing Windows API Functions

For those familiar with conventional Windows programming, WILX provides three functions offering direct access to heavily used Windows API functions.

To get the handle of a child window belonging to a specified parent, use:

```
xGetChildHwnd( parent-hwnd, child-text, child-seq )
```

The other two are the "real" `MessageBox` and `SendMessage` API functions:

Introduction to Programming

```
xMessageBox( title, text, style )  
xSendMessage( hWnd, msg, wParam, lParam )
```

Verifying Credit Card Numbers

Perhaps one of the more interesting of the WILX provisions is the `xVerifyCCard` function:

```
xVerifyCCard( cardnum )
```

This function performs a simple verification of the credit card number itself. This means that the function reports whether the *format* of the credit card number is valid; it does not report whether the card itself is valid and does not validate a credit card transaction.

Using Utility Functions

Finally, the WILX extender offers two utility functions. The `xGetElapsed` function is used with time values:

```
xGetElapsed( time1, time2 )
```

This function calculates the difference between two values obtained with `GetExactTime` (a standard WinBatch function).

The `xEnumStreams` function is used to enumerate 'named streams' in a file.

```
xEnumStreams( filename, flags )
```

This function creates and returns a 2-dimension array. There are [n] rows in the array, where 'n' is the number of named streams in the file. Each row has 4 columns. Each row contains information about one returned stream. The columns are as follows:

Column	Value								
0	stream name								
1	stream size								
2	stream type -- one of the following values: <table><tr><th>Value</th><th>Meaning</th></tr><tr><td>1</td><td>Standard data</td></tr><tr><td>2</td><td>Extended attribute data</td></tr><tr><td>3</td><td>Security descriptor data</td></tr></table>	Value	Meaning	1	Standard data	2	Extended attribute data	3	Security descriptor data
Value	Meaning								
1	Standard data								
2	Extended attribute data								
3	Security descriptor data								

	4 Alternative data streams 5 Hard link information 6 Property data 7 Objects identifiers 8 Reparse points 9 Sparse file						
3	stream attributes -- zero or more of the following values: <table> <tr> <th><u>Value</u></th><th><u>Meaning</u></th></tr> <tr> <td>1</td><td>Stream contains data that is modified when read</td></tr> <tr> <td>2</td><td>Stream contains security data</td></tr> </table>	<u>Value</u>	<u>Meaning</u>	1	Stream contains data that is modified when read	2	Stream contains security data
<u>Value</u>	<u>Meaning</u>						
1	Stream contains data that is modified when read						
2	Stream contains security data						

NETWORK EXTENDERS

WIL Extender Libraries offer extensive functionality for a wide variety of network systems.

WinBatch does not supply the network functions as part of its core operations. However many of the networking Extenders are installed by default.

Given the support for network operations, your programs can include a variety of features:

- Mapping drives
- Adding and removing users and groups
- Setting security permissions for resources
- Get information about the network

Identifying the Network

WinBatch does provide one network function outside the WIL Extender Libraries: the `NetInfo` function. This function provides the initial option to identify which networks are installed and, from this information, to determine which extender library (or libraries) should be installed.

The `NetInfo` function is called as:

```
NetInfo( requestcode )
```

The `requestcode` argument can be either of two values:

0	Reports a list of primary network names
1	Reports a secondary network list

Introduction to Programming

The `NetInfo` function is particularly important in a mixed network environment. You need to determine the types of networks running on a workstation before loading the appropriate network extender DLLs and calling the corresponding functions for network operations.

Windows Platform Version

It is also helpful to identify the version of the Windows platform the script is running on. The `WinVersion` and `WinMetrics` functions can be used to determine useful information about the Windows platform.

The `WinVersion` function provides the version number of the current Windows system.

```
WinVersion (level)
```

The `WinMetrics` function provides basic Windows system information.

```
WinMetrics (request#)
```

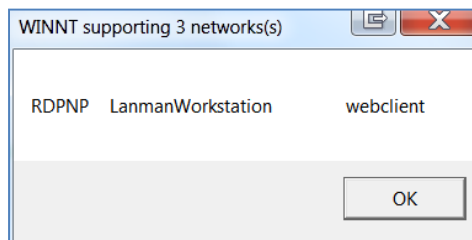
The [PlatformInfo.wbt](#) program demonstrates using the `WinVersion` and `WinMetrics` function to determine the Windows version, the 'bitness' and the latest service pack that has been installed.

Querying across the Network

The [NetTest.wbt](#) program demonstrates using the network extenders to get information across a network. It begins by determining the networking capabilities on a system.

[NetTest.wbt](#) may require modifications as appropriate for your operating system and network.

As a simple example, on a home network with a DSL server and firewall, the initial call to `NetInfo` reports:



The network type is unique to the running network. This value associates resources with a specific network when the resources are persistent or stored in links.

You can find a complete list of network types in the C header file Winnetwk.h. The purpose of a header file is to hold declarations for other files to use. Header files get installed with various Developer tools (i.e. Visual C and Visual C++). You can also find the contents of these header files online.

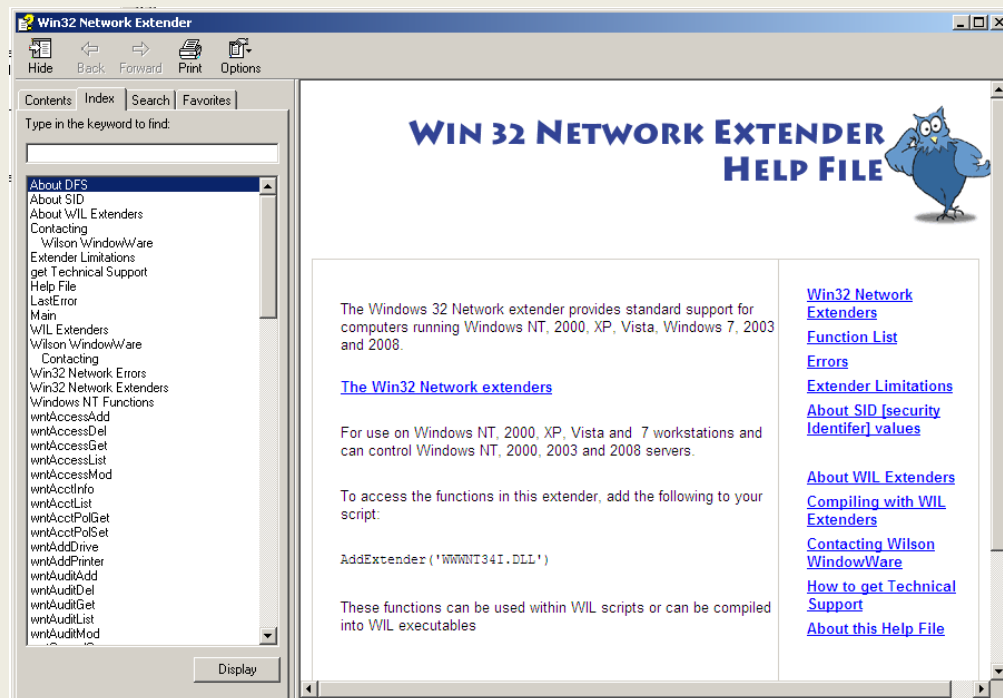
Note: RDPNP = Remote Desktop Protocol Network Provider.

At this point, we can use this initial information to decide which network is available.

Once we have information about which network is installed, we can load one (or more) of the WIL Extender Libraries to provide support for the network. For network operations, we have a choice of WIL extenders depending on the network and operating system(s) in use. (Refer to the online documentation for specifics and details.) For our example, with a Windows 7 system operating on a Windows NT network, the `WWWNT34I.DLL` is the appropriate extender for network functions.

Online Documentation

The online documentation for the network extenders is found in the `NetWareX.CHM`, `Win32Network.CHM` and the `ADSI.CHM` files (in your `\WinBatch` directory). The Win32 Network Extender help file appears below:



The network functions are extensive and fall into several different categories according to operating system and networks supported.

ADSI Extender: WWADS44I.DLL

The WinBatch Active Directory Service Interfaces (ADSI) extender provides access to the powerful functionality of Microsoft's Active Directory Service Interfaces in a style familiar to WinBatch users. With the ADSI extender, you can manage network resources in several directory services with a single, easy to use, set of functions.

Please note: We're assuming you have some prior knowledge of how to use Active Directory Services Interfaces in order to effectively utilize this extender. Along with a general understanding of ADSI, it goes without saying that you will need to have knowledge about the directory service that you will be using this extender to manipulate.

The ADSI extender is used to create directory service objects including users, computers, groups and organizational units. In addition, you can change object attributes and move objects between locations. Further, special functions are present for managing object membership in security and distribution groups; for investigating the directory structure including LDAP style search functions; and for handling Active Directory object security.

To access the functions in this extender, add the following to your script:

```
AddExtender ( 'WWADS44I.DLL' )
```

Compiler Options for WIL Extenders

The WinBatch+ Compiler supports five options for compiling scripts into executables:

- A standalone Windows EXE file.
- A small Windows EXE file.
- An encoded and encrypted WinBatch script file.
- A password protected WinBatch script file.
- Windows NT native service

When any extender functions are used in a script, the corresponding extender (.DLL) must be either compiled into the executable or placed where the executable can access it.

The Standalone EXE option of the compiler has an additional button. The EXTENDERS button displays a list of extenders that can be chosen and compiled into the executable. You can choose more than one extender.

When a stand-alone executable is launched on a PC, it looks for the necessary DLLs in the current directory, on the directory path, and in the Windows directory. If the DLLs are not found, the DLL is automatically written into the current directory.

If the DLLs cannot be written for some reason, such as because the directory is set to be Read Only, the compiled file will not be able to execute.

The DLLs can also be copied into a directory on a computer's directory path, and the compiled EXE will find them there and run. The compiler's Small EXE for Networked PC's option takes advantage of this. The DLLs need to be placed on the path only once. Subsequent EXE files installed on this same machine can be compiled under the Small EXE option.

While prudence recommends against including a directory path with the file name when calling the `AddExtender` statement (since paths can change with undesired results), it is possible to include an explicit reference to the current directory (the directory where the WIL executable resides) as:

```
AddExtender( DirScript() : "WWWNT34I.DLL" )
```

Custom Extender DLLs

You've seen how the Extender Libraries supplied by WinBatch can provide functionality beyond the standard procedures available.

In like fashion, third-party developers can also create their own function libraries as custom extender DLLs, adding virtually any function desired to the WIL language. (A WIL Extender SDK is available.) As an example, the source code for the 32-bit WILX extender, discussed at the beginning of this chapter, is included in the WIL-SDK subdirectory on the WinBatch and WinBatch+Compiler CD-ROM.

To develop custom DLLs, you need to use the appropriate programming tools, such as Microsoft Visual C++. In fact, creating a custom DLL is not a simple task, and you should probably have experience in using C/C++ for Windows programming before attempting to do so.

Summary

This concludes *Introduction to Programming*. At this point, you've gone from learning to use the WinBatch Studio IDE and creating the simplest possible program to creating graphics applications and debugging programs. Finally, you've learned about using libraries to extend the functionality supplied by WinBatch.

Although this has not been a brief journey, it has still been only an introduction and the first step. To go from here to becoming a competent programmer will require work, practice, work, dedication, work, and intelligence.

Along the way, you will also develop an insight into what isn't readily visible in your applications—a sense of what is actually happening and why operations work or fail.

Introduction to Programming

What you may see has no ready name, but when you learn to recognize it, no name will be necessary and any single name would be inadequate. However, this is the point where you begin to become a programmer.

Oh, yes, did I mention early in this book that there would be a test?

There will, except that the test is not in this book ... and there are no answers provided to score the results.

Because the test is simple — programming. And scoring the test is automatic: Do your programs work, how well do they work, and can people use them?

With that said, it only remains to wish you ...

Good luck and happy hacking!

APPENDIX A : WINBATCH DEMOS

REAL WORLD WIL SCRIPTS

Chapter 1 Samples

WordCnt.wbt

```

;*****
;**
;**  [Chapter 1]
;**  WordCnt.wbt
;**  Counts the number of space-delimited words in a text file
;**
;*****

; Set working directory to the same directory the script
DirChange( DirScript() )

file = AskFileName( "Choose a file name", "C:\", "Text
Files|*.txt", "*.txt", 1 )

; Open the selected file for text input
hFile = FileOpen( file, "READ" )
count = 0
line = FileRead( hFile )
While line != "*EOF*"
    ParseData( Line )
    ; Update number of words
    count = count + param0
    ; Read the next line
    Line = FileRead( hFile )
EndWhile

; Close the input file
FileClose( hFile )

```

Introduction to Programming

```
; Display the result
Message( "Output", "File " : file : " has " : count : " words" )
exit
```

Chapter 2 Samples

Hello World.wbt

```
;*****
; **
; ** [Chapter 2]
; ** Hello World.wbt
; ** Simple script to display 'Hello, World'
; **
;*****

Display( 5, "Hello, World", "How are you?" )
Pause( "Okay", "Now, wasn't that easy?" )
exit
```

Chapter 3 Samples

AskYesNo.wbt

```
;*****
; **
; ** [Chapter 3]
; ** AskYesNo.wbt
; ** Simple AskYesNo example
; **
;*****

answer = AskYesNo( "This is the title", "To be or not to be, that is
the question..." )

If answer == @YES
    answerstr = "Yep"
Else
```

```

        answerstr = "Nope"
Endif

Pause( "Answer", answerstr )
exit

:CANCEL
Message( "Answer", "CANCELLED!" )

```

AskLine.wbt

```

;*****
;
; **
; ** [Chapter 3]
; ** AskLine.wbt
; ** Simple AskLine example
; **
;*****

sName = AskLine( "Question", "What is your name?", "Simon Le'Gree" )
Pause( "Your name is" , sName )
exit

```

WILDdialog.wbt

```

;*****
;
; **
; ** [Chapter 3]
; ** WILDdialog.wbt
; ** WIL Dialog containing various control types
; **
;*****

DirChange( DirScript() )

ibVariable1 =
"Red":@tab:"White":@tab:"Blue":@tab:"Green":@tab:"Black":@tab:"Gray":@t
ab:"Orange":@tab:"Yellow":@tab:"Mauve":@tab:"Chartruse":@tab:"Peach":@t
ab:"Apricot"

MyDialogFormat=`WWWDLGED,6.2`

```

Introduction to Programming

```
MyDialogCaption=`WIL Dialog`
MyDialogX=-1
MyDialogY=-1
MyDialogWidth=392
MyDialogHeight=332
MyDialogNumControls=028
MyDialogProcedure=`DEFAULT`
MyDialogFont=`DEFAULT`
MyDialogTextColor=`DEFAULT`
MyDialogBackground=`DEFAULT,DEFAULT`
MyDialogConfig=0

MyDialog001=`023,009,180,192,GROUPBOX,"GroupBox_1",DEFAULT,"GroupBox",D
EFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog002=`039,023,038,010,RADIOBUTTON,"RadioButton_1",rbVariable1,"Ra
dioButton",1,20,DEFAULT,"Microsoft Sans
Serif|5325|40|34","255|0|0",DEFAULT`
MyDialog003=`039,053,038,008,CHECKBOX,"CheckBox_1",cbVariable1,"CheckBo
x",1,30,DEFAULT,"Microsoft Sans Serif|5325|40|34","0|255|0",DEFAULT`
MyDialog004=`039,075,030,010,EDITBOX,"EditBox_1",ebVariable1,"EditBox",
DEFAULT,40,DEFAULT,"Microsoft Sans Serif|5325|40|34","0|0|255",DEFAULT`
MyDialog005=`039,101,038,010,STATICTEXT,"StaticText_1",DEFAULT,"StaticT
ext",DEFAULT,50,DEFAULT,"Microsoft Sans
Serif|5325|40|34","0|255|255",DEFAULT`
MyDialog006=`039,125,036,010,VARYTEXT,"VaryText_1",vtVariable1,"VaryTex
t",DEFAULT,60,DEFAULT,"Modern|5632|40|65330","128|128|128",DEFAULT`
MyDialog007=`037,147,152,046,MULTILINEBOX,"MultiLineBox_1",mlVariable1,
"MultiLineBox",DEFAULT,70,DEFAULT,"Microsoft Sans
Serif|5325|140|34","255|0|255",DEFAULT`
MyDialog008=`143,023,048,020,PICTUREBUTTON,"PictureButton_1",DEFAULT,"P
ict button 1",2,80,DEFAULT,DEFAULT,DEFAULT,"buddha_figure.bmp"`
MyDialog009=`143,053,048,023,DROPLISTBOX,"DropListBox_1",dlVariable1,DE
FAULT,DEFAULT,90,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog010=`143,075,046,010,SPINNER,"Spinner_1",spVariable1,"1",DEFAUL
T,100,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog011=`143,101,044,032,PICTURE,"Picture_1",DEFAULT,"Picture",DEFA
ULT,110,DEFAULT,DEFAULT,DEFAULT,"buddha_figure.bmp"`
MyDialog012=`261,013,100,032,FILELISTBOX,"FileListBox_1",flVariable1,
"WILDialog.wbt",DEFAULT,120,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog013=`263,067,100,032,ITEMBOX,"ItemBox_1",ibVariable1,DEFAULT,DE
FAULT,130,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog014=`263,119,100,080,CALENDAR,"Calendar_1",caVariable1,DEFAULT,
DEFAULT,140,DEFAULT,DEFAULT`
```



```

MyDialog015=`025,223,336,074,COMCONTROL,"ComControl_URL",DEFAULT,"http:
//www.winbatch.com",DEFAULT,150,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

MyDialog016=`103,023,034,008,STATICTEXT,"StaticText_PictureButton",DEFA
ULT,"PictureButton",DEFAULT,160,DEFAULT,"Microsoft Sans
Serif|5632|40|34","0|128|0",DEFAULT`

MyDialog017=`103,053,036,008,STATICTEXT,"StaticText_DropListBox",DEFAUL
T,"DropListBox",DEFAULT,170,DEFAULT,"microsoft Sans
Serif|5632|40|34","128|128|0",DEFAULT`

MyDialog018=`103,075,036,008,STATICTEXT,"StaticText_Spinner",DEFAULT,"S
pinner",DEFAULT,180,DEFAULT,"Microsoft Sans
Serif|5632|40|34","128|0|0",DEFAULT`

MyDialog019=`103,103,032,012,STATICTEXT,"StaticText_Picture",DEFAULT,"P
icture",DEFAULT,190,DEFAULT,"Microsoft Sans
Serif|5632|40|34","0|255|0",DEFAULT`

MyDialog020=`217,025,028,012,STATICTEXT,"StaticText_FileListBox",DEFAUL
T,"FileListBox",DEFAULT,200,DEFAULT,"Microsoft Sans
Serif|5632|40|34","0|0|128",DEFAULT`

MyDialog021=`217,075,028,012,STATICTEXT,"StaticText_ItemBox",DEFAULT,"I
temBox",DEFAULT,210,DEFAULT,"Microsoft Sans
Serif|5632|40|34","128|0|128",DEFAULT`

MyDialog022=`217,147,032,012,STATICTEXT,"StaticText_Calendar",DEFAULT,"
Calendar",DEFAULT,220,DEFAULT,"Microsoft Sans
Serif|5632|40|34","0|128|128",DEFAULT`

MyDialog023=`025,207,044,012,STATICTEXT,"StaticText_COMCONTROL",DEFAULT
,"COMControl",DEFAULT,230,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

MyDialog024=`000,000,000,000,MENUBAR,"Dialog_Bar"`

MyDialog025=`000,000,000,000,MENUITEM,"mbil_Help","Dialog_Bar","Help",D
EFAULT,10,DEFAULT`

MyDialog026=`000,000,000,000,MENUITEM,"mbi2_About","mbil_Help","About",
DEFAULT,10,DEFAULT`

MyDialog027=`121,303,044,014,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,
240,32,DEFAULT,DEFAULT,DEFAULT`

MyDialog028=`235,303,044,014,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Ca
ncel",0,250,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

```

```
ButtonPushed = Dialog("MyDialog")
```

PushButton.wbt

```

;*****
;
;**
;**  [Chapter 3]
;**  PushButton.wbt
;**  WIL Dialog containing multiple PushButtons
;
;*****

```

Introduction to Programming

```
PushButtonDialogFormat = `WWWDLGED,6.2`

PushButtonDialogCaption = `Push Button Dialog`
PushButtonDialogX = -1
PushButtonDialogY = -1
PushButtonDialogWidth = 230
PushButtonDialogHeight = 070
PushButtonDialogMinWidth=2 30
PushButtonDialogMinHeight= 070
PushButtonDialogNumControls= 007
PushButtonDialogProcedure = `DEFAULT`
PushButtonDialogFont = `DEFAULT`
PushButtonDialogTextColor = `DEFAULT`
PushButtonDialogBackground = `DEFAULT,DEFAULT`
PushButtonDialogConfig = 0

PushButtonDialog001=`085,049,036,012,PUSHBUTTON,"PushButton_Cancel",DEF
AULT,"Cancel",0,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PushButtonDialog002=`017,009,036,012,PUSHBUTTON,"PushButton_1",DEFAULT,
"Push 1",1,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PushButtonDialog003=`053,009,036,012,PUSHBUTTON,"PushButton_2",DEFAULT,
"Push 2",2,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PushButtonDialog004=`089,009,036,012,PUSHBUTTON,"PushButton_3",DEFAULT,
"Push 3",3,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PushButtonDialog005=`125,009,036,012,PUSHBUTTON,"PushButton_4",DEFAULT,
"Push 4",4,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PushButtonDialog006=`161,009,036,012,PUSHBUTTON,"PushButton_5",DEFAULT,
"Push 5",5,60,DEFAULT,DEFIT,DEFAULT,DEFAULT`
PushButtonDialog007=`063,031,088,010,VARYTEXT,"VaryText_1",vtVariable1,
"No buttons have been pushed
yet...",DEFAULT,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

While @TRUE
    ButtonPushed=Dialog("PushButtonDialog")
    Select ButtonPushed
        case 1
            vtVariable1 = "The Push button 1 was pushed"
            break
        case 2
```

```

        vtVariable1 = "The Push button 2 was pushed"
        break
    case 3
        vtVariable1 = "The Push button 3 was pushed"
        break
    case 4
        vtVariable1 = "The Push button 4 was pushed"
        break
    case 5
        vtVariable1 = "The Push button 5 was pushed"
        break
    EndSelect
EndWhile
exit

:CANCEL
    Message( "Attention", "The Cancel button was pushed" )
    Message( "Attention", "...ergo. we're quitting now" )

```

RadioButton.wbt

```

;*****
;**
;** [Chapter 3]
;** RadioButton.wbt
;** WIL Dialog containing multiple RadioButton groupings
;**
;*****

; Set default radio buttons
rbColor = 2
rbSize  = 3
rbStyle = 1

MyDialogFormat=`WWWDLGED,6.2`

MyDialogCaption=`RadioButton Groupings`
MyDialogX=-1

```

Introduction to Programming

```
MyDialogY=-1
MyDialogWidth=182
MyDialogHeight=059
MyDialogNumControls=011
MyDialogProcedure=`DEFAULT`
MyDialogFont=`DEFAULT`
MyDialogTextColor=`DEFAULT`
MyDialogBackground=`DEFAULT,DEFAULT`
MyDialogConfig=0

MyDialog001=`140,006,033,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,
10,32,DEFAULT,DEFAULT,DEFAULT`
MyDialog002=`138,041,034,011,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Ca
ncel",0,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog003=`001,004,041,011,RADIOBUTTON,"RadioButton_1",rbColor,"Red",
1,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog004=`001,022,041,010,RADIOBUTTON,"RadioButton_2",rbColor,"Blue"
,2,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog005=`001,041,041,011,RADIOBUTTON,"RadioButton_3",rbColor,"Green
",3,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog006=`042,004,042,011,RADIOBUTTON,"RadioButton_4",rbSize,"Small"
,1,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog007=`042,022,042,010,RADIOBUTTON,"RadioButton_5",rbSize,"Medium
",2,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog008=`042,041,042,011,RADIOBUTTON,"RadioButton_6",rbSize,"Large"
,3,80,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog009=`087,004,041,011,RADIOBUTTON,"RadioButton_7",rbStyle,"Moder
n",1,90,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog010=`087,022,041,010,RADIOBUTTON,"RadioButton_8",rbStyle,"Class
ic",2,100,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog011=`087,041,041,011,RADIOBUTTON,"RadioButton_9",rbStyle,"Fancy
",3,110,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed=Dialog("MyDialog")

switch rbColor
    case 1
        color = "Red"
        break
    case 2
        color = "Blue"
```

```

        break
    case 3
        color = "Green"
        break
endswitch

switch rbSize
    case 1
        size = "Small"
        break
    case 2
        size = "Medium"
        break
    case 3
        size = "Large"
        break
endswitch

switch rbStyle
    case 1
        style = "Modern"
        break
    case 2
        style = "Classic"
        break
    case 3
        style = "Fancy"
        break
endswitch

; Format data to display
data = "Color: " : color : @lf : "Size: " : size : @lf : "Style: ":
style

; Display results
Pause( "Results", data )
exit

```

Introduction to Programming

CheckBox.wbt

```
;*****  
;**  
;**  [Chapter 3]  
;**  CheckBox.wbt  
;**  WIL Dialog containing multiple CheckBoxes  
;**  
;*****  
  
MyDialogFormat=`WWWDLGED,6.2`  
  
MyDialogCaption=`How do you like your burger?`  
MyDialogX=-1  
MyDialogY=-1  
MyDialogWidth=235  
MyDialogHeight=059  
MyDialogNumControls=007  
MyDialogProcedure=`DEFAULT`  
MyDialogFont=`DEFAULT`  
MyDialogTextColor=`DEFAULT`  
MyDialogBackground=`DEFAULT,DEFAULT`  
MyDialogConfig=0  
  
MyDialog001=`012,031,033,011,PUSHBUTTON,"PushButton_OK",DEFAULT,"Serve"  
  ,1,10,32,DEFAULT,DEFAULT,DEFAULT`  
MyDialog002=`188,030,033,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Ca  
  ncel",0,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog003=`010,007,042,011,CHECKBOX,"CheckBox_1",cbVariable1,"Mustard"  
  ,1,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog004=`050,007,042,011,CHECKBOX,"CheckBox_2",cbVariable2,"Catsup"  
  ,1,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog005=`089,007,041,011,CHECKBOX,"CheckBox_3",cbVariable3,"Mayonna  
  ise",1,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog006=`134,007,042,011,CHECKBOX,"CheckBox_4",cbVariable4,"Secret  
  Sauce",1,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
MyDialog007=`186,007,042,011,CHECKBOX,"CheckBox_5",cbVariable5,"Barbequ  
  e",1,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
  
ButtonPushed=Dialog("MyDialog")
```

```

; See if check box was checked
; Build string of ingredients to display to user
ingredients = ''
if cbVariable1 then ingredients = ingredients : @lf : 'Mustard'
if cbVariable2 then ingredients = ingredients : @lf : 'Catsup'
if cbVariable3 then ingredients = ingredients : @lf : 'Mayonnaise'
if cbVariable4 then ingredients = ingredients : @lf : 'Secret Sauce'
if cbVariable5 then ingredients = ingredients : @lf : 'Barbeque'

ingredients = StrSub( ingredients, 2, -1 ) ; Remove leading @lf
Pause('Hamburger Ingredients', ingredients )
exit

```

EditBox.wbt

```

;*****
;**
;**  [Chapter 3]
;**  EditBox.wbt
;**  WIL Dialog containing multiple EditBoxes
;**
;*****

EditTestFormat=`WWWDLGED,6.2`

EditTestCaption=`Edit Test`
EditTestX=-1
EditTestY=-1
EditTestWidth=218
EditTestHeight=054
EditTestNumControls=006
EditTestProcedure=`DEFAULT`
EditTestFont=`DEFAULT`
EditTestTextColor=`DEFAULT`
EditTestBackground=`DEFAULT,DEFAULT`
EditTestConfig=0

```

Introduction to Programming

```
EditTest001=`015,035,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,
30,32,DEFAULT,DEFAULT,DEFAULT`
EditTest002=`167,035,032,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Ca
ncel",0,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EditTest003=`117,015,082,010,EDITBOX,"EditBox_Password",ebPswd,DEFAULT,
DEFAULT,20,16,DEFAULT,DEFAULT,DEFAULT`
EditTest004=`015,015,086,010,EDITBOX,"EditBox_Name",ebName,DEFAULT,DEFA
ULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EditTest005=`015,007,040,006,STATICTEXT,"StaticText_1",DEFAULT,"Name",D
EFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EditTest006=`117,005,040,008,STATICTEXT,"StaticText_2",DEFAULT,"Passwor
d",DEFAULT,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed=Dialog("EditTest")

Pause( "Name", ebName )
Pause( "Password", ebPswd )
exit
```

Listbox.wbt

```
;*****
;
;
; ** [Chapter 3]
; ** ListBox.wbt
; ** WIL Dialog containing sample Listbox
; **
;*****

LBTFormat=`WWDLGED,6.2`

LBTCaption=`ListBox Test`
LBTX=-1
LBTY=-1
LBTWidth=182
LBTHeight=118
LBNumControls=005
LBTProcedure=`DEFAULT`
LBFont=`DEFAULT`
LBTextColor=`DEFAULT`
```



```

LBTBackground=`DEFAULT,DEFAULT`
LBTConfig=0

LBT001=`111,045,042,014,PUSHBUTTON,"PushButton_Select",DEFAULT,"Select"
,1,20,32,DEFAULT,DEFAULT,DEFAULT`

LBT002=`111,071,042,014,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Exit",0
,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

LBT003=`015,007,048,010,STATICTEXT,"StaticText_1",DEFAULT,"Select
color",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

LBT004=`015,023,064,084,ITEMBOX,"ItemBox_1",ibSelected,DEFAULT,DEFAULT,
10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

LBT005=`097,019,076,022,VARYTEXT,"VaryText_1",vtVariable1,"No
selection",DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

colorlist =
"Red":@tab:"White":@tab:"Blue":@tab:"Green":@tab:"Gray":@tab:"Black":@t
ab:"Orange":@tab:"Yellow":@tab:"Mauve":@tab:"Chartreuse":@tab:"Peach":@
tab:"Apricot"

ibSelected = colorlist          ;initialize ITEMBOX with colorlist
While @TRUE
    ButtonPushed = Dialog( "LBT" )
    vtVariable1 = ibSelected ; update vary text w/ user selection
    ibSelected = colorlist      ; re-initialize ITEMBOX with the colorlist
EndWhile
exit

```

FileListBox.wbt

```

;*****
;
;
;**   [Chapter 3]
;**   FileListBox.wbt
;**   WIL Dialog containing a FileListBox
;
;*****

FileSelectFormat=`WWWDLGED,6.2`

FileSelectCaption=`File Selection Dialog`
FileSelectX=-1
FileSelectY=-1

```

Introduction to Programming

```
FileSelectWidth=244
FileSelectHeight=106
FileSelectNumControls=007
FileSelectProcedure=`DEFAULT`
FileSelectFont=`DEFAULT`
FileSelectTextColor=`DEFAULT`
FileSelectBackground=`DEFAULT,DEFAULT`
FileSelectConfig=0

FileSelect001=`123,081,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"Select",1,20,32,DEFAULT,DEFAULT,DEFAULT`
FileSelect002=`177,081,032,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Exit",0,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileSelect003=`013,009,080,088,FILELISTBOX,"FileListBox_1",flVariable1,DEFAULT,DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileSelect004=`109,045,042,008,STATICTEXT,"StaticText_2",DEFAULT,"Selection:",DEFAULT,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileSelect005=`109,055,122,012,VARYTEXT,"VaryText_2",vtVariable2,DEFAULT,DEFAULT,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileSelect006=`109,017,044,008,STATICTEXT,"StaticText_1",DEFAULT,"Current Directory:",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileSelect007=`109,027,118,012,VARYTEXT,"VaryText_1",vtVariable1,DEFAULT,DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

vtVariable1 = DirScript()
flVariable1 = "*.wbt"
```

```
While @TRUE
    ButtonPushed=Dialog("FileSelect")
    vtVariable1 = DirGet() ; update varytext with current directory
    vtVariable2 = flVariable1 ; update varytext with selection
    flVariable1 = "*.wbt" ; redefine filemask for filelistbox
EndWhile
exit
```

ComControl.wbt

```
*****
**
** [Chapter 3]
** ComControl.wbt
```

```

; **  WIL Dialog containing a WebBrowser ComControl
; **
; *****

strUrl = "http://www.winbatch.com"
MyDialogFormat=`WWWDLGED,6.2`

MyDialogCaption=`WinBatch Web Browser`
MyDialogX=-1
MyDialogY=-1
MyDialogWidth=352
MyDialogHeight=217
MyDialogNumControls=003
MyDialogProcedure=`MyDialogCallbackProc`
MyDialogFont=`DEFAULT`
MyDialogTextColor=`DEFAULT`
MyDialogBackground=`DEFAULT,DEFAULT`
MyDialogConfig=0

MyDialog001=`009,007,120,010,STATICTEXT,"StaticText_URL:",DEFAULT,"URL:
":strUrl:",DEFAULT,3,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MyDialog002=`305,007,036,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit
",0,2,32,DEFAULT,DEFAULT,DEFAULT`
MyDialog003=`005,025,336,184,COMCONTROL,"ComControl_1",DEFAULT,"":strUr
l:",DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed=Dialog("MyDialog")
exit

```

Chapter 4 Samples

StringTest.wbt

```

; *****
; **
; **  [Chapter 4]
; **  StringTest.wbt
; **  Demonstrates string delimiters
; **
; *****

```

Introduction to Programming

```
sQuote = 'A simple string'
Message( "This one's easy", sQuote )

sQuote = `It's Tommy this an' Tommy that,`:@CRLF:`  an' "Chuck 'im
out, the brute".`:@CRLF:`But it's "Saviour of 'is country,"`:@CRLF:`
when the guns begin to shoot.`:@CRLF:`- R. Kipling`

Message( "I quote, of course", sQuote )

Exit
```

ArrayTest.wbt

```
;*****
;
;
;**  [Chapter 4]
;**  ArrayTest.wbt
;**  Demonstrates arrays in WinBatch
;
;*****

nNumber = 1

ArrayTestFormat=`WWWDLGED,6.2`

ArrayTestCaption=`Color Selection`
ArrayTestX=025
ArrayTestY=042
ArrayTestWidth=144
ArrayTestHeight=078
ArrayTestNumControls=006
ArrayTestProcedure=`DEFAULT`
ArrayTestFont=`DEFAULT`
ArrayTestTextColor=`DEFAULT`
ArrayTestBackground=`DEFAULT,DEFAULT`
ArrayTestConfig=0

ArrayTest001=`011,057,034,012,PUSHBUTTON,"PushButton_Test",DEFAULT,"Test",1,10,32,DEFAULT,DEFAULT,DEFAULT`
ArrayTest002=`051,057,034,012,PUSHBUTTON,"PushButton_Add",DEFAULT,"Add",2,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```

ArrayTest003=`091,057,034,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit",0,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ArrayTest004=`013,007,112,012,VARYTEXT,"VaryText_1",sPrompt,DEFAULT,DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ArrayTest005=`013,039,112,012,VARYTEXT,"VaryText_2",sReport,DEFAULT,DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ArrayTest006=`041,023,048,012,EDITBOX,"EditBox_1",nNumber,DEFAULT,DEFAULT,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

; Create an array from a list of elements
ColorArray = Arrayize("White,Yellow,Magenta,Chartreuse,Light Blue,Dark Blue,Green,Brown,Gray,Black", ", " )

; Get the number of elements
nMax = ArrInfo( ColorArray, 1 )

While @TRUE
    nSelectedColor = ""
    sPrompt = "Select a color by entering a number from 1 to " : nMax
    ButtonPushed=Dialog("ArrayTest")
    Switch ButtonPushed
        case 1 ; Test button
            If( nNumber > 0 && nNumber <= nMax )
                nSelectedColor = ColorArray[nNumber-1]
                sReport = "The color you selected was " : nSelectedColor
            Else
                sReport = "The color you selected was NOT VALID"
            Endif
            break

        case 2 ; Add button
            Gosub NEWCOLOR
            break
    EndSwitch
EndWhile
exit

;*****

```

Introduction to Programming

```
;**
;**  Subroutine
;**
;*****

:NEWCOLOR
NewColorFormat=`WWWDLGED,6.2`
NewColorCaption=`New Color Entry`
NewColorX=035
NewColorY=053
NewColorWidth=122
NewColorHeight=054
NewColorNumControls=005
NewColorProcedure=`DEFAULT`
NewColorFont=`DEFAULT`
NewColorTextColor=`DEFAULT`
NewColorBackground=`DEFAULT,DEFAULT`
NewColorConfig=0

NewColor001=`011,035,038,012,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,
10,32,DEFAULT,DEFAULT,DEFAULT`
NewColor002=`071,035,038,012,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Ca
ncel",0,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
NewColor003=`021,001,080,012,VARYTEXT,"VaryText_1",sNewColorPrompt,DEFA
ULT,DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
NewColor004=`013,017,038,012,STATICTEXT,"StaticText_1",DEFAULT,"Enter
color",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
NewColor005=`057,017,050,012,EDITBOX,"EditBox_1",sNewColor,DEFAULT,DEFA
ULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

nNext = nMax + 1
sNewColorPrompt = "The new item will be #" : nNext
sNewColor = ""
While @True
    ButtonPushed=Dialog("NewColor")
    if sNewColor == "" then continue ; loop until color is specified
    nMax = nNext
    ArrayRedim( ColorArray, nMax )
    ColorArray[nMax-1] = sNewColor
```

```

        break
    Endwhile
return

```

HugeMath.wbt

```

;*****
;**
;**  [Chapter 4]
;**  HugeMath.wbt
;**  Adds two huge numbers together using the Huge Math Extender
;**
;**  Requires that the Huge Math Extender is installed
;**  http://files.winbatch.com/wwwftp/wb01/wwhug34i.zip
;*****

AddExtender( "WWHUG34I.DLL" )
num1 = "12345678901234567890"
num2 = "98765432109876543210"
ret = huge_Add( num1, num2 )
Message( "Result of large number addition", ret )
exit

```

VariTest.wbt

```

;*****
;**
;**  [Chapter 4]
;**  VariTest.wbt
;**  Demonstrates variable assignments
;**
;*****

n = 2
m = 1.01
Message( "Result: step 1", "m = " : m : ", n = " : n )

n = n * m
Message( "Result: step 2", "m = " : m : ", n = " : n )

```

Introduction to Programming

```
n = "now I'm a string"
Message( "Result: step 3", "m = " : m : ", n = " : n )

n = "2"      ; this is a string
m = "2.02"   ; also a string representing a floating-point value
m * n        ; n is converted to an integer and m to a float
a = m * n
Message( "Result: step 4", "m = " : m : ", a = " : a )

n = "two"    ; this is a string
m = "two point zero two" ; this is also a string
a = m * n    ; we expect this step to fail
Message( "Result: step 5", "m = " : m : ", a = " : a )
exit
```

ListTest.wbt

```
;*****
; **
; ** [Chapter 4]
; ** ArrayTest.wbt
; ** Demonstrates AskItemList function
; **
;*****

listFruits = "apple,pear,orange,banana,peach,apricot,plum"
sFruit = AskItemList( "Fruits", listFruits, ",", @SORTED, @SINGLE )
Message( "The selected fruit is:", sFruit )
exit
```

Chapter 5 Samples

MathTest.wbt

```
;*****
; **
; ** [Chapter 5]
; ** MathTest.wbt
; ** Demonstrates operators
; **
```



```

;*****

nOperation = 0

MathTestFormat=`WWDLGED,6.2`

MathTestCaption=`Math Test`
MathTestX=012
MathTestY=031
MathTestWidth=108
MathTestHeight=100
MathTestNumControls=008
MathTestProcedure=`DEFAULT`
MathTestFont=`DEFAULT`
MathTestTextColor=`DEFAULT`
MathTestBackground=`DEFAULT,DEFAULT`
MathTestConfig=0

MathTest001=`009,081,038,012,PUSHBUTTON,"PushButton_Test",DEFAULT,"Test
",1,10,32,DEFAULT,DEFAULT,DEFAULT`
MathTest002=`061,081,038,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit
",0,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MathTest003=`011,017,088,012,RADIOBUTTON,"RadioButton_Precedence",nOper
ation,"Precedence",1,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MathTest004=`011,029,088,012,RADIOBUTTON,"RadioButton_2",nOperation,"Di
vision operations",2,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MathTest005=`011,041,088,012,RADIOBUTTON,"RadioButton_3",nOperation,"Mo
dulus operations",3,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MathTest006=`011,053,088,012,RADIOBUTTON,"RadioButton_4",nOperation,"Ex
ponential operations",4,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MathTest007=`011,065,088,012,RADIOBUTTON,"RadioButton_5",nOperation,"Bi
twise Shift operations",5,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
MathTest008=`001,003,104,012,STATICTEXT,"StaticText_1",DEFAULT,"Select
the type of operation to
demonstrate",DEFAULT,80,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

While @TRUE                                ; Loop forever

    nOperation = nOperation + 1

```

Introduction to Programming

```
if( nOperation > 5 ) then nOperation = 1

ButtonPushed = Dialog("MathTest")

Switch nOperation
case 1
    gosub precedence
    break

case 2
    gosub division
    break

case 3
    gosub modulus
    break

case 4
    gosub power
    break

case 5
    gosub testshift
    break
EndSwitch

Endwhile

exit

:precedence
a = 2.5
b = 3.7
c = ( a / b ) * 4.1
d = a / ( b * 4.1 )
Message( "Precedence:", "( " : a : " / " : b : " ) * 4.1 = " : c :
@CRLF: a : " / ( " : b : " * 4.1 ) = " : d )
return
```

```

:division
n = 9
m = 3
fResult = n / m
Message( "Division: step 1", n : " / " : m : " = " : fResult )

n = 2
m = 3
fResult = n / m
Message( "Division: step 2", n : " / " : m : " = " : fResult )

fResult = m / n
Message( "Division: step 3", m : " / " : n : " = " : fResult )

n = 3.0
m = 2
fResult = n / m
Message( "Division: step 4", n : " / " : m : " = " : fResult )

n = 2
m = 3.0
fResult = n / m
Message( "Division: step 5", n : " / " : m : " = " : fResult )

n = 6.0
m = 3.0
fResult = n / m
Message( "Division: step 6", n : " / " : m : " = " : fResult )
return

:modulus
n = 6.0
m = 3.0
fResult = n mod m
Message( "Modulus: step 1", n : " mod " : m : " = " : fResult )

```

Introduction to Programming

```
n = 7
m = 3
fResult = n mod m
Message( "Modulus: step 2", n : " mod " : m : " = " : fResult )

n = 7.53
m = 3.1
fResult = n mod m
Message( "Modulus: step 3", n : " mod " : m : " = " : fResult )
return

:power
n = 2
m = 3
fResult = n ** m
Message( "Exponential: step 1", n : " ** " : m : " = " : fResult )

n = 2.5
m = 3
fResult = n ** m
Message( "Exponential: step 2", n : " ** " : m : " = " : fResult )

n = -2.5
m = 3
fResult = n ** m
Message( "Exponential: step 3", n : " ** " : m : " = " : fResult )

n = 2.5
m = -3
fResult = n ** m
Message( "Exponential: step 4", n : " ** " : m : " = " : fResult )
return

:testshift
n = 16393
```

```

m = 2
fResult = n >> m
Message( "Bit Shift: step 1", n : " >> " : m : " = " : fResult )

fResult = n << m
Message( "Bit Shift: step 2", n : " << " : m : " = " : fResult )
return

```

SimpleCalculator.wbt

```

;*****
;**
;**  [Chapter 5]
;**  SimpleCalculator.wbt
;**  WIL Dialog for simple math calculations
;**
;*****

CalcFormat=`WWDLGED,6.2`
CalcCaption=`Simple Calculator`
CalcX=079
CalcY=107
CalcWidth=204
CalcHeight=098
CalcNumControls=012
CalcProcedure=`DEFAULT`
CalcFont=`DEFAULT`
CalcTextColor=`DEFAULT`
CalcBackground=`DEFAULT,DEFAULT`
CalcConfig=0

Calc001=`031,075,064,012,PUSHBUTTON,"PushButton_Ok",DEFAULT,"Ok",1,10,3
2,DEFAULT,DEFAULT,DEFAULT`
Calc002=`121,075,064,012,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel
",0,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Calc003=`091,025,036,012,RADIOBUTTON,"RadioButton_-",op,"-
",2,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Calc004=`055,025,036,012,RADIOBUTTON,"RadioButton_+",op,"+",1,40,DEFAULT,
T,DEFAULT,DEFAULT,DEFAULT`

```

Introduction to Programming

```
Calc005=`127,025,036,012,RADIOBUTTON,"RadioButton_*",op,"*",3,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
Calc006=`163,025,036,012,RADIOBUTTON,"RadioButton_/",op,"/",4,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
Calc007=`009,025,046,012,STATICTEXT,"StaticText_1",DEFAULT,"Select Operation",DEFAULT,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
Calc008=`009,007,046,012,STATICTEXT,"StaticText_2",DEFAULT,"Enter first value",DEFAULT,80,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
Calc009=`009,043,054,012,STATICTEXT,"StaticText_3",DEFAULT,"Enter second value",DEFAULT,90,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
Calc010=`009,061,036,012,CHECKBOX,"CheckBox_Exit",stop,"Exit",1,100,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
Calc011=`065,007,112,012,EDITBOX,"EditBox_1",operand1,DEFAULT,DEFAULT,110,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
Calc012=`065,043,112,012,EDITBOX,"EditBox_2",operand2,DEFAULT,DEFAULT,120,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
While @true                                ; Loop forever
    ButtonPushed=Dialog("Calc")
    If stop == 1 Then Exit                  ; exit?

    ; Repeat loop if either operand is an empty string
    If operand1 == "" Then continue
    If operand2 == "" Then continue

    ; Repeat loop if either operand is not a number
    If ! IsNumber(operand1) Then continue
    If ! IsNumber(operand2) Then continue

    opStr = StrSub("+-*/", op, 1)          ; get the operator
    result = operand1 %opStr% operand2     ; execute calculation
                                           ; show the results
    Message("Output", operand1 : " " : opStr : " " : operand2 : " = " :
result)
EndWhile                                    ; repeat until either exit or cancel
exit
```

Chapter 6 Samples

SearchList.txt

MyApp1 Apples Oranges Pears

MyApp2 23 456.4 93.0 128.6 93.2 35.6 87.456 7.65

MyApp3 %n1% %n2% %n3%

SearchTest.wbt

```
;*****
;
;**
;** [Chapter 6]
;** SearchTest.wbt
;** Open and parse a text file using ParseData
;**
;*****

; Set working directory to the same directory the script is in
DirChange( DirScript() )

fileCmd = "SearchList.txt"

hFile = FileOpen( fileCmd, "READ" )      ; open the file and get a handle

While @TRUE
    line = ""                            ; reset the line string
    line = FileRead( hFile )
    if line == "*EOF*" then break        ; break out of while loop
    nCmds = ParseData( line )
    cmdLine = "There are " : nCmds : " commands: |"
    For i = 1 to nCmds
        cmdLine = StrCat( cmdLine, param%i%, " | " )
    Next
    Message( "Results", cmdLine )
EndWhile

FileClose( hFile )                      ; close the input file
```

Introduction to Programming

```
AskFileText( fileCmd, fileCmd, @UNSORTED, @SINGLE ) ; view entire file
exit
```

SearchTest2.wbt

```
;*****
; **
; ** [Chapter 6]
; ** SearchTest2.wbt
; ** Open and parse a text file using StrScan & StrSub
; **
;*****
```

```
DirChange( DirScript() )
```

```
fileCmd = "SearchList.txt"
```

```
hFile = FileOpen( fileCmd, "READ" ) ; open the output file and get a
handle
```

```
While @TRUE
```

```
    line = "" ; reset the line string
```

```
    line = FileRead( hFile )
```

```
    If line == "*EOF*" Then Break ; break out of while loop
```

```
    nPos1 = 1 ; set starting point
```

```
    nCmds = 0 ; zero the count
```

```
    bDone = @FALSE ; and set an end flag
```

```
    While @TRUE
```

```
        nPos2 = StrScan( line, ',,: ', nPos1, @FWDSCAN ) ; checking for
three characters
```

```
        If( nPos2 == 0 ) ; we've hit the end of the line
```

```
            nPos2 = StrLen( line ) + 1 ; find end of line
```

```
            bDone = @TRUE ; and set an end flag
```

```
        Endif
```

```
        nCmds = nCmds + 1
```

```
        param%nCmds% = StrSub( line, nPos1, nPos2 - nPos1 )
```

```
        nPos1 = nPos2 + 1 ; set a new starting point
```



```

    If( bDone == @TRUE ) Then Break ; reached end, exit loop
EndWhile
param0 = nCmds

cmdLine = "There are " : nCmds : " commands: | "
For i = 1 to nCmds
    cmdLine = StrCat( cmdLine, param%i%, " | " )
Next
Message( "Results", cmdLine )
EndWhile

FileClose( hFile ) ; close the input file

AskFileText( fileCmd, fileCmd, @UNSORTED, @SINGLE ) ; view entire file
Exit

```

SearchTest3.wbt

```

;*****
;**
;** [Chapter 6]
;** SearchTest3.wbt
;** Open and parse a text file using ItemCount & ItemExtract
;**
;*****

; Set working directory to the same directory the script is in
DirChange( DirScript() )

fileCmd = "SearchList.txt"

hFile = FileOpen( fileCmd, "READ" ) ; open the output file and get a
handle

While @TRUE
    line = "" ; reset the line string
    line = FileRead( hFile )
    If line == "*EOF*" Then Break ; end of file reached

```

Introduction to Programming

```
nCmds = ItemCount( line, " " ) ; get the number of commands
For i = 1 to nCmds
    param%i% = ItemExtract( i, line, " " ) ; extract each substring
Next

cmdLine = "There are " : nCmds : " commands: | "
For i = 1 to nCmds
    cmdLine = StrCat( cmdLine, param%i%, " | " )
Next
Message( "Results", cmdLine )
EndWhile

FileClose( hFile ) ; close the input file

AskFileText( fileCmd, fileCmd, @UNSORTED, @SINGLE ) ; view entire file
exit
```

SearchTest4.wbt

```
;*****
; **
; ** [Chapter 6]
; ** SearchTest4.wbt
; ** Open and parse a text file using arrays
; **
;*****

; Set working directory to the same directory the script is in
DirChange( DirScript() )

fileCmd = "SearchList.txt"

ArrayCmd = ArrayFileGet( fileCmd ) ; read the file as an array
nSets = ArrInfo( ArrayCmd, 1 ) ; how many lines were read?
cmdLine = "There are " : nSets : " data sets"
Message( "Array Contents", cmdLine )

For i = 1 to nSets
    Message( "Set: " : i, ArrayCmd[i-1] )
```

Next

```
AskFileText( fileCmd, fileCmd, @UNSORTED, @SINGLE ); view entire file
exit
```

StrIndex.wbt

```
;*****
; **
; ** [Chapter 6]
; ** StrIndex.wbt
; ** Locates all occurrences of the substring
; **
;*****

sSample = "The quick red fox jumped over the lazy brown dog"
sTarget = "brown dog"
nPos = 0
While @TRUE
    nPos = StrIndex( sSample, sTarget, nPos, @FWDSCAN )
    If nPos == 0 Then Break ; time to quit
    Pause( "Found at position", nPos )
    nPos = nPos + 1 ; start one place further
EndWhile
exit
```

Blake.txt

```
And there is a frown of hate;
And there is a frown of disdain,
And there is a frown of frowns
Which you strive to forget in vain.

    Blake -- The Smile, Stanza 2
```

SearchReplace.wbt

```
;*****
; **
; ** [Chapter 6]
; ** SearchReplace.wbt
; ** Demonstrates a selective search-and-replace operation. Reads from
```

Introduction to Programming

```
;** a text file containing an excerpt from Blake's The Smile,
;** where one word is repeated four times in three lines. To show that
;** the replacement is selective rather than global, as would happen
;** using the StrReplace function, we change only the first and third
;** occurrences of the string, leaving the second & fourth as they are
;** in the original.
;*****

; Set working directory to the same directory the script is in
DirChange( DirScript() )

sFileIn = "Blake.txt"
sFileOut = "Blake.out"
sTarget = "frown"
sReplace = "smile"
nLen = StrLen( sTarget )
nCount = 1

AskFileText(sFileIn, sFileIn, @UNSORTED, @SINGLE) ; view original file
hFileIn = FileOpen( sFileIn, "READ" )
If hFileIn ; the source file exists
    hFileOut = FileOpen( sFileOut, "WRITE" )
    While @TRUE
        sLineIn = FileRead( hFileIn );
        If sLineIn == "*EOF*" Then Break ; break out of while loop
        nPos = 0
        sLineOut = sLineIn
        While @TRUE
            nPos = StrIndexNc( sLineIn, sTarget, nPos, @FWDSCAN )
            If nPos == 0 Then Break ; that's it, jump to next line
            If nCount mod 2 == 1
                nPos2 = nPos + nLen
                nLen2 = StrLen( sLineIn ) - nPos2 + 1
                sTemp1 = StrSub( sLineIn, 1, nPos-1 )
                sTemp2 = StrSub( sLineIn, nPos2, nLen2 )
                sLineOut = StrCat( sTemp1, sReplace, sTemp2)
            Endif
        Endif
    Endif
```

```

        nCount = nCount + 1
        nPos = nPos + 1 ; start the next search one place further
    EndWhile
    FileWrite( hFileOut, sLineOut )
EndWhile
Endif

FileClose( hFileIn )           ; close the input file
FileClose( hFileOut )          ; close the output file
; view the new file
AskFileText(sFileOut, sFileOut, @UNSORTED, @SINGLE)
exit

```

StrCmp.wbt

```

;*****
;**
;**  [Chapter 6]
;**  StrCmp.wbt
;**  Demonstrates string comparison
;**
;*****

StringCmpFormat=`WWWDLGED,6.2`

StringCmpCaption=`String Comparison`
StringCmpX=025
StringCmpY=042
StringCmpWidth=180
StringCmpHeight=056
StringCmpNumControls=005
StringCmpProcedure=`DEFAULT`
StringCmpFont=`DEFAULT`
StringCmpTextColor=`DEFAULT`
StringCmpBackground=`DEFAULT,DEFAULT`
StringCmpConfig=0

StringCmp001=`005,003,082,012,EDITBOX,"EditBox_1",sTemp1,"one
banana",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

```

Introduction to Programming

```
StringCmp002=`093,003,080,012,EDITBOX,"EditBox_2",sTemp2,"ONE
APPLE",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
StringCmp003=`007,039,038,012,PUSHBUTTON,"PushButton_Test",DEFAULT,"Tes
t",1,30,32,DEFAULT,DEFAULT,DEFAULT`
StringCmp004=`135,039,038,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exi
t",0,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
StringCmp005=`007,021,166,012,VARYTEXT,"VaryText_1",sReport,DEFAULT,DEF
AULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
While @TRUE
    ButtonPushed = Dialog("StringCmp")
    nResult = StrCmp( sTemp1, sTemp2 )
    Select nResult
        case -1                                ; less than
            sResult = " is less than "
            break
        case 0                                ; equal
            sResult = " is equal to "
            break
        case 1                                ; greater than
            sResult = " is greater than "
            break
    EndSelect
    sReport = '"' : sTemp1 : '"' : sResult: '"' : sTemp2 : '"'
EndWhile
exit
```

RelationalOperators.wbt

```
;*****
; **
; ** [Chapter 6]
; ** RelationalOperators.wbt
; ** Demonstrates string comparison using relational operators
; **
;*****

StringCmpFormat=`WWWDLGED,6.2`

StringCmpCaption=`String Comparison (case sensitive)`
```

```

StringCmpX=025
StringCmpY=042
StringCmpWidth=180
StringCmpHeight=056
StringCmpNumControls=005
StringCmpProcedure=`DEFAULT`
StringCmpFont=`DEFAULT`
StringCmpTextColor=`DEFAULT`
StringCmpBackground=`DEFAULT,DEFAULT`
StringCmpConfig=0

StringCmp001=`005,003,082,012,EDITBOX,"EditBox_1",sTemp1,"one
apple",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
StringCmp002=`093,003,080,012,EDITBOX,"EditBox_2",sTemp2,"ONE
APPLE",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
StringCmp003=`007,039,038,012,PUSHBUTTON,"PushButton_Test",DEFAULT,"Tes
t",1,30,32,DEFAULT,DEFAULT,DEFAULT`
StringCmp004=`135,039,038,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exi
t",0,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
StringCmp005=`007,021,166,012,VARYTEXT,"VaryText_1",sReport,DEFAULT,DEF
AULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

```

```
While @TRUE
```

```

    ButtonPushed = Dialog("StringCmp")
    if sTemp1 == sTemp2 then sResult = " is equal to "
    if sTemp1 > sTemp2 then sResult = " is greater than "
    if sTemp1 < sTemp2 then sResult = " is less than "
    sReport = "'" : sTemp1 : "'" : sResult : "'" : sTemp2 : "'"

```

```
EndWhile
```

```
exit
```

Parts.lst

```

Webley Defaminizer      5    2456-3468-8921    27
Finagle Bolix Grinder 2  3905-1298-7892    12B
Acme Jetpack      3    9834-0909-8721    14
Hobart Skyhook 1    6435-2348-0971    8E
Forward Mass Detector 9    3498-3465-1871    5
Dyson Sphere      1    0000-0000-0001    M27-139-235-890
Niven Transporter (Pad model) 3    9872-2317-2345    17A, 18B, 21C

```

Introduction to Programming

Murphey's Law Guide 2 1313-9872-3458 ??

ListSelection.wbt

```
;*****  
;**  
;** [Chapter 6]  
;** ListSelection.wbt  
;** Demonstrates demonstrates loading a listbox and making a selection  
;**  
;*****
```

```
DirChange( DirScript() )
```

```
SelectionFormat=`WWWDLGED,6.2`  
SelectionCaption=`Acme Parts Catalog`  
SelectionX=057  
SelectionY=074  
SelectionWidth=114  
SelectionHeight=160  
SelectionNumControls=007  
SelectionProcedure=`DEFAULT`  
SelectionFont=`DEFAULT`  
SelectionTextColor=`DEFAULT`  
SelectionBackground=`DEFAULT,DEFAULT`  
SelectionConfig=0
```

```
Selection001=`001,001,108,106,ITEMBOX,"ItemBox_1",listDisplay,DEFAULT,DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
Selection002=`007,145,034,012,PUSHBUTTON,"PushButton_Check",DEFAULT,"Check",1,20,32,DEFAULT,DEFAULT,DEFAULT`  
Selection003=`049,145,034,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit",0,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
Selection004=`005,113,036,012,VARYTEXT,"VaryText_1",sItemSelect,"Item",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
Selection005=`005,129,036,012,VARYTEXT,"VaryText_2",sStkQuant,"Quant",DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
Selection006=`047,113,046,012,VARYTEXT,"VaryText_3",sPN,"P/N",DEFAULT,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
Selection007=`047,129,046,012,VARYTEXT,"VaryText_4",sLoc,"Loc",DEFAULT,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```



```

fileIn = "Parts.lst" ; source file for data
listItem      = "" ; initialize all of the lists
listStkPN     = ""
listStkQuant  = ""
listStkLoc    = ""
listDisplay   = ""

nItemCount = 1
hFileIn = FileOpen( fileIn, "READ" ) ; open input file and get a handle
While @TRUE
    sLineIn = FileRead( hFileIn )
    If sLineIn == "*EOF*" Then Break ; break out of while loop
    If sLineIn == "" Then Continue ; repeat while loop

    sListItem = ItemExtract( 1, sLineIn, @TAB ) ; extract item name
    sListItem = StrFix( sListItem, "", 100 ) ; pad with spaces
    sListItem = StrCat( sListItem, nItemCount ) ; then add the index
number
    listItem = ItemInsert( sListItem, 0, listItem, @TAB ) ; insert at
first of list

    sListItem = ItemExtract( 2, sLineIn, @TAB ) ; extract quantity
    listStkQuant = ItemInsert( sListItem, -1, listStkQuant, @TAB ) ;
insert at end of list

    sListItem = ItemExtract( 3, sLineIn, @TAB ) ; extract P/N
    listStkPN = ItemInsert( sListItem, -1, listStkPN, @TAB ) ; insert
at end of list

    sListItem = ItemExtract( 4, sLineIn, @TAB ) ; extract location
    listStkLoc = ItemInsert( sListItem, -1, listStkLoc, @TAB ) ;
insert at end of list
    nItemCount = nItemCount + 1
EndWhile
FileClose( hFileIn ) ; close the input file

While @TRUE

```

Introduction to Programming

```
listDisplay = ItemSort( listItem, @TAB ) ; initialize listbox with
sorted list
ButtonPushed=Dialog("Selection")
If listDisplay != "" ; is there a selection?
    nLen = StrLen( listDisplay ) ; get entry length
    sIndex = StrSub( listDisplay, nLen - 10, -1 ) ; get rightmost
chars
    sIndex = StrTrim( sIndex ) ; trim for index value
    sItemSelect = StrSub( listDisplay, 1, nLen - 10 ) ; drop
rightmost chars
    sItemSelect = StrTrim( sItemSelect ) ; trim for item name
    ; get the rest of the particulars from the separate lists
    sStkQuant = "In stock: ":ItemExtract( sIndex, listStkQuant, @TAB)
    sPN = "P/N: ": ItemExtract( sIndex, listStkPN, @TAB )
    sLoc = "Location: ": ItemExtract( sIndex, listStkLoc, @TAB)
EndIf
EndWhile
exit
```

ListSelection2.wbt

```
;*****
; **
; ** [Chapter 6]
; ** ListSelection2.wbt
; ** Demonstrates a slightly different method of loading a
; ** listbox and making a selection
; **
;*****
```

```
DirChange( DirScript() )
```

```
SelectionFormat=`WWWDLGED,6.2`
SelectionCaption=`Acme Parts Catalog`
SelectionX=057
SelectionY=074
SelectionWidth=114
SelectionHeight=160
SelectionNumControls=007
```

```

SelectionProcedure=`DEFAULT`
SelectionFont=`DEFAULT`
SelectionTextColor=`DEFAULT`
SelectionBackground=`DEFAULT,DEFAULT`
SelectionConfig=0

Selection001=`001,001,108,106,ITEMBOX,"ItemBox_1",listDisplay,DEFAULT,DE
FAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Selection002=`007,145,034,012,PUSHBUTTON,"PushButton_Check",DEFAULT,"Ch
eck",1,20,32,DEFAULT,DEFAULT,DEFAULT`
Selection003=`049,145,034,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exi
t",0,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Selection004=`005,113,036,012,VARYTEXT,"VaryText_1",sItemSelect,"Item",
DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Selection005=`005,129,036,012,VARYTEXT,"VaryText_2",sStkQuant,"Quant",D
EFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Selection006=`047,113,046,012,VARYTEXT,"VaryText_3",sPN,"P/N",DEFAULT,6
0,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Selection007=`047,129,046,012,VARYTEXT,"VaryText_4",sLoc,"Loc",DEFAULT,
70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

fileIn = "Parts.lst" ; source file for data
listItem      = ""      ; initialize all of the lists
listStkData   = ""

hFileIn = FileOpen( fileIn, "READ" ) ; open the input file and get a
handle
While @TRUE
    sLineIn = FileRead( hFileIn )
    If sLineIn == "*EOF*" Then Break ; break out of while loop
    If sLineIn == "" Then Continue ; repeat loop

    sListItem = ItemExtract( 1, sLineIn, @TAB ) ; extract the item name
    listItem = ItemInsert( sListItem, -1, listItem, @TAB ) ; insert at
end of list
    listStkData = ItemInsert( listStkData, -1, sLineIn, @CR ) ; insert
at end of list
EndWhile
FileClose( hFileIn ) ; close the input file

While @TRUE

```

Introduction to Programming

```
listDisplay = ItemSort( listItem, @TAB ) ; initialize listbox with
sorted list
ButtonPushed = Dialog( "Selection" )
If listDisplay != "" ; is there a selection?
    sItemSelect = listDisplay ; copy to report display
    For i = 1 to ItemCount( listStkData, @CR )
        listTemp = ItemExtract( i, listStkData, @CR )
        If ItemLocate( sItemSelect, listTemp, @TAB ) == 1 Then Break
    Next
    ; get the rest of the particulars from the sublist
    sStkQuant = "In stock: " : ItemExtract( 2, listTemp, @TAB )
    sPN = "P/N: " : ItemExtract( 3, listTemp, @TAB )
    sLoc = "Location: " : ItemExtract( 4, listTemp, @TAB )
EndIf
EndWhile
exit
```

Password.wbt

```
;*****
; **
; ** [Chapter 6]
; ** Password.wbt
; ** Demonstrates a password with simple encoding
; **
;*****

PasswordEntryFormat=`WWWDLGED,6.2`

PasswordEntryCaption=`Enter a password and confirm`
PasswordEntryX=012
PasswordEntryY=031
PasswordEntryWidth=094
PasswordEntryHeight=056
PasswordEntryNumControls=005
PasswordEntryProcedure=`DEFAULT`
PasswordEntryFont=`DEFAULT`
PasswordEntryTextColor=`DEFAULT`
PasswordEntryBackground=`DEFAULT,DEFAULT`
```

```
PasswordEntryConfig=0
```

```
PasswordEntry001=`005,009,032,012,STATICTEXT,"StaticText_Password",DEFAULT,"Password",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
PasswordEntry002=`005,023,032,012,STATICTEXT,"StaticText_Confirm",DEFAULT,"Confirm",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
PasswordEntry003=`037,009,050,012,EDITBOX,"EditBox_1",pw_Password1,DEFAULT,DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
PasswordEntry004=`037,023,050,012,EDITBOX,"EditBox_2",pw_Password2,DEFAULT,DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
PasswordEntry005=`017,039,050,012,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,50,32,DEFAULT,DEFAULT,DEFAULT`
```

```
While @TRUE
```

```
    pw_Password1 = ""
```

```
    pw_Password2 = ""
```

```
    ButtonPushed=Dialog("PasswordEntry")
```

```
    If( ( pw_Password1 != "" ) && ( pw_Password1 == pw_Password2 ) )
```

```
Then Break
```

```
EndWhile
```

```
nKeycode = 0
```

```
For i = 1 to StrLen( pw_Password1 )
```

```
    nKeycode = nKeycode + ( Char2Num( StrSub( pw_Password1, i, 1 ) ) * i )
```

```
Next
```

```
pw_Password = AskPassword( "Enter password", "Your checksum is " :  
nKeycode )
```

```
nCodeTest = 0
```

```
For i = 1 to StrLen( pw_Password )
```

```
    nCodeTest = nCodeTest + ( Char2Num( StrSub( pw_Password, i, 1 ) ) *  
i )
```

```
Next
```

```
If( nCodeTest == nKeycode ) Then Message( "Report", "We have a match" )
```

```
Else Message( "Report", "You goofed!" )
```

```
exit
```

Introduction to Programming

WaitForKey.wbt

```
;*****  
;**  
;** [Chapter 6]  
;** WaitForKey.wbt  
;** Demonstrates a the WaitForKey function  
;**  
;*****  
  
BoxOpen( "Waiting for keystroke", "Select F1, F2, F5, <spacebar> or  
Insert" )  
nKey = WaitForKey( "{F1}", "{F2}", "{F5}", " ", "{INSERT}" )  
Select nKey  
    Case 1  
        Message( "You pressed", "the F1 key" )  
        Break  
    Case 2  
        Message( "You pressed", "the F2 key" )  
        Break  
    Case 3  
        Message( "You pressed", "the F5 key" )  
        Break  
    Case 4  
        Message( "You pressed", "the spacebar" )  
        Break  
    Case 5  
        Message( "You pressed", "the Insert key" )  
        Break  
EndSelect  
exit
```

Chapter 7 Samples

Music.txt

SKU	TITLE	URL
001	The Beat Goes On	http://www.youtube.com/watch?v=F5fsqYctXgM
002	Sixteen Tons	http://www.youtube.com/watch?v=Joo90ZWwUkU

```

003 Love Potion Number 9
    http://www.youtube.com/watch?v=7rXhXLsNJL8&NR=1
004 Fifty Ways To Leave Your Lover  http://www.youtube.com/watch?v=b5--
Sje98jI
005 For All We Know    http://www.youtube.com/watch?v=exhiNToY3eI
006 L.A. International
Airport,http://www.youtube.com/watch?v=Aj8f30Iguw0

```

HyperLink.wbt

```

;*****
;
;**  [Chapter 7]
;**  HyperLink.wbt
;**  Demonstrates subprocedures and creating formatted files
;**  requires music.txt in same directory
;*****

DirChange( DirScript() )
sFileIn  = "music.txt"
sFileOut = ""

gosub selectFile
gosub selectColumn
gosub testSelection
gosub processFile
exit

;=====
;  Select the input (source) file
;=====
:selectFile
    sFileIn= AskFileName( "Select tab-delimited source", "", "Text
Files|*.txt", "*.txt", 1)
return

;=====
;  Show the fields to permit column selection
;=====
:selectColumn

```

Introduction to Programming

```
SelectColumnFormat=`WWWDLGED,6.2`

SelectColumnCaption=`Column selection`
SelectColumnX=035
SelectColumnY=053
SelectColumnWidth=147
SelectColumnHeight=116
SelectColumnNumControls=008
SelectColumnProcedure=`DEFAULT`
SelectColumnFont=`DEFAULT`
SelectColumnTextColor=`DEFAULT`
SelectColumnBackground=`DEFAULT,DEFAULT`
SelectColumnConfig=0

SelectColumn001=`004,004,067,011,STATICTEXT,"StaticText_1",DEFAULT,"
Select LABEL column",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

SelectColumn002=`004,020,067,067,ITEMBOX,"ItemBox_1",listLabelColumn
,DEFAULT,DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

SelectColumn003=`076,004,067,011,STATICTEXT,"StaticText_2",DEFAULT,"
Select HYPERLINK column",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

SelectColumn004=`076,020,067,067,ITEMBOX,"ItemBox_2",listHyperlinkCo
lumn,DEFAULT,DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

SelectColumn005=`006,089,087,011,STATICTEXT,"StaticText_3",DEFAULT,"
Select output file type",DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

SelectColumn006=`006,100,042,011,RADIOBUTTON,"RadioButton_HTML",rbHy
pertext,"HTML",1,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

SelectColumn007=`049,100,043,011,RADIOBUTTON,"RadioButton_TAB-
delimited",rbHypertext,"TAB-
delimited",2,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

SelectColumn008=`100,097,043,011,PUSHBUTTON,"PushButton_OK",DEFAULT,
"OK",1,80,32,DEFAULT,DEFAULT,DEFAULT`

listColumns = ""
listLabelColumn = ""
listHyperlinkColumn = ""
hFileIn = FileOpen( sFileIn, "READ" ); open the input file and get a
handle
sLineIn = FileRead( hFileIn )
For i = 1 to ItemCount( sLineIn, @TAB )
    sEntry = ItemExtract( i, sLineIn, @TAB )
    listColumns = ItemInsert( sEntry, -1, listLabelColumn, @TAB )
```



```

        listLabelColumn = listColumns
        listHyperlinkColumn = listColumns
    Next
    Dialog( "SelectColumn" )
return

;=====
;   Requests confirmation of the file and option selection
;=====
:testSelection
    nHyperLink = ItemLocate( listHyperlinkColumn, listColumns, @TAB )
    nLabel = ItemLocate( listLabelColumn, listColumns, @TAB )
    If( nLabel == 0 ) || ( nHyperLink == 0 ) Then exit
    GoSub formatLine
    sReport = 'The output format will appear in the format:
[':sLineOut:']. If this is correct, select "Yes" to continue.'
    If AskYesNo( "Question", sReport ) == @NO Then exit
return

;=====
;   Processes the input file according to selections
;=====
:processFile
    sFileIn = FileFullName( sFileIn )
    sFileOut = StrFix( sFileIn, "", StrLen( sFileIn ) - 4 )
    If rbHypertext == 1 then
        sFileOut = StrCat( sFileOut, ".HTML" )
    Else
        sFileOut = StrCat( sFileOut, ".LST" )
    EndIf
    hFileOut = FileOpen( sFileOut, "WRITE" ) ; open the output file and
get a handle
    If rbHypertext == 1
        FileWrite( hFileOut, "<html>" )
        FileWrite( hFileOut, "<body>" )
    EndIf
    While @TRUE
        sLineIn = FileRead( hFileIn )

```

Introduction to Programming

```
    If sLineIn == "*EOF*" Then break
    gosub formatLine
    If sRef != ""
        FileWrite( hFileOut, sLineOut )
    EndIf
EndWhile

If rbHypertext == 1
    FileWrite( hFileOut, "</body>" )
    FileWrite( hFileOut, "</html>" )
EndIf
FileClose( hFileIn ) ; close the input file
FileClose( hFileOut ) ; close the output file
Message( "Done", sFileOut )
Return

;=====
;   Format the column fields for output
;=====

:formatLine
    sTemp = ""
    If rbHyperText == 2
        For i = 1 to ( Min( nLabel, nHyperlink ) - 1 )
            sTemp = StrCat( sTemp, ItemExtract( i, sLineIn, @TAB ), "
" )
        Next
    EndIf
    sRef = ItemExtract( nHyperlink, sLineIn, @TAB )
    sName = ItemExtract( nLabel, sLineIn, @TAB )
    If sRef != ""
        sLineOut = StrCat( sTemp, '<a href=', sRef, '>', sName,
'</a><br>' )
    Endif
Return
```

GetData.wbt

```
;*****
```

```

; **
; ** [Chapter 7]
; ** GetData.wbt
; ** Creates a list from the contents of a file.
; ** Parameters:
; **      param1: file to be searched
; **      param2: name of var to return
; **      param3: name of var to return number of items in list
; ****
; ****

If param0 < 3 ; insufficient arguments
    Message("Attention","This script is not meant to use used alone.
    It is used by other scripts")
    exit
Endif

If IsNumber( param3 ) Then exit ; parameter 3 isn't a variable name
If IsNumber( param2 ) Then exit ; parameter 2 isn't a variable name
If IsNumber( param1 ) Then exit ; parameter 1 isn't a filename
If FileExist( param1 ) == 0
    %param2% = "File Error"
    %param3% = 0
    return
Endif

nIndex = 0 ; initialize a count index
sResult = ""

hFileIn = FileOpen( param1, "READ" ) ; open the input file and get a
handle
While @TRUE
    sTemp = ""
    sLineIn = FileRead( hFileIn )
    If( sLineIn == "*EOF*" ) Then break
    nIndex = nIndex + 1
    sTemp = ItemExtract( 1, sLineIn, @TAB )
    sResult = StrCat( sResult, sTemp, @TAB )
EndWhile

```

Introduction to Programming

```
FileClose( hFileIn ) ; close the input file
%param2% = sResult ; assign the result string to param2
%param3% = nIndex ; assign the count to param3
Drop( sLineIn, nIndex, sResult, sTemp ) ; discard local variables
Return
```

Parts.lst

```
Webley Defaminizer      5    2456-3468-8921    27
Finagle Bolix Grinder 2    3905-1298-7892    12B
Acme Jetpack    3    9834-0909-8721    14
Hobart Skyhook 1    6435-2348-0971    8E
Forward Mass Detector 9    3498-3465-1871    5
Dyson Sphere    1    0000-0000-0001    M27-139-235-890
Niven Transporter (Pad model) 3    9872-2317-2345    17A, 18B, 21C
```

ExternCall.wbt

```
*****
;**
;**  [Chapter 7]
;**  ExternCall.wbt
;**  Demonstrates calling external .WBT programs as subroutines
;**
*****

nCount = 0
sList = ""
DirChange( DirScript() )
; open the source file and return an array of entries
Call( 'GetData.WBT', '"Parts List.lst" sList nCount' )
Message( "List count", "Found ":nCount:" items in [":sList:"]" )

Call( "SortData.WBT", "sList @TAB" )
Message( "Sort results", "Sorted ":nCount:" items as [":sList:"]" )
Exit
```

SortData.wbt

```
*****
```

```

; **
; **  [Chapter 7]
; **  SortData.wbt
; **  Demonstrates sorting a list using ItemSort
; **
; *****

If param0 < 2 Then exit
If IsNumber( param1 ) Then exit      ; should be list of data
If IsNumber( param2 ) Then exit      ; should be char (delimiter)

sList = ItemSort( %param1%, %param2% )
return

```

Run_EXE.wbt

```

; *****
; **
; **  [Chapter 7]
; **  Run_EXE.wbt
; **  Demonstrates launching an external EXE
; **
; *****

DirChange( DirScript() )

:loop
sFileRun = AskFileName( "Select executable application", DirWindows(2),
"EXE Files|*.exe", "notepad.exe", 1 )
sArgs = AskLine( "Enter arguments", "Enter arguments: (optional)", "" )
if Run( sFileRun, sArgs ) then goto loop
Message( "Error", "Can not run " : sFileRun )
goto loop
exit

```

Chapter 8 Samples

Logic.wbt

```

; *****

```

Introduction to Programming

```
;**
;** [Chapter 8]
;** Logic.wbt
;** Demonstrates Boolean logic
;**
;*****

Logic1Format=`WWDDLGED,6.2`

Logic1Caption=`Boolean Logic`
Logic1X=057
Logic1Y=074
Logic1Width=092
Logic1Height=042
Logic1NumControls=005
Logic1Procedure=`DEFAULT`
Logic1Font=`DEFAULT`
Logic1TextColor=`DEFAULT`
Logic1Background=`DEFAULT,DEFAULT`
Logic1Config=0

Logic1001=`002,001,083,011,STATICTEXT,"StaticText_1",DEFAULT,"Enter two
numbers for comparison",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Logic1002=`004,012,038,011,EDITBOX,"EditBox_1",nNumber1,DEFAULT,DEFAULT
,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Logic1003=`047,012,038,011,EDITBOX,"EditBox_2",nNumber2,DEFAULT,DEFAULT
,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Logic1004=`004,025,038,011,PUSHBUTTON,"PushButton_Test",DEFAULT,"Test",
1,40,32,DEFAULT,DEFAULT,DEFAULT`
Logic1005=`047,025,038,011,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit",
0,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed = Dialog("Logic1")

If( nNumber1 - nNumber2 )
    Message( "Boolean TRUE", nNumber1 : " does not equal " : nNumber2 )
Else
    Message( "Boolean FALSE", nNumber1 : " equals ": nNumber2 )
Endif
```

```
Exit
```

Select1.wbt

```
;*****
; **
; ** [Chapter 8]
; ** Select1.wbt
; ** Demonstrates nested if logic
; **
;*****
```

```
Country1 = "France"
Country2 = "Egypt"
Country3 = "Russia"
Country4 = "Japan"
Country5 = "England"
```

```
While @TRUE      ; loop forever
```

```
    nItem = Random( 4 ) + 1
    CountryKey = Country%nItem%
```

```
    sResponse = AskLine( "Select1", "What is the capital of " :
CountryKey : "? (case-sensitive)", "" )
```

```
    If( CountryKey == "France" ) then
        If( sResponse == "Paris" ) then
            sMessage = "Correct"
        Else
            sMessage = "Wrong"
        EndIf
    Else
        If( CountryKey == "Egypt" ) then
            If( sResponse == "Cairo" ) then
                sMessage = "Correct"
```

Introduction to Programming

```
        Else
            sMessage = "Wrong"
        EndIf
    Else
        If( CountryKey == "Russia" ) then
            If( sResponse == "Moscow" ) then
                sMessage = "Correct"
            Else
                sMessage = "Wrong"
            EndIf
        Else
            If( CountryKey == "Japan" ) then
                If( sResponse == "Tokyo" ) then
                    sMessage = "Correct"
                Else
                    sMessage = "Wrong"
                EndIf
            Else
                If( CountryKey == "England" ) then
                    If( sResponse == "London" ) then
                        sMessage = "Correct"
                    Else
                        sMessage = "Wrong"
                    EndIf
                EndIf
            EndIf
        EndIf
    EndIf
EndIf

Message( "Your guess was:", sMessage )

Endwhile
exit
```

Select2.wbt

```
;*****
```



```

; **
; ** [Chapter 8]
; ** Select2.wbt
; ** Demonstrates nested if..elseif logic
; **
; *****

listCountry = "France,Egypt,Russia,Japan,England"

While @TRUE      ; loop forever

    nItem = Random( 4 ) + 1
    sCountry = ItemExtract( nItem, listCountry, "," )
    sResponse = AskLine( "Select2", "What is the capital of " : sCountry
: "? (case-sensitive)", "" )

    If( nItem == 1 )
        sCapital = "Paris"
    ElseIf( nItem == 2 )
        sCapital = "Cairo"
    ElseIf( nItem == 3 )
        sCapital = "Moscow"
    ElseIf( nItem == 4 )
        sCapital = "Tokyo"
    ElseIf( nItem == 5 )
        sCapital = "London"
    EndIf

    If( sResponse == sCapital )
        sMessage = "Correct"
    Else
        sMessage = "Wrong"
    Endif

    Message( "Your guess was:", sMessage )

EndWhile
exit

```

Introduction to Programming

Select3.wbt

```
;*****  
;**  
;** [Chapter 8]  
;** Select3.wbt  
;** Demonstrates switch\case logic  
;**  
;*****  
  
listCountry = "France,Egypt,Russia,Japan,England"  
  
While @TRUE      ; loop forever  
  
    nItem = Random( 4 ) + 1  
    sCountry = ItemExtract( nItem, listCountry, "," )  
    sResponse = AskLine( "Select3", "What is the capital of " : sCountry  
: "? (case-sensitive)", "" )  
  
Switch nItem  
    case 1  
        sCapital = "Paris"  
        break  
    case 2  
        sCapital = "Cairo"  
        break  
    case 3  
        sCapital = "Moscow"  
        break  
    case 4  
        sCapital = "Tokyo"  
        break  
    case 5  
        sCapital = "London"  
        break  
EndSwitch  
  
If( sResponse == sCapital )
```

```

        sMessage = "Correct"
    Else
        sMessage = "Wrong"
    EndIf

    Message( "Your guess was:", sMessage )

Endwhile
exit

```

Prime.wbt

```

;*****
;**
;**  [Chapter 8]
;**  Prime.wbt
;**  Demonstrates for loops by searching for prime numbers
;**
;*****

sList = ""
For i = 9 to 1001 by 2
    bPrime = @TRUE
    For j = 3 to Sqrt( i )
        If( i mod j == 0 )
            bPrime = @FALSE
            break
        Endif
    Next ; j
    if bPrime then sList = sList : i : " "
Next ; i
Message( "Primes found are:", sList )
exit

```

ForEach.wbt

```

;*****
;**

```

Introduction to Programming

```
;** [Chapter 8]
;** ForEach.wbt
;** Demonstrates foreach by looping thru elements in an array
;**
;*****

strList = "a,b,c,d,e,f"
; Convert list to an array
arrA = ObjectType ("ARRAY", Arrayize (strList, ","))

intItem = 0
; Access each element in the array
ForEach arrElement in arrA
    intItem = intItem + 1
    Message (intItem, arrElement)
Next
exit
```

Prime2.wbt

```
;*****
;**
;** [Chapter 8]
;** Prime2.wbt
;** Demonstrates while loops by searching for prime numbers
;**
;*****

sList = ""
i = 9
While( i < 1001 )
    bPrime = @TRUE
    j = 3
    While( j <= Sqrt( i ) )
        If( i mod j == 0 )
            bPrime = @FALSE
            break
        Endif
    j = j + 2
```

```

    EndWhile
    if bPrime then sList = sList : i : " "
    i = i + 2
EndWhile
Message( "Primes found are:", sList )
exit

```

Chapter 9 Samples

Average.wbt

```

;*****
;**
;** [Chapter 9]
;** Average.wbt
;** Demonstrates the Average function
;**
;*****

fList = "1.2, 2.3, 3.4, 4.5, 5.6, 6.7, 7.8, 8.9, 9.0"
fAvg = Average( %fList% )
Message( "Average", "The average value of ":fList:" is ":fAvg )
exit

```

Floor_Ceiling.wbt

```

;*****
;**
;** [Chapter 9]
;** Floor_Ceiling.wbt
;** Demonstrates the Floor/Ceiling functions
;**
;*****

fVal = AskLine( "Ceiling/Floor", "Please enter a decimal number (ie.
1.234)", "3.1415" )
nCeiling = Ceiling( fVal )
nFloor = Floor( fVal )
Message( "Ceiling and Floor of ":fVal, "Ceiling: ":nCeiling:", Floor:
":nFloor )

```

Introduction to Programming

exit

Decimals.wbt

```
;*****  
;**  
;** [Chapter 9]  
;** Decimals.wbt  
;** Demonstrates the Decimals function  
;**  
;*****  
  
fVal = 0.9876543210  
Decimals( -1 )  
Message( "Decimals (full)", fVal )  
For d = 0 to 10  
    Decimals( d )  
    Message( "Decimals: " : d, fVal )  
Next  
exit
```

Min_Max.wbt

```
;*****  
;**  
;** [Chapter 9]  
;** Min_Max.wbt  
;** Demonstrates the Min/Max functions  
;**  
;*****  
  
fMin = Min( -7.8, 1.2, 560, 0.34, 45, 6.7, 8.9, -2.3, -90 )  
fMax = Max( -7.8, 1.2, 560, 0.34, 45, 6.7, 8.9, -2.3, -90 )  
Message( "Min/Max", "The largest value is ":fMax:", the smallest is  
":fMin )  
exit
```

TestNumber.wbt

```
;*****  
;**  
;** [Chapter 9]
```

```

; ** TestNumber.wbt
; ** Demonstrates the IsNumber, IsFloat and IsInt functions
; **
; *****

sStringVal = "This is not a number"
sFloatVal = "0.0123456"
sIntVal = "123456"

If IsNumber( sStringVal )
    sNumber = "is a number"
Else
    sNumber = "is not a number"
EndIf

If IsFloat( sStringVal )
    sFloat = "is a floating point value"
Else
    sFloat = "is not a floating point value"
EndIf

If IsInt( sStringVal )
    sInt = "is an integer"
Else
    sInt = "is not an integer"
EndIf

Message( 'Testing',
'':sStringVal:':':@CRLF:sNumber:@CRLF:sFloat:@CRLF:sInt )

If IsNumber( sFloatVal )
    sNumber = "is a number"
Else
    sNumber = "is not a number"
EndIf

If IsFloat( sFloatVal )
    sFloat = "is a floating point value"

```

Introduction to Programming

```
Else
    sFloat = "is not a floating point value"
EndIf

If IsInt( sFloatVal )
    sInt = "is an integer"
Else
    sInt = "is not an integer"
EndIf

Message( "Testing", sFloatVal:@CRLF:sNumber:@CRLF:sFloat:@CRLF:sInt )

If IsNumber( sIntVal )
    sNumber = "is a number"
Else
    sNumber = "is not a number"
EndIf

If IsFloat( sIntVal )
    sFloat = "is a floating point value"
Else
    sFloat = "is not a floating point value"
EndIf

If IsInt( sIntVal )
    sInt = "is an integer"
Else
    sInt = "is not an integer"
EndIf

Message( "Testing", sIntVal:@CRLF:sNumber:@CRLF:sFloat:@CRLF:sInt )
exit
```

Random.wbt

```
;*****
; **
; ** [Chapter 9]
; ** TestNumber.wbt
```



```

; ** Demonstrates the Random function
; **
; *****

RandomFormat=`WWDLGED,6.2`

RandomCaption=`Random`
RandomX=056
RandomY=057
RandomWidth=104
RandomHeight=049
RandomNumControls=005
RandomProcedure=`DEFAULT`
RandomFont=`DEFAULT`
RandomTextColor=`DEFAULT`
RandomBackground=`DEFAULT,DEFAULT`
RandomConfig=0

Random001=`062,001,038,011,EDITBOX,"EditBox_1",nMax,"100",DEFAULT,10,DE
FAULT,DEFAULT,DEFAULT,DEFAULT`
Random002=`004,004,056,011,STATICTEXT,"StaticText_1",DEFAULT,"Enter a
maximum value",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Random003=`006,033,038,011,PUSHBUTTON,"PushButton_1",DEFAULT,"New
Number",1,30,32,DEFAULT,DEFAULT,DEFAULT`
Random004=`004,017,096,011,VARYTEXT,"VaryText_1",sReport,"A random
number will be reported
here",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Random005=`057,033,038,011,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Canc
el",0,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

While @TRUE
    ButtonPushed = Dialog("Random")
    nRandom = Random( nMax )
    sReport = StrCat( "The random number is: ", nRandom )
EndWhile
exit

```

Exponential.wbt

```

; *****
; **

```

Introduction to Programming

```
;** [Chapter 9]
;** Exponential.wbt
;** Demonstrates the Random function
;**
;*****

ExponentialFormat=`WWWDLGED,6.2`

ExponentialCaption=`Exponential`
ExponentialX=056
ExponentialY=057
ExponentialWidth=174
ExponentialHeight=054
ExponentialNumControls=005
ExponentialProcedure=`DEFAULT`
ExponentialFont=`DEFAULT`
ExponentialTextColor=`DEFAULT`
ExponentialBackground=`DEFAULT,DEFAULT`
ExponentialConfig=0

Exponential001=`069,005,038,010,EDITBOX,"EditBox_1",fExp,"2.302585093",
DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Exponential002=`011,005,056,010,STATICTEXT,"StaticText_1",DEFAULT,"Enter a value",
DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Exponential003=`029,035,038,010,PUSHBUTTON,"PushButton_Calculate",DEFAULT,
LT,"Calculate",1,30,32,DEFAULT,DEFAULT,DEFAULT`
Exponential004=`011,019,156,010,VARYTEXT,"VaryText_1",sReport,"The
exponential value will be reported
here",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Exponential005=`083,035,038,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,
"Cancel",0,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

While @TRUE
    ButtonPushed = Dialog("Exponential")
    fVal = Exp( fExp )
    sReport = "The value of Exp(": fExp : ") is: " : fVal
EndWhile
exit
```

LogE.wbt

```

;*****
;**
;**  [Chapter 9]
;**  LogE.wbt
;**  Demonstrates the LogE function
;**
;*****

```

```
LogEFormat=`WWDDLGED,6.2`
```

```
LogECaption=`LogE`
```

```
LogEX=056
```

```
LogEY=057
```

```
LogEWidth=156
```

```
LogEHeight=054
```

```
LogENumControls=005
```

```
LogEProcedure=`DEFAULT`
```

```
LogEFont=`DEFAULT`
```

```
LogETextColor=`DEFAULT`
```

```
LogEBackground=`DEFAULT,DEFAULT`
```

```
LogEConfig=0
```

```
LogE001=`071,005,038,010,EDITBOX,"EditBox_1",fVal,"100",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
LogE002=`013,005,056,010,STATICTEXT,"StaticText_1",DEFAULT,"Enter a value",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
LogE003=`029,035,038,010,PUSHBUTTON,"PushButton_Calculate",DEFAULT,"Calculate",1,30,32,DEFAULT,DEFAULT,DEFAULT`
```

```
LogE004=`013,017,132,010,VARYTEXT,"VaryText_1",sReport,"The natural log will be reported here",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
LogE005=`079,035,038,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel",0,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
While @TRUE
```

```
    ButtonPushed = Dialog("LogE")
```

```
    fExp = LogE( fVal )
```

```
    sReport = "The natural log of " : fVal: " is: " : fExp
```

```
EndWhile
```

Introduction to Programming

exit

Log10.wbt

```
;*****  
;**  
;** [Chapter 9]  
;** Log10.wbt  
;** Demonstrates the Log10 function  
;**  
;*****
```

Log10Format=`WWDLGED,6.2`

Log10Caption=`Log10`

Log10X=056

Log10Y=057

Log10Width=158

Log10Height=054

Log10NumControls=005

Log10Procedure=`DEFAULT`

Log10Font=`DEFAULT`

Log10TextColor=`DEFAULT`

Log10Background=`DEFAULT,DEFAULT`

Log10Config=0

Log10001=`069,003,038,010,EDITBOX,"EditBox_1",fVal,"10000",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Log10002=`013,005,056,010,STATICTEXT,"StaticText_1",DEFAULT,"Enter a value",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Log10003=`025,035,038,010,PUSHBUTTON,"PushButton_Calculate",DEFAULT,"Calculate",1,30,32,DEFAULT,DEFAULT,DEFAULT`

Log10004=`011,017,134,010,VARYTEXT,"VaryText_1",sReport,"The base-10 log will be reported here",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Log10005=`075,035,038,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel",0,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

While @TRUE

 ButtonPushed = Dialog("Log10")

 fExp = Log10(fVal)

```

    sReport = "The base-10 log of " : fVal : " is: " : fExp
EndWhile
exit

```

SquareRoot.wbt

```

;*****
;**
;**  [Chapter 9]
;**  SquareRoot.wbt
;**  Demonstrates the Sqrt function
;**
;*****

```

```
SquareRootFormat=`WWWDLGED,6.2`
```

```
SquareRootCaption=`Square Root`
```

```
SquareRootX=056
```

```
SquareRootY=057
```

```
SquareRootWidth=154
```

```
SquareRootHeight=058
```

```
SquareRootNumControls=005
```

```
SquareRootProcedure=`DEFAULT`
```

```
SquareRootFont=`DEFAULT`
```

```
SquareRootTextColor=`DEFAULT`
```

```
SquareRootBackground=`DEFAULT,DEFAULT`
```

```
SquareRootConfig=0
```

```
SquareRoot001=`063,003,038,010,EDITBOX,"EditBox_1",fVal,"456",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
SquareRoot002=`005,005,056,010,STATICTEXT,"StaticText_1",DEFAULT,"Enter a value",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
SquareRoot003=`025,037,038,010,PUSHBUTTON,"PushButton_Calculate",DEFAULTT,"Calculate",1,30,32,DEFAULT,DEFAULT,DEFAULT`
```

```
SquareRoot004=`005,019,132,010,VARYTEXT,"VaryText_1",sReport,"The square root will be reported here",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
SquareRoot005=`075,037,038,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel",0,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
While @TRUE
```

Introduction to Programming

```
ButtonPushed = Dialog("SquareRoot")
fSquareRoot = Sqrt( Fabs( fVal ) )
sSign = ""
if fVal != Fabs( fVal ) then sSign = "i"
sReport = "The square root of " : fVal : " is: " : fSquareRoot :
sSign
EndWhile
exit
```

Trig.wbt

```
;*****
; **
; ** [Chapter 9]
; ** Trig.wbt
; ** Demonstrates the trigonometric operations
; **
;*****
```

```
TrigFormat=`WWDDLGED,6.2`
```

```
TrigCaption=`Trig`
```

```
TrigX=056
```

```
TrigY=056
```

```
TrigWidth=102
```

```
TrigHeight=073
```

```
TrigNumControls=008
```

```
TrigProcedure=`DEFAULT`
```

```
TrigFont=`DEFAULT`
```

```
TrigTextColor=`DEFAULT`
```

```
TrigBackground=`DEFAULT,DEFAULT`
```

```
TrigConfig=0
```

```
Trig001=`060,001,035,011,EDITBOX,"EditBox_1",nDegrees,"135",DEFAULT,10,
DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
Trig002=`004,001,056,011,STATICTEXT,"StaticText_1",DEFAULT,"Enter angle
in degrees",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
Trig003=`002,017,096,011,VARYTEXT,"VaryText_1",sReport,"The trigometric
values appear here",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```

Trig004=`004,026,094,011,VARYTEXT,"VaryText_2",sSine,DEFAULT,DEFAULT,40
,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Trig005=`004,036,094,011,VARYTEXT,"VaryText_3",sCosine,DEFAULT,DEFAULT,
50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Trig006=`004,046,094,010,VARYTEXT,"VaryText_4",sTangent,DEFAULT,DEFAULT
,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
Trig007=`004,058,035,011,PUSHBUTTON,"PushButton_Calculate",DEFAULT,"Cal
culate",1,70,32,DEFAULT,DEFAULT,DEFAULT`
Trig008=`060,058,035,011,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel
",0,80,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

```

```

fSine = 0.0
fCosine = 0.0
fTangent = 0.0

```

```

While @TRUE
    ButtonPushed = Dialog("Trig")
    While( nDegrees < -360 )
        nDegrees = nDegrees + 360
    EndWhile
    While( nDegrees > 360 )
        nDegrees = nDegrees - 360
    EndWhile

    fRadians = nDegrees * @DEG2RAD
    fSine = Sin( fRadians )
    fCosine = Cos( fRadians )
    fTangent = Tan( fRadians )
    sReport = StrCat( "For an angle of ", nDegrees, " degrees:" )
    sSine = StrCat( "the sine is ", fSine )
    sCosine = StrCat( "the cosine is ", fCosine )
    sTangent = StrCat( "the tangent is ", fTangent )
EndWhile
exit

```

ArcSin.wbt

```

;*****
;
;
;** [Chapter 9]

```

Introduction to Programming

```
;**  ArcSin.wbt
;**  Demonstrates the ASin function
;**
;*****

ArcSinFormat=`WWWDLGED,6.2`

ArcSinCaption=`ArcSin`
ArcSinX=-001
ArcSinY=-001
ArcSinWidth=146
ArcSinHeight=072
ArcSinNumControls=006
ArcSinProcedure=`DEFAULT`
ArcSinFont=`DEFAULT`
ArcSinTextColor=`DEFAULT`
ArcSinBackground=`DEFAULT,DEFAULT`
ArcSinConfig=0

ArcSin001=`099,003,034,010,EDITBOX,"EditBox_1",fSine,"0.70710678",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
ArcSin002=`003,003,090,010,STATICTEXT,"StaticText_1",DEFAULT,"Enter the sine as -1.0 to 1.0",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
ArcSin003=`015,023,094,010,VARYTEXT,"VaryText_1",sReport,"The angle in degrees appears here",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
ArcSin004=`015,035,090,010,VARYTEXT,"VaryText_2",sAngle,DEFAULT,DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
ArcSin005=`023,051,034,010,PUSHBUTTON,"PushButton_Calculate",DEFAULT,"Calculate",1,50,32,DEFAULT,DEFAULT,DEFAULT`
ArcSin006=`085,051,034,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel",0,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

While @TRUE
    ButtonPushed = Dialog("ArcSin")
    If( fabs( fSine ) > 1.0 )
        sReport = "value out of range"
    Else
        fRadians = ASin( fSine )
        nDegrees = fRadians * @RAD2DEG
```



```

sReport = StrCat( "For a sine of ", fSine )
sAngle  = StrCat( "the angle in degrees is ", nDegrees )

Endif
EndWhile
exit

```

ArcCosine.wbt

```

;*****
;**
;**  [Chapter 9]
;**  ArcCosine.wbt
;**  Demonstrates the ACos function
;**
;*****

ArcCosineFormat=`WWWDLGED,6.2`

ArcCosineCaption=`ArcCosine`
ArcCosineX=-001
ArcCosineY=-001
ArcCosineWidth=132
ArcCosineHeight=060
ArcCosineNumControls=006
ArcCosineProcedure=`DEFAULT`
ArcCosineFont=`DEFAULT`
ArcCosineTextColor=`DEFAULT`
ArcCosineBackground=`DEFAULT,DEFAULT`
ArcCosineConfig=0

ArcCosine001=`081,003,034,010,EDITBOX,"EditBox_1",fCosine,"-
0.70710678",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
ArcCosine002=`005,005,074,010,STATICTEXT,"StaticText_1",DEFAULT,"Enter
the cosine as -1.0 to 1.0",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
ArcCosine003=`013,023,096,010,VARYTEXT,"VaryText_1",sReport,"The angle
in degrees appears here",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
ArcCosine004=`015,033,094,010,VARYTEXT,"VaryText_2",sAngle,DEFAULT,DEFA
ULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
ArcCosine005=`017,045,034,010,PUSHBUTTON,"PushButton_Calculate",DEFAULT
,"Calculate",1,50,32,DEFAULT,DEFAULT,DEFAULT`

```

Introduction to Programming

```
ArcCosine006=`069,045,034,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel",0,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
While @TRUE
    ButtonPushed = Dialog("ArcCosine")
    If( fabs( fCosine ) > 1.0 )
        sReport = "value out of range"
    Else
        fRadians = ACos( fCosine )
        nDegrees = fRadians * @RAD2DEG
        sReport = StrCat( "For a cosine of ", fCosine )
        sAngle = StrCat( "the angle in radians is ", nDegrees )
    Endif
EndWhile
exit
```

ArcTangent.wbt

```
*****
; **
; ** [Chapter 9]
; ** ArcTangent.wbt
; ** Demonstrates the ATan function
; **
*****
```

```
ArcTangentFormat=`WWDLGED,6.2`
```

```
ArcTangentCaption=`ArcTangent`
```

```
ArcTangentX=056
```

```
ArcTangentY=056
```

```
ArcTangentWidth=102
```

```
ArcTangentHeight=054
```

```
ArcTangentNumControls=006
```

```
ArcTangentProcedure=`DEFAULT`
```

```
ArcTangentFont=`DEFAULT`
```

```
ArcTangentTextColor=`DEFAULT`
```

```
ArcTangentBackground=`DEFAULT,DEFAULT`
```

```
ArcTangentConfig=0
```

```

ArcTangent001=`060,001,035,011,EDITBOX,"EditBox_1",fTangent,"2.84147099
",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ArcTangent002=`004,001,056,011,STATICTEXT,"StaticText_1",DEFAULT,"Enter
the tangent",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ArcTangent003=`002,018,096,011,VARYTEXT,"VaryText_1",sReport,"The angle
in degrees appears here",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ArcTangent004=`004,028,094,011,VARYTEXT,"VaryText_2",sAngle,DEFAULT,DEF
AULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

ArcTangent005=`004,038,035,010,PUSHBUTTON,"PushButton_Calculate",DEFAULT
T,"Calculate",1,50,32,DEFAULT,DEFAULT,DEFAULT`

ArcTangent006=`060,038,035,010,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"
Cancel",0,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

```

```

While @TRUE
    ButtonPushed = Dialog("ArcTangent")
    If fTangent == 0.0
        sReport = "value out of range"
    Else
        fRadians = ATan( fTangent )
        nDegrees = fRadians * @RAD2DEG
        sReport = StrCat( "For a tangent of ", fTangent )
        sAngle = StrCat( "the angle in degrees is ", nDegrees )
    Endif
EndWhile
exit

```

HyperTrig.wbt

```

;*****
;**
;**  [Chapter 9]
;**  HyperTrig.wbt
;**  Demonstrates the Hyperbolic functions
;**
;*****

```

```
HyperTrigFormat=`WWWDLGED,6.2`
```

```
HyperTrigCaption=`Hyperbolic`
```

```
HyperTrigX=-001
```

Introduction to Programming

```
HyperTrigY=-001
HyperTrigWidth=102
HyperTrigHeight=073
HyperTrigNumControls=008
HyperTrigProcedure=`DEFAULT`
HyperTrigFont=`DEFAULT`
HyperTrigTextColor=`DEFAULT`
HyperTrigBackground=`DEFAULT,DEFAULT`
HyperTrigConfig=0

HyperTrig001=`060,001,035,011,EDITBOX,"EditBox_1",nDegrees,"135",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
HyperTrig002=`004,001,056,011,STATICTEXT,"StaticText_1",DEFAULT,"Enter angle in degrees",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
HyperTrig003=`002,017,096,011,VARYTEXT,"VaryText_1",sReport,"The hyperbolic values appear here",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
HyperTrig004=`004,026,094,011,VARYTEXT,"VaryText_2",sSine,DEFAULT,DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
HyperTrig005=`004,036,094,011,VARYTEXT,"VaryText_3",sCosine,DEFAULT,DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
HyperTrig006=`004,046,094,010,VARYTEXT,"VaryText_4",sTangent,DEFAULT,DEFAULT,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
HyperTrig007=`004,058,035,011,PUSHBUTTON,"PushButton_Calculate",DEFAULT,"Calculate",1,70,32,DEFAULT,DEFAULT,DEFAULT`
HyperTrig008=`060,058,035,011,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel",0,80,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

fSine = 0.0
fCosine = 0.0
fTangent = 0.0

While @TRUE
    ButtonPushed = Dialog("HyperTrig")
    While( nDegrees < -360 )
        nDegrees = nDegrees + 360
    EndWhile
    While( nDegrees > 360 )
        nDegrees = nDegrees - 360
    EndWhile
```

```

fRadians = nDegrees * @DEG2RAD
fSineH = SinH( fRadians )
fCosineH = CosH( fRadians )
fTangentH = TanH( fRadians )
sReport = StrCat( "For an angle of ", nDegrees, " degrees:" )
sSine = StrCat( "the hyperbolic sine is ", fSineH )
sCosine = StrCat( "the hyperbolic cosine is ", fCosineH )
sTangent = StrCat( "the hyperbolic tangent is ", fTangentH )
EndWhile
exit

```

TimeCheck.wbt

```

;*****
;**
;** [Chapter 9]
;** TimeCheck.wbt
;** Demonstrates the TimeDate function
;**
;*****

TimeCheckFormat=`WWWDLGED,6.2`

TimeCheckCaption=`Time Check`
TimeCheckX=-001
TimeCheckY=-001
TimeCheckWidth=131
TimeCheckHeight=054
TimeCheckNumControls=003
TimeCheckProcedure=`DEFAULT`
TimeCheckFont=`DEFAULT`
TimeCheckTextColor=`DEFAULT`
TimeCheckBackground=`DEFAULT,DEFAULT`
TimeCheckConfig=0

TimeCheck001=`009,031,033,011,PUSHBUTTON,"PushButton_OK",DEFAULT,"Check
",1,10,32,DEFAULT,DEFAULT,DEFAULT`

```

Introduction to Programming

```
TimeCheck002=`084,031,033,011,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel",0,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
TimeCheck003=`009,010,108,011,VARYTEXT,"dtReport",dtReport,DEFAULT,DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
While @TRUE
    dtReport = TimeDate()
    ButtonPushed = Dialog("TimeCheck")
EndWhile
exit
```

TimeCheck2.wbt

```
;*****
; **
; ** [Chapter 9]
; ** TimeCheck2.wbt
; ** Demonstrates the TimeYmdHms function
; **
;*****
```

```
TimeCheckFormat=`WWWDLGED,6.2`
```

```
TimeCheckCaption=`Time Check 2`
```

```
TimeCheckX=-001
```

```
TimeCheckY=-001
```

```
TimeCheckWidth=131
```

```
TimeCheckHeight=054
```

```
TimeCheckNumControls=003
```

```
TimeCheckProcedure=`DEFAULT`
```

```
TimeCheckFont=`DEFAULT`
```

```
TimeCheckTextColor=`DEFAULT`
```

```
TimeCheckBackground=`DEFAULT,DEFAULT`
```

```
TimeCheckConfig=0
```

```
TimeCheck001=`009,031,033,011,PUSHBUTTON,"PushButton_OK",DEFAULT,"Check",1,10,32,DEFAULT,DEFAULT,DEFAULT`
```

```
TimeCheck002=`084,031,033,011,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel",0,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
TimeCheck003=`009,010,108,011,VARYTEXT,"dtReport",dtReport,DEFAULT,DEFA
ULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
While @TRUE
    dtReport = TimeYmdHms()
    ButtonPushed = Dialog("TimeCheck")
EndWhile
exit
```

TimeCheck3.wbt

```
;*****
; **
; ** [Chapter 9]
; ** TimeCheck3.wbt
; ** Demonstrates Julian dates
; **
;*****
```

```
TimeCheckFormat=`WWWDLGED,6.2`
```

```
TimeCheckCaption=`Time Check 3`
TimeCheckX=-001
TimeCheckY=-001
TimeCheckWidth=131
TimeCheckHeight=054
TimeCheckNumControls=003
TimeCheckProcedure=`DEFAULT`
TimeCheckFont=`DEFAULT`
TimeCheckTextColor=`DEFAULT`
TimeCheckBackground=`DEFAULT,DEFAULT`
TimeCheckConfig=0
```

```
TimeCheck001=`009,031,033,011,PUSHBUTTON,"PushButton_OK",DEFAULT,"Check
",1,10,32,DEFAULT,DEFAULT,DEFAULT`
```

```
TimeCheck002=`084,031,033,011,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"C
ancel",0,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
TimeCheck003=`009,010,108,011,VARYTEXT,"dtReport",dtReport,DEFAULT,DEFA
ULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

Introduction to Programming

```
While @TRUE
    nJulianDay = TimeJulianDay( TimeYmdHms() )
    nDayCount = ( ( nJulianDay + 5 ) mod 7 )
    sWeekDay = ItemExtract( nDayCount + 1, "Sunday Monday Tuesday
Wednesday Thursday Friday Saturday", " " )
    dtReport = StrCat( "Julian date: ", nJulianDay, @CRLF, "Day of week:
", sWeekDay )
    ButtonPushed = Dialog("TimeCheck")
EndWhile
exit
```

Mortgage.wbt

```
;*****
; **
; ** [Chapter 9]
; ** Mortgage.wbt
; ** Demonstrates a mortgage calculation
; **
;*****
```

```
DirChange( DirScript() )
```

```
MortgageFormat=`WWWDLGED,6.2`
```

```
MortgageCaption=`Mortgage Calculator`
```

```
MortgageX=057
```

```
MortgageY=074
```

```
MortgageWidth=174
```

```
MortgageHeight=079
```

```
MortgageNumControls=014
```

```
MortgageProcedure=`DEFAULT`
```

```
MortgageFont=`DEFAULT`
```

```
MortgageTextColor=`DEFAULT`
```

```
MortgageBackground=`DEFAULT,DEFAULT`
```

```
MortgageConfig=0
```

```
Mortgage001=`006,004,050,011,STATICTEXT,"StaticText_1",DEFAULT,"Loan
Amount ($)",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```



```

Mortgage002=`004,014,051,010,EDITBOX,"EditBox_1",sPrincipal,"100000",DE
FAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage003=`062,004,050,011,STATICTEXT,"StaticText_2",DEFAULT,"%%
Interest (APR)",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage004=`060,014,051,010,EDITBOX,"EditBox_2",sPercent,"6.5%",DEFAU
LT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage005=`118,004,050,011,STATICTEXT,"StaticText_3",DEFAULT,"Term
(months)",DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage006=`116,014,051,010,EDITBOX,"EditBox_3",sTerm,"360",DEFAULT,60
,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage007=`006,036,054,011,STATICTEXT,"StaticText_4",DEFAULT,"1st
Payment Date",DEFAULT,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage008=`004,044,051,011,VARYTEXT,"VaryText_1",sStartDate,"date",DE
FAULT,80,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage009=`062,036,050,011,STATICTEXT,"StaticText_5",DEFAULT,"Payment
Amount",DEFAULT,90,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage010=`062,044,050,011,VARYTEXT,"VaryText_2",sPaymentAmt,DEFAULT,
DEFAULT,100,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage011=`118,036,050,011,STATICTEXT,"StaticText_6",DEFAULT,"Total
Interest",DEFAULT,110,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage012=`118,044,050,011,VARYTEXT,"VaryText_3",sInterestTot,DEFAULT
,DEFAULT,120,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

Mortgage013=`004,062,051,010,PUSHBUTTON,"PushButton_Calculate",DEFAULT,
"Calculate",1,130,32,DEFAULT,DEFAULT,DEFAULT`

Mortgage014=`118,062,050,010,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit
",0,140,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

bExternal = @FALSE

sStartDate = ""
Decimals( 2 )

While @TRUE
    ButtonPushed = Dialog( "Mortgage" )

    fPrincipal = sPrincipal
    fPercentAPR = sPercent
    nTerm = sTerm
    sDate = TimeAdd( TimeYmdHms(), "0000:00:30:00:00:00" )
    sStartDate = StrCat( ItemExtract( 2, sDate, ":" ), " / ",
ItemExtract( 3, sDate, ":" ), " / ", ItemExtract( 1, sDate, ":" ) )

```

Introduction to Programming

```
fPrincipal = StrTrim( fPrincipal ) ; trim all leading or trailing
blanks

If fPrincipal == ""
    Display( 10, "Entry error", "A loan amount is required" )
    Continue ; loop_on_error
EndIf

If ! IsNumber( fPrincipal )
    fPrincipal = StrReplace( fPrincipal, "$", "" ) ; remove any $
    fPrincipal = StrReplace( fPrincipal, ",", "" ) ; remove any
commas
    fPrincipal = StrTrim( fPrincipal ) ; trim a second time
    If ! IsNumber( fPrincipal ) ; test a second time
        sPrincipal = ""
        Display( 10, "Entry error", "Principal amount entry is
invalid" )
        Continue ; loop_on_error
    EndIf
EndIf

If bExternal ; use this code to call the external subroutine
    sPrincipal = fPrincipal
    Call( "FormatCurrency.wbt", "sPrincipal" )
Else ; use this code to call the internal subroutine
    sTempStr = fPrincipal
    GoSub Format_Dollar_string
    sPrincipal = sTempStr
EndIf

fPercentAPR = StrTrim( fPercentAPR ) ; trim all leading or trailing
blanks

If fPercentAPR == ""
    Display( 10, "Entry error", "A loan rate is required" )
    Continue ; loop_on_error
EndIf

If ! IsNumber( fPercentAPR )
```

```

        fPercentAPR = StrReplace( fPercentAPR, "%%", "" ) ; remove %%
sign
        fPercentAPR = StrTrim( fPercentAPR ); repeat trim
        If StrScan( fPercentAPR, ".", 1, @FWDSCAN ) == 1 ; check leading
decimal
            fPercentAPR = StrCat( "0", fPercentAPR ) ; add a leading zero
        EndIf
        If ! IsNumber( fPercentAPR ) ; check to ensure this is a number
            sPercent = ""
            Display( 10, "Entry error", "Percentage entry is invalid" )
            Continue ; loop_on_error
        EndIf
    EndIf
    If fPercentAPR >= 1.0 ; must be percentage, not decimal
        fPercentAPR = fPercentAPR / 100.0
    EndIf
    sPercent = StrCat( ( fPercentAPR * 100.0 ), "%%" ) ; reformat for
display

    If nTerm == ""
        Display( 10, "Entry error", "A loan term is required" )
        Continue ; loop_on_error
    EndIf

    nTerm = StrTrim( nTerm )
    If ! IsNumber( nTerm )
        nTerm = ""
        Display( 10, "Entry error", "Loan term entry is invalid" )
        Continue ; loop_on_error
    EndIf

    ;=====
    ; all information is provided, now calculate the payment
    ; and interest for the loan
    ;=====

    GoSub DoTheMath

EndWhile

```

Introduction to Programming

```
exit

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;;;;;;;;;;;;;;

:DoTheMath

fRate = fPercentAPR / 12.0          ; convert APR to monthly interest
rate

;== the formula is: =====
;
;               fPrincipal * fRate
;
;   -----
;   Pmt =              1
;
;   1 - -----
;
;   ( 1 + fRate ) ** nTerm
;=====

fFactor = 1 - ( 1 / ( ( 1 + fRate ) ** nTerm ) )
fPayment = ( fPrincipal * fRate ) / fFactor
fInterest = ( fPayment * nTerm ) - fPrincipal

;== now format the results before reporting =====
If bExternal ; use this code to call the external subroutine
    sPaymentAmt = fPayment
    Call( "FormatCurrency.wbt", "sPaymentAmt" )
    sInterestTot = fInterest
    Call( "FormatCurrency.wbt", "sInterestTot" )
Else ; use this code to call the internal subroutine
    sTempStr = fPayment
    GoSub Format_Dollar_String
    sPaymentAmt = sTempStr

    sTempStr = fInterest
    GoSub Format_Dollar_string
    sInterestTot = sTempStr
EndIf
return
```

```

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;

```

```

:Format_Dollar_String

```

```

    sTarget = ""

```

```

    nDecimal = StrIndex( sTempStr, ".", 1, @FWDSCAN )

```

```

    If( nDecimal )

```

```

        nLen = nDecimal

```

```

    Else

```

```

        nLen = StrLen( sTempStr ) + 1

```

```

        sTempStr = StrCat( sTempStr, ".00" ) ; add decimal places

```

```

    EndIf

```

```

    If( nLen > 4 ) ; have at least four digits

```

```

        sSubStr = StrSub( sTempStr, nLen - 3, -1 )

```

```

        sTempStr = StrSub( sTempStr, 1, nLen - 4 )

```

```

        sTempStr = StrCat( sTempStr, ",", sSubStr )

```

```

    EndIf

```

```

    nLen = StrIndex( sTempStr, ",", 1, @FWDSCAN ) ; now see if more
commas are needed

```

```

    While( nLen > 4 )

```

```

        sSubStr = StrSub( sTempStr, nLen - 3, -1 )

```

```

        sTempStr = StrSub( sTempStr, 1, nLen - 4 )

```

```

        sTempStr = StrCat( sTempStr, ",", sSubStr )

```

```

        nLen = StrIndex( sTempStr, ",", 1, @FWDSCAN )

```

```

    EndWhile

```

```

    sTempStr = StrCat( "$", sTempStr ) ; add the leading dollar sign

```

```

Return

```

FormatCurrency.wbt

```

;*****

```

```

; **

```

```

; ** [Chapter 9]

```

```

; ** FormatCurrency.wbt

```

```

; ** Formats a string as currency

```

Introduction to Programming

```
;** Parameters:
;**          param1: string to be formatted
;**
;*****

If param0 < 1 ; insufficient arguments
    Message("Format Currency","This is a subroutine meant to be called
from other WBT files and not used directly.")
    exit
Endif

; use variable substitution to store off value of passed variable
sTempStr = %param1%
nDecimal = StrIndex( sTempStr, ".", 1, @FWDSCAN )
If nDecimal
    nLen = nDecimal
Else
    nLen = StrLen( sTempStr ) + 1
    sTempStr = StrCat( sTempStr, ".00" ) ; add decimal places
EndIf

If nLen > 4 ; have at least four digits
    sSubStr = StrSub( sTempStr, nLen - 3, -1 )
    sTempStr = StrSub( sTempStr, 1, nLen - 4 )
    sTempStr = StrCat( sTempStr, ",", sSubStr )
EndIf

nLen = StrIndex( sTempStr, ",", 1, @FWDSCAN ) ; now see if more commas
are needed
While nLen > 4
    sSubStr = StrSub( sTempStr, nLen - 3, -1 )
    sTempStr = StrSub( sTempStr, 1, nLen - 4 )
    sTempStr = StrCat( sTempStr, ",", sSubStr )
    nLen = StrIndex( sTempStr, ",", 1, @FWDSCAN )
EndWhile

%param1% = StrCat( "$", sTempStr ) ; add the leading dollar sign
Drop( sTempStr, sSubStr, nLen, nDecimal ) ; discard the local variables
```

```
return
```

Chapter 10 Samples

CallFileList.wbt

```
;*****
; **
; **  [Chapter 10]
; **  CallFileList.wbt
; **  Displays a file list for selection
; **  Parameters:
; **      param1: initial filespec
; **              defaults to *.* , returns selected file name
; **      param2: initial directory  (optional)
; **              defaults to current dir, returns selected dir.
;*****
```

```
FileSelectFormat=`WWDLGED,6.2`
```

```
FileSelectCaption=`File Selection`
```

```
FileSelectX=080
```

```
FileSelectY=040
```

```
FileSelectWidth=134
```

```
FileSelectHeight=172
```

```
FileSelectNumControls=007
```

```
FileSelectProcedure=`DEFAULT`
```

```
FileSelectFont=`DEFAULT`
```

```
FileSelectTextColor=`DEFAULT`
```

```
FileSelectBackground=`DEFAULT,DEFAULT`
```

```
FileSelectConfig=0
```

```
FileSelect001=`005,003,038,012,STATICTEXT,"StaticText_Directory:",DEFAU
LT,"Directory:",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
FileSelect002=`033,003,096,012,VARYTEXT,"VaryText_1",selectFile,DEFAULT
,DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
FileSelect003=`005,015,038,012,STATICTEXT,"StaticText_2",DEFAULT,"File
Spec",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
FileSelect004=`033,015,096,012,EDITBOX,"EditBox_1",selectFile,DEFAULT,D
EFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

Introduction to Programming

```
FileSelect005=`005,031,122,124,FILELISTBOX,"FileListBox_1",selectFile,DEFAULT,DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
FileSelect006=`075,153,050,012,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"&Cancel",0,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
FileSelect007=`007,153,050,012,PUSHBUTTON,"PushButton_Select",DEFAULT,"&Select",1,70,32,DEFAULT,DEFAULT,DEFAULT`
```

```
selectFile = "*.*"      ; set default mask for filelistbox
selectDir = ""          ; set default directory
```

```
Switch param0
```

```
    case 2
```

```
        selectDir = %param2%      ; two parameters supplied
                                    ; second must be initial directory
```

```
    case 1
```

```
        selectFile = %param1%     ; first parameter is file spec
        break
```

```
    case 0
```

```
        ; must have one parameter
```

```
        exit
```

```
EndSwitch
```

```
If selectDir != ""
```

```
    If DirExist (selectDir) == @FALSE
```

```
        Message( "Drive or Directory Error", selectDir : " was not a
valid drive/directory specification" )
```

```
        return
```

```
    Endif
```

```
    DirChange( selectDir )
```

```
EndIf
```

```
; Display the dialog, then wait for one of the pushbuttons to be
activated.
```

```
While @TRUE
```

```
    If Dialog( "FileSelect" ) == @FALSE then return
```

```
    ; if a valid file was not selected, redisplay the dialog
```

```
    If FileExist( selectFile ) then break
```

```
Endwhile
```

```
If param0 > 1 then %param2% = DirGet()      ; optional parameter
```

```
%param1% = selectFile                      ; always return this value
```



```
return
```

DirTest.wbt

```
*****
; **
; ** [Chapter 10]
; ** DirTest.wbt
; ** Calls CallFileList.wbt for a drive/directory/file selection
; **
*****
```

```
DirTestFormat=`WWWDLGED,6.2`
```

```
DirTestCaption=`Directory Test`
```

```
DirTestX=035
```

```
DirTestY=053
```

```
DirTestWidth=106
```

```
DirTestHeight=048
```

```
DirTestNumControls=005
```

```
DirTestProcedure=`DEFAULT`
```

```
DirTestFont=`DEFAULT`
```

```
DirTestTextColor=`DEFAULT`
```

```
DirTestBackground=`DEFAULT,DEFAULT`
```

```
DirTestConfig=0
```

```
DirTest001=`063,001,038,012,EDITBOX,"EditBox_1",sDirDrive,DEFAULT,DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
DirTest002=`063,017,038,012,EDITBOX,"EditBox_2",sFileSpec,"*.*",DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
DirTest003=`025,033,050,012,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,30,32,DEFAULT,DEFAULT,DEFAULT`
```

```
DirTest004=`003,017,060,012,STATICTEXT,"StaticText_1",DEFAULT,"Enter a file specification",DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
DirTest005=`003,003,060,012,STATICTEXT,"StaticText_2",DEFAULT,"Enter a drive or directory",DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
DirChange( DirScript() ) ; be sure we're in the right directory
```

```
Dialog( "DirTest" )
```

Introduction to Programming

```
Call( "CallFileList.wbt", "sFileSpec sDirDrive" )
Message( "The selected file is:", sDirDrive:sFileSpec )
exit
```

DirTest2.wbt

```
;*****
;**
;**  [Chapter 10]
;**  DirTest2.wbt
;**  Uses AskFileName function for a drive/directory/file selection
;**
;*****

sDirDrive = ""           ; default is always the current directory
sFileTypes = "All Files|*.*|WIL files|*.wbt;*.mnu|Text files|*.txt|"
sFileSpec = AskFileName( "Select a file", sDirDrive, sFileTypes, "",1 )
Message( "The selected file is:", sFileSpec )
exit
```

Free Disk Space.wbt

```
;*****
;**
;**  [Chapter 10]
;**  Free Disk Space.wbt
;**  Reports total free space on the system
;**
;*****

DirChange( DirScript() )

sDrives = DiskScan( 6 )      ; 6 = 4 + 2 = Network and Local Drives
nMax = StrLen( sDrives )
BoxOpen( "Network Free Space", "Looking for space on hard drives" )
TimeDelay( 1 )
nDrive = 1
TotalSize = 0
TotalFree = 0
```

```

DriveReport = "Drive":@TAB:"
Free":@TAB:"%% Free":@CRLF:@CRLF

Total":@TAB:"

While @TRUE
    NextDrive = StrSub( sDrives, nDrive, 1 )
    nSize = DiskSize( NextDrive ) / 1024      ; convert to kilobytes
    nFree = DiskFree( NextDrive ) / 1024

    If nSize > 10240
        sUnit = " Mb"
        nSize = nSize / 1024.0                ; convert both to
Megabytes
        nFree = nFree / 1024.0
    Else
        sUnit = " Kb"
    EndIf

    TotalSize = TotalSize + nSize
    sSize = Int( nSize )
    Call( "FormatNumber.wbt", "sSize" )
    sSize = StrFixCharsL( sSize, " ", 16 )

    TotalFree = TotalFree + nFree
    sFree = Int( nFree )
    Call( "FormatNumber.wbt", "sFree" )
    sFree = StrFixCharsL( sFree, " ", 16 )

    If nSize > 0
        sPercent = Int ( ( nFree / 1.0 ) / ( nSize * 1.0 ) * 100 )
        sPercent = StrCat( sPercent, "%%" )
        sPercent = StrFixCharsL( sPercent, " ", 8 )
    Else
        sPercent = ""
    EndIf

    BoxText( "Checking ":NextDrive:":" )
    DriveReport = StrCat( DriveReport, NextDrive, ":" )
    DriveReport = StrCat( DriveReport, @TAB, sSize, sUnit )

```

Introduction to Programming

```
DriveReport = StrCat( DriveReport, @TAB, sFree, sUnit )
DriveReport = StrCat( DriveReport, @TAB, sPercent, @CRLF )
nDrive = nDrive + 3 ; each entry is 3 bytes long
If nDrive > nMax then break
EndWhile

DriveReport = StrCat( DriveReport, @CRLF, sUnit, "ytes" )

sTotalSize = Int( TotalSize )
Call( "FormatNumber.wbt", "sTotalSize" )
sTotalSize = StrFixLeft( sTotalSize, " ", 16 )
n = StrLen( sTotalSize )
DriveReport = StrCat( DriveReport, @TAB, sTotalSize )

sTotalFree = Int( TotalFree )
Call( "FormatNumber.wbt", "sTotalFree" )
sTotalFree = StrFixCharsL( sTotalFree, " ", 16 )
DriveReport = StrCat( DriveReport, @TAB, sTotalFree )

sPercent = Int( ( TotalFree / 1.0 ) / ( TotalSize * 1.0 ) * 100 )
sPercent = StrFixCharsL( sPercent, " ", 8 )
DriveReport = StrCat( DriveReport, @TAB, sPercent, "%%" )

BoxShut()
TotalFree = Int( TotalFree )
Call( "FormatNumber.wbt", "TotalFree" )
Message( "Total Space Available = ":TotalFree:" ":sUnit, DriveReport )
Drop( TotalSize, TotalFree, DriveReport, Drives, NextDrive )
exit
```

FormatNumber.wbt

```
*****
; **
; ** [Chapter 10]
; ** FormatNumber.wbt
; ** Formats a number string
; ** Parameters:
; **     param1: string to be formatted
```

```

;*****

If param0 < 1 ; insufficient arguments
    Message("Format Argument","This is a subroutine meant to be called
from other WBT files and not used directly.")
    exit
Endif

sTempStr = %param1%
nDecimal = StrIndex( sTempStr, ".", 1, @FWDSCAN )
If nDecimal
    nLen = nDecimal ; move back one space
Else
    nLen = StrLen( sTempStr ) + 1 ; find the string length
EndIf

While nLen > 4
    sSubStr = StrSub( sTempStr, nLen - 3, -1 )
    sTempStr = StrSub( sTempStr, 1, nLen - 4 )
    sTempStr = StrCat( sTempStr, ",", sSubStr )
    nLen = StrIndex( sTempStr, ",", 1, @FWDSCAN )
EndWhile

%param1% = sTempStr
Drop( sTempStr, sSubStr, nLen, nDecimal ) ; discard the local variables
return

```

Phone.lst

Albert Einstein	34 McFadden Apt 7	Princeton	MA	01234	(222)	357-9246
Han Solo	87 Cloud St	#92 Hollywood	CA	89034	(111)	345-6789
Joe Friday	1 Police Plaza	New York	NY	01010	(333)	135-7924
John Jacob	Jingleheimer Schmidt	999 9th St		Stumptown	CA	
	95446	(707)	999-8765			
Rob Roy	123 4th Ave	Gurrock	MN	78945	(555)	567-8901
S. F. Katt	1 Park Ave	New York	NY	10016	(781)	393-3700
Sol Rosencranz	#2 Old Stone Bridge	Daubmore	MS	68758	(999)	555-2345

Introduction to Programming

DilbertCubicle 23 Silicon Valley CA 95444 (890) 555-5893

ShowList.wbt

```
*****
;**
;** [Chapter 10]
;** ShowList.wbt
;** Demonstrates AskFileText function
;**
*****

sTemp = AskFileText( "Select any name", "Phone.lst", @SORTED, @SINGLE )
if sTemp == "" then exit
sName      = ItemExtract( 1, sTemp, @TAB )
sAddress1  = ItemExtract( 2, sTemp, @TAB )
sAddress2  = ItemExtract( 3, sTemp, @TAB )
sCity      = ItemExtract( 4, sTemp, @TAB )
sState     = ItemExtract( 5, sTemp, @TAB )
sZip       = ItemExtract( 6, sTemp, @TAB )
sPhone     = ItemExtract( 7, sTemp, @TAB )
Message( "Thank you", sName:@CRLF:"can be reached at:":@CRLF:sPhone )
exit
```

PhoneList.wbt

```
*****
;**
;** [Chapter 10]
;** PhoneList.wbt
;** Demonstrates file operations including:
;** FileOpen, FileRead, FileWrite, FileAppend and FileClose
;**
*****

DirChange( DirScript() )

PhoneListFormat=`WWWDLGED,6.2`
```

```

PhoneListCaption=`Phone List`
PhoneListX=044
PhoneListY=064
PhoneListWidth=246
PhoneListHeight=114
PhoneListNumControls=017
PhoneListProcedure=`DEFAULT`
PhoneListFont=`DEFAULT`
PhoneListTextColor=`DEFAULT`
PhoneListBackground=`DEFAULT,DEFAULT`
PhoneListConfig=0

PhoneList001=`001,003,072,092,ITEMBOX,"ItemBox_1",lbNames,DEFAULT,DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList002=`075,005,154,012,EDITBOX,"EditBox_1",sName,DEFAULT,DEFAULT,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList003=`079,019,038,012,STATICTEXT,"StaticText_address",DEFAULT,"address",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList004=`119,019,110,012,EDITBOX,"EditBox_2",sAddress1,DEFAULT,DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList005=`079,033,038,012,STATICTEXT,"StaticText_2",DEFAULT,"address",DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList006=`119,033,110,012,EDITBOX,"EditBox_3",sAddress2,DEFAULT,DEFAULT,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList007=`079,049,038,012,STATICTEXT,"StaticText_3",DEFAULT,"city / state",DEFAULT,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList008=`119,049,066,012,EDITBOX,"EditBox_4",sCity,DEFAULT,DEFAULT,80,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList009=`191,049,038,012,EDITBOX,"EditBox_5",sState,DEFAULT,DEFAULT,90,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList010=`079,063,038,012,STATICTEXT,"StaticText_4",DEFAULT,"zip code",DEFAULT,100,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList011=`119,063,066,012,EDITBOX,"EditBox_6",sZip,DEFAULT,DEFAULT,110,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList012=`079,077,038,012,STATICTEXT,"StaticText_phone",DEFAULT,"phone",DEFAULT,120,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList013=`119,077,076,012,EDITBOX,"EditBox_7",sPhone,DEFAULT,DEFAULT,130,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList014=`157,097,038,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit",0,140,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
PhoneList015=`003,097,038,012,PUSHBUTTON,"PushButton_Select",DEFAULT,"Select",1,150,32,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

```

Introduction to Programming

```
PhoneList016=`107,097,038,012,PUSHBUTTON,"PushButton_3",DEFAULT,"Save  
New",2,160,DEFAULT,DEFAULT,DEFAULT,DEFAULT`  
PhoneList017=`055,097,038,012,PUSHBUTTON,"PushButton_Clear",DEFAULT,"Cl  
ear",3,170,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
sDataFile = "Phone.lst"
```

```
hFile = FileOpen( sDataFile, "READ" )
```

```
listPhone = ""
```

```
nCount = 0
```

```
While @TRUE
```

```
    sLineIn = FileRead( hFile )
```

```
    If( sLineIn == "*EOF*" ) then break
```

```
    listPhone = ItemInsert( sLineIn, -1, listPhone, @CR )
```

```
    nCount = nCount + 1
```

```
EndWhile
```

```
FileClose( hFile ) ; close the input file
```

```
If nCount > 0
```

```
    gosub Prepare_List
```

```
    nSelect = 1
```

```
    gosub Extract_Entry
```

```
Endif
```

```
; Display List
```

```
While @TRUE
```

```
    nRequest = Dialog( "PhoneList" )
```

```
    Switch nRequest
```

```
        case 1 ; Select -- update the display with a new item
```

```
        sSelect = lbNames ; list now has only one entry, the  
selection
```

```
        For i = 1 to nCount
```

```
            sTemp = ItemExtract( i, listPhone, @CR )
```

```
            If ItemLocate( sSelect, sTemp, @TAB ) == 1 then
```

```
                nSelect = i ; this is the item
```

```
                gosub Extract_Entry ; update the display
```

```
                break
```

```
            EndIf
```



```

        Next
    gosub Prepare_List ; rebuild the name list
    break

    case 2 ; Save New -- update the list and file
        If sName != ""
            nSelect = 0
            sNewEntry = StrCat( sName, @TAB, sAddress1, @TAB,
sAddress2, @TAB, sCity, @TAB, sState, @TAB, sZip, @TAB, sPhone )
            For i = 1 to ItemCount( listPhone, @CR ) ; see if there is
an existing match
                sTemp = ItemExtract( i, listPhone, @CR )
                If ItemLocate( sName, sTemp, @TAB ) == 1 then
                    nSelect = I ; match found, this is the item
                    gosub Delete_Entry ; remove the existing entry
                    break
                EndIf
            Next
            gosub Add_Entry ; add the new entry to the list
        EndIf
        gosub Prepare_List ; rebuild the name list
    break

    case 3 ; Clear -- clear the current entries but not the list
        gosub Clear_Entry
        gosub Prepare_List
    break

EndSwitch

EndWhile

exit

:Extract_Entry
    sTemp      = ItemExtract( nSelect, listPhone, @CR )
    sName      = ItemExtract( 1, sTemp, @TAB )
    sAddress1  = ItemExtract( 2, sTemp, @TAB )
    sAddress2  = ItemExtract( 3, sTemp, @TAB )

```

Introduction to Programming

```
sCity      = ItemExtract( 4, sTemp, @TAB )
sState     = ItemExtract( 5, sTemp, @TAB )
sZip       = ItemExtract( 6, sTemp, @TAB )
sPhone     = ItemExtract( 7, sTemp, @TAB )
return

:Clear_Entry
sName      = ""
sAddress1  = ""
sAddress2  = ""
sCity      = ""
sState     = ""
sZip       = ""
sPhone     = ""
return

:Prepare_List
listPhone = ItemSort( listPhone, @CR )
lbNames = ""
For i = 1 to ItemCount( listPhone, @CR )
    sTemp = ItemExtract( i, listPhone, @CR )
    sItem = ItemExtract( 1, sTemp, @TAB )
    lbNames = ItemInsert( sItem, -1, lbNames, @TAB )
Next
return

:Add_Entry
hFile = FileOpen( sDataFile, "APPEND" )
FileWrite( hFile, sNewEntry )
FileClose( hFile ) ; close the input file
listPhone = ItemInsert( sNewEntry, -1, listPhone, @CR )
listPhone = ItemSort( listPhone, @CR )
return

:Delete_Entry
listPhone = ItemRemove( nSelect, listPhone, @CR )
hFile = FileOpen( sDataFile, "WRITE" )
```

```

For i = 1 to ItemCount( listPhone, @CR )
    sTemp = ItemExtract( i, listPhone, @CR )
    FileWrite( hFile, sTemp )
Next
FileClose( hFile ) ; close the input file
return

```

Phone.lst

```

Albert Einstein»34 McFadden Apt 7»Princeton»MA»01234»(222) 357-9246
Han Solo»87 Cloud St #92»Hollywood»CA»89034»(111) 345-6789
Joe Friday»1 Police Plaza»New York»NY»01010»(333) 135-7924
John Jacob Jingleheimer Schmidt»999 9th St»Stumptown»CA»95446»(707)
999-8765
Rob Roy»123 4th Ave»Gurrock»MN»78945»(555) 567-8901
S. F. Katt»1 Park Ave»New York»NY»10016» (781) 393-3700
Sol Rosencranz»#2 Old Stone Bridge»Daubmore»MS»68758»(999) 555-2345
Dilbert»Cubicle 23»Silicon Valley»CA»95444»(890) 555-5893

```

FileAttr.wbt

```

;*****
;**
;** [Chapter 10]
;** FileAttr.wbt
;** Displays the file name, size, date/time stamp, and attributes.
;** Allows you to "touch" the file and set its attributes.
;**
;*****

```

```
FileInfoFormat=`WWWDLGED,6.2`
```

```
FileInfoCaption=`File Information`
```

```
FileInfoX=044
```

```
FileInfoY=064
```

```
FileInfoWidth=196
```

```
FileInfoHeight=098
```

Introduction to Programming

```
FileInfoNumControls=012
FileInfoProcedure=`DEFAULT`
FileInfoFont=`DEFAULT`
FileInfoTextColor=`DEFAULT`
FileInfoBackground=`DEFAULT,DEFAULT`
FileInfoConfig=0

FileInfo001=`021,005,038,012,STATICTEXT,"StaticText_2",DEFAULT,"File
Name:",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo002=`063,005,124,012,VARYTEXT,"VaryText_2",sFileName,DEFAULT,DE
FAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo003=`021,017,038,012,STATICTEXT,"StaticText_3",DEFAULT,"File
Size:",DEFAULT,50,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo004=`063,017,124,012,VARYTEXT,"VaryText_3",sFileSize,DEFAULT,DE
FAULT,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo005=`003,029,056,012,STATICTEXT,"StaticText_4",DEFAULT,"Date/Ti
me Stamp:",DEFAULT,70,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo006=`063,029,124,012,VARYTEXT,"VaryText_4",sTimeStamp,DEFAULT,D
EFAULT,80,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo007=`019,043,038,012,STATICTEXT,"StaticText_Attributes:",DEFAUL
T,"Attributes:",DEFAULT,90,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo008=`063,043,068,012,EDITBOX,"EditBox_1",sFileAttr,DEFAULT,DEFA
ULT,100,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo009=`031,079,038,012,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,
110,32,DEFAULT,DEFAULT,DEFAULT`
FileInfo010=`101,079,038,012,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Ca
ncel",0,120,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo011=`021,059,050,012,CHECKBOX,"CheckBox_1",cbTouchDate,"Touch
Date",1,130,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
FileInfo012=`087,059,078,012,CHECKBOX,"CheckBox_2",cbFileAttr,"Change
file attributes",1,140,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

sDirDrive = "" ; default is always the current directory
sFileTypes = "All Files|*.*|Executable files|.exe;*.com|Text
files|.txt|WIL files|.wbt;*.mnu|"
curDir = DirGet() ; store the current directory in the variable curDir

If param0 > 0
    sFileName = Param1
Else
    sFileName = AskFileName( "Select a file",sDirDrive,sFileTypes,"",1 )
Endif
```

```

sFileSize = FileSize( sFileName ) ; obtain the file size
sTimeStamp = FileTimeGet( sFileName ) ; obtain the date/time stamps
sFileAttr = FileAttrGet( sFileName ) ; obtain the file attributes
Dialog( "FileInfo" )

if cbTouchDate == 1 then FileTimeTouch( sFileName )
if cbFileAttr == 1 then FileAttrSet( sFileName, sFileAttr )
exit

```

Binary.wbt

```

;*****
;**
;**  [Chapter 10]
;**  Binary.wbt
;**  Edits the Config.sys file
;**  by adding a new line to the bottom of the file.
;*****

fs = FileSize("C:\CONFIG.SYS")
; Allocate a buffer the size of your file + 100 bytes.
binbuf = BinaryAlloc(fs+100)
If binbuf == 0
    Message("Error", "BinaryAlloc Failed")
Else
    ; Read the file into the buffer.
    BinaryRead(binbuf, "C:\CONFIG.SYS")
    ; Append a line to the end of the file in buffer.
    BinaryPokeStr(binbuf, fs, "DEVICE=C:\FLOOGLE.SYS%~crlf%")
    ; Write modified file back to the file from the buffer.
    BinaryWrite(binbuf, "C:\CONFIG.SYS")
    binbuf = BinaryFree(binbuf)
EndIf
Message("BinaryAlloc", "Done.")
exit

```

Chapter 11 Samples

Hello Windows.wbt

```

;*****
;
; **
; ** [Chapter 11]
; ** Hello Windows.wbt
; ** Uses Box functions to create a window display
; **
;*****

;=====
; definitions for assorted colors
;=====

;===== gray scale =====
BLACK      = " 0,  0,  0"
DKGRAY     = " 64, 64, 64"
GRAY       = "128, 128, 128"
LTGRAY     = "192, 192, 192"
WHITE      = "255, 255, 255"
;===== dark colors =====
DKBLUE     = " 0,  0, 128"
DKGREEN    = " 0, 160,  0"
DKRED      = "128,  0,  0"
DKCYAN     = " 0, 128, 128"
DKMAGENTA  = "128,  0, 128"
BROWN      = "128, 128,  0"
;===== light colors =====
BLUE       = " 0,  0, 255"
GREEN      = " 0, 255,  0"
RED        = "255,  0,  0"
CYAN       = " 0, 255, 255"
MAGENTA    = "255,  0, 255"
YELLOW     = "255, 255,  0"

;=====
; Window identifiers

```

```

;=====
mainID = 1      ; requires IDs less than 9
drawID = 2
noteID = 3

;=====
; Button identifiers
;=====

bExit  = 1

;=====
; Generic Initialization
;   allows windows to exit without warning (1)
;               + quiet termination (4)
;=====
IntControl( 12, 5, 0, 0, 0 )

;=====
; Creates the top-level Window
;=====
BoxesUp( "100, 100, 900, 900", @NORMAL )
TimeDelay(1)

;=====
; This section creates the main window
;=====
BoxColor( mainID, BLACK, 7 ) ; third param sets shaded background
BoxDrawRect( mainID, "0, 0, 1000, 1000", 2 ) ; size in logical units
BoxCaption( mainID, "Hello Windows.WBT Demo" ) ; window caption

;=====
; This section creates the fancy banner headline in a box
;=====
rectNote = "100, 100, 900, 340" ; set the size of the banner box
BoxNew( noteID, rectNote, 1 ) ; create the box
BoxColor( noteID, LTGRAY, 0 ) ; background is Light Gray, no gradient
BoxDrawRect( noteID, "", 2 ) ; fill banner box with background color

```

Introduction to Programming

```
;=====
;  Next create a 3-D outline around the box
;=====

penWidthA = 20                                ; note that all units are
line1A = "  0,    0, 1000,    0"              ; logical units relative to
line2A = "1000, 1000, 1000,    0"              ; the notebox which (by default)
line3A = "  0, 1000, 1000, 1000"              ; has a logical size of
line4A = "  0,    0,    0, 1000"              ; 1000 x 1000 units

;=====
; draw the outer outline
;=====

BoxPen( noteID, WHITE, penWidthA )            ; line color top and left
BoxDrawLine( noteID, line1A )                  ; top
BoxDrawLine( noteID, line4A )                  ; left
BoxPen( noteID, GRAY,  penWidthA )            ; line color bottom and right
BoxDrawLine( noteID, line2A )                  ; right
BoxDrawLine( noteID, line3A )                  ; bottom

;=====
; draw the inner outline
;=====

penWidthB = 10
line1B = " 40,  150,  960,  150"              ; top
line2B = " 960,  840,  960,  150"              ; right
line3B = " 40,  840,  960,  840"              ; bottom
line4B = " 40,  150,   40,  840"              ; left

BoxPen( noteID, WHITE, penWidthB )
BoxDrawLine( noteID, line2B )                  ; right
BoxDrawLine( noteID, line3B )                  ; bottom
BoxPen( noteID, GRAY,  penWidthB )
BoxDrawLine( noteID, line1B )                  ; top
BoxDrawLine( noteID, line4B )                  ; left

;=====
```



```

; Now put text in the banner headline box
;=====
noteHeight    = 400
BoxTextFont( noteID, "Arial", noteHeight, 170, 0 ) ; set headline font
rectNoteText = " 70, 200, 950, 800"
BoxTextColor( noteID, RED )
; creates the headline text - this line can be copied
; anywhere in the program where the headline needs to be changed
BoxDrawText( noteID, rectNoteText, "Hello Windows", 1, 4 )

;=====
; And display a message in the window
;=====
BoxTextFont( mainID, "Times", 80, 80, 0 | 0 ) ; initial font
BoxTextColor( mainID, YELLOW ) ; initial font color
BoxDrawText( mainID, "10, 500, 990, 600", "Now, this wasn't too
difficult, was it?", 0, 1 | 4 )

;=====
; This section creates the Exit buttons
;=====
BoxButtonDraw( mainID, bExit, "E&xit", "750, 820, 900, 890" )

;=====
; Wait for a button to be selected (clicked)
;=====
BoxButtonWait()
exit

```

Progress.wbt

```

;*****
;**
;** [Chapter 11]
;** Progress.wbt
;** Uses Box functions to create a progress bar display
;**
;*****

```

Introduction to Programming

```
===== Generic Initialization =====;
; allows windows to exit without warning (1) + quiet termination (4)
IntControl( 12, 5, 0, 0, 0 )

===== Assorted Colors =====
DKBLUE   = "  0,  0, 128"
BLUE     = "  0,  0, 255"
LTGRAY   = "192, 192, 192"
GRAY     = "128, 128, 128"
DKGRAY   = " 64,  64,  64"
DKGREEN  = "  0, 160,  0"
GREEN    = "  0, 255,  0"
RED      = "255,  0,  0"
BLACK    = "  0,  0,  0"
WHITE    = "255, 255, 255"
YELLOW   = "255, 255,  0"

===== Drawing the Main Box =====
BoxesUp( "100, 100, 900, 900", @NORMAL )

; Window identifiers
mainID = 1 ; requires IDs less than 10
drawID = 2
noteID = 3
progID = 8

bExit  = 1 ; buttons must be successive
bBegin = 2

=====
; This section creates the main window
=====
BoxColor( mainID, GREEN, 4 ) ; third param sets shaded background
BoxCaption( mainID, "Progress.wbt Demo" ) ; window caption
BoxDrawRect( mainID, " 0, 0, 1000, 1000", 2 ) ; size is in logical
units
BoxTextFont( mainID, "Times", 80, 80, 0 | 0 ) ; initial font
information
```

```

BoxTextColor( mainID, "255, 255, 0" ) ; initial font color

;=====
; And this puts a message in the window
;=====

BoxDrawText( mainID, "245, 500, 700, 600", "Pick [Begin] to start", 0,
0 )

;=====
; This section creates the fancy banner headline in a box
;=====

rectNote = "100, 100, 900, 340" ; set the size of the banner box
BoxNew( noteID, rectNote, 1 ) ; create the box
BoxColor( noteID, LTGRAY, 0 ) ; background is Light Gray, no gradient
BoxDrawRect( noteID, "", 2 ) ; fill the entire banner box with the
background color

;=====
; Next create a 3-D outline around the box
;=====

notePenWidthA = 20 ; note that all units are
rectNoteLine1A = " 0, 0, 1000, 0" ; logical units relative to
rectNoteLine2A = "1000, 1000, 1000, 0" ; the notebbox which (by
default)
rectNoteLine3A = " 0, 1000, 1000, 1000" ; has a logical size of
rectNoteLine4A = " 0, 0, 0, 1000" ; 1000 x 1000 units

; draw the outer outline
BoxPen( noteID, WHITE, notePenWidthA ) ; line color top and left
BoxDrawLine( noteID, rectNoteLine1A )
BoxDrawLine( noteID, rectNoteLine4A )
BoxPen( noteID, GRAY, notePenWidthA ) ; line color bottom and right
BoxDrawLine( noteID, rectNoteLine2A )
BoxDrawLine( noteID, rectNoteLine3A )

; draw the inner outline
notePenWidthB = 10
rectNoteLine1B = " 40, 150, 960, 150"

```

Introduction to Programming

```
rectNoteline2B = " 960, 840, 960, 150"
rectNoteLine3B = " 40, 840, 960, 840"
rectNoteLine4B = " 40, 150, 40, 840"

BoxPen( noteID, WHITE, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine2B )
BoxDrawLine( noteID, rectNoteLine3B )
BoxPen( noteID, GRAY, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine1B )
BoxDrawLine( noteID, rectNoteLine4B )

;=====
; Now put text in the banner headline box
;=====

noteHeight = 400
BoxTextFont( noteID, "Arial", noteHeight, 170, 0 ) ; set the headline
font
rectNoteText = " 70, 200, 950, 800"
BoxTextColor( noteID, RED )
; this next line creates the headline text -- this line can be copied
; anywhere in the program where the headline text needs to be changed
BoxDrawText( noteID, rectNoteText, "Progress Bar Demo", 1, 4 )

;=====
; This section creates the Begin and Exit buttons
;=====
BoxButtonDraw( mainID, bBegin, "&Begin", "100, 820, 250, 890" )
BoxButtonDraw( mainID, bExit, "E&xit", "750, 820, 900, 890" )

;=====
; Wait for a button to be selected (clicked)
;=====

iBox = 0
BoxButtonWait()
While iBox == 0
    For x = 1 to 2 ; sequential buttons required
        If BoxButtonStat( mainID, x ) Then iBox = x
    Next
```

```

EndWhile

;=====
; poll the buttons to decide which was pressed (clicked)
;=====

if iBox
    BoxDataClear( mainID, "TOP" )
    Switch iBox
        case bExit
            exit
            Break
        case bBegin
            gosub DO_PROGRESS
            Break
    EndSwitch
Endif
exit

;=====
; PROGRESS BAR SUBROUTINE
;=====

:DO_PROGRESS

;=====
; Make the main window look nice
;=====

BoxColor( mainID, RED, 1 ) ; select Red gradient
BoxDrawRect( mainID, "0, 0, 1000, 1000", 2 ) ; redraw the main window
BoxCaption( mainID, "Are we making any progress here?" ) ; change
caption

;=====
; Create a window to contain the progress bar
;=====

BoxNew( noteID, rectNote, 1 ) ; do note box
BoxColor( noteID, LTGRAY, 0 ) ; Light Gray, no gradient
BoxDrawRect( noteID, "", 2 )

BoxTextFont( noteID, "Arial", noteHeight, 170, 0 )

```

Introduction to Programming

```
BoxTextColor( noteID, RED )

rectNoteLine1A = " 0, 0, 1000, 0"
rectNoteline2A = "1000, 1000, 1000, 0"
rectNoteLine3A = " 0, 1000, 1000, 1000"
rectNoteLine4A = " 0, 0, 0, 1000"
notePenWidthA = 20

rectNoteLine1B = " 40, 150, 960, 150"
rectNoteline2B = " 960, 840, 960, 150"
rectNoteLine3B = " 40, 840, 960, 840"
rectNoteLine4B = " 40, 150, 40, 840"
notePenWidthB = 10

BoxPen( noteID, WHITE, notePenWidthA )
BoxDrawLine( noteID, rectNoteLine1A )
BoxDrawLine( noteID, rectNoteLine4A )
BoxPen( noteID, GRAY, notePenWidthA )
BoxDrawLine( noteID, rectNoteLine2A )
BoxDrawLine( noteID, rectNoteLine3A )

BoxPen( noteID, WHITE, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine2B )
BoxDrawLine( noteID, rectNoteLine3B )
BoxPen( noteID, GRAY, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine1B )
BoxDrawLine( noteID, rectNoteLine4B )

BoxDataTag( noteID, "NOTE" )
BoxDrawText( noteID, rectNoteText, "Initializing...", 1, 4 )

TimeDelay( 1 ) ; brief pause (1 second)

;=====
; Now create a window for the progress bar proper
;=====
rectProg = " 200, 550, 800, 750" ; size/position progress bar
; (relative to main window)
```

```

BoxNew( progID, rectProg, 2 )
BoxUpdates( progID, 0 )
BoxCaption( progID, "Name Goes Here" )

TimeDelay( 1 ) ; brief pause (1 second)

BoxColor( progID, LTGRAY, 0 )
BoxDrawRect( progID, "", 2 )
; Draw updating progress bar here
; there are 3 virtual pixels per percent
; we have to draw 3 boxes and some text...
BoxDataTag( progID, "NULL" )
BoxCaption( progID, "Please wait. Stepping through example..." )

;=====
; Settings for progress bar
;=====

barLf  = 104 ; settings for progress bar
barRt  = 896
barTop = 454
barBtm = 796

textTop = barTop + 40 ; settings for percentage display
textBtm = barBtm - 40

outlnLf  = barLf - 4 ; settings for progress bar outline
outlnRt  = barRt + 4
outlnTop = barTop - 4
outlnBtm = barBtm + 4

; outline for progress bar
rectProgLine1 = " %outlnLf%, %outlnTop%, %outlnRt%, %outlnTop%"
rectProgLine2 = " %outlnRt%, %outlnTop%, %outlnRt%, %outlnBtm%"
rectProgLine3 = " %outlnRt%, %outlnBtm%, %outlnLf%, %outlnBtm%"
rectProgLine4 = " %outlnLf%, %outlnBtm%, %outlnLf%, %outlnTop%"
progPenWidth = 20

filesToCopy = 16 ; simulate file copy operation for progress bar

```

Introduction to Programming

```
filesCopied = 0
rectProgText = " 100,    50, 1000,  250" ; progress text message

;=====
;  Now simulate operation and paint the progress bar
;=====

For filesCopied = 1 to filesToCopy

    BoxDataClear( progID, "NULL" )
    BoxUpdates( progID, 0 )

    ;===== setup the progress position =====
    xPos = 100 + ( ( 800 * filesCopied ) / filesToCopy )
    per = ( 100.0 * filesCopied ) / filesToCopy
    per = Int( per )

    ;===== draw completed portion of bar =====
    BoxColor( progID, GREEN, 0 )
    BoxDrawRect( progID, "%barLf%, %barTop%, %xPos%, %barBtm%", 2 )

    ;===== draw incomplete portion of bar =====
    BoxColor( progID, GRAY, 0 )
    BoxDrawRect( progID, "%xPos%, %barTop%, %barRt%, %barBtm%", 2 )

    ;===== draw outline around progress bar =====
    BoxPen( progID, BLACK, progPenWidth )
    BoxDrawLine( progID, rectProgLine1 )
    BoxDrawLine( progID, rectProgLine2 )
    BoxDrawLine( progID, rectProgLine3 )
    BoxDrawLine( progID, rectProgLine4 )

    ;===== center the percentage on the green bar =====
    textLf = Max( 104, ( ( xPos / 2 ) - 15 ) )
    textRt = textLf + 30
    rectProgPercent = "%textLf%, %textTop%, %textRt%, %textBtm%"
    BoxDrawText( progID, rectProgPercent, "%per%%", 0, 0 )

    ;===== now report progress as text
```



```

BoxTextColor( progID, BLACK )
BoxColor( progID, LTGRAY, 0 )
BoxDrawText( progID, rectProgText, "Steps completed:
%filesCopied%", 1, 0 )
BoxDrawText( noteID, rectNoteText, "Showing Progress", 1, 4 )
BoxUpdates( progID, 2 )

TimeDelay( Random( 1.0 ) ) ; Fake passage of time
Next

;==== now report completion ====
BoxTextColor( noteID, DKGREEN )
BoxDrawText( noteID, rectNoteText, "Example Complete", 1, 4 )
Message( "Company Name Goes Here", "Example Complete!" )

;==== and clean up ====
BoxDestroy( progID )
BoxDestroy( noteID )
Return

```

Text Fonts.wbt

```

;*****
;**
;** [Chapter 11]
;** Text Fonts.wbt
;** Uses Box functions for text and font display
;**
;*****

;===== gray scale =====
;          -R-  -G-  -B-
BLACK      = "  0,   0,   0"   ; Black
DKGRAY     = " 64,  64,  64"   ; Dark Gray
GRAY       = "128, 128, 128"   ; Gray
LTGRAY     = "192, 192, 192"   ; Light Gray
OFFWHITE   = "236, 236, 236"   ; Off-White
WHITE      = "255, 255, 255"   ; White

```

Introduction to Programming

```
;===== dark colors =====
;           -R-  -G-  -B-
DKBLUE      = "  0,   0, 128" ; Dark Blue
DKGREEN     = "  0, 160,   0" ; Dark Green
DKRED       = "128,   0,   0" ; Dark Red
DKCYAN      = "  0, 128, 128" ; Dark Cyan
DKMAGENTA   = "128,   0, 128" ; Dark Magenta
BROWN       = "128,  96,  48" ; Brown

;===== light colors =====
;           -R-  -G-  -B-
BLUE        = "  0,   0, 255" ; Blue
GREEN       = "  0, 255,   0" ; Green
RED         = "255,   0,   0" ; Red
CYAN        = "  0, 255, 255" ; Cyan
MAGENTA     = "255,   0, 255" ; Magenta
YELLOW      = "255, 255,   0" ; Yellow

;===== Generic Initialization =====;
; allows windows to exit without warning (1) quiet termination (4)
IntControl( 12, 5, 0, 0, 0 )

;===== Drawing the Main Box =====
BoxesUp( "100, 100, 900, 900", @NORMAL )

; Window identifiers
mainID = 1 ; requires successive IDs
drawID = 2
noteID = 3
progID = 8

bExit    = 1
bBegin   = 2
bColorUp = 2
bColorDn = 3

nColorStep = 3
```

```

dRed    = nColorStep
dBlue   = nColorStep
dGreen  = nColorStep

; Note the use of while @TRUE. This use of while maintains
; the boxes until a user clicks on a button and exits the
; while construction
While @TRUE

    ;=====
    ; This section creates the main window
    ;=====
    BoxColor( mainID, GREEN, 4 ) ; third param sets shaded background
    BoxCaption( mainID, "Text Fonts.wbt Demo" ) ; window caption
    BoxDrawRect( mainID, " 0, 0, 1000, 1000", 2 ) ; size logical units
    BoxTextFont( mainID, "Times", 80, 80, 0 | 0 ) ; init font info
    BoxTextColor( mainID, "255, 255, 0" ) ; initial font color

    ;=====
    ; And this puts a message in thw window
    ;=====
    BoxDrawText( mainID, "245, 500, 700, 600", "Pick [Begin] to start",
0, 0 )

    ;=====
    ; This section creates the fancy banner headline in a box
    ;=====
    rectNote = "100, 100, 900, 340" ; set the size of the banner box
    BoxNew( noteID, rectNote, 1 ) ; create the box
    BoxColor( noteID, LTGRAY, 0 ) ; background Light Gray, no gradient
    BoxDrawRect( noteID, "", 2 ) ; fill banner box with background color

    ;=====
    ; Next create a 3-D outline around the box
    ;=====
    ; note that all units are logical units relative to
    ; the notebox which (by default) has a logical size of
    ; 1000 x 1000 units

```

Introduction to Programming

```
notePenWidthA = 20
rectNoteLine1A = "  0,    0, 1000,    0"
rectNoteLine2A = "1000, 1000, 1000,    0"
rectNoteLine3A = "    0, 1000, 1000, 1000"
rectNoteLine4A = "    0,    0,    0, 1000"

; draw the outer outline
BoxPen( noteID, WHITE, notePenWidthA ) ; line color top and left
BoxDrawLine( noteID, rectNoteLine1A )
BoxDrawLine( noteID, rectNoteLine4A )
BoxPen( noteID, GRAY, notePenWidthA ) ; line color bottom and right
BoxDrawLine( noteID, rectNoteLine2A )
BoxDrawLine( noteID, rectNoteLine3A )

; draw the inner outline
notePenWidthB = 10
rectNoteLine1B = "  40,  150,  960,  150"
rectNoteLine2B = " 960,  840,  960,  150"
rectNoteLine3B = "  40,  840,  960,  840"
rectNoteLine4B = "  40,  150,   40,  840"

BoxPen( noteID, WHITE, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine2B )
BoxDrawLine( noteID, rectNoteLine3B )
BoxPen( noteID, GRAY, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine1B )
BoxDrawLine( noteID, rectNoteLine4B )

;=====
; Now put text in the banner headline box
;=====

noteHeight = 400
BoxTextFont( noteID, "Arial", noteHeight, 170, 0 ) ; headline font
rectNoteText = " 70, 200, 950, 800"
BoxTextColor( noteID, RED )

; next line creates the headline text. This can be copied anywhere
; in the program where the headline text needs to be changed
```

```

BoxDrawText( noteID, rectNoteText, "Text & Font Demo", 1, 4 )

;=====
; This section creates the Begin and Exit buttons
;=====

BoxButtonDraw( mainID, bBegin,    "&Begin", "100, 820, 250, 890" )
BoxButtonDraw( mainID, bExit,     "E&xit",  "750, 820, 900, 890" )

;=====
; Wait for a button to be selected (clicked)
;=====

iBox = 0
BoxButtonWait()
While iBox == 0
    For x = 1 to 2 ; sequential buttons required
        If BoxButtonStat( mainID, x ) then iBox = x
    Next
EndWhile

;=====
; poll the buttons to decide which was pressed (clicked)
;=====

If iBox
    BoxDataClear( mainID, "TOP" )
    Switch iBox
        case bExit
            exit
            break
        case bBegin
            gosub DO_TEXT
            break
    EndSwitch
EndIf
EndWhile
exit

;=====

```

Introduction to Programming

```
;      TEXT / FONT SUBROUTINE
;=====

:DO_TEXT
BoxCaption( mainID, "Fontopia" )
BoxNew( drawID, "0, 0, 1000, 500", 0 )
BoxColor( drawID, LTGRAY, 2)
BoxColor( mainID, LTGRAY, 0 )
BoxButtonDraw( mainID, bExit, "E&xit", "750, 860, 900, 930" )
BoxDrawRect( drawID, "0, 0, 1000, 1000", 2 )
BoxDrawRect( mainID, "0, 500, 1000, 1000", 2 )
listWords = "T'was brillig and the slithey toves did gyre gimble on
wabe all mimsey were borogoives momraths outgrabe"
wc = ItemCount( listWords, " " )
BoxTextColor( mainID, BLACK )

BoxTextFont( mainID, "", 20, 0, 0 )
BoxTextColor( mainID, RED )
BoxDrawText( mainID, "25, 510, 1000, 545", "Results shown will vary
with installed fonts...", 0, 0 )
BoxTextColor( mainID, BLACK )

fontStyleList = "Normal":@TAB:"Italic":@TAB:"Bold":@TAB:"Bold Italic"
fontTypeList = "0,100,70,170"

For i = 1 to 4
    fontStyle = ItemExtract( i, fontStyleList, @TAB )
    fontType = ItemExtract( i, fontTypeList, "," )
    xLf = 25 + ( i - 1 ) * 250
    xRt = xLf + 225
    BoxTextFont( mainID, "", 30, fontType, 0 )
    BoxDrawText( mainID, "%xLf%, 550, %xRt%, 600", fontStyle, 0, 0 )
    BoxTextFont( mainID, "Times New Roman", 30, fontType, 16 )
    BoxDrawText( mainID, "%xLf%, 600, %xRt%, 650", "Roman", 0, 0 )
    BoxTextFont( mainID, "Arial", 30, fontType, 32 )
    BoxDrawText( mainID, "%xLf%, 650, %xRt%, 700", "Swiss", 0, 0 )
    BoxTextFont( mainID, "Courier New", 30, fontType, 48 )
    BoxDrawText( mainID, "%xLf%, 700, %xRt%, 750", "Modern", 0, 0 )
    BoxTextFont( mainID, "Black Adder", 30, fontType, 64 )
```

```

BoxDrawText( mainID, "%xLf%, 750, %xRt%, 800", "Script", 0, 0 )
BoxTextFont( mainID, "Old English", 30, fontType, 80 )
BoxDrawText( mainID, "%xLf%, 800, %xRt%, 850", "Decorative", 0, 0 )
Next

BoxDataTag( drawID, "NULL" )
brk = 0
cnt = 0
wash = 0
BoxDestroy( noteID )
While @TRUE
    For i = 1 to wc
        cnt = cnt + 1
        GoSub RANDOM_COLOR
        BoxTextColor( drawID, "%rVal%, %gVal%, %bVal%" )
        x = Random( 800 )
        y = Random( 800 )
        s = x + Random( 200 )
        t = y + Random( 200 )
        fontStyle = 10 + Random(80) + ( 100 * Int( Random(1) ) )
        fontSize = Max( 10, Min( t - y, Random(200) ) )
        fontPitch = Random ( 2 )
        Switch Random ( 5 )
            case 0
                fontFamily = 0 ; default
                break
            case 1
                fontFamily = 16 ; Roman (Times Roman, etc)
                break
            case 2
                fontFamily = 32 ; Swiss (Helvetica, Arial, Swiss, etc)
                break
            case 3
                fontFamily = 48 ; Modern (Pica, Elite, Courier, etc)
                break
            case 4
                fontFamily = 64 ; Script

```

Introduction to Programming

```
        break
    case 5
        fontFamily = 80 ; Decorative (Old English, etc)
        break
    EndSwitch

    BoxTextFont( drawID, "", fontSize, fontStyle, fontPitch &
fontFamily )

    BoxDrawText( drawID, "%x%, %y%, %s%, %t%", ItemExtract( i,
listWords, " " ), 0, 0 )

    If BoxButtonStat( mainID, bExit )
        brk = 1
        BoxButtonKill( mainID, bExit )
        break
    Endif

    If cnt == 200
        cnt = 0
        wash = wash + 1
        if wash == 8 then wash = 0
        BoxColor( drawID, "%rVal%, %gVal%, %bVal%", wash )
        BoxDrawRect( drawID, "0, 0, 1000, 1000", 2 )
    Endif

    Next

    BoxDataClear( drawID, "NULL" )

    If brk then break
Endwhile
BoxDestroy( drawID )
return

;=====
;    SPARE SUBROUTINE
;=====

:SPARE
Message( "Empty", "This can be used for your own purposes." )
return

;=====
;    ADDITIONAL SUBROUTINES
```



```

=====
:SET_COLOR_CAPTION
BoxCaption( mainID, "That Old Line [Color Step = %nColorStep%]" )
return

:STEP_COLOR
If( rVal >= 250 ) then dRed = -nColorStep
If( rVal <= 5 ) then dRed = nColorStep
rVal = rVal + dRed

If( gVal >= 250 ) then dGreen = -nColorStep
If( gVal <= 5 ) then dGreen = nColorStep
gVal = gVal + dGreen

If( bVal >= 250 ) then dBlue = -nColorStep
If( bVal <= 5 ) then dBlue = nColorStep
bVal = bVal + dBlue
return

:RANDOM_HUE
Switch Random(3)
    case 1
        If( dRed == nColorStep )
            dRed = -nColorStep
        Else
            dRed = nColorStep
        Endif
        break
    case 2
        If( dGreen == nColorStep )
            dGreen = -nColorStep
        Else
            dGreen = nColorStep
        Endif
        break
    case 3
        If( dBlue == nColorStep )

```

Introduction to Programming

```
        dBlue = -nColorStep
    Else
        dBlue = nColorStep
    Endif
    break
EndSwitch
return
```

:RANDOM_COLOR

```
rVal = Int( Random(255) )
gVal = Int( Random(255) )
bVal = Int( Random(255) )
```

:CHECK_COLOR

```
q = Min( rVal, gVal, bVal )
If q == rVal then rVal = 0
If q == gVal then gVal = 0
If q == bVal then bVal = 0
q = Max( rVal, gVal, bVal )
If q == rVal then rVal = 255
If q == gVal then gVal = 255
If q == bVal then bVal = 255
return
```

Colors.wbt

```
;*****
;
;
; [Chapter 11]
; Colors.wbt
; Uses Box functions to create a color display
; showing three palettes of six colors each
;
;*****

;=====
; definitions for assorted colors
;=====

;===== gray scale =====
```

```

;           -R-  -G-  -B-
COLOR1  = "  0,   0,   0" ; Black
COLOR2  = " 64,  64,  64" ; Dark Gray
COLOR3  = "128, 128, 128" ; Gray
COLOR4  = "192, 192, 192" ; Light Gray
COLOR5  = "236, 236, 236" ; Off-White
COLOR6  = "255, 255, 255" ; White

;===== dark colors =====
;           -R-  -G-  -B-
COLOR7  = "  0,   0, 128" ; Dark Blue
COLOR8  = "  0, 160,   0" ; Dark Green
COLOR9  = "128,   0,   0" ; Dark Red
COLOR10 = "  0, 128, 128" ; Dark Cyan
COLOR11 = "128,   0, 128" ; Dark Magenta
COLOR12 = "128,  96,  48" ; Brown

;===== light colors =====
;           -R-  -G-  -B-
COLOR13 = "  0,   0, 255" ; Blue
COLOR14 = "  0, 255,   0" ; Green
COLOR15 = "255,   0,   0" ; Red
COLOR16 = "  0, 255, 255" ; Cyan
COLOR17 = "255,   0, 255" ; Magenta
COLOR18 = "255, 255,   0" ; Yellow

listColors = "Black,Dark Gray,Gray,Light Gray,Off-White,White,Dark
Blue,Dark Green,Dark Red,Dark Cyan,Dark
Magenta,Brown,Blue,Green,Red,Cyan,Magenta,Yellow"

;=====
; Window identifiers
;=====
mainID = 1           ; requires IDs less than 9

;=====
; Color rectangles
;=====

```

Introduction to Programming

```
rectColor1 = " 50, 100, 330, 400"
rectColor2 = "360, 100, 640, 400"
rectColor3 = "670, 100, 950, 400"
rectColor4 = " 50, 450, 330, 750"
rectColor5 = "360, 450, 640, 750"
rectColor6 = "670, 450, 950, 750"

;=====
; Button identifiers
;=====

bExit  = 1
bGrays = 2
bDark  = 3
bLight = 4

;=====
; Generic Initialization
;   allows windows to exit without warning (1)
;                               + quiet termination (4)
;=====
IntControl( 12, 5, 0, 0, 0 )

;=====
; Creates the top-level Window
;=====
BoxesUp( "100, 100, 900, 900", @NORMAL )

;===== Drawing the Main Box =====
; Note the use of while @TRUE. This use of while maintains
; the boxes until a user clicks on a button and exits
; the while construction
;=====

nOfs = 0
While @TRUE
    ;=====
    ; This section clears all previous drawing commands
```

```

; if any, form the "Box Stack"
; See the section in the Winbatch helpfile on
; Drawing Stack Management.
;=====

BoxDataClear( mainID, "TOP")

;=====
; This section creates the main window
;=====
BoxColor( mainID, COLOR14, 4 ) ; third param sets shaded background
BoxCaption( mainID, "Colors.wbt Demo" ) ; window caption
BoxDrawRect( mainID, " 0, 0, 1000, 1000", 2 ) ; size logical units

;=====
; This section creates the Color and Exit buttons
;=====
BoxButtonDraw( mainID, bGrays, "Gray Scale", "100, 820, 250, 890" )
BoxButtonDraw( mainID, bDark, "Dark Colors", "275, 820, 425, 890" )
BoxButtonDraw( mainID, bLight, "Light Colors", "450, 820, 600, 890" )
BoxButtonDraw( mainID, bExit, "E&xit", "750, 820, 900, 890" )

If nOfs == 12 then textColor = 1
If nOfs == 6 then textColor = 6
BoxTextFont( mainID, "Times", 40, 40, 0 | 0 ) ; initial font info
For i = 1 to 6
    nColor = i + nOfs ; box color
    If nOfs == 0 then textColor = 7 - i ; text color for gray scale
    label = ItemExtract( nColor, listColors, "," )
    label = StrCat( @CRLF, " ", label, @CRLF, @TAB, COLOR%nColor% )

    BoxColor( mainID, COLOR%nColor%, 0 ) ; background color
    BoxPen( mainID, COLOR%textColor%, 5 ) ; outline color
    BoxDrawRect( mainID, rectColor%i%, 1 ) ; fill banner box with
background color
    BoxTextColor( mainID, COLOR%textColor% )
    BoxDrawText( mainID, rectColor%i%, label, 0, 0 )
Next
;=====

```

Introduction to Programming

```
; Wait for a button to be selected (clicked)
;=====
iBox = 0
BoxButtonWait()

;=====
; and poll the buttons to decide which was pressed (clicked)
;=====
While iBox == 0
    For x = 1 to 4 ; sequential buttons required
        if BoxButtonStat( mainID, x ) then iBox = x
    Next
EndWhile

;=====
; now do something with the event
;=====
If iBox
    Switch iBox
        case bExit
            exit
            break
        case bGrays
            nOfs = 0
            break
        case bDark
            nOfs = 6
            break
        case bLight
            nOfs = 12
            break
    EndSwitch
Endif
EndWhile
exit
```

Buttons.wbt

```

;*****
; **
; **  [Chapter 11]
; **  Buttons.wbt
; **  Uses Box functions to demonstrate buttons
; **
;*****

;=====
; Window identifiers
;=====
mainID = 1      ; requires IDs less than 9

;=====
; Button identifiers
;=====
b1      = 1
b2      = 2
b3      = 3
b4      = 4
b5      = 5
b6      = 6
b7      = 7
b8      = 8
b9      = 9
b10     = 10
b11     = 11
b12     = 12

IntControl( 12, 5, 0, 0, 0 )

;===== Drawing the Main Box =====
BoxesUp( "100, 100, 900, 900", @NORMAL )
BoxColor( mainID, "0, 0, 0", 7 )
BoxDrawRect( mainID, "0, 0, 1000, 1000", 1 )

```

Introduction to Programming

```
BoxButtonDraw( mainID, b1, "1", "100, 100, 300, 250" )
BoxButtonDraw( mainID, b2, "2", "400, 100, 600, 250" )
BoxButtonDraw( mainID, b3, "3", "700, 100, 900, 250" )
BoxButtonDraw( mainID, b4, "4", "100, 300, 300, 450" )
BoxButtonDraw( mainID, b5, "5", "400, 300, 600, 450" )
BoxButtonDraw( mainID, b6, "6", "700, 300, 900, 450" )
BoxButtonDraw( mainID, b7, "7", "100, 500, 300, 650" )
BoxButtonDraw( mainID, b8, "8", "400, 500, 600, 650" )
BoxButtonDraw( mainID, b9, "9", "700, 500, 900, 650" )
BoxButtonDraw( mainID, b10, "10", "100, 700, 300, 850" )
BoxButtonDraw( mainID, b11, "11", "400, 700, 600, 850" )
BoxButtonDraw( mainID, b12, "12", "700, 700, 900, 850" )

; Randomly select an Exit button
bExit = Random( 11 ) + 1

While @TRUE
    ;=====
    ; Wait for a button to be selected (clicked)
    ;=====

    BoxButtonWait()
    If BoxButtonStat( mainID, b%bExit% ) then exit
    For i = 1 to 12
        If BoxButtonStat( mainID, b%i% ) == @TRUE
            Switch Random(8)
                case 1
                    Display( 1, "Forgetit", "Not even close" )
                    break
                case 2
                    Display( 1, "Try again", "It's here somewhere" )
                    break
                case 3
                    Display( 1, "Missed", "Your aim's lousy" )
                    break
                case 4
                    Display( 1, "Nayya", "No way, Fred" )
                    break
```



```

        case 5
            Display( 1, "Huh?", "You mean me?" )
            break
        case 6
            Display( 1, "Whoops", "Come on, you can do better" )
            break
        case 7
            Display( 1, "Ouch", "Hey, hands off" )
            break
        case 8
            Display( 1, "Wrong", "Guess again" )
            break
    EndSwitch
EndIf
Next
EndWhile
exit

```

Lines.wbt

```

;*****
;**
;**  [Chapter 11]
;**  Lines.wbt
;**  Uses Box functions to create rectangles and circles
;**
;*****

;===== gray scale =====
;
;      -R-  -G-  -B-
BLACK      = "  0,   0,   0"   ; Black
DKGRAY     = " 64,  64,  64"   ; Dark Gray
GRAY       = "128, 128, 128"   ; Gray
LTGRAY     = "192, 192, 192"   ; Light Gray
OFFWHITE   = "236, 236, 236"   ; Off-White
WHITE      = "255, 255, 255"   ; White

;===== dark colors =====

```

Introduction to Programming

```
;          -R-  -G-  -B-
DKBLUE     = "  0,   0, 128" ; Dark Blue
DKGREEN    = "  0, 160,   0" ; Dark Green
DKRED      = "128,   0,   0" ; Dark Red
DKCYAN     = "  0, 128, 128" ; Dark Cyan
DKMAGENTA  = "128,   0, 128" ; Dark Magenta
BROWN      = "128,  96,  48" ; Brown

;===== light colors =====
;          -R-  -G-  -B-
BLUE       = "  0,   0, 255" ; Blue
GREEN      = "  0, 255,   0" ; Green
RED        = "255,   0,   0" ; Red
CYAN       = "  0, 255, 255" ; Cyan
MAGENTA    = "255,   0, 255" ; Magenta
YELLOW     = "255, 255,   0" ; Yellow

;===== Generic Initialization =====;
; allows windows to exit without warning (1)
;          + quiet termination (4)
IntControl( 12, 5, 0, 0, 0 )

;===== Drawing the Main Box =====
BoxesUp( "100, 100, 900, 900", @NORMAL )

; Window identifiers
mainID = 1          ; requires successive IDs
drawID = 2
noteID = 3
progID = 8

bExit   = 1
bBegin  = 2
bColorUp = 4
bColorDn = 5

nColorStep = 3
```

```

dRed    = nColorStep
dBlue   = nColorStep
dGreen  = nColorStep

;=====
;  This section creates the main window
;=====
BoxColor( mainID, GREEN, 4 )           ; third param sets
shaded background
BoxCaption( mainID, "Lines.wbt Demo" ) ; window caption
BoxDrawRect( mainID, " 0, 0, 1000, 1000", 2 ) ; size is in logical
units
BoxTextFont( mainID, "Times", 80, 80, 0 | 0 ) ; initial font
information
BoxTextColor( mainID, "255, 255, 0" ) ; initial font color

;=====
;  And this puts a message in the window
;=====
BoxDrawText( mainID, "245, 500, 700, 600", "Pick [Begin] to start", 0,
0 )

;=====
;  This section creates the fancy banner headline in a box
;=====
rectNote = "100, 100, 900, 340"      ; set the size of the banner box
BoxNew( noteID, rectNote, 1 )         ; create the box
BoxColor( noteID, LTGRAY, 0 )         ; background is Light Gray, no
gradient
BoxDrawRect( noteID, "", 2 )          ; fill the entire banner box with
the background color

;=====
;  Next create a 3-D outline around the box
;=====
; note that all units are logical units relative to the textbox
; which (by default) has a logical size of 1000 x 1000 units
notePenWidthA = 20
rectNoteLine1A = "    0,    0, 1000,    0"

```

Introduction to Programming

```
rectNoteline2A = "1000, 1000, 1000, 0"
rectNoteLine3A = " 0, 1000, 1000, 1000"
rectNoteLine4A = " 0, 0, 0, 1000"

; draw the outer outline
BoxPen( noteID, WHITE, notePenWidthA ) ; line color top and left
BoxDrawLine( noteID, rectNoteLine1A )
BoxDrawLine( noteID, rectNoteLine4A )
BoxPen( noteID, GRAY, notePenWidthA ) ; line color bottom and right
BoxDrawLine( noteID, rectNoteLine2A )
BoxDrawLine( noteID, rectNoteLine3A )

; draw the inner outline
notePenWidthB = 10
rectNoteLine1B = " 40, 150, 960, 150"
rectNoteline2B = " 960, 840, 960, 150"
rectNoteLine3B = " 40, 840, 960, 840"
rectNoteLine4B = " 40, 150, 40, 840"

BoxPen( noteID, WHITE, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine2B )
BoxDrawLine( noteID, rectNoteLine3B )
BoxPen( noteID, GRAY, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine1B )
BoxDrawLine( noteID, rectNoteLine4B )

;=====
; Now put text in the banner headline box
;=====
noteHeight = 400
BoxTextFont( noteID, "Arial", noteHeight, 170, 0 ) ; set headline font
rectNoteText = " 70, 200, 950, 800"
BoxTextColor( noteID, RED )
; this next line creates the headline text -- this line can be copied
; anywhere in the program where the headline text needs to be changed
BoxDrawText( noteID, rectNoteText, "Line Drawing Demo", 1, 4 )
```

```

;=====
;  This section creates the Begin and Exit buttons
;=====
BoxButtonDraw( mainID, bBegin,    "&Begin", "100, 820, 250, 890" )
BoxButtonDraw( mainID, bExit,    "E&xit",  "750, 820, 900, 890" )

;=====
;  Wait for a button to be selected (clicked)
;=====
iBox  =  0
BoxButtonWait()
While iBox == 0
    For x = 1 to 2                ; sequential buttons required
        If BoxButtonStat( mainID, x ) then iBox = x
    Next
EndWhile

;=====
;  poll the buttons to decide which was pressed (clicked)
;=====
If iBox
    BoxDataClear( mainID, "TOP" )
    Switch iBox
        case bExit
            exit
            break
        case bBegin
            GoSub DO_LINES
            break
    EndSwitch
EndIf
exit

;=====
;      DRAW LINES SUBROUTINE
;=====
:DO_LINES

```

Introduction to Programming

```
boxLf = 0
boxRt = 1000
boxTop = 0
boxBtm = 1000
GoSub RANDOM_COLOR           ; get an initial color value

rectBox = "%boxLf%, %boxTop%, %boxRt%, %boxBtm%"
GoSub SET_COLOR_CAPTION
BoxNew( drawID, rectBox, 0 )
BoxColor( drawID, BLACK, 0 )
BoxDrawRect( drawID, rectBox, 2 )
BoxButtonDraw( drawID, bColorUp, "Inc Color", " 70, 860, 220, 930" )
BoxButtonDraw( drawID, bColorDn, "Dec Color", "250, 860, 400, 930" )
BoxButtonDraw( drawID, bExit, "E&xit", "750, 860, 900, 930" )

BoxDataTag( drawID, "NULL" )

point_1 = 200
point_2 = 200
point_3 = 400
point_4 = 400

vertex_1 = 20 - Random( 40 )
vertex_2 = 20 - Random( 40 )
vertex_3 = 20 - Random( 40 )
vertex_4 = 20 - Random( 40 )

bForward = @TRUE
While @TRUE
    If BoxButtonStat( drawID, bExit ) == 1 then
        BoxButtonKill( drawID, bColorUp )
        BoxButtonKill( drawID, bColorDn )
        BoxButtonKill( drawID, bExit )
        break
    EndIf

    If BoxButtonStat( drawID, bColorUp ) == 1 then
```

```

    If nColorStep < 10 then nColorStep = nColorStep + 1
    GoSub SET_COLOR_CAPTION
EndIf

If BoxButtonStat( drawID, bColorDn ) == 1 then
    If nColorStep > 1 then nColorStep = nColorStep - 1
    GoSub SET_COLOR_CAPTION
EndIf

BoxDataClear( drawID, "NULL" )

If bForward == @TRUE
    GoSub RANDOM_HUE
    bForward = @FALSE ; change step direction
EndIf

BoxPen( drawID, "%rVal%, %gVal%, %bVal%", 1 )
BoxDrawLine( drawID, "%point_1%,%point_2%,%point_3%,%point_4%" )
GoSub STEP_COLOR
bForward = @FALSE

For q = 1 to 4
    point_%q% = point_%q% + vertex_%q%
    If point_%q% <= boxLf
        point_%q% = boxLf
        vertex_%q% = Int( Random(10) ) + 1
        bForward = @TRUE ; change step direction
    EndIf
    If point_%q% >= boxRt
        point_%q% = boxRt
        vertex_%q% = -Int( Random(10) ) - 1
        bForward = @TRUE ; change step direction
    EndIf
Next
EndWhile
BoxDestroy( drawID )
return

```

Introduction to Programming

```
:SET_COLOR_CAPTION
    BoxCaption( mainID, "That Old Line    [Color Step = %nColorStep%]" )
return

:STEP_COLOR
    If( rVal >= 245 ) then dRed = -nColorStep
    If( rVal <= 10 )  then dRed = nColorStep
    rVal = rVal + dRed

    If( gVal >= 245 ) then dGreen = -nColorStep
    If( gVal <= 10 )  then dGreen = nColorStep
    gVal = gVal + dGreen

    If( bVal >= 245 ) then dBlue = -nColorStep
    If( bVal <= 10 )  then dBlue = nColorStep
    bVal = bVal + dBlue
return

:RANDOM_HUE
    switch Random(3)
        case 1
            If( dRed == nColorStep )
                dRed = -nColorStep
            else
                dRed = nColorStep
            EndIf
            break
        case 2
            If( dGreen == nColorStep )
                dGreen = -nColorStep
            else
                dGreen = nColorStep
            EndIf
            break
        case 3
            If( dBlue == nColorStep )
```



```

        dBlue = -nColorStep
    else
        dBlue = nColorStep
    EndIf
    break
endSwitch
return

```

```
:RANDOM_COLOR
```

```

    rVal = Int( Random(255) )
    gVal = Int( Random(255) )
    bVal = Int( Random(255) )

```

```
:CHECK_COLOR
```

```

    q = Min( rVal, gVal, bVal )
    If q == rVal then rVal = 0
    If q == gVal then gVal = 0
    If q == bVal then bVal = 0
    q = Max( rVal, gVal, bVal )
    If q == rVal then rVal = 255
    If q == gVal then gVal = 255
    If q == bVal then bVal = 255

```

```
return
```

Shapes.wbt

```

;*****
;**
;** [Chapter 11]
;** Shapes.wbt
;** Uses Box functions to create rectangles and circles
;**
;*****

;===== gray scale =====
;
;           -R-  -G-  -B-
BLACK      = "  0,   0,   0"   ; Black
DKGRAY     = " 64,  64,  64"   ; Dark Gray
GRAY       = "128, 128, 128"   ; Gray

```

Introduction to Programming

```
LTGRAY      = "192, 192, 192"    ; Light Gray
OFFWHITE    = "236, 236, 236"    ; Off-White
WHITE       = "255, 255, 255"    ; White

;===== dark colors =====
;           -R-  -G-  -B-
DKBLUE      = "  0,   0, 128"    ; Dark Blue
DKGREEN     = "  0, 160,   0"    ; Dark Green
DKRED       = "128,   0,   0"    ; Dark Red
DKCYAN      = "  0, 128, 128"    ; Dark Cyan
DKMAGENTA   = "128,   0, 128"    ; Dark Magenta
BROWN       = "128,  96,  48"    ; Brown

;===== light colors =====
;           -R-  -G-  -B-
BLUE        = "  0,   0, 255"    ; Blue
GREEN       = "  0, 255,   0"    ; Green
RED         = "255,   0,   0"    ; Red
CYAN        = "  0, 255, 255"    ; Cyan
MAGENTA     = "255,   0, 255"    ; Magenta
YELLOW      = "255, 255,   0"    ; Yellow

;===== Generic Initialization =====;
; allows windows to exit without warning (1) + quiet termination (4)
IntControl( 12, 5, 0, 0, 0 )

;===== Drawing the Main Box =====
BoxesUp( "100, 100, 900, 900", @NORMAL )

; Window identifiers
mainID = 1          ; requires successive IDs
drawID = 2
noteID = 3
progID = 8

bExit   = 1
bBegin  = 2
```

```
bChange = 2
```

```

;=====
;  This section creates the main window
;=====
BoxColor( mainID, GREEN, 4 ); third param sets shaded background
BoxCaption( mainID, "Shapes.wbt Demo" ) ; window caption
BoxDrawRect( mainID, " 0, 0, 1000, 1000", 2 ) ; size in logical units
BoxTextFont( mainID, "Times", 80, 80, 0 | 0 ) ; initial font info
BoxTextColor( mainID, "255, 255, 0" ) ; initial font color

;=====
;  And this puts a message in the window
;=====
BoxDrawText( mainID, "245, 500, 700, 600", "Pick [Begin] to start", 0,
0 )

;=====
;  This section creates the fancy banner headline in a box
;=====
rectNote = "100, 100, 900, 340" ; set the size of the banner box
BoxNew( noteID, rectNote, 1 ) ; create the box
BoxColor( noteID, LTGRAY, 0 ) ; background is Light Gray, no gradient
BoxDrawRect( noteID, "", 2 ) ; fill banner box with background color

;=====
;  Next create a 3-D outline around the box
;=====
; note that all units are logical units relative to the notebbox
; which (by default) has a logical size of 1000 x 1000 units
notePenWidthA = 20
rectNoteLine1A = " 0, 0, 1000, 0"
rectNoteLine2A = "1000, 1000, 1000, 0"
rectNoteLine3A = " 0, 1000, 1000, 1000"
rectNoteLine4A = " 0, 0, 0, 1000"

; draw the outer outline
BoxPen( noteID, WHITE, notePenWidthA ) ; line color top and left

```

Introduction to Programming

```
BoxDrawLine( noteID, rectNoteLine1A )
BoxDrawLine( noteID, rectNoteLine4A )
BoxPen( noteID, GRAY, notePenWidthA ) ; line color bottom and right
BoxDrawLine( noteID, rectNoteLine2A )
BoxDrawLine( noteID, rectNoteLine3A )

; draw the inner outline
notePenWidthB = 10
rectNoteLine1B = " 40, 150, 960, 150"
rectNoteLine2B = " 960, 840, 960, 150"
rectNoteLine3B = " 40, 840, 960, 840"
rectNoteLine4B = " 40, 150, 40, 840"

BoxPen( noteID, WHITE, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine2B )
BoxDrawLine( noteID, rectNoteLine3B )
BoxPen( noteID, GRAY, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine1B )
BoxDrawLine( noteID, rectNoteLine4B )

;=====
; Now put text in the banner headline box
;=====
noteHeight = 300
BoxTextFont( noteID, "Arial", noteHeight, 170, 0 ) ; set headline font
rectNoteText = " 70, 200, 950, 800"
BoxTextColor( noteID, RED )
; this next line creates the headline text -- this line can be copied
; anywhere in the program where the headline text needs to be changed
BoxDrawText( noteID, rectNoteText, "Shape Drawing Demo", 1, 1+4 )

;=====
; This section creates the Begin and Exit buttons
;=====
BoxButtonDraw( mainID, bBegin, "&Begin", "100, 820, 250, 890" )
BoxButtonDraw( mainID, bExit, "E&xit", "750, 820, 900, 890" )
```

```

;=====
;  Wait for a button to be selected (clicked)
;=====

iBox  =  0
BoxButtonWait()
While iBox == 0
    For x = 1 to 2 ; sequential buttons required
        If BoxButtonStat( mainID, x ) then iBox = x
    Next
EndWhile

;=====
;  poll the buttons to decide which was pressed (clicked)
;=====

If iBox
    BoxDataClear( mainID, "TOP" )
    Switch iBox
        case bExit
            exit
            break
        case bBegin
            gosub DO_SHAPES
            break
    EndSwitch
EndIf

exit

;=====
;      DRAW SHAPES SUBROUTINE
;=====

:DO_SHAPES
    bRectangles = @TRUE
    BoxCaption( mainID, "Random Shapes" )
    BoxNew( drawID, "0, 0, 1000, 1000", 0 )
    BoxButtonDraw( drawID, bChange, "Circles", " 70, 860, 220, 930" )
    BoxButtonDraw( drawID, bExit,  "E&xit",  "750, 860, 900, 930" )

```

Introduction to Programming

```
;=====
; Set the point to clear the drawing stack
; for a BoxDataClear function later
;=====
BoxDataTag( drawID, "NULL" )

While @TRUE

    ;=====
    ; see if the Exit button was clicked
    ;=====
    If BoxButtonStat( drawID, bExit ) == 1 then
        BoxButtonKill( drawID, bChange )
        BoxButtonKill( drawID, bExit )
        break
    EndIf

    ;=====
    ; update the button label
    ;=====
    If BoxButtonStat( drawID, bChange ) == 1 then
        If bRectangles then
            bRectangles = @FALSE
            BoxButtonDraw( drawID, bChange, "Rectangles", "70,860,220,930" )
        Else
            bRectangles = @TRUE
            BoxButtonDraw( drawID, bChange, "Circles", "70,860,220,930" )
        EndIf
    EndIf

    ;=====
    ; random coordinates for shape
    ;=====
    boxLf = Random(1000)
    boxRt = Random(1000)
    boxTop = Random(1000)
```

```

boxBtm = Random(1000)

;=====
; create a random color
;=====
rVal = Random(255)
gVal = Random(255)
bVal = Random(255)
BoxColor( drawID, "%rVal%, %gVal%, %bVal%", 0 )

;=====
; invert the color for the outline
;=====
rPen = 255 - rVal
gPen = 255 - gVal
bPen = 255 - bVal
BoxPen( drawID, "%rPen%, %gPen%, %bPen%", 5 )

;=====
; draw either a rectangle or a circle
;=====
If bRectangles then
    BoxDrawRect( drawID, "%boxLf%, %boxTop%, %boxRt%, %boxBtm%", 1 )
Else
    BoxDrawCircle( drawID, "%boxLf%, %boxTop%, %boxRt%, %boxBtm%", 1 )
EndIf

;=====
; Clear part of the drawing stack at point set by BoxDataTag.
;=====
BoxDataClear( drawID, "NULL" )

EndWhile
BoxDestroy( drawID )
return

```

Introduction to Programming

Phone.lst

```
Albert Einstein    34 McFadden Apt 7 Princeton    MA    01234 (222) 357-
9246
Han Solo    87 Cloud St    #92 Hollywood    CA    89034 (111) 345-6789
Joe Friday 1 Police Plaza    New York    NY    01010 (333) 135-7924
John Jacob Jingleheimer Schmidt    999 9th St    Stumptown    CA
95446 (707) 999-8765
Rob Roy123 4th Ave    Gurrock    MN    78945 (555) 567-8901
S. F. Katt 1 Park Ave    New York    NY    10016 (781) 393-3700
Sol Rosencranz #2 Old Stone Bridge    Daubmore    MS    68758 (999)
555-2345
DilbertCubicle 23    Silicon ValleyCA    95444 (890) 555-5893
```

PhoneListBox.wbt

```
;*****
;
; **
; ** [Chapter 11]
; ** PhoneListBox.wbt
; ** Uses Box functions for formatted text dieplay
; **
;*****

; Set working directory to the same directory the script is in
; in case somehow it got started un in some strange manner
DirChange( DirScript() )

;===== gray scale =====
;          -R-  -G-  -B-
BLACK      = "  0,   0,   0"    ; Black
DKGRAY     = " 64,  64,  64"    ; Dark Gray
GRAY       = "128, 128, 128"    ; Gray
LTGRAY     = "192, 192, 192"    ; Light Gray
OFFWHITE   = "236, 236, 236"    ; Off-White
WHITE      = "255, 255, 255"    ; White

;===== dark colors =====
;          -R-  -G-  -B-
DKBLUE     = "  0,   0, 128"    ; Dark Blue
DKGREEN    = "  0, 160,   0"    ; Dark Green
```



```

DKRED      = "128,  0,  0"    ; Dark Red
DKCYAN     = "  0, 128, 128"   ; Dark Cyan
DKMAGENTA  = "128,  0, 128"   ; Dark Magenta
BROWN      = "128, 96, 48"    ; Brown

;===== light colors =====
;          -R-  -G-  -B-
BLUE       = "  0,  0, 255"    ; Blue
GREEN      = "  0, 255,  0"    ; Green
RED        = "255,  0,  0"     ; Red
CYAN       = "  0, 255, 255"   ; Cyan
MAGENTA    = "255,  0, 255"   ; Magenta
YELLOW     = "255, 255,  0"    ; Yellow

;===== Generic Initialization =====;
; allows windows to exit without warning (1)
;                               + quiet termination (4)
IntControl( 12, 5, 0, 0, 0 )

;===== Drawing the Main Box =====
BoxesUp( "100, 100, 900, 900", @NORMAL )

; Window identifiers
mainID = 1          ; requires successive IDs
drawID = 2
noteID = 3

bExit  = 1
bBegin = 2

nColor = 3          ; initial drawing color
nSelect = 0         ; no selection to start
sDelimiter = '|'

;=====
; This section creates the main window

```

Introduction to Programming

```
;=====
BoxColor( mainID, GREEN, 4 ) ; third param sets shaded background
BoxCaption( mainID, "PhoneListBox.wbt" ) ; window caption
BoxDrawRect( mainID, " 0, 0, 1000, 1000", 2 ) ; size logical units
BoxTextFont( mainID, "Times", 80, 80, 0 | 0 ) ; initial font info
BoxTextColor( mainID, "255, 255, 0" ) ; initial font color

;=====
; And this puts a message in thw window
;=====
BoxDrawText( mainID, "245, 500, 700, 600", "Pick [Begin] to start", 0,
0 )

;=====
; This section creates the fancy banner headline in a box
;=====
rectNote = "100, 100, 900, 340" ; set the size of the banner box
BoxNew( noteID, rectNote, 1 ) ; create the box
BoxColor( noteID, LTGRAY, 0 ) ; background Lt Gray, no gradient
BoxDrawRect( noteID, "", 2 ) ; fill banner box with background color

;=====
; Next create a 3-D outline around the box
;=====
; note that all units are logical units relative to the notebox
; which (by default) has a logical size of 1000 x 1000 units
notePenWidthA = 20
rectNoteLine1A = " 0, 0, 1000, 0"
rectNoteline2A = "1000, 1000, 1000, 0"
rectNoteLine3A = " 0, 1000, 1000, 1000"
rectNoteLine4A = " 0, 0, 0, 1000"

; draw the outer outline
BoxPen( noteID, WHITE, notePenWidthA ) ; line color top and left
BoxDrawLine( noteID, rectNoteLine1A )
BoxDrawLine( noteID, rectNoteLine4A )
BoxPen( noteID, GRAY, notePenWidthA ) ; line color bottom and right
BoxDrawLine( noteID, rectNoteline2A )
```

```

BoxDrawLine( noteID, rectNoteLine3A )

; draw the inner outline
notePenWidthB = 10
rectNoteLine1B = " 40, 150, 960, 150"
rectNoteLine2B = " 960, 840, 960, 150"
rectNoteLine3B = " 40, 840, 960, 840"
rectNoteLine4B = " 40, 150, 40, 840"

BoxPen( noteID, WHITE, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine2B )
BoxDrawLine( noteID, rectNoteLine3B )
BoxPen( noteID, GRAY, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine1B )
BoxDrawLine( noteID, rectNoteLine4B )

;=====
; Now put text in the banner headline box
;=====
noteHeight = 400
BoxTextFont( noteID, "Arial", noteHeight, 170, 0 ) ; set the headline
font
rectNoteText = " 70, 200, 950, 800"
BoxTextColor( noteID, RED )
; this next line creates the headline text -- this line can be copied
; anywhere in the program where the headline text needs to be changed
BoxDrawText( noteID, rectNoteText, "Phone List Box", 1, 4 )

;=====
; This section creates the Begin and Exit buttons
;=====
BoxButtonDraw( mainID, bBegin, "&Begin", "100, 820, 250, 890" )
BoxButtonDraw( mainID, bExit, "E&xit", "750, 820, 900, 890" )

;=====
; Wait for a button to be selected (clicked)
;=====
iBox = 0

```

Introduction to Programming

```
BoxButtonWait()  
While iBox == 0  
    For x = 1 to 2                ; sequential buttons required  
        If BoxButtonStat( mainID, x ) then iBox = x  
    Next  
EndWhile  
  
;=====;  
; poll the buttons to decide which was pressed (clicked)  
;=====;  
  
If iBox  
    BoxDataClear( mainID, "TOP" )  
    Switch iBox  
        case bExit  
            exit  
            break  
        case bBegin  
            BoxDestroy( noteID )  
            GoSub DO_LIST  
            break  
    EndSwitch  
EndIf  
  
exit  
  
:DO_LIST  
BoxCaption( mainID, "Left-Click to select, right-click to exit" ) ;  
window caption  
sDataFile = "Phone.lst"  
hFile = FileOpen( sDataFile, "READ" )  
listPhone = ""  
While @TRUE  
    sLineIn = FileRead( hFile )  
    If( sLineIn == "*EOF*" ) then break  
    listPhone = StrCat( listPhone, sLineIn, sDelimiter )  
EndWhile  
  
FileClose( hFile ) ; close the input file
```

```

listPhone = ItemSort( listPhone, sDelimiter )
GoSub DISPLAY_LIST
exit

:DISPLAY_LIST
nCount = 0
nCheck = 0
nRow = 0

nFontHeight = 30
nRowHeight = Int( nFontHeight * 1.2 )
rectBox = "0, 0, 1000, 1000"
BoxNew( drawID, rectBox, 0 )
BoxColor( drawID, BLACK, 0 )
BoxDrawRect( drawID, rectBox, 2 )
BoxTextFont( drawID, "Arial", nFontHeight, 40, 0 | 0 )
BoxDataTag( drawID, "LIST" )

While @TRUE      ; Redo list
    BoxDataClear( drawID, "LIST" )
    For i = 1 to ItemCount( listPhone, sDelimiter )
        If i == nSelect
            BoxTextColor( drawID, GREEN )
        Else
            BoxTextColor( drawID, WHITE )
        EndIf
        sTemp      = ItemExtract( i, listPhone, sDelimiter )
        sName      = ItemExtract( 1, sTemp, @TAB )
        sPhone     = ItemExtract( 7, sTemp, @TAB )
        yTop = Int( nRowHeight * ( i - 1 ) )
        yBottom = yTop + nFontHeight
        BoxDrawText( drawID, "10, %yTop%, 490, %yBottom%", sName, 0, 0 )
        BoxDrawText( drawID, "500, %yTop%, 990, %yBottom%", sPhone, 0, 0 )
    Next

nButton = 0

```

Introduction to Programming

```
doRedraw=@FALSE
While nButton == 0 ; loop while waiting for mouse event
    nCheck = nCheck + 1
    nButton = MouseInfo( 4 )
    If nButton & 1 then return ; use the right mouse button to exit
    If nButton & 4
        nCount = nCount + 1
        sPosition = MouseInfo( 6 ) ; get position
        nRow = ItemExtract( 2, sPosition, " " )
        nSelect = Int( nRow / nRowHeight ) + 1
        If nCount > 2 ; is this the third click?
            sTemp      = ItemExtract( nSelect, listPhone, sDelimiter )
            sName       = ItemExtract( 1, sTemp, @TAB )
            sAddress1   = ItemExtract( 2, sTemp, @TAB )
            sAddress2   = ItemExtract( 3, sTemp, @TAB )
            sCity       = ItemExtract( 4, sTemp, @TAB )
            sState      = ItemExtract( 5, sTemp, @TAB )
            sZip        = ItemExtract( 6, sTemp, @TAB )
            sPhone      = ItemExtract( 7, sTemp, @TAB )
            sReport     = StrCat( sName, @CRLF, sAddress1, @CRLF )
            if sAddress2 != "" then sReport = StrCat( sReport,
sAddress2, @CRLF )
            sReport     = StrCat( sReport, sCity, ", ", sState, " ", sZip,
@CRLF, sPhone )
            Message( "Selection", sReport )
        EndIf
        doRedraw=@TRUE
        break ; redisplay the list
    EndIf
    If nCheck > 199
        nCount = 0 ; long enough, reset the count
        nCheck = 0
        nSelect = 0 ; reset selection
        doRedraw=@TRUE
        break ; Redisplay the list
    EndIf
EndWhile ; nButton == 0
If doRedraw==@FALSE then return
```

```
Endwhile ;redo list
return
```

Chapter 12 Samples

Freehand.wbt

```
;*****
; **
; ** [Chapter 12]
; ** Freehand.wbt
; ** Uses Box functions for freehand drawing routine tracks mouse
; ** and mouse button status.
; **
;*****

;===== gray scale =====
;          -R-  -G-  -B-
BLACK      = "  0,   0,   0" ; Black
DKGRAY     = " 64,  64,  64" ; Dark Gray
GRAY       = "128, 128, 128" ; Gray
LTGRAY     = "192, 192, 192" ; Light Gray
OFFWHITE   = "236, 236, 236" ; Off-White
WHITE      = "255, 255, 255" ; White

;===== dark colors =====
;          -R-  -G-  -B-
DKBLUE     = "  0,   0, 128" ; Dark Blue
DKGREEN    = "  0, 160,   0" ; Dark Green
DKRED      = "128,   0,   0" ; Dark Red
DKCYAN     = "  0, 128, 128" ; Dark Cyan
DKMAGENTA  = "128,   0, 128" ; Dark Magenta
BROWN      = "128,  96,  48" ; Brown

;===== light colors =====
;          -R-  -G-  -B-
BLUE       = "  0,   0, 255" ; Blue
GREEN      = "  0, 255,   0" ; Green
```

Introduction to Programming

```
RED      = "255,  0,  0"   ; Red
CYAN     = "  0, 255, 255"  ; Cyan
MAGENTA  = "255,  0, 255"  ; Magenta
YELLOW   = "255, 255,  0"   ; Yellow

;===== drawing palette =====
COLOR1 = BLUE
COLOR2 = GREEN
COLOR3 = RED
COLOR4 = CYAN
COLOR5 = MAGENTA
COLOR6 = YELLOW

;===== Generic Initialization =====;
; allows windows to exit without warning (1)
;                                     + quiet termination (4)
IntControl( 12, 5, 0, 0, 0 )

;===== Drawing the Main Box =====
BoxesUp( "100, 100, 900, 900", @NORMAL )

; Window identifiers
mainID = 1 ; requires successive IDs
drawID = 2
noteID = 3

bExit  = 1
bBegin = 2

nColor = 3 ; initial drawing color

;=====
; This section creates the main window
;=====
BoxColor( mainID, GREEN, 4 ) ; third param sets shaded background
BoxCaption( mainID, "Freehand.wbt Demo" ) ; window caption
BoxDrawRect( mainID, " 0, 0, 1000, 1000", 2 ) ; size is in logical
units
```



```

BoxTextFont( mainID, "Times", 80, 80, 0 | 0 ) ; initial font
information
BoxTextColor( mainID, "255, 255, 0" ) ; initial font color

;=====
; And this puts a message in thw window
;=====

BoxDrawText( mainID, "245, 500, 700, 600", "Pick [Begin] to start", 0,
0 )

;=====
; This section creates the fancy banner headline in a box
;=====

rectNote = "100, 100, 900, 340" ; set the size of the banner box
BoxNew( noteID, rectNote, 1 ) ; create the box
BoxColor( noteID, LTGRAY, 0 ) ; background is Light Gray, no gradient
BoxDrawRect( noteID, "", 2 ) ; fill the entire banner box with the
background color

;=====
; Next create a 3-D outline around the box
;=====

; note that all units are logical units relative to the textbox
; which (by default) has a logical size of 1000 x 1000 units
notePenWidthA = 20
rectNoteLine1A = " 0, 0, 1000, 0"
rectNoteline2A = "1000, 1000, 1000, 0"
rectNoteLine3A = " 0, 1000, 1000, 1000"
rectNoteLine4A = " 0, 0, 0, 1000"

; draw the outer outline
BoxPen( noteID, WHITE, notePenWidthA ) ; line color top and left
BoxDrawLine( noteID, rectNoteLine1A )
BoxDrawLine( noteID, rectNoteLine4A )
BoxPen( noteID, GRAY, notePenWidthA ) ; line color bottom and right
BoxDrawLine( noteID, rectNoteLine2A )
BoxDrawLine( noteID, rectNoteLine3A )

```

Introduction to Programming

```
; draw the inner outline
notePenWidthB = 10
rectNoteLine1B = " 40, 150, 960, 150"
rectNoteLine2B = " 960, 840, 960, 150"
rectNoteLine3B = " 40, 840, 960, 840"
rectNoteLine4B = " 40, 150, 40, 840"

BoxPen( noteID, WHITE, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine2B )
BoxDrawLine( noteID, rectNoteLine3B )
BoxPen( noteID, GRAY, notePenWidthB )
BoxDrawLine( noteID, rectNoteLine1B )
BoxDrawLine( noteID, rectNoteLine4B )

;=====
; Now put text in the banner headline box
;=====
noteHeight = 400
BoxTextFont( noteID, "Arial", noteHeight, 170, 0 ); set the headline
font
rectNoteText = " 70, 200, 950, 800"
BoxTextColor( noteID, RED )
; this next line creates the headline text -- this line can be copied
; anywhere in the program where the headline text needs to be changed
BoxDrawText( noteID, rectNoteText, "Freehand Drawing", 1, 4 )

;=====
; This section creates the Begin and Exit buttons
;=====
BoxButtonDraw( mainID, bBegin, "&Begin", "100, 820, 250, 890" )
BoxButtonDraw( mainID, bExit, "&Exit", "750, 820, 900, 890" )

;=====
; Wait for a button to be selected (clicked)
;=====
iBox = 0
BoxButtonWait()
While iBox == 0
```

```

    For x = 1 to 2 ; sequential buttons required
        If BoxButtonStat( mainID, x ) then iBox = x
    Next
EndWhile

;=====
; poll the buttons to decide which was pressed (clicked)
;=====

If iBox
    Switch iBox
        case bExit
            exit
            break
        case bBegin
            gosub DO_DRAW
            break
    EndSwitch
EndIf

exit

;=====
;     FREEHAND DRAW SUBROUTINE
;=====

:DO_DRAW
    nPenResult = 0
    lastPoint = "-1,-1"
    savePoint = "-1,-1"
    BoxCaption( mainID, "Freehand" )
    BoxNew( drawID, "0, 0, 1000, 1000", 0 )
    BoxPen( drawID, COLOR%nColor%, 1 )
    BoxButtonDraw( drawID, bExit, "E&xit", "750, 860, 900, 930" )
    BoxDataTag( drawID, "NULL" )
    Exclusive( @ON )
    While @TRUE
        BoxDataClear( drawID, "NULL" )
        nButton = 0

```

Introduction to Programming

```
nUp = 0

While nButton == 0 ; loop while waiting for mouse event
    nButton = MouseInfo( 4 )
    nUp = nUp + 1
    if nUp == 10 then lastPoint = "-1,-1"
EndWhile

If( nButton & 04 ) ; left button is down
    Point = MouseInfo( 6 )
    Point = StrReplace( Point, " ", ",")
    If lastPoint != "-1,-1" then BoxDrawLine( drawID,
"%lastPoint%, %Point%" )
        lastPoint = Point
        savePoint = Point
    EndIf

If( nButton & 01 ) ; right button is down
    Point = MouseInfo( 6 )
    Point = StrReplace( Point, " ", ",")
    If savePoint != "-1,-1" then BoxDrawLine( drawID,
"%savePoint%, %Point%" )
        savePoint = Point
    EndIf

If BoxButtonStat( drawID, bExit ) == 1
    BoxButtonKill( drawID, bExit )
    break
EndIf
Endwhile
BoxDestroy( drawID )
return
```

Chapter 13 Samples

SelfTest.wbt

```
;*****
```

```

; **
; ** [Chapter 13]
; ** SelfTest.wbt
; ** Sample IntControl function.
; **
; *****

IntControl( 1, "Argument 1", "Argument 2", "Argument 3", "Argument 4" )
exit

```

Chapter 14 Samples

Exercise A.wbt

```

; =====
; =====
; =====

#DefineSubRoutine InitDialogConstants()
    ; DialogprocOptions Constants
    MSG_INIT=0 ; The one-time initialization
    MSG_TIMER=1 ; Timer event
    MSG_BUTTONPUSHED=2 ; Pushbutton or Picturebutton
    MSG_RADIOPUSHED=3 ; Radiobutton clicked
    MSG_CHECKBOX=4 ; Checkbox clicked
    MSG_EDITBOX=5 ; Editbox or Multilinebox
    MSG_FILESELECT=6 ; Filelistbox
    MSG_ITEMSELECT=7 ; Itembox
    MSG_COMBOCHANGE=8 ; Combobox/Droplistbox
    MSG_CALENDAR=9 ; Calendar date change
    MSG_SPINNER=10 ; Spinner number change
    MSG_CLOSEVIA49=11 ; Close clicked (Enabled via DialogProcOptions
1002
    MSG_FILEBOXDOUBLECLICK=12 ; Get double-click message on a
FileListBox
    MSG_ITEMBOXDOUBLECLICK=13 ; Get double-click message on an ItemBox
    MSG_COMEVENT=14 ; COMCONTROL Event notification from DialogObject
(NOT DialogProcOptions)
    MSG_MENUITEM=15 ; MenuItem selected
    MSG_MENUITEMINIT=16 ; MenuItem initialized
    MSG_RESIZE=17 ; Dialog resized

```

Introduction to Programming

```
DPO_DISABLESTATE=1000 ; codes -1=GetSetting 0=EnableDialog
1=DisableDialog

DPO_CHANGEBACKGROUND=1001 ; -1=Get Current otherwise bitmap or color
string

DPO_CHANGESYSMENU=1002 ; -1=Get Current 0=none 1=close 2=close/min
3=close/max 4=close/min/max

DPO_CHANGETITLE=1003 ; Set/Get Dialog Title - (-1 to get)

DPO_GETNAME=1004 ; Returns the name associated with a control's
number.

DPO_GETNUMBER=1005 ; Returns the number associated with a control's
name.

DPO_GETCLIENTAREA=1007 ; Returns a space delimited list of the width
and height of the client area.


;DialogControlState Constants

DCSTATE_SETFOCUS=1 ; Give Control Focus
DCSTATE_QUERYSTYLE=2 ; Query control's style
DCSTATE_ADDSTYLE=3 ; Add control style
DCSTATE_REMOVESTYLE=4 ; Remove control style
DCSTATE_GETFOCUS=5 ; Get control that has focus
DCSTATE_MOVEMOUSEOVER=6 ; Move the mouse over the control


DCSTYLE_DEFAULT=0 ; Set Default Style
DCSTYLE_INVISIBLE=1 ; Set Control Invisible
DCSTYLE_DISABLED=2 ; Set Control Disabled
DCSTYLE_NOUSERDATA=4 ; Note: Setable via DialogControlState function
ONLY SPINNER control only
DCSTYLE_READONLY=8 ; Sets control to read-only (user cannot type in
data) EDITBOX MULTILINEBOX SPINNER
DCSTYLE_PASSWORD=16 ; Sets 'password mode' where only *'s are
displayed EDITBOX
DCSTYLE_DEFAULTBUTTON=32 ; Sets a button as the default button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_DIGITONLY=64 ; Set edit box to accept digits only EDITMOX
MULTILINEBOX
DCSTYLE_FLAT=128 ; Makes a 'flat' hyperlink-looking button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_NOADJUST=256 ; Turns off auto-height adjustment ITEMBOX
FILELISTBOX
DCSTYLE_TEXTCENTER=512 ; Center text in control VARYTEXT STATICTEXT
```

```

DCSTYLE_TEXTRIGHT=1024 ; Flush-Right text in control VARYTEXT
STATICTEXT

DCSTYLE_NOSELCURLEFT=2048 ; No selection, cursor left EDITBOX
MULTILINEBOX

DCSTYLE_NOSELCURRIGHT=4096 ; No selection, cursor right EDITBOX
MULTILINEBOX

DCSTYLE_SHIELD=8192 ; Display Security Shield icon on button
(Vista/7 and newer) PUSHBUTTON PICTUREBUTTON

DCSTYLE_MENUCHECK=32768 ; Adds a check mark to the left of a menu
item MENUITEM

DCSTYLE_MENURADIO=65536 ; Adds a radio button like dot graphic to
the left of a menu item MENUITEM

DCSTYLE_MENUSEP=131072 ; Separator bar graphic MENUITEM

DCSTYLE_MENUBREAK=262144 ; column break MENUBAR


;DialogControlSet / DialogControlGet Constants
DC_CHECKBOX=1 ; CHECKBOX
DC_RADIOBUTTON=2 ; RADIOBUTTON
DC_EDITBOX=3 ; EDITBOX MULTILINEBOX
DC_TITLE=4 ; PICTURE RADIOBUTTON CHECKBOX PICTUREBUTTON VARYTEXT
STATICTEXT GROUPBOX PUSHBUTTON MENUITEM
DC_ITEMBOXCONTENTS=5 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXSELECT=6 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_CALENDAR=7 ; CALENDAR
DC_SPINNER=8 ; SPINNER
DC_MULTITABSTOPS=9 ; MULTILINEBOX
DC_ITEMSCROLLPOS=10 ; ITEMBOX FILELISTBOX
DC_BACKGROUNDCOLOR=11 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT
GROUPBOX PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX
MULTILINEBOX
DC_PICTUREBITMAP=12 ; PICTURE PICTUREBUTTON
DC_TEXTCOLOR=13 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT GROUPBOX
PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX MULTILINEBOX
DC_ITEMBOXADD=14 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXREMOVE=15 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_RADIOVALUE=16 ; RADIOBUTTON
DC_POSITION=17 ; ALL CONTROLS (Except MENUBAR and MENUITEM)
DC_MENUNAMES=18 ; ALL CONTROLS
DC_HANDLE=19 ; ALL CONTROLS (Except MENUBAR and MENUITEM)

```

Introduction to Programming

```
    ;DialogObject constants

    DLGOBJECT_ADDEVENT=1 ; Call dialog callback when the specified event
occurs

    DLGOBJECT_STOPEVENT=2 ; Stop calling dialog callback when an event
previously requested with

    DLGOBJECT_GETOBJECT=3 ; Return an object references to the specified
control

    DLGOBJECT_GETPICTURE=4 ; Create and return an object reference to a
picture object


    ;Return code constants

    RET_DO_CANCEL=0 ; Cancels dialog
    RET_DO_DEFAULT= -1 ; Continue with default processing for control
    RET_DO_NOT_EXIT= -2 ; Do not exit the dialog

    return
#EndSubroutine

;=====
;=====
;=====

#DefineFunction
EXACallbackProc (EXA_Handle, EXA_Message, EXA_Name, EXA_EventInfo, EXA_ChangeInfo)

    InitDialogConstants() ; Initialize Dialog Constants
    ON_EQUAL = @TRUE ; Initialize variable ON_EQUAL
    Switch EXA_Message ; Switch based on Dialog Message type
        case MSG_INIT ; Standard Initialization message
;            DialogProcOptions (EXA_Handle, MSG_TIMER, 1000)
;            DialogProcOptions (EXA_Handle, MSG_BUTTONPUSHED, @TRUE)
;            DialogProcOptions (EXA_Handle, MSG_CHECKBOX, @TRUE)
            return (RET_DO_DEFAULT)

;        case MSG_BUTTONPUSHED ; ID "PushButton_OK" OK
;            return (RET_DO_DEFAULT)

;        case MSG_CHECKBOX ; ID "CheckBox_1" MyCheckBox Click Me
;            return (RET_DO_DEFAULT)

    Endswitch ; EXA_Message
```



```

    return(RET_DO_DEFAULT)
#EndFunction ; End of Dialog Callback EXACallbackProc

;=====
;=====
;=====

" <X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---
<X>--^!---<X>--^!---<X>--^!---"

" <X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---
<X>--^!---<X>--^!---<X>--^!---"

" <X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---
<X>--^!---<X>--^!---<X>--^!---"

"REMEMBER UPDATE EXAProcedure VARIABLE AS BELOW AND DELETE THESE LINES"
EXAProcedure=`EXACallbackProc`

" <X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---
<X>--^!---<X>--^!---<X>--^!---"

" <X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---
<X>--^!---<X>--^!---<X>--^!---"

" <X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---<X>--^!---
<X>--^!---<X>--^!---<X>--^!---"

EXAFormat=`WWWDLGED,6.2`

EXACaption=`Example A`
EXAX=9999 ; -01
EXAY=9999 ; -01
EXAWidth=060
EXAHeight=045
EXANumControls=002
EXAProcedure=`DEFAULT`
EXAFont=`Microsoft Sans Serif|7373|70|34`
EXATextColor=`0|0|128`
EXABackground=`DEFAULT,128|255|255`
EXAConfig=0

EXA001=`005,005,046,010,CHECKBOX,"CheckBox_1",MyCheckBox,"Click
Me",1,1,DEFAULT,DEFAULT,DEFAULT,DEFAULT`

```

Introduction to Programming

```
EXA002=`009,023,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,2,DEF  
AULT,DEFAULT,DEFAULT,DEFAULT`
```

```
ButtonPushed=Dialog("EXA")
```

```
exit
```

Exercise B.wbt

```
*****  
**  
** [Chapter 14]  
** Exercise B.wbt  
** Sample Dynamic Dialog - Exercise B.  
**  
*****  
  
;=====
```

```
;  
;=====
```

```
;  
;=====
```

```
#DefineSubRoutine InitDialogConstants()  
    ;DialogprocOptions Constants  
    MSG_INIT=0 ; The one-time initialization  
    MSG_TIMER=1 ; Timer event  
    MSG_BUTTONPUSHED=2 ; Pushbutton or Picturebutton  
    MSG_RADIOPUSHED=3 ; Radiobutton clicked  
    MSG_CHECKBOX=4 ; Checkbox clicked  
    MSG_EDITBOX=5 ; Editbox or Multilinebox  
    MSG_FILESELECT=6 ; Filelistbox  
    MSG_ITEMSELECT=7 ; Itembox  
    MSG_COMBOCHANGE=8 ; Combobox/Droplistbox  
    MSG_CALENDAR=9 ; Calendar date change  
    MSG_SPINNER=10 ; Spinner number change  
    MSG_CLOSEVIA49=11 ; Close clicked (Enabled via DialogProcOptions  
1002  
    MSG_FILEBOXDOUBLECLICK=12 ; Get double-click message on a  
FileListBox  
    MSG_ITEMBOXDOUBLECLICK=13 ; Get double-click message on an ItemBox  
    MSG_COMEVENT=14 ; COMCONTROL Event notification from DialogObject  
(NOT DialogProcOptions)  
    MSG_MENUITEM=15 ; MenuItem selected
```

```

MSG_MENUITEMINIT=16 ; MenuItem initialized
MSG_RESIZE=17 ; Dialog resized

DPO_DISABLESTATE=1000 ; codes -1=GetSetting 0=EnableDialog
1=DisableDialog

DPO_CHANGEBACKGROUND=1001 ; -1=Get Current otherwise bitmap or color
string

DPO_CHANGESYSMENU=1002 ; -1=Get Current 0=none 1=close 2=close/min
3=close/max 4=close/min/max

DPO_CHANGETITLE=1003 ; Set/Get Dialog Title - (-1 to get)

DPO_GETNAME=1004 ; Returns the name associated with a control's
number.

DPO_GETNUMBER=1005 ; Returns the number associated with a control's
name.

DPO_GETCLIENTAREA=1007 ; Returns a space delimited list of the width
and height of the client area.

;DialogControlState Constants
DCSTATE_SETFOCUS=1 ; Give Control Focus
DCSTATE_QUERYSTYLE=2 ; Query control's style
DCSTATE_ADDSTYLE=3 ; Add control style
DCSTATE_REMOVESTYLE=4 ; Remove control style
DCSTATE_GETFOCUS=5 ; Get control that has focus
DCSTATE_MOVEMOUSEOVER=6 ; Move the mouse over the control

DCSTYLE_DEFAULT=0 ; Set Default Style
DCSTYLE_INVISIBLE=1 ; Set Control Invisible
DCSTYLE_DISABLED=2 ; Set Control Disabled
DCSTYLE_NOUSERDATA=4 ; Note: Setable via DialogControlState function
ONLY SPINNER control only
DCSTYLE_READONLY=8 ; Sets control to read-only (user cannot type in
data) EDITBOX MULTILINEBOX SPINNER
DCSTYLE_PASSWORD=16 ; Sets 'password mode' where only *'s are
displayed EDITBOX
DCSTYLE_DEFAULTBUTTON=32 ; Sets a button as the default button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_DIGITONLY=64 ; Set edit box to accept digits only EDITMOX
MULTILINEBOX
DCSTYLE_FLAT=128 ; Makes a 'flat' hyperlink-looking button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_NOADJUST=256 ; Turns off auto-height adjustment ITEMBOX
FILELISTBOX

```

Introduction to Programming

```
DCSTYLE_TEXTCENTER=512 ; Center text in control VARYTEXT STATICTEXT
DCSTYLE_TEXTRIGHT=1024 ; Flush-Right text in control VARYTEXT
STATICTEXT
DCSTYLE_NOSELCLURLEFT=2048 ; No selection, cursor left EDITBOX
MULTILINEBOX
DCSTYLE_NOSELCLURRIGHT=4096 ; No selection, cursor right EDITBOX
MULTILINEBOX
DCSTYLE_SHIELD=8192 ; Display Security Shield icon on button
(Vista/7 and newer) PUSHBUTTON PICTUREBUTTON
DCSTYLE_MENUCHECK=32768 ; Adds a check mark to the left of a menu
item MENUITEM
DCSTYLE_MENURADIO=65536 ; Adds a radio button like dot graphic to
the left of a menu item MENUITEM
DCSTYLE_MENUSEP=131072 ; Separator bar graphic MENUITEM
DCSTYLE_MENUBREAK=262144 ; column break MENUBAR

;DialogControlSet / DialogControlGet Constants
DC_CHECKBOX=1 ; CHECKBOX
DC_RADIOBUTTON=2 ; RADIOBUTTON
DC_EDITBOX=3 ; EDITBOX MULTILINEBOX
DC_TITLE=4 ; PICTURE RADIOBUTTON CHECKBOX PICTUREBUTTON VARYTEXT
STATICTEXT GROUPBOX PUSHBUTTON MENUITEM
DC_ITEMBOXCONTENTS=5 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXSELECT=6 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_CALENDAR=7 ; CALENDAR
DC_SPINNER=8 ; SPINNER
DC_MULTITABSTOPS=9 ; MULTILINEBOX
DC_ITEMSCROLLPOS=10 ; ITEMBOX FILELISTBOX
DC_BACKGROUNDOLOR=11 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT
GROUPBOX PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX
MULTILINEBOX
DC_PICTUREBITMAP=12 ; PICTURE PICTUREBUTTON
DC_TEXTCOLOR=13 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT GROUPBOX
PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX MULTILINEBOX
DC_ITEMBOXADD=14 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXREMOVE=15 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_RADIOVALUE=16 ; RADIOBUTTON
DC_POSITION=17 ; ALL CONTROLS (Except MENUBAR and MENUITEM)
DC_MENUNAMES=18 ; ALL CONTROLS
DC_HANDLE=19 ; ALL CONTROLS (Except MENUBAR and MENUITEM)
```

```

;DialogObject constants
DLGOBJECT_ADDEVENT=1 ; Call dialog callback when the specified event
occurs
DLGOBJECT_STOPEVENT=2 ; Stop calling dialog callback when an event
previously requested with
DLGOBJECT_GETOBJECT=3 ; Return an object references to the specified
control
DLGOBJECT_GETPICTURE=4 ; Create and return an object reference to a
picture object

;Return code constants
RET_DO_CANCEL=0 ; Cancels dialog
RET_DO_DEFAULT= -1 ; Continue with default processing for control
RET_DO_NOT_EXIT= -2 ; Do not exit the dialog
return
#EndSubroutine

;=====
;=====
;=====

#DefineFunction
EXBCallbackProc (EXB_Handle, EXB_Message, EXB_Name, EXB_EventInfo,
EXB_ChangeInfo)
    InitDialogConstants() ; Initialize Dialog Constants
    ON_EQUAL = @TRUE ; Initialize variable ON_EQUAL
    Switch EXB_Message ; Switch based on Dialog Message type
        Case MSG_INIT ; Standard Initialization message
;            DialogProcOptions (EXB_Handle, MSG_TIMER, 1000)
;            DialogProcOptions (EXB_Handle, MSG_BUTTONPUSHED, @TRUE)
;            DialogProcOptions (EXB_Handle, MSG_CHECKBOX, @TRUE)
            Return (RET_DO_DEFAULT)

;        case MSG_BUTTONPUSHED ; ID "PushButton_OK" PushButton_OK
;            return (RET_DO_DEFAULT)

;        case MSG_CHECKBOX ; ID "CheckBox_1" CheckBox_1 MyCheckBox
;            return (RET_DO_DEFAULT)

    EndSwitch ; EXB_Message

```

Introduction to Programming

```
    Return (RET_DO_DEFAULT)
#EndFunction ; End of Dialog Callback EXBCallbackProc

;=====
;=====
;=====

EXBFormat=`WWIDLGED,6.2`

EXBCaption=`Example B`
EXBX=9999 ; -01
EXBY=9999 ; -01
EXBWidth=060
EXBHeight=045
EXBNumControls=002
EXBProcedure=`EXBCallbackProc`
EXBFont=`Microsoft Sans Serif|7373|70|34`
EXBTextColor=`0|0|128`
EXBBackground=`DEFAULT,128|255|255`
EXBConfig=0

EXB001=`005,005,046,010,CHECKBOX,"CheckBox_1",MyCheckBox,"Click
Me",1,1,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EXB002=`009,023,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,2,DEF
AULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed=Dialog("EXB")

Message("CheckBox Value on Dialog Exit is", MyCheckbox)
exit
```

Exercise C.wbt

```
;*****
;**
;** [Chapter 14]
;** Exercise C.wbt
;** Sample Dynamic Dialog - Exercise C.
```

```

; **
; *****

;=====
;=====
;=====
#DefineSubRoutine InitDialogConstants()
    ;DialogprocOptions Constants
    MSG_INIT=0 ; The one-time initialization
    MSG_TIMER=1 ; Timer event
    MSG_BUTTONPUSHED=2 ; Pushbutton or Picturebutton
    MSG_RADIOPUSHED=3 ; Radiobutton clicked
    MSG_CHECKBOX=4 ; Checkbox clicked
    MSG_EDITBOX=5 ; Editbox or Multilinebox
    MSG_FILESELECT=6 ; Filelistbox
    MSG_ITEMSELECT=7 ; Itembox
    MSG_COMBOCHANGE=8 ; Combobox/Droplistbox
    MSG_CALENDAR=9 ; Calendar date change
    MSG_SPINNER=10 ; Spinner number change
    MSG_CLOSEVIA49=11 ; Close clicked (Enabled via DialogProcOptions
1002
    MSG_FILEBOXDOUBLECLICK=12 ; Get double-click message on a
FileListBox
    MSG_ITEMBOXDOUBLECLICK=13 ; Get double-click message on an ItemBox
    MSG_COMEVENT=14 ; COMCONTROL Event notification from DialogObject
(NOT DialogProcOptions)
    MSG_MENUITEM=15 ; MenuItem selected
    MSG_MENUITEMINIT=16 ; MenuItem initialized
    MSG_RESIZE=17 ; Dialog resized

    DPO_DISABLESTATE=1000 ; codes -1=GetSetting 0=EnableDialog
1=DisableDialog
    DPO_CHANGEBACKGROUND=1001 ; -1=Get Current otherwise bitmap or color
string
    DPO_CHANGESYSMENU=1002 ; -1=Get Current 0=none 1=close 2=close/min
3=close/max 4=close/min/max
    DPO_CHANGETITLE=1003 ; Set/Get Dialog Title - (-1 to get)
    DPO_GETNAME=1004 ; Returns the name associated with a control's
number.

```

Introduction to Programming

DPO_GETNUMBER=1005 ; Returns the number associated with a control's name.

DPO_GETCLIENTAREA=1007 ; Returns a space delimited list of the width and height of the client area.

;DialogControlState Constants

DCSTATE_SETFOCUS=1 ; Give Control Focus

DCSTATE_QUERYSTYLE=2 ; Query control's style

DCSTATE_ADDSTYLE=3 ; Add control style

DCSTATE_REMOVESTYLE=4 ; Remove control style

DCSTATE_GETFOCUS=5 ; Get control that has focus

DCSTATE_MOVEMOUSEOVER=6 ; Move the mouse over the control

DCSTYLE_DEFAULT=0 ; Set Default Style

DCSTYLE_INVISIBLE=1 ; Set Control Invisible

DCSTYLE_DISABLED=2 ; Set Control Disabled

DCSTYLE_NOUSERDATA=4 ; Note: Setable via DialogControlState function ONLY SPINNER control only

DCSTYLE_READONLY=8 ; Sets control to read-only (user cannot type in data) EDITBOX MULTILINEBOX SPINNER

DCSTYLE_PASSWORD=16 ; Sets 'password mode' where only '*'s are displayed EDITBOX

DCSTYLE_DEFAULTBUTTON=32 ; Sets a button as the default button PUSHBUTTON PICTUREBUTTON

DCSTYLE_DIGITONLY=64 ; Set edit box to accept digits only EDITBOX MULTILINEBOX

DCSTYLE_FLAT=128 ; Makes a 'flat' hyperlink-looking button PUSHBUTTON PICTUREBUTTON

DCSTYLE_NOADJUST=256 ; Turns off auto-height adjustment ITEMBOX FILELISTBOX

DCSTYLE_TEXTCENTER=512 ; Center text in control VARYTEXT STATICTEXT

DCSTYLE_TEXTRIGHT=1024 ; Flush-Right text in control VARYTEXT STATICTEXT

DCSTYLE_NOSELCLURLEFT=2048 ; No selection, cursor left EDITBOX MULTILINEBOX

DCSTYLE_NOSELCURRIGHT=4096 ; No selection, cursor right EDITBOX MULTILINEBOX

DCSTYLE_SHIELD=8192 ; Display Security Shield icon on button (Vista/7 and newer) PUSHBUTTON PICTUREBUTTON

DCSTYLE_MENUCHECK=32768 ; Adds a check mark to the left of a menu item MENUITEM


```

DCSTYLE_MENURADIO=65536 ; Adds a radio button like dot graphic to
the left of a menu item MENUITEM

DCSTYLE_MENUSEP=131072 ; Separator bar graphic MENUITEM

DCSTYLE_MENUBREAK=262144 ; column break MENUBAR


;DialogControlSet / DialogControlGet Constants

DC_CHECKBOX=1 ; CHECKBOX

DC_RADIOBUTTON=2 ; RADIOBUTTON

DC_EDITBOX=3 ; EDITBOX MULTILINEBOX

DC_TITLE=4 ; PICTURE RADIOBUTTON CHECKBOX PICTUREBUTTON VARYTEXT
STATICTEXT GROUPBOX PUSHBUTTON MENUITEM

DC_ITEMBOXCONTENTS=5 ; ITEMBOX FILELISTBOX DROPLISTBOX

DC_ITEMBOXSELECT=6 ; ITEMBOX FILELISTBOX DROPLISTBOX

DC_CALENDAR=7 ; CALENDAR

DC_SPINNER=8 ; SPINNER

DC_MULTITABSTOPS=9 ; MULTILINEBOX

DC_ITEMSCROLLPOS=10 ; ITEMBOX FILELISTBOX

DC_BACKGROUNDCOLOR=11 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT
GROUPBOX PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX
MULTILINEBOX

DC_PICTUREBITMAP=12 ; PICTURE PICTUREBUTTON

DC_TEXTCOLOR=13 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT GROUPBOX
PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX MULTILINEBOX

DC_ITEMBOXADD=14 ; ITEMBOX FILELISTBOX DROPLISTBOX

DC_ITEMBOXREMOVE=15 ; ITEMBOX FILELISTBOX DROPLISTBOX

DC_RADIOVALUE=16 ; RADIOBUTTON

DC_POSITION=17 ; ALL CONTROLS (Except MENUBAR and MENUITEM)

DC_MENUNAMES=18 ; ALL CONTROLS

DC_HANDLE=19 ; ALL CONTROLS (Except MENUBAR and MENUITEM)


;DialogObject constants

DLGOBJECT_ADDEVENT=1 ; Call dialog callback when the specified event
occurs

DLGOBJECT_STOPEVENT=2 ; Stop calling dialog callback when an event
previously requested with

DLGOBJECT_GETOBJECT=3 ; Return an object references to the specified
control

DLGOBJECT_GETPICTURE=4 ; Create and return an object reference to a
picture object

```

Introduction to Programming

```
;Return code constants
RET_DO_CANCEL=0 ; Cancels dialog
RET_DO_DEFAULT=-1 ; Continue with default processing for control
RET_DO_NOT_EXIT=-2 ; Do not exit the dialog
return
#EndSubroutine

;=====
;=====
;=====

#DefineFunction
EXCCallbackProc (EXC_Handle, EXC_Message, EXC_Name, EXC_EventInfo, EXC_ChangeInfo)
    InitDialogConstants() ; Initialize Dialog Constants
    ON_EQUAL = @TRUE ; Initialize variable ON_EQUAL
    Switch EXC_Message ; Switch based on Dialog Message type
        Case MSG_INIT ; Standard Initialization message
;            DialogProcOptions (EXC_Handle, MSG_TIMER, 1000)
;            DialogProcOptions (EXC_Handle, MSG_BUTTONPUSHED, @TRUE)
            DialogProcOptions (EXC_Handle, MSG_CHECKBOX, @TRUE)
            Return (RET_DO_DEFAULT)

;        case MSG_BUTTONPUSHED ; ID "PushButton_OK" PushButton_OK
;            return (RET_DO_DEFAULT)

        Case MSG_CHECKBOX ; ID "CheckBox_1" CheckBox_1 MyCheckBox
            Return (RET_DO_DEFAULT)

    EndSwitch ; EXC_Message
    Return (RET_DO_DEFAULT)
#EndFunction ; End of Dialog Callback EXCCallbackProc

;=====
;=====
;=====

EXCFormat=`WWDLGED, 6.2`
```

```

EXCCaption=`Example C`
EXCX=9999 ; -01
EXCY=9999 ; -01
EXCWidth=060
EXCHeight=045
EXCNumControls=002
EXCProcedure=`EXCCallbackProc`
EXCFont=`Microsoft Sans Serif|7373|70|34`
EXCTextColor=`0|0|128`
EXCBackground=`DEFAULT,128|255|255`
EXCConfig=0

EXC001=`005,005,046,010,CHECKBOX,"CheckBox_1",MyCheckBox,"Click
Me",1,1,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EXC002=`009,023,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,2,DEF
AULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed=Dialog("EXC")

Message( "CheckBox Value on Dialog Exit is", MyCheckbox )
exit

```

Exercise D.wbt

```

;*****
;**
;** [Chapter 14]
;** Exercise D.wbt
;** Sample Dynamic Dialog - Exercise D.
;**
;*****

;=====
;=====
;=====

#DefineSubRoutine InitDialogConstants()
    ;DialogprocOptions Constants
    MSG_INIT=0 ; The one-time initialization

```

Introduction to Programming

```
MSG_TIMER=1 ; Timer event
MSG_BUTTONPUSHED=2 ; Pushbutton or Picturebutton
MSG_RADIOPUSHED=3 ; Radiobutton clicked
MSG_CHECKBOX=4 ; Checkbox clicked
MSG_EDITBOX=5 ; Editbox or Multilinebox
MSG_FILESELECT=6 ; Filelistbox
MSG_ITEMSELECT=7 ; Itembox
MSG_COMBOCHANGE=8 ; Combobox/Droplistbox
MSG_CALENDAR=9 ; Calendar date change
MSG_SPINNER=10 ; Spinner number change
MSG_CLOSEVIA49=11 ; Close clicked (Enabled via DialogProcOptions
1002
MSG_FILEBOXDOUBLECLICK=12 ; Get double-click message on a
FileListBox
MSG_ITEMBOXDOUBLECLICK=13 ; Get double-click message on an ItemBox
MSG_COMEVENT=14 ; COMCONTROL Event notification from DialogObject
(NOT DialogProcOptions)
MSG_MENUITEM=15 ; MenuItem selected
MSG_MENUITEMINIT=16 ; MenuItem initialized
MSG_RESIZE=17 ; Dialog resized

DPO_DISABLESTATE=1000 ; codes -1=GetSetting 0=EnableDialog
1=DisableDialog
DPO_CHANGEBACKGROUND=1001 ; -1=Get Current otherwise bitmap or color
string
DPO_CHANGESYSMENU=1002 ; -1=Get Current 0=none 1=close 2=close/min
3=close/max 4=close/min/max
DPO_CHANGETITLE=1003 ; Set/Get Dialog Title - (-1 to get)
DPO_GETNAME=1004 ; Returns the name associated with a control's
number.
DPO_GETNUMBER=1005 ; Returns the number associated with a control's
name.
DPO_GETCLIENTAREA=1007 ; Returns a space delimited list of the width
and height of the client area.

;DialogControlState Constants
DCSTATE_SETFOCUS=1 ; Give Control Focus
DCSTATE_QUERYSTYLE=2 ; Query control's style
DCSTATE_ADDSTYLE=3 ; Add control style
DCSTATE_REMOVESTYLE=4 ; Remove control style
```

```

DCSTATE_GETFOCUS=5 ; Get control that has focus
DCSTATE_MOVEMOUSEOVER=6 ; Move the mouse over the control

DCSTYLE_DEFAULT=0 ; Set Default Style
DCSTYLE_INVISIBLE=1 ; Set Control Invisible
DCSTYLE_DISABLED=2 ; Set Control Disabled
DCSTYLE_NOUSERDATA=4 ; Note: Setable via DialogControlState function
ONLY SPINNER control only
DCSTYLE_READONLY=8 ; Sets control to read-only (user cannot type in
data) EDITBOX MULTILINEBOX SPINNER
DCSTYLE_PASSWORD=16 ; Sets 'password mode' where only '*'s are
displayed EDITBOX
DCSTYLE_DEFAULTBUTTON=32 ; Sets a button as the default button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_DIGITONLY=64 ; Set edit box to accept digits only EDITBOX
MULTILINEBOX
DCSTYLE_FLAT=128 ; Makes a 'flat' hyperlink-looking button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_NOADJUST=256 ; Turns off auto-height adjustment ITEMBOX
FILELISTBOX
DCSTYLE_TEXTCENTER=512 ; Center text in control VARYTEXT STATICTEXT
DCSTYLE_TEXTRIGHT=1024 ; Flush-Right text in control VARYTEXT
STATICTEXT
DCSTYLE_NOSELCLURLEFT=2048 ; No selection, cursor left EDITBOX
MULTILINEBOX
DCSTYLE_NOSELCLURRIGHT=4096 ; No selection, cursor right EDITBOX
MULTILINEBOX
DCSTYLE_SHIELD=8192 ; Display Security Shield icon on button
(Vista/7 and newer) PUSHBUTTON PICTUREBUTTON
DCSTYLE_MENUCHECK=32768 ; Adds a check mark to the left of a menu
item MENUITEM
DCSTYLE_MENURADIO=65536 ; Adds a radio button like dot graphic to
the left of a menu item MENUITEM
DCSTYLE_MENUSEP=131072 ; Separator bar graphic MENUITEM
DCSTYLE_MENUBREAK=262144 ; column break MENUBAR

;DialogControlSet / DialogControlGet Constants
DC_CHECKBOX=1 ; CHECKBOX
DC_RADIOBUTTON=2 ; RADIOBUTTON
DC_EDITBOX=3 ; EDITBOX MULTILINEBOX
DC_TITLE=4 ; PICTURE RADIOBUTTON CHECKBOX PICTUREBUTTON VARYTEXT
STATICTEXT GROUPBOX PUSHBUTTON MENUITEM

```

Introduction to Programming

```
DC_ITEMBOXCONTENTS=5 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXSELECT=6 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_CALENDAR=7 ; CALENDAR
DC_SPINNER=8 ; SPINNER
DC_MULTITABSTOPS=9 ; MULTILINEBOX
DC_ITEMSCROLLPOS=10 ; ITEMBOX FILELISTBOX
DC_BACKGROUNDCOLOR=11 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT
GROUPBOX PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX
MULTILINEBOX
DC_PICTUREBITMAP=12 ; PICTURE PICTUREBUTTON
DC_TEXTCOLOR=13 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT GROUPBOX
PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX MULTILINEBOX
DC_ITEMBOXADD=14 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXREMOVE=15 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_RADIOVALUE=16 ; RADIOBUTTON
DC_POSITION=17 ; ALL CONTROLS (Except MENUBAR and MENUITEM)
DC_MENUNAMES=18 ; ALL CONTROLS
DC_HANDLE=19 ; ALL CONTROLS (Except MENUBAR and MENUITEM)

;DialogObject constants
DLGOBJECT_ADDEVENT=1 ; Call dialog callback when the specified event
occurs
DLGOBJECT_STOPEVENT=2 ; Stop calling dialog callback when an event
previously requested with
DLGOBJECT_GETOBJECT=3 ; Return an object references to the specified
control
DLGOBJECT_GETPICTURE=4 ; Create and return an object reference to a
picture object

;Return code constants
RET_DO_CANCEL=0 ; Cancels dialog
RET_DO_DEFAULT= -1 ; Continue with default processing for control
RET_DO_NOT_EXIT= -2 ; Do not exit the dialog
return
#EndSubroutine
;=====
;=====
;=====
```

```

#DefineFunction
EXDCallbackProc (EXD_Handle, EXD_Message, EXD_Name, EXD_EventInfo, EXD_ChangeInfo)

    InitDialogConstants() ; Initialize Dialog Constants
    ON_EQUAL = @TRUE ; Initialize variable ON_EQUAL
    Switch EXD_Message ; Switch based on Dialog Message type
        Case MSG_INIT ; Standard Initialization message
;           DialogProcOptions (EXD_Handle, MSG_TIMER, 1000)
;           DialogProcOptions (EXD_Handle, MSG_BUTTONPUSHED, @TRUE)
            DialogProcOptions (EXD_Handle, MSG_CHECKBOX, @TRUE)
            Return (RET_DO_DEFAULT)

;       case MSG_BUTTONPUSHED ; ID "PushButton_OK" PushButton_OK
;           return (RET_DO_DEFAULT)

        Case MSG_CHECKBOX ; ID "CheckBox_1" CheckBox_1 MyCheckBox
            flag=AskYesNo ("Example D", "Do you really want to change the
value of this checkbox?")
            If flag==@NO
                ;Set it back to what it was
                cbval=DialogControlGet (EXD_Handle, "CheckBox_1", DC_CHECKBOX)
; get new state
                DialogControlSet (EXD_Handle, "CheckBox_1", DC_CHECKBOX, !cbval)
; put back opposite state
            EndIf
            Return (RET_DO_DEFAULT)

        EndSwitch ; EXD_Message
    Return (RET_DO_DEFAULT)
#EndFunction ; End of Dialog Callback EXDCallbackProc

;=====
;=====
;=====

EXDFormat=`WWWDLGED, 6.2`

EXDCaption=`Example D`
EXDX=9999 ; -01

```

Introduction to Programming

```
EXDY=9999 ; -01
EXDWidth=060
EXDHeight=045
EXDNumControls=002
EXDProcedure=`EXDCallbackProc`
EXDFont=`Microsoft Sans Serif|7373|70|34`
EXDTextColor=`0|0|128`
EXDBackground=`DEFAULT,128|255|255`
EXDConfig=0

EXD001=`005,005,046,010,CHECKBOX,"CheckBox_1",MyCheckBox,"Click
Me",1,1,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EXD002=`009,023,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,2,DEF
AULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed=Dialog("EXD")

Message("CheckBox Value on Dialog Exit is",MyCheckbox)
exit
```

Exercise E.wbt

```
;*****
; **
; ** [Chapter 14]
; ** Exercise E.wbt
; ** Sample Dynamic Dialog - Exercise E.
; **
;*****

;=====
;=====
;=====

#DefineSubRoutine InitDialogConstants()
    ;DialogprocOptions Constants
    MSG_INIT=0 ; The one-time initialization
    MSG_TIMER=1 ; Timer event
    MSG_BUTTONPUSHED=2 ; Pushbutton or Picturebutton
    MSG_RADIOPUSHED=3 ; Radiobutton clicked
```



```

MSG_CHECKBOX=4 ; Checkbox clicked
MSG_EDITBOX=5 ; Editbox or Multilinebox
MSG_FILESELECT=6 ; Filelistbox
MSG_ITEMSELECT=7 ; Itembox
MSG_COMBOCHANGE=8 ; Combobox/Droplistbox
MSG_CALENDAR=9 ; Calendar date change
MSG_SPINNER=10 ; Spinner number change
MSG_CLOSEVIA49=11 ; Close clicked (Enabled via DialogProcOptions
1002
MSG_FILEBOXDOUBLECLICK=12 ; Get double-click message on a
FileListBox
MSG_ITEMBOXDOUBLECLICK=13 ; Get double-click message on an ItemBox
MSG_COMEVENT=14 ; COMCONTROL Event notification from DialogObject
(NOT DialogProcOptions)
MSG_MENUITEM=15 ; MenuItem selected
MSG_MENUITEMINIT=16 ; MenuItem initialized
MSG_RESIZE=17 ; Dialog resized

DPO_DISABLESTATE=1000 ; codes -1=GetSetting 0=EnableDialog
1=DisableDialog
DPO_CHANGEBACKGROUND=1001 ; -1=Get Current otherwise bitmap or color
string
DPO_CHANGESYSMENU=1002 ; -1=Get Current 0=none 1=close 2=close/min
3=close/max 4=close/min/max
DPO_CHANGETITLE=1003 ; Set/Get Dialog Title - (-1 to get)
DPO_GETNAME=1004 ; Returns the name associated with a control's
number.
DPO_GETNUMBER=1005 ; Returns the number associated with a control's
name.
DPO_GETCLIENTAREA=1007 ; Returns a space delimited list of the width
and height of the client area.

;DialogControlState Constants
DCSTATE_SETFOCUS=1 ; Give Control Focus
DCSTATE_QUERYSTYLE=2 ; Query control's style
DCSTATE_ADDSTYLE=3 ; Add control style
DCSTATE_REMOVESTYLE=4 ; Remove control style
DCSTATE_GETFOCUS=5 ; Get control that has focus
DCSTATE_MOVEMOUSEOVER=6 ; Move the mouse over the control

```

Introduction to Programming

```
DCSTYLE_DEFAULT=0 ; Set Default Style
DCSTYLE_INVISIBLE=1 ; Set Control Invisible
DCSTYLE_DISABLED=2 ; Set Control Disabled
DCSTYLE_NOUSERDATA=4 ; Note: Setable via DialogControlState function
ONLY SPINNER control only
DCSTYLE_READONLY=8 ; Sets control to read-only (user cannot type in
data) EDITBOX MULTILINEBOX SPINNER
DCSTYLE_PASSWORD=16 ; Sets 'password mode' where only '*'s are
displayed EDITBOX
DCSTYLE_DEFAULTBUTTON=32 ; Sets a button as the default button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_DIGITONLY=64 ; Set edit box to accept digits only EDITBOX
MULTILINEBOX
DCSTYLE_FLAT=128 ; Makes a 'flat' hyperlink-looking button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_NOADJUST=256 ; Turns off auto-height adjustment ITEMBOX
FILELISTBOX
DCSTYLE_TEXTCENTER=512 ; Center text in control VARYTEXT STATICTEXT
DCSTYLE_TEXTRIGHT=1024 ; Flush-Right text in control VARYTEXT
STATICTEXT
DCSTYLE_NOSELCURLEFT=2048 ; No selection, cursor left EDITBOX
MULTILINEBOX
DCSTYLE_NOSELCURRIGHT=4096 ; No selection, cursor right EDITBOX
MULTILINEBOX
DCSTYLE_SHIELD=8192 ; Display Security Shield icon on button
(Vista/7 and newer) PUSHBUTTON PICTUREBUTTON
DCSTYLE_MENUCHECK=32768 ; Adds a check mark to the left of a menu
item MENUITEM
DCSTYLE_MENURADIO=65536 ; Adds a radio button like dot graphic to
the left of a menu item MENUITEM
DCSTYLE_MENUSEP=131072 ; Separator bar graphic MENUITEM
DCSTYLE_MENUBREAK=262144 ; column break MENUBAR

;DialogControlSet / DialogControlGet Constants
DC_CHECKBOX=1 ; CHECKBOX
DC_RADIOBUTTON=2 ; RADIOBUTTON
DC_EDITBOX=3 ; EDITBOX MULTILINEBOX
DC_TITLE=4 ; PICTURE RADIOBUTTON CHECKBOX PICTUREBUTTON VARYTEXT
STATICTEXT GROUPBOX PUSHBUTTON MENUITEM
DC_ITEMBOXCONTENTS=5 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXSELECT=6 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_CALENDAR=7 ; CALENDAR
```

```

DC_SPINNER=8 ; SPINNER
DC_MULTITABSTOPS=9 ; MULTILINEBOX
DC_ITEMSCROLLPOS=10 ; ITEMBOX FILELISTBOX
DC_BACKGROUNDCOLOR=11 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT
GROUPBOX PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX
MULTILINEBOX
DC_PICTUREBITMAP=12 ; PICTURE PICTUREBUTTON
DC_TEXTCOLOR=13 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT GROUPBOX
PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX MULTILINEBOX
DC_ITEMBOXADD=14 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXREMOVE=15 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_RADIOVALUE=16 ; RADIOBUTTON
DC_POSITION=17 ; ALL CONTROLS (Except MENUBAR and MENUITEM)
DC_MENUNAMES=18 ; ALL CONTROLS
DC_HANDLE=19 ; ALL CONTROLS (Except MENUBAR and MENUITEM)

;DialogObject constants
DLGOBJECT_ADDEVENT=1 ; Call dialog callback when the specified event
occurs
DLGOBJECT_STOPEVENT=2 ; Stop calling dialog callback when an event
previously requested with
DLGOBJECT_GETOBJECT=3 ; Return an object references to the specified
control
DLGOBJECT_GETPICTURE=4 ; Create and return an object reference to a
picture object

;Return code constants
RET_DO_CANCEL=0 ; Cancels dialog
RET_DO_DEFAULT=-1 ; Continue with default processing for control
RET_DO_NOT_EXIT=-2 ; Do not exit the dialog
return
#EndSubroutine

;=====
;=====
;=====

#DefineFunction
EXECallbackProc (EXE_Handle, EXE_Message, EXE_Name, EXE_EventInfo, EXE_ChangeInfo)
InitDialogConstants() ; Initialize Dialog Constants

```

Introduction to Programming

```
ON_EQUAL = @TRUE ; Initialize variable ON_EQUAL
Switch EXE_Message ; Switch based on Dialog Message type
    Case MSG_INIT ; Standard Initialization message
;        DialogProcOptions (EXE_Handle,MSG_TIMER,1000)
;        DialogProcOptions (EXE_Handle,MSG_BUTTONPUSHED,@TRUE)
        DialogProcOptions (EXE_Handle,MSG_CHECKBOX,@TRUE)
        Return (RET_DO_DEFAULT)

;    case MSG_BUTTONPUSHED ; ID "PushButton_OK" PushButton_OK
;        return (RET_DO_DEFAULT)

    Case MSG_CHECKBOX
        Switch ON_EQUAL
            Case EXE_Name == "CheckBox_1" ; ID "CheckBox_1" CheckBox_1
MyCheckBox
                flag=AskYesNo ("Example D","Do you really want to change
the value of this checkbox?")
                If flag==@NO
                    ;Set it back to what it was

cbval=DialogControlGet (EXE_Handle,"CheckBox_1",DC_CHECKBOX) ; get new
state

DialogControlSet (EXE_Handle,"CheckBox_1",DC_CHECKBOX,!cbval) ; put back
opposite state

                EndIf
                Return (RET_DO_DEFAULT)

            Case EXE_Name == "CheckBox_2" ; ID "CheckBox_2" CheckBox_2
MyOtherCheckBox
                Return (RET_DO_DEFAULT)

        EndSwitch ; EXE_Name
        Return (RET_DO_DEFAULT)

    EndSwitch ; EXE_Message
    Return (RET_DO_DEFAULT)
#EndFunction ; End of Dialog Callback EXECallbackProc

;=====
```

```

;=====
;=====

EXEFormat=`WWIDLGED,6.2`

EXECaption=`Example E`
EXEX=9999 ; -01
EXEY=9999 ; -01
EXEWidth=068
EXEHeight=057
EXENumControls=003
EXEProcedure=`EXECallbackProc`
EXEFont=`Microsoft Sans Serif|7373|70|34`
EXETextColor=`0|0|128`
EXEBackground=`DEFAULT,128|255|255`
EXEConfig=0

EXE001=`005,005,056,010,CHECKBOX,"CheckBox_1",MyCheckBox,"&Confirm
Me",1,1,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EXE002=`005,023,056,010,CHECKBOX,"CheckBox_2",MyOtherCheckBox,"&But not
me",1,2,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EXE003=`015,039,032,010,PUSHBUTTON,"PushButton_OK",DEFAULT,"&OK",1,3,DE
FAULT,DEFAULT,DEFAULT,DEFAULT`

ButtonPushed=Dialog("EXE")

Message("CheckBox Values
are",StrCat("EXE001=",MyCheckBox,@CRLF,"EXE002=",MyOtherCheckBox))
exit

```

Exercise F.wbt

```

;*****
;**
;** [Chapter 14]
;** Exercise F.wbt
;** Sample Dynamic Dialog - Exercise F (Extra Credit).
;**
;*****

```

Introduction to Programming

```
;=====
;=====
;=====
#DefineSubRoutine InitDialogConstants()
    ;DialogprocOptions Constants
    MSG_INIT=0 ; The one-time initialization
    MSG_TIMER=1 ; Timer event
    MSG_BUTTONPUSHED=2 ; Pushbutton or Picturebutton
    MSG_RADIOPUSHED=3 ; Radiobutton clicked
    MSG_CHECKBOX=4 ; Checkbox clicked
    MSG_EDITBOX=5 ; Editbox or Multilinebox
    MSG_FILESELECT=6 ; Filelistbox
    MSG_ITEMSELECT=7 ; Itembox
    MSG_COMBOCHANGE=8 ; Combobox/Droplistbox
    MSG_CALENDAR=9 ; Calendar date change
    MSG_SPINNER=10 ; Spinner number change
    MSG_CLOSEVIA49=11 ; Close clicked (Enabled via DialogProcOptions
1002
    MSG_FILEBOXDOUBLECLICK=12 ; Get double-click message on a
FileListBox
    MSG_ITEMBOXDOUBLECLICK=13 ; Get double-click message on an ItemBox
    MSG_COMEVENT=14 ; COMCONTROL Event notification from DialogObject
(NOT DialogProcOptions)
    MSG_MENUITEM=15 ; MenuItem selected
    MSG_MENUITEMINIT=16 ; MenuItem initialized
    MSG_RESIZE=17 ; Dialog resized

    DPO_DISABLESTATE=1000 ; codes -1=GetSetting 0=EnableDialog
1=DisableDialog
    DPO_CHANGEBACKGROUND=1001 ; -1=Get Current otherwise bitmap or color
string
    DPO_CHANGESYSMENU=1002 ; -1=Get Current 0=none 1=close 2=close/min
3=close/max 4=close/min/max
    DPO_CHANGETITLE=1003 ; Set/Get Dialog Title - (-1 to get)
    DPO_GETNAME=1004 ; Returns the name associated with a control's
number.
    DPO_GETNUMBER=1005 ; Returns the number associated with a control's
name.
    DPO_GETCLIENTAREA=1007 ; Returns a space delimited list of the width
and height of the client area.
```

```

;DialogControlState Constants
DCSTATE_SETFOCUS=1 ; Give Control Focus
DCSTATE_QUERYSTYLE=2 ; Query control's style
DCSTATE_ADDSTYLE=3 ; Add control style
DCSTATE_REMOVESTYLE=4 ; Remove control style
DCSTATE_GETFOCUS=5 ; Get control that has focus
DCSTATE_MOVEMOUSEOVER=6 ; Move the mouse over the control

DCSTYLE_DEFAULT=0 ; Set Default Style
DCSTYLE_INVISIBLE=1 ; Set Control Invisible
DCSTYLE_DISABLED=2 ; Set Control Disabled
DCSTYLE_NOUSERDATA=4 ; Note: Setable via DialogControlState function
ONLY SPINNER control only
DCSTYLE_READONLY=8 ; Sets control to read-only (user cannot type in
data) EDITBOX MULTILINEBOX SPINNER
DCSTYLE_PASSWORD=16 ; Sets 'password mode' where only '*'s are
displayed EDITBOX
DCSTYLE_DEFAULTBUTTON=32 ; Sets a button as the default button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_DIGITONLY=64 ; Set edit box to accept digits only EDITBOX
MULTILINEBOX
DCSTYLE_FLAT=128 ; Makes a 'flat' hyperlink-looking button
PUSHBUTTON PICTUREBUTTON
DCSTYLE_NOADJUST=256 ; Turns off auto-height adjustment ITEMBOX
FILELISTBOX
DCSTYLE_TEXTCENTER=512 ; Center text in control VARYTEXT STATICTEXT
DCSTYLE_TEXTRIGHT=1024 ; Flush-Right text in control VARYTEXT
STATICTEXT
DCSTYLE_NOSELCLLEFT=2048 ; No selection, cursor left EDITBOX
MULTILINEBOX
DCSTYLE_NOSELCLRIGHT=4096 ; No selection, cursor right EDITBOX
MULTILINEBOX
DCSTYLE_SHIELD=8192 ; Display Security Shield icon on button
(Vista/7 and newer) PUSHBUTTON PICTUREBUTTON
DCSTYLE_MENUCHECK=32768 ; Adds a check mark to the left of a menu
item MENUITEM
DCSTYLE_MENURADIO=65536 ; Adds a radio button like dot graphic to
the left of a menu item MENUITEM
DCSTYLE_MENUSEP=131072 ; Separator bar graphic MENUITEM
DCSTYLE_MENUBREAK=262144 ; column break MENUBAR

```

Introduction to Programming

```
;DialogControlSet / DialogControlGet Constants
DC_CHECKBOX=1 ; CHECKBOX
DC_RADIOBUTTON=2 ; RADIOBUTTON
DC_EDITBOX=3 ; EDITBOX MULTILINEBOX
DC_TITLE=4 ; PICTURE RADIOBUTTON CHECKBOX PICTUREBUTTON VARYTEXT
STATICTEXT GROUPBOX PUSHBUTTON MENUITEM
DC_ITEMBOXCONTENTS=5 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXSELECT=6 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_CALENDAR=7 ; CALENDAR
DC_SPINNER=8 ; SPINNER
DC_MULTITABSTOPS=9 ; MULTILINEBOX
DC_ITEMSCROLLPOS=10 ; ITEMBOX FILELISTBOX
DC_BACKGROUNDCOLOR=11 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT
GROUPBOX PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX
MULTILINEBOX
DC_PICTUREBITMAP=12 ; PICTURE PICTUREBUTTON
DC_TEXTCOLOR=13 ; RADIOBUTTON CHECKBOX VARYTEXT STATICTEXT GROUPBOX
PUSHBUTTON ITEMBOX FILELISTBOX DROPLISTBOX SPINNER EDITBOX MULTILINEBOX
DC_ITEMBOXADD=14 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_ITEMBOXREMOVE=15 ; ITEMBOX FILELISTBOX DROPLISTBOX
DC_RADIOVALUE=16 ; RADIOBUTTON
DC_POSITION=17 ; ALL CONTROLS (Except MENUBAR and MENUITEM)
DC_MENUNAMES=18 ; ALL CONTROLS
DC_HANDLE=19 ; ALL CONTROLS (Except MENUBAR and MENUITEM)

;DialogObject constants
DLGOBJECT_ADDEVENT=1 ; Call dialog callback when the specified event
occurs
DLGOBJECT_STOPEVENT=2 ; Stop calling dialog callback when an event
previously requested with
DLGOBJECT_GETOBJECT=3 ; Return an object references to the specified
control
DLGOBJECT_GETPICTURE=4 ; Create and return an object reference to a
picture object

;Return code constants
RET_DO_CANCEL=0 ; Cancels dialog
RET_DO_DEFAULT= -1 ; Continue with default processing for control
RET_DO_NOT_EXIT= -2 ; Do not exit the dialog
return
```



```

#EndSubroutine

;=====
;=====
;=====

#DefineFunction
ExtraCallbackProc(Extra_Handle,Extra_Message,Extra_Name,Extra_EventInfo
,EXF_ChangeInfo)

    InitDialogConstants() ; Initialize Dialog Constants
    ON_EQUAL = @TRUE ; Initialize variable ON_EQUAL
    Switch Extra_Message ; Switch based on Dialog Message type
        Case MSG_INIT ; Standard Initialization message
;           DialogProcOptions(Extra_Handle,MSG_TIMER,1000)
           DialogProcOptions(Extra_Handle,MSG_BUTTONPUSHED,@TRUE)
;           DialogProcOptions(Extra_Handle,MSG_EDITBOX,@TRUE)
           Return(RET_DO_DEFAULT)

        Case MSG_BUTTONPUSHED
            Switch ON_EQUAL
                Case Extra_Name == "PushButton_Browse" ; ID
"PushButton_Browse" PushButton_Browse
                    fname=AskFilename("Choose a FileName", "", "All
Files|*.*", "*.*", 1 )
                    DialogControlSet(Extra_Handle,"EditBox_1", DC_EDITBOX,
fname)

                    :CANCEL
                    Return(RET_DO_NOT_EXIT) ; don't exit

                Case Extra_Name == "PushButton_OK" ; ID "PushButton_OK"
PushButton_OK
                    Return(RET_DO_DEFAULT)

                Case Extra_Name == "PushButton_Cancel" ; ID
"PushButton_Cancel" PushButton_Cancel
                    Return(RET_DO_DEFAULT)

            EndSwitch ; Extra_Name
            Return(RET_DO_DEFAULT)

;       case MSG_EDITBOX ; ID "EditBox_1" EditBox_1 MyFileName

```

Introduction to Programming

```
;          return(RET_DO_DEFAULT)

    EndSwitch ; Extra_Message
    Return(RET_DO_DEFAULT)
#EndFunction ; End of Dialog Callback ExtraCallbackProc

;=====
;=====
;=====

ExtraFormat=`WWWDLGED,6.2`

ExtraCaption=`Extra Credit Assignment`
ExtraX=9999 ; -01
ExtraY=9999 ; -01
ExtraWidth=254
ExtraHeight=081
ExtraNumControls=005
ExtraProcedure=`ExtraCallbackProc`
ExtraFont=`DEFAULT`
ExtraTextColor=`DEFAULT`
ExtraBackground=`DEFAULT,255|128|0`
ExtraConfig=0

Extra001=`009,007,042,008,STATICTEXT,"StaticText_FileName:",DEFAULT,"File
leName:",DEFAULT,1,DEFAULT,"Microsoft Sans
Serif|7373|70|34","0|0|128",DEFAULT`
Extra002=`007,021,192,014,EDITBOX,"EditBox_1",MyFileName,DEFAULT,DEFAULT
T,2,DEFAULT,"Microsoft Sans Serif|7373|70|34","0|0|128","255|128|0`
Extra003=`209,021,034,014,PUSHBUTTON,"PushButton_Browse",DEFAULT,"Browse
e",2,3,DEFAULT,"Microsoft Sans
Serif|7373|70|34","0|0|128","255|128|64`
Extra004=`033,055,034,014,PUSHBUTTON,"PushButton_OK",DEFAULT,"OK",1,4,DE
EFAULT,"Microsoft Sans Serif|7373|70|34","0|0|128","255|128|64`
Extra005=`127,055,034,014,PUSHBUTTON,"PushButton_Cancel",DEFAULT,"Cancel
l",0,5,DEFAULT,"Microsoft Sans
Serif|7373|70|34","0|0|128","255|128|64`

ButtonPushed=Dialog("Extra")
```

```
Message("Selected Filename is",MyFileName)
exit
```

Chapter 15 Samples

Debug01.wbt

```
;*****
; **
; **  [Chapter 15]
; **  Debug01.wbt
; **  Debugging Sample 01.
; **
;*****

EditTestFormat=`WWDDLGED,6.2`

EditTestCaption=`Edit Test (Debug01.wbt)`
EditTestX=025
EditTestY=042
EditTestWidth=142
EditTestHeight=086
EditTestNumControls=006
EditTestProcedure=`DEFAULT`
EditTestFont=`DEFAULT`
EditTestTextColor=`DEFAULT`
EditTestBackground=`DEFAULT,DEFAULT`
EditTestConfig=0

EditTest001=`007,007,128,012,STATICTEXT,"StaticText_1",DEFAULT,"Enter a
string, integer or floating point value
here",DEFAULT,10,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
EditTest002=`007,019,128,012,EDITBOX,"EditBox_1",edText,DEFAULT,DEFAULT
,20,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

Introduction to Programming

```
EditTest003=`007,033,128,012,STATICTEXT,"StaticText_2",DEFAULT,"Enter a password in this field",DEFAULT,30,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
EditTest004=`007,047,128,012,EDITBOX,"EditBox_2",pw_Password,DEFAULT,DEFAULT,40,DEFAULT,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
EditTest005=`007,065,050,012,PUSHBUTTON,"PushButton_Test",DEFAULT,"Test",1,50,32,DEFAULT,DEFAULT,DEFAULT`
```

```
EditTest006=`083,065,050,012,PUSHBUTTON,"PushButton_Exit",DEFAULT,"Exit",0,60,DEFAULT,DEFAULT,DEFAULT,DEFAULT`
```

```
ButtonPushed = Dialog( "EditTest" )
```

```
Message( "Report",  
edtTest )
```

```
if pw_Password <> ""
```

```
    Message("Password", "The password entry is ":pw_Password )
```

```
endif
```

```
exit
```

Debug02.wbt

```
;*****
```

```
;**
```

```
;** [Chapter 15]
```

```
;** Debug02.wbt (VariTest.wbt)
```

```
;** Demonstrates variable assignments
```

```
;**
```

```
;*****
```

```
n = 2
```

```
m = 1.01
```

```
Message( "Result: step 1", "m = " : m : ", n = " : n )
```

```
n = n * m
```

```
Message( "Result: step 2", "m = " : m : ", n = " : n )
```

```
n = "now I'm a string"
```

```
Message( "Result: step 3", "m = " : m : ", n = " : n )
```

```
n = "2"
```

```
m = "2.02"
```

```

m * n
a = m * n
Message( "Result: step 4", "m = " : m : ", a = " : a )

n = "two"
m = "two point zero two"
a = m + n           ; we can confidently expect this step to fail
Message( "Result: step 5", "m = " : m : ", a = " : a )
exit

```

Parts List.lst

```

Webley Defaminizer      5    2456-3468-8921    27
Finagle Bolix Grinder 2    3905-1298-7892    12B
Acme Jetpack      3    9834-0909-8721    14
Hobart Skyhook 1    6435-2348-0971    8E
Forward Mass Detector 9    3498-3465-1871    5
Dyson Sphere      1    0000-0000-0001    M27-139-235-890
Niven Transporter (Pad model) 3    9872-2317-2345    17A, 18B, 21C

```

ExternCall.wbt

```

;*****
;**
;**  [Chapter 15]
;**  ExternCall.wbt
;**  Demonstrates calling external .WBT programs as subroutines
;**
;*****

nCount = 0
sList = ""
DirChange( DirScript() )
; open the source file and return an array of entries
Call( 'GetData.WBT', '"Parts List.lst" sList nCount' )
Message( "List count", "Found ":nCount:" items in [":sList:"]" )

Call( "SortData.WBT", "sList @TAB" )
Message( "Sort results", "Sorted ":nCount:" items as [":sList:"]" )

```

Introduction to Programming

Exit

GetData.wbt

```
;*****
; **
; ** [Chapter 15]
; ** GetData.wbt
; ** Creates a list from the contents of a file.
; ** Parameters:
; **      param1: file to be searched
; **      param2: name of var to return
; **      param3: name of var to return number of items in list
;*****

If param0 < 3 ; insufficient arguments
    Message("Attention","This script is not meant to use used alone.
    It is used by other scripts")
    exit
Endif

If IsNumber( param3 ) Then exit ; parameter 3 isn't a variable name
If IsNumber( param2 ) Then exit ; parameter 2 isn't a variable name
If IsNumber( param1 ) Then exit ; parameter 1 isn't a filename
If FileExist( param1 ) == 0
    %param2% = "File Error"
    %param3% = 0
    return
Endif

nIndex = 0 ; initialize a count index
sResult = ""

hFileIn = FileOpen( param1, "READ" ) ; open the input file and get a
handle
While @TRUE
    sTemp = ""
    sLineIn = FileRead( hFileIn )
    If( sLineIn == "*EOF*" ) Then break
```

```

    nIndex = nIndex + 1
    sTemp = ItemExtract( 1, sLineIn, @TAB )
    sResult = StrCat( sResult, sTemp, @TAB )
EndWhile

FileClose( hFileIn ) ; close the input file
%param2% = sResult ; assign the result string to param2
%param3% = nIndex ; assign the count to param3
Drop( sLineIn, nIndex, sResult, sTemp ) ; discard local variables
Return

```

SortData.wbt

```

;*****
; **
; ** [Chapter 15]
; ** SortData.wbt
; ** Demonstrates sorting a list using ItemSort
; **
;*****

If param0 < 2 Then exit
If IsNumber( param1 ) Then exit ; should be list of data
If IsNumber( param2 ) Then exit ; should be char (delimiter)

sList = ItemSort( %param1%, %param2% )
return

```

Debug03a.wbt

```

;*****
; **
; ** [Chapter 15]
; ** Debug03a.wbt
; ** Demonstrates a long execution script with error
; **
;*****

listPrimes = ""

```

Introduction to Programming

```
nCount = 0
For i = 1 to 1000000
    bPrime = @TRUE
    For j = 2 to Sqrt( i )
        if( i / j ) == ( ( i * 1.0 ) / ( j * 1.0 ) ) then bPrime = @FALSE
    Next
    If bPrime then
        nCount = nCount + 1
        listResult = StrCat( listPrimes, ", ", i )
    EndIf
Next
Message( "Found ":nCount:" primes", listPrimes )
Exit
```

Debug03b.wbt

```
;*****
; **
; ** [Chapter 15]
; ** Debug03b.wbt
; ** Demonstrates Debug Function on a long execution script with error
; **
;*****
```

```
Debug(@ON)
```

```
listPrimes = ""
nCount = 0
For i = 1 to 1000000
    bPrime = @TRUE
    For j = 2 to Sqrt( i )
        if( i / j ) == ( ( i * 1.0 ) / ( j * 1.0 ) ) then bPrime = @FALSE
    Next
    If bPrime then
        nCount = nCount + 1
        listResult = StrCat( listPrimes, ", ", i )
    EndIf
Next
```



```

Message( "Found ":nCount:" primes", listPrimes )
Exit

```

Debug03c.wbt

```

;*****
;**
;**  [Chapter 15]
;**  Debug03c.wbt
;**  Demonstrates DebugTrace on a long execution script with error
;**
;*****

listPrimes = ""
nCount = 0
For i = 1 to 1000000
    bPrime = @TRUE
    For j = 2 to Sqrt( i )
        if( i / j ) == ( ( i * 1.0 ) / ( j * 1.0 ) ) then bPrime = @FALSE
    Next
    If bPrime then
        DebugTrace( @ON, "TRACE.TXT" )
        nCount = nCount + 1
        listResult = StrCat( listPrimes, ", ", i )
        DebugTrace( 203, listprimes ) ; variable info as string
    Else
        DebugTrace( @OFF )
    EndIf
Next
Message( "Found ":nCount:" primes", listPrimes )
exit

```

Debug03d.wbt

```

;*****
;**
;**  [Chapter 15]
;**  Debug03d.wbt
;**  Demonstrates Pause on a long execution script with error
;**

```

Introduction to Programming

```
;*****
listPrimes = ""
nCount = 0
For i = 1 to 1000000
    bPrime = @TRUE
    For j = 2 to Sqrt( i )
        If( i / j ) == ( ( i * 1.0 ) / ( j * 1.0 ) ) then bPrime = @FALSE
    Next
    If ( i == 100 ) then Pause( "Checkpoint", "[ " : listprimes : " ]" )
    If bPrime then
        nCount = nCount + 1
        listResult = StrCat( listPrimes, ", ", i )
    EndIf
Next
Message( "Found ":nCount:" primes", listPrimes )
exit
```

Chapter 16 Samples

PlatformInfo.wbt

```
;*****
; **
; ** [Chapter 16]
; ** PlatformInfo.wbt
; ** Demonstrates WinVersion and WinMetrics functions
; **
;*****

v = WinVersion(5)

; Determine Platform Version
Platform="Unknown"
If v=="1-4-0" then Platform="Windows 95"
If v=="1-4-10" then Platform="Windows 98"
If v=="1-4-90" then Platform="Windows ME"
If v=="2-3-51" then Platform="Windows NT 3.51"
If v=="2-4-0" then Platform="Windows NT 4.0"
```

```

If v=="2-5-0" then Platform="Windows 2000"
If v=="2-5-1" then Platform="Windows XP"
If v=="2-5-2" then Platform="Windows 2003 Server"
If v=="2-6-0" then Platform="Windows Vista"
If v=="2-6-1" then Platform="Windows 7"

; Determine Bitness
bit = WinMetrics( -7 )
If bit==2 then bitness="64 bit"
Else bitness="32 bit"

; Determine Service Pack
csdver = WinVersion(3)

Message("Windows Version", platform : " " : bitness : " " : csdver )
exit

```

NetTest.wbt

```

;*****
;**
;** [Chapter 16]
;** NetTest.wbt
;** Determines Windows network configuration
;**
;*****

sNetwork = NetInfo(0)
sNetClient = NetInfo(1)
nCount = ItemCount( sNetClient, @TAB)
Message(sNetwork:" supporting ":nCount:" networks(s)",sNetClient)

AddExtender( "WWWNT34I.DLL" )

sUser = wntUserInfo( 0 )
sDomain = wntUserInfo( 1 )
Message( "Logon info", sUser:" logged on to ":sDomain )

;Return a list of all servers and workstations

```

Introduction to Programming

```
ALLSERVERS = -1

listServers = wntServerlist("", "", ALLSERVERS ) ;Specify -1 for all
servers and workstations

sServer = AskItemList( "Workstations and Servers", listServers, @TAB,
@SORTED, @SINGLE )

; Return information about a server's type.
nServerInf = wntServiceInf( sServer )
sResult = ""
For iStep = 0 to 22
    If( (2 ** iStep) & nServerInf )
        gosub add_info
    EndIf
Next
Message( sServer:" identified as:", sResult )
exit

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
:add_info
If( sResult != "" ) then sResult = sResult : @CRLF
sResult = sResult : 2 ** iStep : @TAB
Switch iStep
    case 0
        sResult = sResult : "All LAN Manager workstation"
        break
    case 1
        sResult = sResult : "All LAN Manager server"
        break
    case 2
        sResult = sResult : "Any server running with Microsoft SQL
Server"
        break
    case 3
        sResult = sResult : "Primary domain controller"
        break
    case 4
        sResult = sResult : "Backup domain controller"
        break
```

```
case 5
    sResult = sResult : "Server running the timesource service"
    break
case 6
    sResult = sResult : "Apple File Protocol servers"
    break
case 7
    sResult = sResult : "Novell servers"
    break
case 8
    sResult = sResult : "LAN Manager 2.x Domain Member"
    break
case 9
    sResult = sResult : "Server sharing print queue"
    break
case 10
    sResult = sResult : "Server running dialin service"
    break
case 11
    sResult = sResult : "Xenix server"
    break
case 12
    sResult = sResult : "Windows NT (either workstation or server)"
    break
case 13
    sResult = sResult : "Server running Windows for Workgroups"
    break
case 14
    sResult = sResult : "Microsoft File and Print for Netware"
    break
case 15
    sResult = sResult : "Windows NT Non-DC server"
    break
case 16
    sResult = sResult : "Server that can run the browser service"
    break
case 17
```

Introduction to Programming

```
sResult = sResult : "Server running a browser service as backup"
break
case 18
sResult = sResult : "Server running the master browser service"
break
case 19
sResult = sResult : "Server running the domain master browser"
break
case 20
sResult = sResult : "Unknown service"
break
case 21
sResult = sResult : "Unknown service"
break
case 22
sResult = sResult : "Windows 95 or newer"
break
EndSwitch
return
```