Turbo Pascal®

User's Guide

Version 5.0

Borland International
1800 Green Hills Road
P.O. Box 660001
Scotts Valley, CA 95066-0001

This manual was produced with
Sprint® The Professional Word Processor

All Borland products are trademarks or registered trademarks of
Borland International, Inc. Other brand and product names are trademarks
or registered trademarks of their respective holders.
Copyright© 1983, 1989 Borland International.

Printed in the U.S.A.

10 9 8

R4

Table of Contents

Introduction 1
Understanding 5.0 . . . 2
  Integrated Environment and Command-Line Compilers . . . 2
  Stand-Alone Debugging Support . . . 3
  Separate Compilation . . . 3
  Programs and Units . . . 3
  Compile, Make, and Build . . . 3
  Pick File List . . . 4
  File Extensions . . . 4
About This Manual . . . 5
  Using Turbo Pascal . . . 5
  Appendixes . . . 6
Hardware and Software Requirements . . . 7
Typography . . . 7
How to Contact Borland . . . 8

Part 1 Using Turbo Pascal

Chapter 1 Getting Started 11
Before You Begin . . . 11
What's On Your Disks . . . 12
Installing Turbo Pascal On Your System . . . 15
  Hard Disk Installation . . . 15
  Installation on a Floppy-Drive System . . . 18
  Special Notes . . . 20
Choosing Between Two Compilers . . . 20
Using This Manual . . . 21

Chapter 2 Beginning Turbo Pascal 23
Using the Integrated Environment . . . 23
  Online Help . . . 23
    Using THELP . . . 24
  Windows and Menus . . . 25
  Using Hot Keys . . . 27
Loading Turbo Pascal . . . 31
Creating Your First Program . . . 31
  Analyzing Your First Program . . . 32
  Saving Your First Program . . . 32
  Compiling Your First Program . . . 33

Running Your First Program 34
Checking the Files You've Created 35
Stepping Up: Your Second Program 35
Debugging Your Program 36
Using the Watch Window 37
Fixing Your Second Program 38
Programming Pizazz: Your Third Program 39
The Turbo Pascal Compiler 41
So, What's a Compiler Anyway? 41
What Gets Compiled? 42
Where's the Code? 42
Compile, Make, and Build 43

Chapter 3 Programming in Turbo Pascal 45
The Seven Basic Elements of Programming 46
Data 47
Data Types 47
Integer Data Types 47
Real Data Types 48
Character and String Data Types 49
Boolean Data Type 51
Pointer Data Type 52
Identifiers 53
Operators 53
Assignment Operators 54
Unary and Binary Arithmetic Operators 54
Bitwise Operators 55
Relational Operators 55
Logical Operators 56
Address Operators 57
Set Operators 57
String Operators 57
Output 57
The Writeln Procedure 57
Input 59
Conditional Statements 59
The If Statement 59
The Case Statement 60
Loops 61
The While Loop 61
The Repeat..Until Loop 62
The For Loop 63
Procedures and Functions 64
Program Structure 65
Procedure and Function Structure 65

Sample Program 66
Program Comments 67

Chapter 4 Units and Related Mysteries 69
What's a Unit, Anyway? 69
A Unit's Structure 70
Interface Section 71
Implementation Section 71
Initialization Section 72
How Are Units Used? 73
Referencing Unit Declarations 74
Implementation Section Uses Clause 76
Circular Unit References 77
Sharing Other Declarations 78
TURBO.TPL 79
Writing Your Own Units 81
Compiling a Unit 82
An Example 82
Units and Large Programs 83
Units as Overlays 84
TPUMOVER 84

Chapter 5 Project Management 87
Program Organization 87
Initialization 88
The Build and Make Options 89
The Make Option 90
The Build Option 90
The Stand-Alone Make Utility 91
A Quick Example 91
Creating a Makefile 92
Using MAKE 93
Conditional Compilation 93
The DEFINE and UNDEF Directives 94
Defining at the Command Line 95
Defining in the Integrated Environment 95
Predefined Symbols 95
The VER50 Symbol 95
The MSDOS and CPU86 Symbols 96
The CPU87 Symbol 96
The IFxxx, ELSE, and ENDIF Symbols 96
The IFDEF and IFNDEF Directives 97
The IFOPT Directive 98
Optimizing Code 99

Chapter 6 Debugging Your Turbo Pascal Programs 101
Compile-Time Errors ... 102
Run-Time Errors ... 102
Logic Errors ... 103
Turbo Pascal's Integrated Debugger ... 103
A Quick Debugging Example ... 107
Why Use the Debugger? ... 109
Tracing ... 110
Go to Cursor ... 110
Breaking ... 110
Watching ... 110
Evaluating ... 111
Modifying ... 111
Navigating ... 111
Preparing to Use the Debugger ... 111
Your Display ... 112
Starting a Debugging Session ... 114
Restarting a Debugging Session ... 114
Ending a Debugging Session ... 115
Stepping Through Your Program ... 115
Using Breakpoints ... 119
Using Ctrl-Break ... 120
Watching Values ... 121
Scope and Qualification ... 123
Types of Watch Expressions ... 126
Format Specifiers ... 127
Typecasting ... 130
Expressions ... 131
Editing the Watch Window ... 133
Evaluating and Modifying ... 134
Modification Issues ... 136
Navigation ... 137
The Call Stack ... 137
Finding Procedures and Functions ... 139
General Issues ... 140
How to Write Programs for Debugging ... 141
Memory Issues ... 142
Recursive Routines ... 143
Where Debugging Won't Go ... 144
Common Pitfalls ... 145
Error-Handling ... 145
Input/Output Error-Checking ... 146
Range-Checking ... 148
Other Error-Handling Abilities ... 150

Turbo Debugger 150

Chapter 7 All About the Integrated Environment 153

Turbo Pascal Command-Line Switches 153
  Menu Structure 155
  The Bottom Line 159
  The Edit Window 159
    Working with Source Files 161
      Creating a New Source File 161
      Loading an Existing Source File 162
      Saving a Source File 162
      Writing an Output File 162
  The Watch Window 163
  The Output Window 163
  The Integrated Debugger 164
    The Debugger Screen Display 164
The File Menu 165
    Load (F3) 165
    Pick (Alt-F3) 166
    New 166
    Save (F2) 167
    Write To 167
    Directory 167
    Change Dir 167
    OS Shell 167
    Quit (Alt-X) 168
The Edit Command 168
The Run Menu 168
    Run (Ctrl-F9) 169
    Program Reset (Ctrl-F2) 169
    Go to Cursor (F4) 170
    Trace Into (F7) 170
    Step Over (F8) 170
    User Screen (Alt-F5) 171
The Compile Menu 171
    Compile (Alt-F9) 171
    Make (F9) 172
    Build 172
    Destination (Memory) 172
    Find Error 173
    Primary File 173
    Get Info 174
The Options Menu 174
    Compiler 174
      Range-Checking (Off) 175

Stack-Checking (On) 175
I/O-Checking (On) 176
Force Far Calls (Off) 176
Overlays Allowed (Off) 176
Align Data (Word) 176
Var-String Checking (Strict) 177
Boolean Evaluation (Short Circuit) 177
Numeric Processing (Software) 177
Emulation (On) 177
Debug Information (On) 178
Local Symbols (On) 178
Conditional Defines 179
Memory Sizes 179
Linker 180
Map File (Off) 180
Link Buffer (Memory) 181
Environment 181
Config Auto Save (Off) 182
Edit Auto Save (Off) 182
Backup Files (On) 182
Tab Size (8) 183
Zoom Windows (Off) 183
Screen Size (25-line) 183
Directories 183
Turbo Directory 183
EXE & TPU Directory 183
Include Directories 184
Unit Directories 184
Object Directories 184
Pick File Name 185
Current Pick File 185
Parameters 185
Save Options 185
Retrieve Options 185
The Debug Menu 186
Evaluate (Ctrl-F4) 186
Call Stack (Ctrl-F3) 187
Find Procedure 187
Integrated Debugging (On) 187
Stand-Alone Debugging (Off) 188
Display Swapping (Smart) 188
Refresh Display 189
The Break/Watch Menu 189
Add Watch (Ctrl-F7) 189

Delete Watch 190
Edit Watch 190
Remove All Watches 190
Toggle Breakpoint (Ctrl-F8) 190
Clear All Breakpoints 190
View Next Breakpoint 190
About the Pick List and Pick File 191
The Pick List (Alt-F3) 191
The Pick File 191
Loading a Pick File 192
Saving Pick Files 192
Configuration Files and the Pick File 192

Chapter 8 Command-Line Reference 193
Using the Compiler 193
Compiler Options 194
Compiler Directive Options 195
The Switch Directive (/$) Option 196
The Conditional Defines (/D) Option 197
Compiler Mode Options 197
The Make (/M) Option 197
The Build All (/B) Option 198
The Find Error (/F) Option 198
The Link Buffer (/L) Option 199
The Quiet (/Q) Option 200
Directory Options 200
The Turbo Directory (/T) Option 200
The EXE & TPU Directory (/E) Option 200
The Include Directories (/I) Option 201
The Unit Directories (/U) Option 201
The Object Directories (/O) Option 201
Debug Options 202
The Map File (/G) Option 202
The Stand-Alone Debugging (/V) Option 203
The TPC.CFG File 203

Part 2 Appendixes

Appendix A Differences Between Turbo Pascal 3.0, 4.0, and 5.0 207
How 4.0 and 5.0 Differ 208
How 3.0 and 5.0 Differ 212
Programming Changes 213
Program Declarations 213
Compiler Directives 213
Input and Output 215

Data Types 217
Expression Evaluation Order 218
Predeclared Identifiers 218
Other Additions and Improvements 220
Using Assembly Language 226
Converting from Turbo Pascal 3.0 227
Using UPGRADE 228
/3 Activate Turbo3 Unit 230
/J Activate Journal File 231
/N No Source Markup 231
/O [d:][path] Output Destination 232
/U Unitize 232
What UPGRADE Can Detect 234
What UPGRADE Cannot Detect 235
An UPGRADE Checklist 236

Appendix B Using the Editor 239
Quick In, Quick Out 239
The Edit Window Status Line 240
Editor Commands 240
Basic Movement Commands 242
Extended Movement Commands 243
Insert and Delete Commands 244
Block Commands 245
Miscellaneous Editing Commands 247
The Turbo Pascal Editor versus WordStar 251

Appendix C Turbo Pascal Utilities 253
Using TPUMOVER, the Unit Mover 253
A Review of Unit Files 253
Using TPUMOVER 254
TPUMOVER Commands 255
Moving Units into TURBO.TPL 256
Deleting Units from TURBO.TPL 257
Moving Units Between .TPL Files 257
Command-Line Shortcuts 258
The Stand-Alone MAKE Utility 258
Creating Makefiles 258
Comments 259
Explicit Rules 259
Implicit Rules 261
Command Lists 264
Macros 266
Defined Test Macro ($d) 268
Base File Name Macro ($*) 268

Full File Name Macro ($<) 269
File Name Path Macro ($:) 269
File Name and Extension Macro ($.) 269
File Name Only Macro ($&) 269
Directives 270
Using MAKE 273
The BUILTINS.MAK File 274
How MAKE Searches for Files 275
MAKE Command-Line Options 275
MAKE Error Messages 276
Fatal Errors 276
Errors 277
THELP: The Online Help Utility 278
Installing THELP on Your System 279
Loading and Invoking THELP 279
The THELP Cursor Keys 279
Summary of THELP Command-line Options 281
The /B Option (Use BIOS for Video) 282
The /C#xx Option (Select Color) 282
The /Dname Option (Full Path for Disk Swapping) 283
The /Fname Option (Fill Path and File Name for Help File) 284
The /H, /?, and ? Options (Display Help Screen) 284
The /Kxxyy Option (Reassign Hot Key) 284
The /Lxx Option (Force Number of Rows Onscreen) 285
The /M+ and /M- Options (Display Help Text) 285
The /Px Option (Pasting Speed) 286
The /R Option (Send Options to Resident THELP) 286
The /Sx Option (Default Swapping Mode) 286
The /U Option (Remove THELP from Memory) 287
The /W Option (Write Options to THELP.COM and Exit) 287
The TOUCH Utility 287
The GREP Utility 287
The GREP Switches 288
How to Search Using GREP 290
Examples Using GREP 291
The BINOBJ Utility 294

Appendix D Customizing Turbo Pascal 297
What Is TINST? 297
Keeping Your Turbo Pascal 4.0 TINST Settings (TINSTXFR) 298
How to Use TINSTXFR 299
Running TINST 299
The Compile Menu 301
The Options Menu 301
The Compiler Menu 301

Range-Checking (Off) 301
Stack-Checking (On) 301
I/O-Checking (On) 301
Force Far Calls (Off) 301
Overlays Allowed (Off) 302
Align Data (Word) 302
Var-String Checking (Strict) 302
Boolean Evaluation (Short Circuit) 302
Numeric Processing (Software) 302
Emulation (On) 302
Debug Information (On) 302
Local Symbols (On) 302
Conditional Defines 302
Memory Sizes (16384, 0, 655360) 303
The Linker Menu 303
Map File (Off) 303
Link Buffer (Memory) 303
The Environment Menu 303
Config Auto Save (Off) 303
Edit Auto Save (Off) 304
Backup Files (On) 304
Zoom Windows (Off) 304
Full Graphics Save (On) 304
Screen Size (25) 304
Options for Editor 304
Insert Mode (On) 305
Autoindent Mode (On) 305
Use Tabs (Off) 305
Optimal Fill (On) 305
Backspace Unindents (On) 305
Tab Size (8) 305
Editor Buffer Size (65,534) 305
Make Use of EMS Memory (On) 306
The Directories Menu 306
Turbo Directory 306
EXE & TPU Directory 307
Include Directories, Unit Directories, and Object Directories 307
Pick File Name 307
The Parameters Setting 307
The Debug Menu 308
Integrated Debugging (On) 308
Stand-Alone Debugging (Off) 308
Display Swapping (Smart) 308
The Editor Commands Menu 308

WordStar-Like Selection 310
Ignore Case Selection 311
Verbatim Selection 312
Allowed Keystrokes 312
Global Rules 312
Turbo Pascal Editor Keystrokes 313
The Mode for Display Menu 314
Default 315
Color 315
Black and White 316
LCD or Composite 316
Monochrome 316
The Set Colors Menu 316
The Resize Windows Menu 317
Quitting the Program 318

Appendix E A DOS Primer 319
What Is DOS? 319
How to Load a Program 320
Directories 321
Subdirectories 322
Where Am I? The $p $g Prompt 322
The AUTOEXEC.BAT File 323
Changing Directories 324

Appendix F Glossary 327

Index 337

List of Figures

Figure 2.1: The Turbo Pascal Main Menu Screen 26
Figure 2.2: A Sample Use of Hot Keys 29
Figure 6.1: The Evaluate Window 135
Figure 6.2: The Call Stack Window 139
Figure 7.1: Turbo Pascal's Menu Structure 156
Figure 7.2: The File Menu 165
Figure 7.3: The Run Menu 169
Figure 7.4: The Compile Menu 171
Figure 7.5: The Options Menu 174
Figure 7.6: The Options/Compiler Menu 175
Figure 7.7: The Options/Linker Menu 180
Figure 7.8: The Map File Menu 181
Figure 7.9: The Options/Environment Menu 182
Figure 7.10: The Options/Directories Menu 184
Figure 7.11: The Debug Menu 186
Figure 7.12: The Break/Watch Menu 189
Figure D.1: The TINST Installation Menu 300

List of Tables

Table 2.1: Turbo Pascal's Hot Keys 30
Table 3.1: Integer Data Types 48
Table 3.2: Real Data Types 49
Table 3.3: Precedence of Operators 54
Table 5.1: Summary of Compiler Directives 94
Table 5.2: Predefined Conditional Symbols 95
Table 6.1: Debugger Commands and Hot Keys 104
Table 6.2: Debug Expression Format Characters 129
Table 6.3: Watch Expression Values 132
Table 8.1: Command-Line Options 195
Table A.1: New and Modified Procedures and Functions 209
Table B.1: Summary of Editor Commands 241
Table B.2: Control Cursor Sequences 243
Table B.3: Quick Movement Commands 243
Table C.1: THELP Command Keys 280
Table C.2: THELP Command-Line Options 281
Table C.3: Color Values for a Standard Color Display 283
Table D.1: Editor Screen Keystrokes 309
Table D.2: Editor Screen Keystrokes in Modify Mode 310

I N T R O D U C T I O N

Welcome to version 5.0 of Turbo Pascal! Turbo Pascal is designed to meet the needs of all types of users of IBM PS/2s, PCs, and compatibles. It's a structured, high-level language you can use to write programs for any type or size application.

This version of Turbo Pascal is the latest generation of Borland's flagship language product. Here's a quick look at what you get with 5.0:

- integrated source-level debugging, complete with single-stepping, breakpoints, examination of variables, data structures and expressions, and the option of assigning new values to variables and data structures while debugging
- complete 8087 floating-point emulation, allowing use of IEEE floating-point types even if you don't have an 8087 math coprocessor
- unit-based overlays, and a state-of-the-art overlay manager
- compatibility with Borland's Turbo Debugger, permitting stand-alone debugging of your 5.0 programs
- expanded memory support on systems running EMS (3.2 or later), including the ability to load overlays into EMS memory, and use of EMS memory by the built-in editor
- two to three times faster compilation speed (lines per minute) than version 3.0
- improved code generation, resulting in faster execution
- a smart built-in linker that removes unused code and data at link time, producing smaller programs
- .EXE files, which can be larger than 64K
- separate compilation of individual units
- built-in project management that performs automatic recompilation of dependent source files (including units)
- several standard units, including *System*, *Dos*, *Crt*, *Overlay*, and *Graph*

- a more powerful assembly language interface, as well as inline assembly options
- the ability to nest Include files up to 15 levels deep
- several new data types, including longint, shortint, word, and IEEE floating-point types (single, double, extended, and comp)
- several new built-in procedures and functions, including *Inc()* and *Dec()*
- built-in 8087/80287/80387 coprocessor support
- short-circuit Boolean expression evaluation
- conditional compilation directives
- a high degree of compatibility with versions 3.0 and 4.0, and a utility and units to aid in converting 3.0 programs to 5.0
- both command-line and integrated environment versions of the compiler

Understanding 5.0

As you're reading through this manual, several major concepts will be introduced. To help clarify these ideas, here's a summary of some of 5.0's features.

Integrated Environment and Command-Line Compilers

The Turbo Pascal compiler is actually two compilers: an integrated development environment and a command-line version. The Borland-style integrated environment combines a text editor and compiler; it provides pull-down menus, windows, input boxes, configuration control, and context-sensitive help. The integrated debugger makes it easy to step through your program line-by-line, examine or modify variables and memory locations, set breakpoints, and stop your program at any time using *Ctrl-Break*. This compiler is the TURBO.EXE file on your disk.

The traditional command-line or batch mode compiler allows you to use your own editor to create and modify program source code. You then run the compiler from either the command line or a batch file, giving the file name and any other compiler options. This compiler is the TPC.EXE file on your disk.

Stand-Alone Debugging Support

Both the command-line compiler and the integrated development environment provide full support for debugging using Borland's stand-alone debugger, Turbo Debugger.

Separate Compilation

Separate compilation lets you break programs into parts and compile them. That way you can test each part to make sure it works. You can then link all the parts together to build a program. This is useful, since you don't have to recompile everything that makes up a program each time you use it. In addition, this feature lets you build up a toolbox of precompiled, tested code that you can use in all your programs.

Programs and Units

A program is the main module of Pascal source code that you write and execute. In order to provide for separate compilation and still maintain Pascal's strict checking among program parts, units are used. A *unit* is a piece of source code that can be compiled as a stand-alone entity. You can think of units as a library of data and program code. They provide a description of the interface between the unit's code and data and other programs that will use that unit. Programs and other units can use units; units don't use programs.

Compile, Make, and Build

It's probable that you may change the source code of several of the units you're using without recompiling them; however, you'll definitely want your main program to use the absolute latest units. We've provided two ways for you to make sure the unit files are brought up to date.

The Make option tells the compiler to go and look at the date and time of any source and compiled unit file used by your main program (or by another unit, since units can use units). If the source file was modified since the unit was compiled, the compiler will recompile the unit to bring it up to date.

The Build option is similar to Make except that it will recompile all of the units used by your main program (or unit) *without* checking date and time.

Use this option if you want to make absolutely sure you have all the latest compiled units.

Pick File List

The pick file contains the state of the integrated environment, so that when you leave TURBO.EXE and return to it later, you are placed at the spot in the file where you left off previously. The pick file list also offers you easy access to files when you are editing multiple files. The last eight file names and the state of each respective file that you've edited are kept in the pick list. When you select a file from the pick list, the file is loaded and the cursor is placed at the point in the file where you were when you left it. You can enable or disable pick file (TURBO.PCK) generation.

File Extensions

All kinds of file name extensions are used in the DOS world; most are application- or program-specific. (Remember that a file name consists of up to eight characters with an optional three-character extension.) Turbo Pascal uses several different file name extensions:

- **.EXE:** An executable file. The two compilers themselves are .EXE files. The compiled programs you'll build with the compilers will be .EXE files. (Turbo Pascal 3.0 created .COM files that were also executable files.)
- **.PAS:** Use this for your Pascal source code files. You can use other file name extensions, but traditionally .PAS is used.
- **.TPU:** A precompiled unit file. When you compile a Pascal unit, the compiler generates a .TPU file with the same first eight characters of the source file. A .TPU file contains the symbol information and compiled code for the unit.
- **.TPL:** A Turbo Pascal library file. You can use only one of these at a time. The standard library file on the disk is called TURBO.TPL. You can modify TURBO.TPL to suit your needs.
- **.TP and .CFG:** Configuration files for the two compilers. These files allow you to override default settings in the compilers and customize compiler default values to your own needs.

  A .TP file is a binary file containing the options you set for the integrated environment. You can have multiple .TP files for different settings.

  TPC.CFG is the configuration file for the command-line version of the compiler. There can be only one TPC.CFG file. It is a text file that contains directories to the compiler, command-line switches, and so on.

- .PCK: The Turbo Pascal pick file extension. The pick file contains the state of the integrated environment so that, when you leave TURBO.EXE and return later on, you are placed at the spot in the file where you were last working. You can enable or disable pick file generation.
- .MAP: This file is generated if you set the Options/Linker/Map File menu command to On in the integrated environment or use the /G command-line compiler option. It contains information about your program that can be used with most standard symbolic debuggers.
- .BAK: Backup source file extension. The editor in the integrated environment renames the existing file on disk to a .BAK file when you save a modified version of the file. You can enable or disable .BAK file generation with TINST (refer to Appendix D, "Customizing Turbo Pascal," for details).

About This Manual

This manual walks you through writing, compiling, and saving Turbo Pascal programs. It explains in detail the many new features and how to use them. It also teaches you how to take existing version 3.0 and 4.0 programs and convert them to run under Turbo Pascal version 5.0.

Sample programs are provided on the distribution disks for you to study. You can also tailor these sample exercises to your particular needs.

Before you get started, you should be somewhat familiar with the basics of operating an IBM PC (or compatible) under MS-DOS (or PC-DOS). You'll need to know how to run programs, copy and delete files, and how to use other basic DOS commands. If you're not sure about how to do these things, spend some time playing with your PC and reviewing the MS-DOS user's manual that came with it; you can also look at Appendix E, "A DOS Primer," to learn some basics. Appendix F lists many of the terms introduced in this manual.

This volume can be considered the tutorial part of the Turbo Pascal documentation; it's divided into two main sections: "Using Turbo Pascal" and "The Appendixes." The second volume, the *Turbo Pascal Reference Guide* gives you a complete technical description of Turbo Pascal 5.0.

Using Turbo Pascal

The first section,"Using Turbo Pascal," introduces you to Turbo Pascal, shows you how to use it, and includes chapters that focus on such specific features as units and debugging. Here's a breakdown of the chapters:

- **Chapter 1: Getting Started** explains how to make backup copies of your Turbo Pascal disks, describes the different files on the disks, and tells you how to use INSTALL to set up Turbo Pascal for your particular system.
- **Chapter 2: Beginning Turbo Pascal** leads you directly from loading Turbo Pascal into writing simple programs, and then on to compiling and running them. A discussion of a few common programming errors and how to use the debugger is also presented. You'll learn some basics about getting around in the integrated environment.
- **Chapter 3: Programming in Turbo Pascal** introduces you to the Pascal programming language.
- **Chapter 4: Units and Related Mysteries** tells you what a unit is, how it's used, what predefined units (libraries) Turbo Pascal provides, and how to write your own. It also describes the general structure of a unit and its interface and implementation portions, as well as how to initialize and compile a unit.
- **Chapter 5: Project Management** tells how to develop large programs using multiple source files and libraries, and discusses conditional compilation.
- **Chapter 6: Debugging Your Turbo Pascal Programs** tells how to use the integrated debugger and gives suggestions on how to track down and eliminate errors in your programs.
- **Chapter 7: All About the Integrated Environment** is a complete guide to the menu commands in Turbo Pascal's integrated environment.
- **Chapter 8: Command-Line Reference** is a complete guide to the command-line version (TPC.EXE) of Turbo Pascal.

Appendixes

Part 2 of this manual contains six appendixes that deal with the following topics:

- **Appendix A: Differences Between Turbo Pascal 3.0, 4.0, and 5.0** lists the differences between the three versions that affect backward compatibility.
- **Appendix B: Using the Editor** explains how to use the built-in editor to open, edit, change, save a file, and more.
- **Appendix C: Turbo Pascal Utilities** discusses some of the other programs that come with Turbo Pascal 5.0, such as TPUMOVER, MAKE, THELP, TOUCH, GREP, and BINOBJ.

- **Appendix D: Customizing Turbo Pascal** shows how to use TINST to customize your copy of Turbo Pascal 5.0, as well as transfer your 4.0 TINST customizations intact.
- **Appendix E: A DOS Primer** talks about MS-DOS, telling you all you need to know in order to use Turbo Pascal 5.0.
- **Appendix F: Glossary** lists commonly used terms in this manual and explains what they mean.

Hardware and Software Requirements

Turbo Pascal runs on the IBM PC family of computers, including the XT, AT, and the PS/2 series, as well as true IBM compatibles. Turbo Pascal requires DOS 2.0 or higher and at least 448K of RAM to run the integrated environment (256K to run the command-line compiler).

Turbo Pascal includes floating-point routines that let your programs make use of an 8087, 80287, or 80387 numeric coprocessor if you have one. An 8087/80287/80387 coprocessor can significantly enhance performance of your programs, but Turbo Pascal does not require one. In fact, Turbo Pascal can link to a run-time library that will emulate the numeric coprocessor if you don't have one.

Turbo Pascal supports expanded memory on systems running EMS drivers conforming to the 3.2 (or later) Lotus/Intel/Microsoft Expanded Memory Specification (EMS).

Typography

This manual was produced by Borland's Sprint: The Professional Word Processor on a PostScript printer. The different typefaces displayed are used for the following purposes:

Italics	In text, this typeface represents constant identifiers, field identifiers, and formal parameter identifiers, as well as unit names, labels, user-defined types, variables, procedures, and functions.
Boldface	Turbo Pascal's reserved words are set in this typeface.
`Monospace`	This type represents text that appears on your screen.

Keycaps This typeface indicates a key on your keyboard. It is often used when describing a key you have to press to perform a particular function; for example, "Press *Esc* to exit from a menu."

How to Contact Borland

If, after reading this manual and using Turbo Pascal, you would like to contact Borland with comments or suggestions, we suggest the following procedures:

- The best way is to log on to Borland's forum on CompuServe: Type GO BPROGA at the main CompuServe menu and follow the menus to section 2. Leave your questions or comments here for the support staff to process.
- If you prefer, write a letter detailing your problem and send it to

  Technical Support Department
  Borland International
  P.O. Box 660001
  1800 Green Hills Road
  Scotts Valley, CA 95066-0001

  Please note: If you include a program example in your message, it must be limited to 100 lines or less. We request that you submit it on disk, include all the necessary support files on that disk, and provide step-by-step instructions on how to reproduce the problem. Before you decide to get technical support, try to replicate the problem with the code contained on a floppy disk, just to be sure we can duplicate the problem using the disk you provide us.
- You can also telephone our Technical Support department at (408) 438-5300. To help us handle your problem as quickly as possible, have these items handy before you call:
  - product name and version number
  - product serial number
  - computer make and model number
  - operating system and version number

If you're not familiar with Borland's No-Nonsense License statement, now's the time to read the agreement at the front of this manual and mail in your completed product registration card.

P A R T

1

Using Turbo Pascal

C H A P T E R

1

Getting Started

In this chapter, we'll get you started using Turbo Pascal by showing how to use INSTALL to load it on your floppy disk or hard disk system. We'll also offer some guidance on how to go about reading this manual, based on your programming experience.

Before You Begin

The three distribution disks that accompany this manual are formatted for standard 5-1/4 inch disks, 360K disk drives, and can be read by IBM PCs and compatibles (those with 3-1/2 inch disk, 720K disk drives will receive two distribution disks). Now, before you do anything else, we want you to make backup copies of these three disks; then, after you run the Turbo Pascal installation program (see page 15), put the originals away. Since there's a replacement charge if you erase or damage the original disks, take heed and use your originals only to install Turbo Pascal or make backup copies. Here's how:

- Get three new (or unused) floppy disks.
- Boot up your computer.
- At the system prompt, type `diskcopy A: B:` and press *Enter*. The message `Insert source diskette in drive A:` will be displayed on your screen. Remove your system disk from Drive A and put distribution disk 1 into Drive A.
- If your system has two floppy disk drives, your screen will also say `Insert destination diskette into Drive B`. In that case you'll need to remove any disk in Drive B, replacing it with a blank disk. If your system

only has one floppy drive, then you'll be swapping disks in Drive A. Just remember that the distribution disk is the *source* disk, the blank disk is the *destination* disk.

- If you haven't done it already, press *Enter*. The computer will start reading from the source disk in Drive A.
- If you have a two-drive system, it will then write out to the destination disk in Drive B and continue reading from A and writing to B until copying is complete. If you have a one-drive system, you'll be asked to put the destination disk in A, then the source disk, then the destination disk, and so on and so forth until it's finished.
- When copying is completed, remove the distribution (source) disk from Drive A, and put it away. Remove the copy (destination) disk from Drive B and label it Install/Compiler.
- Repeat the preceding process with the second and third distribution disks and the other blank floppies.

Now that you've made your backup copies, we can get on to the meat of this chapter.

What's On Your Disks

The distribution disks that come with this manual include two different versions of the Pascal compiler: an integrated environment version and a stand-alone, command-line version.

You might not need all the files that come on your distribution disks. Use the INSTALL program and then delete the files you don't need from your working disks. The README file contains a complete file list. For your reference, here's a summary of most of the files on disks and how to determine which ones to retain:

README — To see any last-minute notes and corrections, type `README` at the system prompt. (If you have a printer, you can print it out.) Once you review this material, keep it around for future reference.

HELPME!.DOC — Contains answers to many common questions about Turbo Pascal 5.0.

TURBO.EXE — This is the integrated (menu-driven) environment version of Turbo Pascal. It lets you edit, compile, run, and debug your programs. See Chapter 7, "All About the Integrated Environment," for more information.

TURBO.TPL	This contains the units (program libraries) that come with Turbo Pascal, including *System, Crt, Dos, Overlay,* and *Printer*—this is a must! See Chapter 12 of the *Reference Guide,* "Standard Units," for more information on these units.
TURBO.HLP	This contains the online, context-sensitive help text used by the integrated environment and the THELP utility. See page 23 for information on online help, and Appendix C, "Turbo Pascal Utilities," for details on the THELP utility.
THELP.COM	This is the memory-resident utility that provides access to Turbo Pascal's context-sensitive help system from any program. See Appendix C, "Turbo Pascal Utilities."
TPC.EXE	This is the command-line version of Turbo Pascal. If you use a separate editor, make heavy use of batch files, and so on, you'll probably want to use this. Refer to Chapter 8, "Command-Line Reference," for information on how to use the command-line compiler.
GRAPH.TPU	This contains the *Graph* unit (the Borland Graphics Interface unit). See the section "The Graph Unit" in Chapter 12 of the *Reference Guide* for more information.
*.ARC files	Packed files that contain documentation files, example programs, graphics device drivers, fonts, interface section listings for Turbo Pascal's units, and more. INSTALL with walk you through the dearchiving of these files (see page 15).
*.BGI files	BGI graphics device drivers.
*.CHR files	BGI graphics stroked character fonts.
*.DOC files	These include the interface section listings for all the standard units.
*.PAS files	These include an overlay example and the MicroCalc source files, as well as other sample programs.
INSTALL.EXE	Installs Turbo Pascal on your system (see page 15 for instructions).
TPUMOVER.EXE	This utility allows you to add units to or remove units from the TURBO.TPL file. Appendix C, "Turbo Pascal Utilities," contains information on TPUMOVER.

TINST.EXE	This utility allows you to customize certain features of TURBO.EXE. See Appendix D, "Customizing Turbo Pascal," for more information.
TINSTXFR.EXE	This utility transfers the customized settings you created with TINST in 4.0 to 5.0. See Appendix D, "Customizing Turbo Pascal."
MAKE.EXE	This is an intelligent project manager that allows you to keep your programs up-to-date and is especially useful when you are mixing assembler and Pascal and using the command-line compiler (TPC.EXE). See Appendix C, "Turbo Pascal Utilities," for more information on using MAKE.
GREP.COM	This is a fast, powerful, text search utility. See Appendix C, "Turbo Pascal Utilities," for more information on using GREP.
TOUCH.COM	This utility changes the date and time of one or more files to the current date and time, making it "newer" than the files that depend on it. It's generally used in conjunction with MAKE.EXE.
BINOBJ.EXE	Use this utility to convert a binary data file to an .OBJ file.
TPCONFIG.EXE	This utility takes your integrated environment configuration file and converts it to work with the command-line compiler (as TPC.CFG). It's helpful if you want to use the integrated environment to set all your options, but want to compile with the command-line version. This utility will also convert a TPC.CFG file to a .TP file.
UPGRADE.DTA UPGRADE.EXE	This utility does a quick upgrade of Turbo Pascal version 3.0 source files, modifying them for compatibility with Turbo Pascal version 5.0. See the section on UPGRADE in Appendix A for more information.
GRAPH3.TPU TURBO3.TPU	These are version 3.0 compatibility units. Refer to Appendix A, "Differences between Turbo Pascal 3.0, 4.0 and 5.0."
README.COM	This is the program to display the README file. Once you've read the README, you can delete this.

Installing Turbo Pascal On Your System

The first thing you'll want to do is install Turbo Pascal on your system. Your Turbo Pascal package includes all the files and programs necessary to run both the integrated environment and command-line versions of the compiler. A new program, INSTALL.EXE, sets up Turbo Pascal on your system. INSTALL works on both hard disk and floppy-based systems. The next section descibes hard disk installation; for floppy installation, skip to the section "Installation on a Floppy-Drive System" on page 18.

Hard Disk Installation

For the sake of simplicity, let's say you're installing Turbo Pascal on your hard disk, which is Drive C. During installation, you will place the Turbo Pascal source disks in Drive A. (You will have three source disks if you ordered 5-1/4 inch, 360K disks, two if you ordered 3-1/2 inch, 720K disks.) Put the Install/Compiler Disk in Drive A (or the Install/Compiler/Help/ Utilities Disk, if you have 3-1/2 inch, 720K disks). Now type

```
A:INSTALL
```

at the C:\> prompt. When you see the installation opening screen, press *Enter*. INSTALL will ask you to specify a source drive. The source drive refers to the drive containing the Turbo Pascal program and utilities, in this case Drive A. If your source disks are in a drive other than Drive A, enter the appropriate drive by typing the drive letter followed by *Enter*.

Now you are asked whether you want to

1. Install Turbo Pascal on a Hard Drive.
2. Update Turbo Pascal 4.0 to Turbo Pascal 5.0 on a Hard Drive.
3. Install Turbo Pascal on a Floppy Drive.

Use the arrow keys to move the highlight bar to the appropriate choice (1 or 2) and press *Enter*. At any time during the installation, you may ask for help by pressing *F1*. If you choose "Install Turbo Pascal on a Hard Drive," the INSTALL program will display a window listing the default directories.

- Turbo Pascal Directory: C:\TP
- Graphics Subdirectory: C:\TP
- Documentation Subdirectory: C:\TP\DOC
- Example Subdirectory: C:\TP
- Turbo Pascal 3.0 Compatibility Subdirectory: C:\TP\TURBO3

By default, INSTALL creates separate subdirectories for the documentation files (*.DOC and HELPME!.DOC) and the Turbo3 compatibility files (UPGRADE.EXE, TURBO3.TPU, GRAPH3.TPU, TURBO3.DOC, and so on). It will place all other files from the distribution disks in the Turbo Pascal Directory. If you would rather separate graphics and example programs into their own subdirectories as well, edit the default paths for those files by moving the highlight bar to the directory you want to change, pressing *Enter*, and then entering the full MS-DOS path name of the directory you want (the INSTALL program will create a new one for you).

INSTALL has an "Unpack Archives" command; it toggles between Yes and No. Archive files (*.ARC) are condensed files that contain example programs and documentation and help files. INSTALL gives you a choice of copying the .ARC files intact ("No") or dearchiving them ("Yes") and copying all the individual, unpacked files onto your hard disk.

Note: If you don't have much space on your hard disk, you might want to switch "Unpack Archives" to No. To do this, use the arrow keys to move the highlight bar to "Unpack Archives" and press *Enter*.

The BGI/Demos/Doc/Turbo3 Disk contains several files with an .ARC file extension: BGI.ARC, DEMOS.ARC, DOC.ARC, MCALC.ARC, and TURBO3.ARC. If you set "Unpack Archives" to No, you can still dearchive them yourself later by using the UNPACK.COM utility. For example, typing

```
UNPACK DEMOS
```

unpacks all the files stored in the DEMOS.ARC archive into the current directory.

Note that INSTALL does not unpack the file stored in BGI.ARC. BGIEXAMP.ARC contains all the BGI (Borland Graphics Interface) program examples from the "Turbo Pascal Reference Lookup" chapter in the *Reference Guide*. If you want to unpack the examples from this file, go to the directory that contains both UNPACK.COM and BGIEXAMP.ARC and type

```
UNPACK BGIEXAMP
```

This will unpack all 69 examples from BGIEXAMP.ARC.

When you are satisfied with your installation settings, move the highlight bar down to "Start Installation" and press *Enter* (or press the INSTALL hot key *F9*). A window will pop open and list the file names as the INSTALL program copies them from the source disk and writes them to the destination disk (in this case, your hard drive). At some point, the disk will stop whirring and you will be asked to put in the Help/Utilities Disk. Insert it in your source drive (Drive A) and press any key to continue.

Note: If you ordered 3-1/2 inch, 720K disks, your first disk combines the Install/Compiler files and the Help/Utilities files; therefore, there's no need for you to switch disks until you are prompted for the "packed files" disk (BGI/Demos/Doc/Turbo3).

Finally, INSTALL prompts you to insert the BGI/Demos/Doc/Turbo3 Disk in the source drive. Do this and press any key. If you have left "Unpack Archives" set to Yes, INSTALL will spend a little bit longer on this disk than on the first two as it uncompresses the archive files.

When it has finished, the INSTALL program reminds you to read the latest about Turbo Pascal in the README file, which contains important, last-minute information about Turbo Pascal 5.0. The HELPME!.DOC file also answers many common technical support questions. Installation is now complete.

At this point, INSTALL reminds you to create a CONFIG.SYS file, if you don't already have one. Be sure to include the line

```
FILES = 20
```

This tells DOS that you want to reserve the maximum space possible for simultaneously open files. By default, DOS allows eight open files at a time, which is usually not enough. Twenty open files may sound like a lot, but every time you run a program, DOS automatically opens three to five files. When you realize that Turbo Pascal itself opens several files, that any files you edit are open files, and that running your program from the integrated environment opens another few open files, you'll see how quickly you could wind up with twelve open files. If you use a lot of include directives in your Pascal source code, you'll add even more.

You should also add C:\TP to the DOS PATH command in your AUTOEXEC.BAT file by including a line like

```
PATH=C:\DOS;C:\TP
```

At some point, you may want to run TINST to configure Turbo Pascal to your own unique quirks (see Appendix D, "Customizing Turbo Pascal," for more information). Note that INSTALL's upgrade option *automatically* runs TINSTXFR.EXE, a new utility that copies your Turbo Pascal 4.0 TINST configuration settings to the new compiler. If you don't use INSTALL to upgrade, you might still want to run TINSTXFR to transfer your settings (see Appendix D, "Customizing Turbo Pascal").

Now that you're finished, skip to "Special Notes" on page 19 for a few more items of interest.

Installation on a Floppy-Drive System

To create a working disk, you need to set up a floppy disk that has at least two files on it: TURBO.EXE and TURBO.TPL. In addition, you'll probably want to copy TURBO.HLP from the Help/Utilities Disk. The INSTALL program will automatically copy the appropriate files. You must first format a blank floppy disk. If you want to make a bootable floppy disk (a disk with DOS system files on it), you will not have room for the help file on a standard 360K floppy. If you do not make a bootable floppy disk, you will have to boot your computer from another system disk and swap in your Turbo Pascal working disk when you want to run Turbo Pascal.

To format a bootable floppy, put your DOS system disk in Drive A and type

```
FORMAT B:/S
```

If you have only one drive, your computer will ask you to insert a DOS disk into Drive A; just insert your regular system boot disk. If you have a two-drive system, place a blank disk into Drive B and press *Enter* when prompted. On a one-drive system, place your blank disk into the drive whenever you are asked to insert a blank disk into Drive B, and place your original boot disk into the drive whenever you are asked to insert a DOS disk in Drive A. If you want to save room for the help file, you can omit the /S option in the FORMAT command, but you won't be able to boot from your working disk.

Now you're ready to transfer files from the distribution disks to your working disk. Make sure you are logged on to Drive A (you should see the A> prompt on the DOS command line). Insert the Install/Compiler Disk and type

```
INSTALL
```

The Turbo Pascal Installation screen greets you. Press *Enter* to continue; keep in mind that you can press *F1* at any time to get help. INSTALL will prompt you to enter a source drive. The source drive refers to the drive containing the Turbo Pascal program and utilities. The INSTALL program uses Drive A as the default source drive, so just press *Enter*.

Now INSTALL asks whether you want to:

1. Install Turbo Pascal on a Hard Drive.
2. Update Turbo Pascal 4.0 to Turbo Pascal 5.0 on a Hard Drive.
3. Install Turbo Pascal on a Floppy Drive.

Move the highlight bar to the third choice, "Install Turbo Pascal on a Floppy Drive," by pressing *Down arrow* twice; now press *Enter*. INSTALL asks

you if you want to install the integrated environment or the command-line version of Turbo Pascal; the "Version of Turbo Pascal to Copy" command toggles between Integrated Environment and Command Line when you press *Enter*. If you don't know which to one to pick, choose Integrated Environment. If you don't want INSTALL to copy the help file, either because you don't need it or because you don't have room on your disk, move the highlight bar down to "Copy Help File" and press *Enter* to toggle it to No.

Now, to start the installation, move the highlight bar down to "Start Installation" and press *Enter*. The INSTALL program asks you to insert your working disk in Drive B. This is the disk you just finished formatting. If you only have a one-floppy system, you will have to keep swapping disks during installation. In this case, put your working disk in Drive A. INSTALL will tell you when you need to swap disks, so keep your eyes on the screen. For two-floppy systems, you will only have to swap disks if you set "Copy Help File" to Yes.

INSTALL informs you that it is working by listing the files it is transferring, like this:

```
Reading files:
  A:\TURBO.EXE, A:\TURBO.TPL
Writing files:
  B:\TURBO.EXE, B:\TURBO.TPL
```

If you chose Yes for "Copy Help File," you will eventually be prompted to insert the Help/Utilities Disk in Drive A. Do this and press *Enter*. From this disk, INSTALL will copy TURBO.HLP (the help file) to your working disk. When this file is copied, installation is complete. The installation utility reminds you that you should create a file on your working disk called CONFIG.SYS (if you don't already have one). You can do this with the Turbo Pascal editor if you like. There should be a line in CONFIG.SYS that says

```
FILES = 20
```

This tells DOS that you want to reserve the maximum space possible for simultaneously open files. This may sound like a lot; see the explanation of FILES = 20 on page 17 under "Hard Disk Installation" if you're curious about why you need that much space for open files.

Once the installation is complete, simply press *Enter* and you will exit to DOS. You now have a working disk.

Special Notes

- If you use INSTALL's Upgrade option, 5.0 files will overwrite any version 4.0 files that have the same names. If you let INSTALL copy 5.0 files into your 4.0 subdirectory, some 4.0 files might still be left on disk and not be overwritten. In this case, you should delete any obsolete 4.0 files after running INSTALL. This is especially important if you have INSTALL build separate subdirectories for 5.0 file groups (DOC, BGI, TURBO3, and so on).
- If you install the graphics files into a separate subdirectory (C:\TP\BGI, for example), remember to specify the full path to the driver and font files when calling *InitGraph*, for example:

  ```
  InitGraph(Driver, Mode, 'C:\TP\BGI');
  ```

- If GRAPH.TPU is not in the current directory, you'll need to add its location to the unit directories with the Options/Directories/Unit Directories command (or with the */U* option in the command-line compiler) in order to compile a BGI program.
- If you have difficulty reading the text displayed by the INSTALL or TINST programs, they will both accept an optional command-line parameter that forces them to use black-and-white colors:

  - A:INSTALL /B — Forces INSTALL into BW80 mode
  - A:TINST /B — Forces TINST into BW80 mode

  Specifying the /B parameter may be necessary if you are using an LCD screen or a system that has a color graphics adapter and a monochrome or composite monitor. To find out how to permanently force the integrated environment to use black-and-white colors with your LCD screen (or CGA and monochome/composite monitor combination), see the note on page 26.

Choosing Between Two Compilers

You've bought two complete versions of the Turbo Pascal compiler. The first, TURBO.EXE, is known as the *integrated environment*. It provides a pull-down menu- and keystroke-driven multiwindow environment. You can load, edit, save, compile, and run your programs without ever leaving it. Most of the chapters that follow this one are devoted to using the integrated environment.

The second version, TPC.EXE, is known as the *command-line compiler*. It presumes that you have created your Pascal program with some other editor (MicroStar, BRIEF, EDLIN, even the integrated environment). You

run it from the MS-DOS system prompt; for example, if your program is in a file named MYFIRST.PAS, you would type at the prompt

```
TPC MYFIRST
```

and then press *Enter*. TPC.EXE compiles and links your program, producing an .EXE file (just like TURBO.EXE). Command-line options allow you to specify a number of things, such as where the system library (TURBO.TPL) resides and whether to recompile any files upon which MYFIRST.PAS depends.

Which version should you use? Chances are you'll find the integrated environment best suits your needs. It provides a complete development system in which you can quickly build and debug programs. On the other hand, if you are currently using a command-line compiler, if you have another editor that you prefer, or if you are making heavy use of an assembler (for external subroutines), you may want to use the command-line compiler in conjunction with a batch file or Make utility.

Using This Manual

Now that you've loaded the Turbo Pascal files and libraries onto the appropriate floppy disks or hard disk directories, you can start digesting this manual and using Turbo Pascal. But, since this user's guide is written for three different types of users, certain chapters are written with your particular Turbo Pascal programming needs in mind. Take a few moments to read the following, then take off programming.

- **Programmers Learning Pascal:** If you're a beginning Pascal programmer, you will want to read Chapters 2 through 5. These are written in tutorial fashion and take you through creating and compiling your first Pascal programs. Along the way, they teach you how to use the integrated environment. (You may also want to look at the *Turbo Pascal Tutor* manual.)
- **Experienced Pascal Programmers:** If you're an experienced Pascal programmer, you should have little difficulty porting your programs to this implementation. You'll want to skim Chapter 7, "All About the Integrated Environment," and Appendix B, "Using the Editor," to get familiar with the integrated environment. Chapters 4, 5, and 6 will introduce you to most features unique to Turbo Pascal 5.0. You'll also want to skim through the *Turbo Pascal Reference Guide*, noting the differences between Turbo Pascal 5.0 and your Pascal compiler.
- **Turbo Pascal Programmers:** Appendix A provides guidelines on the things you'll need to convert your program from version 3.0 or 4.0 to

version 5.0. It also highlights the differences between 3.0, 4.0, and 5.0. You'll definitely want to read Chapter 6 in this manual to acquaint yourself with the integrated debugger.

Whatever your approach, welcome to the world of Turbo Pascal 5.0!

C H A P T E R

2

Beginning Turbo Pascal

Turbo Pascal is more than just a fast Pascal compiler; it is an efficient Pascal compiler with an easy-to-learn and easy-to-use integrated development environment. With Turbo Pascal, you don't need to use a separate editor, compiler, linker, and debugger in order to create, debug, and run your Pascal programs (although you can use the command-line version). All these features are built into Turbo Pascal, and they are all accessible from the Turbo Pascal integrated development environment (IDE).

You can begin writing your first Turbo Pascal program using the IDE compiler. By the end of this chapter, you'll have learned the basics of this development environment, written and saved three small programs, and learned a few basic programming skills.

Using the Integrated Environment

In this section, we describe the components of the Turbo Pascal main screen, and explain briefly how to move around in the environment. For greater detail, refer to Chapter 7, "All About the Integrated Environment." For more on the editor, refer to Appendix B, "Using the Editor."

Online Help

Turbo Pascal provides context-sensitive onscreen help at the touch of a single key. You can get help at any point (except when *your* program has control) by pressing *F1*. The Help window details the functions of the item

on which you're currently positioned. Any help screen can contain one or more *keywords* (highlighted items) on which you can get more information. Use the arrow keys to move to any keyword, and press *Enter* to get more detailed help on the chosen item. Use the *Home* and *End* keys to go to the first and last keywords on a screen, respectively. You can think of the organization of the help system as a tree or an outline structure with some additional links that make it easy to move between screens which cover associated topics.

To get to the help index, which lists keywords for 18 general topics, press *F1* again once you're in the help system. The help index lets you access both language and environment help. While you're in the editor, you can also get help on any standard Pascal unit, function, variable, constant, or type by positioning the cursor on the item and pressing *Ctrl-F1*. (**Note:** *Ctrl-F1* is an editor command that can be redefined using TINST, as described in Appendix D, "Customizing Turbo Pascal.")

If you want to return to a previous help screen from inside the Help system, or even from outside, press *Alt-F1*. (You can back up through 20 previous help screens.) Within a help group (a series of related help screens), *Alt-F1* remembers the group as one screen viewed rather than remembering each screen individually. In a help group, wherever *PgUp* and *PgDn* occur, *PgUp* takes you back a screen, and *PgDn* takes you forward. To exit from help and return to your menu choice, press *Esc* (or any of the hot keys described later in this chapter).

Using THELP

THELP is a memory-resident help utility you can use with the command-line version of Turbo Pascal, TPC.EXE. You might be using the command-line compiler instead of the integrated environment because you use a stand-alone editor like Sprint, MicroStar or Brief. With these editors, THELP works like the online help system in the integrated environment, with one major difference: To invoke THELP, you press the THELP hot key (*5* on the numeric keypad) instead of *F1*, as you would in the IDE. THELP is ideal for use with Turbo Debugger, Borland's stand-alone debugger that works on Turbo Pascal, Turbo C, and Turbo Assembler .EXE files. Since THELP is memory-resident, you must load it in memory before you invoke your editor. To load THELP, type

```
THELP
```

at the `C:>` prompt on the DOS command line. Now invoke your editor as usual. Once in the editor, press the THELP hot key, *5*, to get help. If your cursor is on a blank line when you press *5*, THELP displays the same help

index screen as in the IDE. Move your cursor to any keyword and press *Enter*.

You can always get back to the THELP index screen from inside the help system by pressing *F1*. Also you can cycle back through the last 20 screens you've viewed by pressing *Alt-F1*. Each time you press *Alt-F1*, you go back one more screen until you wrap around again to the most recently accessed screen.

More than likely, the help you'll want will be on a particular function, procedure, or other element of Turbo Pascal. Place your cursor over the word in your editor and press *5*. If THELP knows about the word, you'll get help associated with it; otherwise, you'll get the THELP index screen. For example, type the word `Chr` into your editor. Leaving the cursor on the same line as *Chr*, press the THELP hot key, *5*. THELP displays the help screen for the function *Chr*, along with a highlighted link to *Ord*, the inverse function of *Chr*.

Place the cursor on a plus or equal sign before invoking THELP to get help on expressions, operators and operands. Place the cursor on **begin** for information on the structure of compound statements in Turbo Pascal. You'll find language help available for most reserved words: **program**, **var**, **type**, **const**, **procedure**, **function**, and so on. You can get help on Turbo Pascal Libraries (TPL's) and units by moving to the reserved word **uses** and pressing *5*.

If you don't want to type a keyword into your editor, you can simply press the *K* key while THELP is active. THELP will prompt you to enter any keyword you need help on. You can also jump to any help screen you know by page number (press *J*), or paste a highlighted word or an entire help screen into your editor (press *I* or *P*, respectively).

Alternatively, you can save the current help screen to a file called THELP.SAV—simply press *S*. If the file already exists, the screen will be appended to the end of THELP.SAV, making it easy to create custom help documents.

For more details about THELP, its command keys, and its command-line options, see Appendix C, "Turbo Pascal Utilities."

Windows and Menus

When you load Turbo Pascal (by typing `TURBO` and pressing *Enter* at the DOS prompt), the program's first screen includes the main screen and product version information (pressing *Shift-F10* at any time will bring up this infor-

mation). When you press any key, the version information disappears, but the windowed environment remains.

Note: If you are using a laptop computer and have difficulty reading text displayed in the integrated development environment, use the TINST utility to change the Mode for Display setting on the TINST main menu to LCD or Composite. This setting will force the integrated environment to use black and white colors. Using the TINST Mode for Display command to customize your system display is also recommended if you have a color graphics adapter and a monochrome or composite monitor. (Refer to page 316 for more information about the Mode for Display command.)

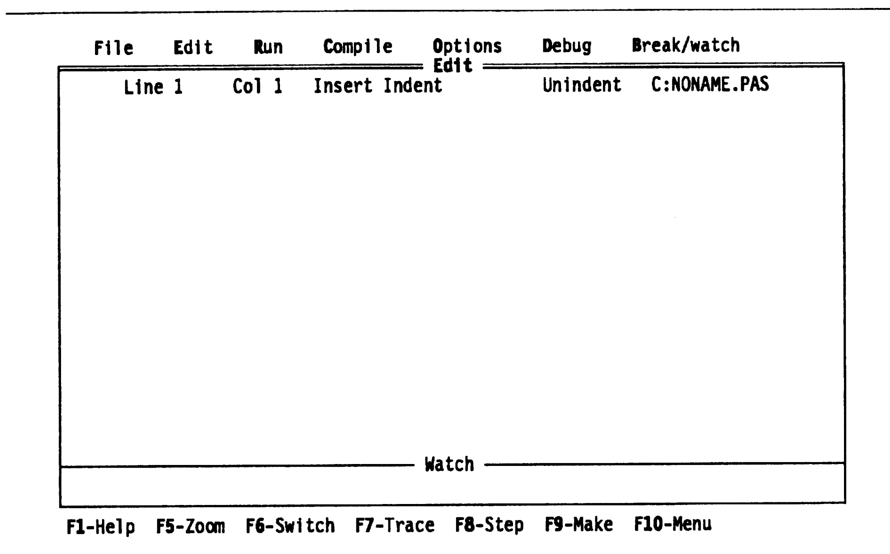


Figure 2.1: The Turbo Pascal Main Menu Screen

Look closely at the screen; it consists of four parts:

- the main menu, from which pull-down menus of commands can be invoked
- the Edit window
- the Watch window
- the *bottom* line (the line at the bottom of the screen that indicates which *hot keys* are currently active)

To introduce you to the Turbo Pascal IDE, here are some navigating basics:

- Type the highlighted capital letter to choose a menu command or use the arrow keys to move to the command and press *Enter*.
- Press *Esc* to leave a menu.
- Press *F10* to toggle between the menu system and the active window.
- If you are in the main menu or a pull-down menu, press *Esc* to go to the previously active window. (When a window is active, it will have a double bar at its top, and its name will be highlighted.)
- Use the *Right* and *Left arrow* keys to move from one pull-down menu to another.

From anywhere in Turbo Pascal:

- Press *F1* to get online help about your current position in the environment (how to use a menu item, what editing commands are available, prompt boxes, and so on).
- Pressing *Alt* plus the first letter of any main menu command (*F, E, R, C, O,D,* or *B*) invokes the command specified. For example, from anywhere in the system, pressing *Alt-E* takes you to the Edit window; pressing *Alt-F* takes you to the File menu.
- Press *F5* to zoom/unzoom the active window.
- Press *F6* to switch windows.
- Press *Alt-F6* to switch the contents of a window. When the Edit window is active, *Alt-F6* switches between the last file and the current file; when the other window (Watch or Output) is active, *Alt-F6* switches between the Watch and Output windows.

To exit Turbo Pascal and return to DOS, press *Alt-X* or go to the File menu and choose Quit (press *Q* or move the highlight bar to Quit and press *Enter*). If you choose Quit without saving your current work file, the editor will ask whether you want to save the file before exiting.

See Chapter 7, "All About the Integrated Environment," for more details on the way windows and menus work in the IDE.

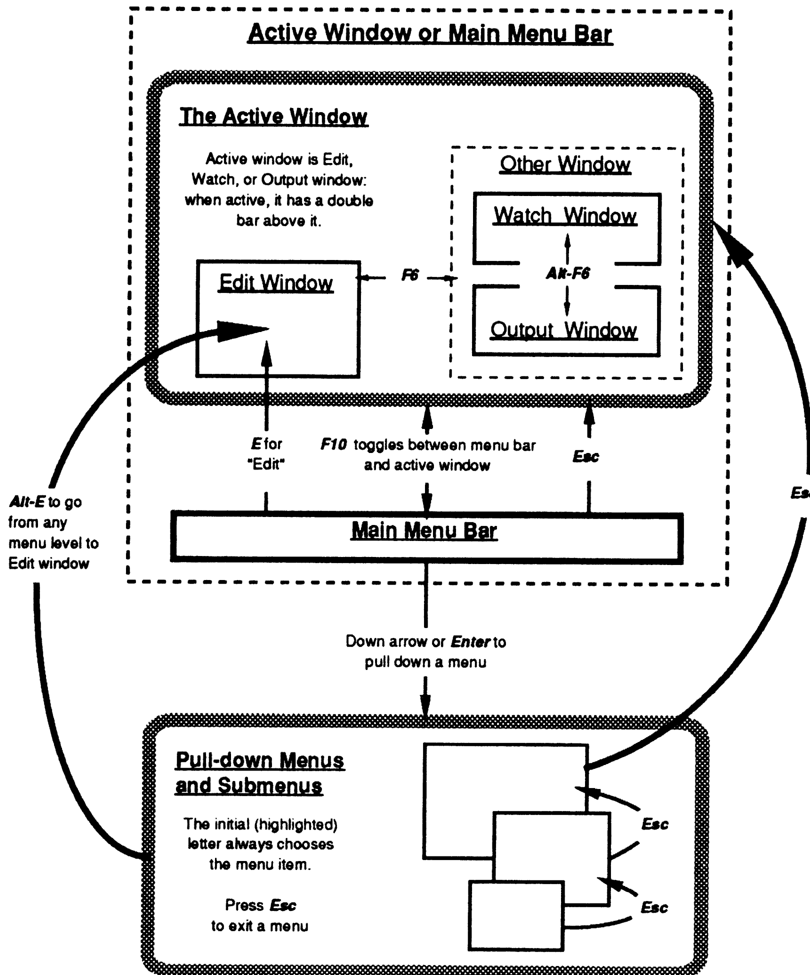
Using Hot Keys

Since there are number of menu items you'll be using again and again, we've provided you with hot keys (shortcuts) to use. *Hot keys* are keys set up to perform a certain function. For example, pressing *Alt* and the first letter of a main menu command will take you to the specified option's menu or perform an action; these are all considered hot keys. The only

other *Alt*/first-letter command is *Alt-X*, which is really just a shortcut for **File/Quit**.

In general, hot keys work from anywhere in the IDE; but there are two exceptions:

- Hot keys are disabled in error boxes and verify boxes. In these cases, you are required to press the key specified.
- You can override the hot keys with editor commands by using TINST. This means that, while you are in the editor, the hot key will behave as an edit command; when you are not in the editor, the hot key will work as originally defined. For example, if you define *Alt-R* to be *PgUp* in the editor, then it will not bring up the **Run** menu when you choose it from within the editor. So you must exit the editor (*F10* or *F6*) before *Alt-R* will bring up the **Run** menu. This gives you the flexibility to define the keys you prefer to use when editing. (Refer to Appendix D, "Customizing Turbo Pascal," for a complete discussion of redefining the editor keys.)



From anywhere in Turbo Pascal 5.0, ***Alt*** plus the first letter of any main menu name (F, E, R, C, O, D, or B) invokes that item, and ***F1*** calls up context-sensitive help information. Press ***Alt*** and hold to see a list of Alt-key shortcuts on the bottom line.

Figure 2.2: A Sample Use of Hot Keys

Table 2.1 lists all the hot keys you can use in the IDE. Remember that when these keys are pressed, their specific function is carried out no matter where you are in the integrated environment.

Table 2.1: Turbo Pascal's Hot Keys

Key(s)	Function	Menu Equivalent
F1	Calls up context-sensitive help	
F2	Saves the file currently in the editor	File/Save
F3	Lets you load a file (an input box will appear)	File/Load
F4	Executes to the cursor location	Run/Go to Cursor
F5	Zooms and unzooms the active window	
F6	Switches the active window	
F7	Traces into subroutines	Run/Trace Into
F8	Steps over subroutine calls	Run/Step Over
F9	Performs a "Make"	Compile/Make
F10	Toggles between main menu and active window	
Alt-F1	Calls up last help screen you were reading	
Alt-F3	Lets you pick a file to load	File/Pick
Alt-F5	Switches to the User screen	Run/User Screen
Alt-F6	Switches the contents of the active window	
Alt-F9	Compiles your program	Compile/Compile
Alt-B	Takes you to the Break/Watch menu	
Alt-C	Takes you to the Compile menu	
Alt-D	Takes you to the Debug menu	
Alt-E	Puts you in the Edit window	
Alt-F	Takes you to the File menu	
Alt-O	Takes you to the Options menu	
Alt-R	Takes you to the Run menu	
Alt-X	Quits Turbo Pascal and takes you to DOS	File/Quit
Ctrl-F1	Gives language help while in the editor	
Ctrl-F2	Terminates a debugging session	Run/Program Reset
Ctrl-F3	Displays call stack when debugging	Debug/Call Stack
Ctrl-F4	Evaluates or modifies a variable	Debug/Evaluate
Ctrl-F7	Adds an expression to the Watch window	B/Add Watch
Ctrl-F8	Toggles breakpoint	B/Toggle Breakpoint
Ctrl-F9	Runs your program	Run/Run
Shift-F10	Displays the version screen	

In this book, we often refer to menu items by an abbreviated name. The abbreviated name for a given menu item is represented by the sequence of letters you press to get to that item from the main menu. For example, at the main menu, the menu offering compile-time options related to memory sizes is Options/Compiler/Memory Sizes; we'll tell you to choose O/C/Memory Sizes (press *O C M*). In addition, we often give the hot key for a menu command in parentheses after the menu command name.

If you feel like you need more help using the IDE, look at Chapter 7, "All About the Integrated Environment," in this manual. If you're comfortable with what you've learned to date, let's get on to actually writing some programs in Turbo Pascal.

Loading Turbo Pascal

If you're using a floppy disk drive, put your Turbo Pascal system disk into Drive A and type the following command at the system prompt:

```
A>TURBO
```

and press *Enter*. This runs the program TURBO.EXE, which brings up the IDE, placing you in the main menu.

If you're using a hard disk, get into the Turbo Pascal subdirectory you created with INSTALL in Chapter 1 (the default is C:\TP) and run TURBO.EXE by typing

```
TURBO
```

at the `C:\TP>` prompt. Now you're ready to write your first Turbo Pascal program.

Creating Your First Program

When you first get into Turbo Pascal, you're placed at the main menu. Press *E* to get to the Edit window (or you can use the arrow keys to move the highlight bar and press *Enter* when positioned at the Edit command). You'll be placed in the editor with the cursor in the upper left-hand corner. You can then start typing in the following program, pressing *Enter* at the end of each line:

```
program MyFirst;
var
  A,B   : integer;
  Ratio : real;
begin
  Write('Enter two numbers: ');
  Readln(A,B);
  Ratio := A / B;
  Writeln('The ratio is ',Ratio);
  Write('Press <Enter>...');
  Readln;
end.
```

Place semicolons exactly where they appear in the manual, and make sure the last **end** is followed by a period. Use the *Backspace* key to make deletions, and use the arrow keys to move around in the Edit window.

If you're unfamiliar with editing commands, Appendix B discusses all the ones at your disposal.

Analyzing Your First Program

While you can type in and run this program without ever knowing how it works, we've provided a brief explanation here. The first line you entered gives the program the name *MyFirst*. This is an optional statement, but it's a good practice to include it.

The next three lines declare some *variables*, with the word **var** signaling the start of variable declarations. *A* and *B* are declared to be of type integer; that is, they can contain whole numbers, such as 52, –421, 0, 32,283, and so on. *Ratio* is declared to be of type real, which means it can hold fractional numbers such as 423.328 and –0.032, in addition to all integer values.

The rest of the program contains the *statements* to be executed. The word **begin** signals the start of the program. The statements are separated by semicolons and contain instructions to write to the screen (*Write* and *Writeln*), to read from the keyboard (*Readln*), and to perform calculations (*Ratio* := *A* / *B*). The *Readln* at the end of the program will cause execution to pause (until you press *Enter*) so you can inspect the program's output. The program's execution starts with the first instruction after **begin** and continues until **end.** is encountered.

Saving Your First Program

After entering your first program, it's a good idea to save it to disk. To do this, choose the Save command from the File menu. Press *F10* (or *Ctrl-K D*) to get out of the Edit window and invoke the main menu. Then press *F* to bring up the File menu and *S* to choose the Save command. By default, your file will have been given the name NONAME.PAS. You can rename it now by typing in `MYFIRST.PAS`, or just `MYFIRST` (the IDE will provide a .PAS extension by default) and then pressing *Enter*. Any time you choose File/ Save after that, your program will be saved as MYFIRST.PAS.

An alternate method of saving your program uses the hot key for File/ Save, *F2*. As when you chose File/Save, you'll be queried whether you want to save this file as NONAME.PAS. Enter the name `MYFIRST` as your file name and the IDE will add a .PAS extension.

Compiling Your First Program

To compile your first program, go to the main menu; if you're still in the Edit window, press *F10* (or *Ctrl-K D*) to do so. Press *C* to bring up the Compile menu, then *C* again to choose the Compile command from that menu; the

hot key for this command is *Alt-F9*. (The Compile menu has several options; see "The Compile Menu" on page 174 for details.)

Turbo Pascal compiles your program, changing it from Pascal (which you can read) to 8086 machine code for the microprocessor (which your PC can execute). You don't see the 8086 machine code; it's stored in memory (or on disk).

Like English, Pascal has rules of grammar you must follow. However, unlike English, Pascal's structure isn't lenient enough to allow for slang or poor syntax—the compiler must always understand what you mean. In Pascal, when you don't use the appropriate words or symbols in a statement or when you organize them incorrectly, you will get a compile-time (syntax) error.

What compile-time errors are you likely to get? Probably the most common error novice Pascal programmers will get is

```
Unknown identifier
```

or

```
';'expected
```

Pascal requires that you declare all variables, data types, constants, and subroutines—in short, all identifiers—before using them. If you refer to an undeclared identifier or if you misspell it, you'll get an error. Other common errors are unmatched **begin..end** pairs, assignment of incompatible data types (such as assigning reals to integers), parameter count and type mismatches in procedure and function calls, and so on.

When you start compiling, a box appears in the middle of the screen, giving information about the compilation taking place. A message flashes across the box to press *Ctrl-Break* to quit compilation. If no errors occurred during compilation, the message `Success: Press any key` flashes across the box. The box remains visible until you press a key. See how fast that went?

If an error occurs during compilation, Turbo Pascal stops, positions the cursor at the point of error in the editor, and displays an error message at the top of the editor, as it does with compile-time error messages. (The first key you press will clear the error message, and *Ctrl-Q W* will bring it back until you change files or recompile. Make the correction, save the updated file, and compile it again.)

Running Your First Program

After you've fixed any typing errors that might have occurred, go to the main menu and choose **Run** to bring up the **Run** menu. Choose the **Run** command from that menu (or press *Ctrl-F9*).

You're placed at the User screen, and the message

```
Enter two numbers:
```

appears on the screen. Type in any two integers (whole numbers), with a space between them, and press *Enter*. The following message will appear:

```
The ratio is
```

followed by the ratio of the first number to the second. On the next line the message `Press <Enter>...` will appear and the program will wait for you to press the *Enter* key.

Once your program has finished running, you're returned to the Edit window. To review your program output, choose **Run/User** Screen (or press *Alt-F5*).

If an error occurs while your program is executing, you'll get a message on the screen that looks like this:

```
Run-time error &lt;errnum&gt; at &lt;segment&gt;:&lt;offset&gt;
```

where *<errnum>* is the appropriate error number (see Appendix D in the *Reference Guide*, "Error Messages and Codes," for information on compiler and run-time error messages), and *<segment>:<offset>* is the memory address where the error occurred. (If you need this number later, look for it in the Output window.) You'll be positioned at the point of error in your program with a descriptive error message displayed on the editor status line. While the message is still on the editor status line, you can press *F1* to get help with that particular error. Any other keystroke clears the error message. If you need to find the error location again, press *Ctrl-Q W* or choose the **F**ind Error command from the **C**ompile menu.

When your program has finished executing, you're returned to the place in your program where you started. You can now modify your program if you wish. If you choose the **Run/Run** command before you make any changes to your program, Turbo Pascal immediately executes it again, without recompiling.

Once you're back in the IDE after executing your program, you can view your program's output by choosing the **Run/U**ser Screen command (or by pressing *Alt-F5*). Choose it again to return to the Turbo Pascal environment.

Checking the Files You've Created

If you exit Turbo Pascal (choose **Quit** from the **File** menu), you can see a directory listing of the source (Pascal) file you've created. To exit, press *O* (for **OS Shell**) in the **File** menu or, alternatively, press *Q* (for **Quit**) and type the following command at the DOS prompt:

```
DIR MYFIRST.*
```

You'll get a listing that looks something like this:

```
MYFIRST    PAS      217  8-10-88   11:07a
```

The file MYFIRST.PAS contains the Pascal program you just wrote. If you saved the program while you were typing, you'll also see a backup file MYFIRST.BAK, which was created automatically by the editor.

(**Note:** You'll only see the executable file if you've changed your default **Destination** setting in the **Compile** menu to Disk. You would then produce a file called MYFIRST.EXE, which would contain the machine code that Turbo Pascal generated from your program. You could then execute that program by typing `MYFIRST` followed by *Enter* at the DOS system prompt.)

Stepping Up: Your Second Program

Now you're going to write a second program, building upon the first. If you exited from Turbo Pascal using the **OS Shell** command from the **File** menu, you can return to the Turbo Pascal environment by typing `EXIT` at the DOS prompt. If you exited using **Quit** from the **File** menu, you would type

```
TURBO MYFIRST.PAS
```

at the prompt in order to return to the IDE. This will place you directly into the editor. Now, modify your MYFIRST.PAS program to look like this:

```
program MySecond;
var
  A,B     : integer;
  Ratio   : real;
begin
  repeat
    Write('Enter two numbers:  ');
    Readln(A,B);
    Ratio := A / B;
    Writeln('The ratio is ',Ratio:8:2);
    Write('Press <Enter>...');
    Readln
```

```
  until B = 0
end.
```

You want to save this as a separate program, so go to the main menu (press *F10*), choose the **File** menu (press *F*), and then **Write To** (press *W*). When prompted for a new name, type `MYSECOND.PAS` and press *Enter*.

Now here's a shortcut: To compile and run your second program, just press *Ctrl-F9*. This is the same as choosing the **Run/Run** command. It tells Turbo Pascal to run your updated program. And since you've made changes to the program, Turbo Pascal knows to compile the program before running it.

A major change has been made to the program: The statements have been enclosed in the **repeat..until** loop. This causes all the statements between **repeat** and **until** to be executed until the expression following **until** is True. A test is made to see if *B* has a value of zero or not. If *B* has a value of zero, then the loop should exit.

Run your program, try out some values, then enter `1 0` and press *Enter*. Your program does exit, but not quite in the way desired: It exits with a *run-time error*. You're placed back in the editor, with the cursor in front of the line

```
Ratio := A/B;
```

and the message

```
Error 200: Division by zero
```

at the top of the Edit window.

Debugging Your Program

If you've programmed before, you may recognize this error and how to fix it. But let's take this opportunity to show you how to use the *integrated debugger* that's built into Turbo Pascal 5.0.

A "bug" is an error in your program; a "debugger" is a utility to remove bugs from your program. Turbo Pascal's integrated debugger allows you to step through your code one line at a time. At the same time, you can watch your variables to see how their values change.

To start the debugging session, choose the **Run/Trace** Into command (or press *F7*). If your program needs to be recompiled, Turbo Pascal will do so. The first statement (**begin** in this case) in the main body of your program is highlighted; from now on we'll call this highlighted bar the *execution bar*.

The first *F7* you pressed initiated the debugging session. Now press *F7* to begin executing the program. The debugger just executed the invisible

startup code. The next executable line in this program is the *Write* statement on line 7.

Press *F7* again. Your screen blinks momentarily, then shows your program with the execution bar on the second statement (*Readln*). What's happening here is that Turbo Pascal switches to the User screen (where your program is executed and its output displayed), executes your first statement (a *Write* statement), then goes back to the editing screen.

Press *F7* again. This time, the User screen comes up and stays there. That's because a *Readln* statement is waiting for you to enter two numbers. Type two integer numbers, separated by a space; be sure the second number isn't a zero. Now press *Enter*. You're back at the Edit window, with the execution bar on the assignment statement on line 9.

Press *F7* and execute the assignment statement. Now the execution bar is on the *Writeln* statement on line 10. Press *F7* twice. Now you're about to execute *Readln* on line 12. Press *F7*, inspect your program's output, and then press *Enter*.

The execution bar is on the **until** clause. Press *F7* one more time, and you're back at the top of the **repeat** loop.

Instead of racing through one program statement after another, the integrated debugger lets you step through your code one line at a time. This is a powerful tool, and we'll go into a more detailed discussion of debugging in Chapter 6. First, we'll give you a quick taste of debugging by tracking down that divide-by-zero error.

Using the Watch Window

Let's take a look at the values of the variables you've declared. Press *Alt-B* to bring up the Break/Watch menu. Choose the Add Watch command; also note that you can invoke that command directly by pressing *Ctrl-F7*. When the Add Watch box comes up, type `A` and press *Enter*. You'll see that *A* has now appeared in the Watch window, along with its current value. Now use the Add Watch command to add *B* and *Ratio* to the Watch window. Finally, use it to add the expression *A/B* to the Watch window. Note that the Watch window grows in size as you add variables and expressions. Also note that you can use the Options/Environment/Zoom Windows command (hot key *F5*) to make the Watch window appear and disappear.

Choose Run/Trace Into (or press *F7*) to step through your program. This time, when you have to enter two numbers, enter `0` for the second number. When you press *Enter* and return to the IDE, look at the expression *A/B*. Instead of having a value after it, it has the phrase `Invalid floating-point`

operation; that's because dividing by zero is undefined. Note, though, that having this expression in your Watch window doesn't cause the program to stop with an error. Instead, the error is reported to you and the debugger does not perform the division in the Watch window.

Now press *F7* again, assigning *A/B* to *Ratio*. At this point, your program does halt, and the error message Division by zero appears at the top of the Edit window again.

Fixing Your Second Program

Now you probably have a good idea of what's wrong with your program: If you enter a value of zero for the second number (*B*), the program halts with a run-time error.

How do you fix it? If *B* has a value of zero, don't divide *B* into *A*. Edit your program so that it looks like this:

```
program MySecond;
var
  A,B     : integer;
  Ratio   : real;
begin
  repeat
    Write('Enter two numbers:  ');
    Readln(A,B);
    if B = 0 then
      Writeln('The ratio is undefined')
    else
    begin
      Ratio := A / B;
      Writeln('The ratio is ',Ratio:8:2);
    end;
    Write('Press <Enter>...');
    Readln
  until B = 0
end.
```

Now run your program (either by itself, or using the debugger). If you do use the debugger, note how the values in the Watch window change as you step through the program. When you're ready to stop, enter 0 for *B*. The program will pause after printing the message The ratio is undefined. Press <Enter>....

Now you have an idea just how powerful the debugger is. You can step through your program line by line; you can display the value of your

program's variables and expressions, and you can watch the values change as your program runs.

Programming Pizazz: Your Third Program

For the last program, let's get a little fancy and dabble in graphics. This program assumes that you have a graphics adapter for your system, and that you are currently set up to use that adapter. If you are in doubt, try the program and see what happens. If an error message appears, then you probably don't have a graphics adapter (or you have one that's not supported by our *Graph* unit). In any case, pressing *Enter* once should get you back to the IDE.

At the main menu, choose File/Load. Enter the file name MYTHIRD.PAS at the prompt, and you'll be placed in the editor. Here's the program to enter:

```
program MyThird;
uses
  Graph;
const
  Start  =  25;
  Finish = 175;
  Step   =   2;
var
  GraphDriver : integer;               { Stores graphics driver number }
  GraphMode   : integer;          { Stores graphics mode for the driver }
  ErrorCode   : integer;                 { Reports an error condition }
  X1,Y1,X2,Y2 : integer;
begin
  GraphDriver := Detect;               { Try to autodetect Graphics card }
  InitGraph(GraphDriver, GraphMode, '');
  ErrorCode := GraphResult;
  if ErrorCode <> grOk then                                   { Error? }
  begin
    Writeln('Graphics error: ', GraphErrorMsg(ErrorCode));
    Writeln('(You probably don''t have a graphics card!)');
    Writeln('Program aborted...');
    Readln;
    Halt(1);
  end;
  Y1 := Start;
  Y2 := Finish;
  X1 := Start;
  while X1 <= Finish do
```

```
    begin
      X2 := (Start+Finish) - X1;
      Line(X1, Y1, X2, Y2);
      X1 := X1 + Step;
    end;
    X1 := Start;
    X2 := Finish;
    Y1 := Start;
    while Y1 <= Finish do
    begin
      Y2 := (Start+Finish) - Y1;
      Line(X1, Y1, X2, Y2);
      Y1 := Y1 + Step;
    end;
    OutText('Press <Enter> to quit:');
    Readln;
    CloseGraph;
  end. { MyThird }
```

After you finish typing in this program, choose File/Save (*F2*) to save it and then choose **Compile/Compile** (*Alt-F9*) to compile. If you have no errors during compilation, choose **Run/Run** (or press *Ctrl-F9*) to run it. This program produces a square with some wavy patterns along the edges. When execution is over, you'll be returned to your program.

The **uses** clause says that the program uses a unit named *Graph*. A *unit* is a library, or collection, of subroutines (procedures and functions) and other declarations. In this case, the unit *Graph* contains the routines you want to use: *InitGraph*, *Line*, and *CloseGraph*.

The section labeled **const** defines three numeric constants—*Start*, *Finish*, and *Step*—that affect the size, location, and appearance of the square. By changing their values, you can change how the square looks.

Warning: Don't set *Step* to anything less than 1; if you do, the program will get stuck in what is known as an *infinite loop* (a loop that circles endlessly). If you've compiled to disk and are running the .EXE from DOS, you won't be able to exit except by pressing *Ctrl-Alt-Del* or by turning your PC off. If you're running from inside the IDE, you can interrupt the program by pressing *Ctrl-Break*.

The variables *X1*, *Y1*, *X2*, and *Y2* hold the values of locations along opposite sides of the square. The square itself is drawn by drawing a straight line from *X1,Y1* to *X2,Y2*. The coordinates are then changed, and the next line drawn. The coordinates always start out in opposite corners: The first line drawn goes from (25,25) to (175,175).

The program itself consists primarily of two loops. The first loop draws a line from (25,25) to (175,175). It then moves the *X* (horizontal) coordinates

by two, so that the next line goes from (27,25) to (173,175). This continues until the loop draws a line from (175,25) to (25,175).

The program then goes into its second loop, which pursues a similar course, changing the *Y* (vertical) coordinates by two each time. The routine *Line* is from the *Graph* unit and draws a line between the endpoints given.

The final *Readln* statement causes the program to wait for you to press a key before it goes back into text mode and returns to the IDE.

You might want to step through this program line-by-line using the integrated debugger and then watch it swap back and forth between the program's graphics mode and the IDE's text mode.

The Turbo Pascal Compiler

You now know how to enter, compile, debug, and run your programs. And because of Turbo Pascal's method of locating errors and its fast compilation, the cycle of entering, testing, and correcting your program takes little time. Let's look at the different aspects of that cycle in more detail.

So, What's a Compiler Anyway?

Your PC, like most microcomputers, has a central processing unit (CPU) that is the workhorse of the machine. On your PC, the CPU is a single chip from a "family" of chips: the iAPx86, a series of microprocessors designed by Intel. The actual chip in your machine could be an 8088, an 8086, an 80186, an 80286, or even an 80386; it doesn't matter, since the code Turbo Pascal produces will run on all of them.

The iAPx86 family has a set of binary-coded instructions that all the chips can execute. By giving the iAPx86 the right set of instructions, you can make it put text on the screen, perform math, move text and data around, draw pictures—in short, do all the things you want it to do. These instructions are known collectively as *machine code.*

Since machine code consists of pure binary information, it's neither easy to write nor easy to read. You can use a program known as an *assembler* to write machine-level instructions in a form that you can read, which means you would then be programming in assembly language. However, you would still have to understand how the iAPx86 microprocessors work. You'd also find that to perform simple operations—such as printing out a number—often requires a large number of instructions.

If you don't want to deal with machine code or assembly language, you use a high-level language such as Pascal. You can easily read and write programs in Pascal because it is designed for humans, not computers. Still, the PC understands only machine code. The Turbo Pascal compiler translates (or compiles) your Pascal program into instructions that the computer can understand. The compiler is just another program that moves data around; in this case, it reads in your program text and writes out the corresponding machine code.

What Gets Compiled?

You can only edit one Turbo Pascal program at a time and—unless you're working with Include files or writing your own units—that's the only program that would be compiled. So when you choose the Compile, Make, or Build commands from the Compile menu, or the Run command from the Run menu, Turbo Pascal compiles the text you're currently editing and produces an .EXE file, a .TPU file, or code in memory.

There are two exceptions to this rule. First, you can specify a *primary file*, using the Primary File command in the Compile menu. Once you've done that, the primary file will be compiled if you Make or Build, but the edit file will be compiled if you choose Compile.

Second, you can ask Turbo Pascal to recompile any units that the program you're compiling might use. You actually have two options here:

1. You can tell Turbo Pascal to recompile any units that have been changed since the last time you compiled your program. This is called a "make."
2. You can tell Turbo Pascal to recompile all units that your program uses. This is called a "build."

Where's the Code?

When you use the Run/Run command, Turbo Pascal (by default) saves the resulting machine code in memory (RAM). This has several advantages. First, the compiler runs much faster because it takes less time to write the machine code out to RAM than to a floppy or hard disk. Second, since your program is already loaded into RAM, Turbo Pascal doesn't have to load it when you run it. Third, the PC more easily returns to Turbo Pascal once your program stops executing, since Turbo Pascal also stays in RAM the whole time.

If compiling to RAM is so wonderful, why wouldn't you want to do it every time? Two reasons. First, because the resulting machine code is never

saved on disk, you could only run your programs from inside Turbo Pascal. There would be no way to execute your program from MS-DOS, nor would you be able to distribute your program.

The second problem is memory—you might not have enough. This could happen if your system doesn't have much memory, if your program is very large, or if your program needs a lot of memory for data.

It's easy to produce an .EXE file (application) you can run from outside Turbo Pascal: Choose the **D**estination command from the **C**ompile menu. This option allows you to toggle between Disk and Memory for the destination of compiled machine code. If you choose Disk and then recompile, Turbo Pascal produces a code file that you can run from MS-DOS by typing its name at the prompt.

The newly produced file has the same name as your source file but with the extension .EXE; for example, the resulting code file of a program named MYFIRST.PAS would be MYFIRST.EXE.

Regardless of whether you are compiling to disk or to memory, the **R**un command still executes the resulting program once the compilation is done.

Compile, Make, and Build

The **C**ompile menu has many options, three of which are compilation commands: **C**ompile, **M**ake, and **B**uild. All three take a source file and produce an .EXE file (if **D**estination is set to Disk) or a .TPU file. Let's look at the differences between them:

- **Compile** always compiles the file in the editor.
- **Make** checks to see whether you have specified a primary file. Once it has determined whether you have, it checks the time and date of the .PAS and .TPU (precompiled unit) files for every unit referenced in the **uses** clause (if there is one) in the program being compiled. (A *unit* is a collection of constants, data types, variables, and procedures and functions; see Chapter 4, "Units and Related Mysteries," for more information.) If the .PAS file has been modified since the corresponding .TPU file was created, then Turbo Pascal will automatically recompile that unit's .PAS file, creating a new .TPU file. Turbo Pascal also recompiles any unit that uses a unit whose interface has changed, whose Include files have been changed, or that links an .OBJ file modified since the unit's .TPU file was built. In short, Turbo Pascal ensures that all the units your program depends on are up to date. Once it's done that, Turbo Pascal compiles and links your program, producing an .EXE file.

- **Build** acts just like the Make command, with one important exception: It recompiles all units used by your program (and all units used by those units, and so on), regardless of whether they are current.
- If Make or Build cannot find the .PAS file corresponding to a given unit, then the unit is considered valid. That way, if your program uses any of the standard units, Turbo Pascal won't try to recompile them.

C H A P T E R

3

Programming in Turbo Pascal

The Pascal language was designed by Niklaus Wirth in the early 1970s to teach programming. Because of that, it's particularly well-suited as a first programming language. And if you've already programmed in other languages, you'll find it easy to pick up Pascal.

To get you started on the road to Pascal programming, in this chapter we'll teach you the basic elements of the Pascal language, and show you how to use them in your programs. Of course, we won't cover everything about programming in Pascal in this chapter. So if you're a Pascal novice, your best bet would be to pick up a copy of the *Turbo Pascal Tutor*, a complete book-plus-disk tutorial about programming in Pascal and using version 5.0 of Turbo Pascal.

Before you work through this chapter, you might want to read Chapter 7, "All About the Integrated Environment," and Appendix B, "Using the Editor," to learn about the menus and text editor in Turbo Pascal. If you haven't already installed Turbo Pascal (made a working copy of your Turbo Pascal disk or copied the files onto your hard disk) as described in Chapter 1, you can do so now. Make sure you've created the file TURBO.TP or installed the .EXE file using TINST.EXE (see Appendix D), otherwise, Turbo Pascal won't know the location of the standard units in TURBO.TPL and the configuration file. (Unless you happen to own MS-DOS 3.x and you have those files in the same directory as TURBO.EXE.)

Once you've done all that, get ready to learn about programming in Turbo Pascal.

The Seven Basic Elements of Programming

Most programs are designed to solve a problem. They solve problems by manipulating information or data. What you as the programmer have to do is

- get the information into the program—input.
- have a place to keep it—data.
- give the right instructions to manipulate the data—operations.
- be able to get the data back out of the program to the user (you, usually)—output.

You can organize your instructions so that

- some are executed only when a specific condition (or set of conditions) is True—conditional execution.
- others are repeated a number of times—loops.
- others are broken off into chunks that can be executed at different locations in your program—subroutines.

We've just described the seven basic elements of programming: *input, data, operations, output, conditional execution, loops,* and *subroutines.* This list is not comprehensive, but it does describe those elements that programs (and programming languages) usually have in common.

Many programming languages, including Pascal, have additional features as well. And when you want to learn a new language quickly, you can find out how that language implements these seven elements, then build from there. Here's a brief description of each element:

Input
This means reading values in from the keyboard, from a disk, or from an I/O port.

Data
These are constants, variables, and structures that contain numbers (integer and real), text (characters and strings), or addresses (of variables and structures).

Operations
These assign one value to another, combine values (add, divide, and so forth), and compare values (equal, not equal, and so on).

Output
This means writing information to the screen, to a disk, or to an I/O port.

Conditional Execution
This refers to executing a set of instructions if a specified condition is True (and skipping them or executing a different set if it is False) or if a data item has a specified value or range of values.

Loops
These execute a set of instructions some fixed number of times, while some condition is True or until some condition is True.

Subroutines
These are separately named sets of instructions that can be executed anywhere in the program just by referencing the name.

Now we'll take a look at how to use these elements in Turbo Pascal.

Data

Data Types

When you write a program, you're working with information that generally falls into one of five basic types: *integers, real numbers, characters* and *strings, boolean,* and *pointers.*

Integers are the whole numbers you learned to count with (1, 5, –21, and 752, for example).

Real numbers have fractional portions (3.14159) and exponents (2.579×10^{24}). These are also sometimes known as *floating-point* numbers.

Characters are any of the letters of the alphabet, symbols, and the numbers 0-9. They can be used individually (*a, Z,* !, 3) or combined into character strings ('This is only a test.').

Boolean expressions have one of two possible values: True or False. They are used in conditional expressions, which we'll discuss later.

Pointers hold the address of some location in the computer's memory, which in turn holds information.

Integer Data Types

Standard Pascal defines the data type integer as consisting of the values ranging from –*MaxInt* through 0 to *MaxInt*, where *MaxInt* is the largest possible integer value allowed by the compiler you're using. Turbo Pascal supports type integer, defines *MaxInt* as equal to 32767, and allows the value –32768 as well. A variable of type integer occupies 2 bytes.

Turbo Pascal also defines a long integer constant, *MaxLongInt*, with a value of 2,147,483,647.

Turbo Pascal also supports four other integer data types, each of which has a different range of values. Table 3.1 shows all five integer types.

Table 3.1: Integer Data Types

Type	Range	Size in Bytes
byte	0..255	1
shortint	–128..127	1
integer	-32768..32767	2
word	0..65535	2
longint	–2147483648..2147483647	4

A final note: Turbo Pascal allows you to use hexadecimal (base-16) integer values. To specify a constant value as hexadecimal, place a dollar sign ($) in front of it; for example, $27 = 39 decimal.

Real Data Types

Standard Pascal defines the data type real as representing floating-point values consisting of a significand (fractional portion) multiplied by an exponent (power of 10). The number of digits (known as *significant digits*) in the significand and the range of values of the exponent are compiler-dependent. Turbo Pascal defines the type real as being 6 bytes in size, with 11 significant digits and an exponent range of 10^{-38} to 10^{38}.

Turbo Pascal also supports the IEEE Standard 754 for binary floating-point arithmetic. This includes the data types single, double, extended, and comp. Single uses 4 bytes, with 7 significant digits and an exponent range of 10^{-45} to 10^{38}; double uses 8 bytes, with 15 significant digits and an exponent range of 10^{-324} to 10^{308}; and extended uses 10 bytes, with 19 significant digits and an exponent range of 10^{-4951} to 10^{4931}.

If you have an 8087 math coprocessor and enable the numeric support compiler directive or environment option ({$*N*+}), Turbo Pascal generates the proper 8087 instructions to support these types and to perform all floating-point operations on the 8087.

If you don't have an 8087 chip, but you still want to use the IEEE Standard types, you can ask Turbo Pascal to *emulate* an 8087 chip, by enabling both the 8087 emulation and numeric processing compiler directives ({$*E*+} and {$*N*+}, respectively). Turbo Pascal then links in a special math library that performs floating-point functions just like an 8087 chip. And if the resulting

program is run on a computer with an 8087 chip, then the hardware is used instead of the library routines.

Table 3.2: Real Data Types

Type	Range	Significant Digits	Size in Bytes
real	2.9 × 10E-39 .. 1.7 × 10E38	11-12	6
single	1.5 × 10E-45 .. 3.4 × 10E38	7-8	4
double	5.0 × 10E-324 .. 1.7 × 10E308	15-16	8
extended	1.9 × 10E-4951 .. 1.1 × 10E4932	19-20	10
comp*	–2E+63+1..2E+63-1	19-20	8

* comp only holds integer values.

Get into the Turbo Pascal editor and enter the following program:

```
program DoRatio;
var
  A,B   : integer;
  Ratio : real;
begin
  Write('Enter two numbers:  ');
  Readln(A,B);
  Ratio := A div B;
  Writeln('The ratio is ',Ratio)
end.
```

Save this as DORATIO.PAS by bringing up the main menu and selecting the **File/Write to** command. Then press *Alt-R* to compile and run the program. Enter two values (such as 10 and 3) and note the result (3.000000).

You were probably expecting an answer of 3.3333333333, and instead you received a 3. That's because you used the **div** operator, which performs integer division. Now go back and change the div statement to read as follows:

```
Ratio := A / B;
```

Save the code (press *F2*), then compile and run. The result is now 3.3333333333, as you expected. Using the division operator (/) gives you the most precise result—a real number.

Character and String Data Types

You've learned how to store numbers in Pascal, now how about characters and strings? Pascal offers a predefined data type char that is 1 byte in size and holds exactly one character. Character constants are represented by surrounding the character with single quotes (for example, 'A', 'e', '?', '2').

Note that '2' means the *character* 2, while 2 means the *integer* 2 (and 2.0 means the *real number* 2).

Here's a modification of DORATIO that allows you to repeat it several times (this also uses a **repeat..until** loop, which we'll discuss a little later):

```
program DoRatio;
var
  A,B   : integer;
  Ratio : real;
  Ans   : char;
begin
  repeat
    Write('Enter two numbers:  ');
    Readln(A,B);
    Ratio := A / B;
    Writeln('The ratio is ',Ratio);
    Write('Do it again? (Y/N)  ');
    Readln(Ans)
  until UpCase(Ans) = 'N'
end.
```

After calculating the ratio once, the program writes the message

```
Do it again? (Y/N)
```

and waits for you to type in a single character, followed by pressing *Enter*. If you type in a lowercase *n* or an uppercase *N*, the **until** condition is met and the loop ends; otherwise, the program goes back to the **repeat** statement and starts over again.

Note that *n* is *not* the same as *N*. This is because they have different ASCII code values. Characters are represented by the ASCII code: Each character has its own 8-bit number (characters take up 1 byte, remember). Appendix C in the *Reference Guide*, "Reference Materials," lists the ASCII codes for all characters.

Turbo Pascal gives you two additional ways of representing character constants: with a caret (^) or a number symbol (#). First, the characters with codes 0 through 31 are known as *control characters* (because historically they were used to control teletype operations). They are referred to by their abbreviations (CR for carriage return, LF for linefeed, ESC for escape, and so on) or by the word *Ctrl* followed by a corresponding letter (meaning the letter produced by adding 64 to the control code). For example, the control character with ASCII code 7 is known as BEL or *Ctrl-G*. Turbo Pascal lets you represent these characters using the caret (^), followed by the corresponding letter (or character). Thus, ^G is a legal representation in your program of *Ctrl-G*, and you could write statements such as *Writeln(^G)*,

causing your computer to beep at you. This method, however, only works for the control characters.

You can also represent *any* character using the number symbol (#), followed by the character's ASCII code. Thus, #7 would be the same as ^G, #65 would be the same as *'A'*, and #233 would represent one of the special IBM PC graphics characters.

Individual characters are nice, but what about strings of characters? After all, that's how you will most often use them. Standard Pascal does not support a separate string data type, but Turbo Pascal does. Take a look at this program:

```
program Hello;
var
  Name  : string[30];
begin
  Write('What is your name?  ');
  Readln(Name);
  Writeln('Hello, ',Name)
end.
```

This declares the variable *Name* to be of type **string**, with space set aside to hold 30 characters. One more byte is set aside internally by Turbo Pascal to hold the current length of the string. That way, no matter how long or short the name is you enter at the prompt, that is exactly how much is printed out in the *Writeln* statement. Unless, of course, you enter a name more than 30 characters long, in which case only the first 30 characters are used, and the rest are ignored.

When you declare a string variable, you can specify how many characters (up to 255) it can hold. Or you can declare a variable (or parameter) to be of type **string** with no length mentioned, in which case the default size of 255 characters is assumed.

Turbo Pascal offers a number of predefined procedures and functions to use with strings; they can be found in Chapter 16 of the *Reference Guide*, "Turbo Pascal Reference Lookup."

Boolean Data Type

Pascal's predefined data type boolean has two possible values: True and False. You can declare a variable to be of type boolean, then assign the variable either a True or False value or (more importantly) an expression that resolves to one of those two values.

A *Boolean expression* is simply an expression that is either True or False. It is made up of relational expressions, Boolean operators, Boolean variables,

and/or other Boolean expressions. For example, the following **while** statement contains a Boolean expression:

```
while (Index <= Limit) and not Done do ...
```

The Boolean expression consists of everything between the keywords **while** and **do**, and presumes that *Done* is a variable (or possibly a function) of type boolean.

Pointer Data Type

All the data types we've discussed until now hold just that—data. A *pointer* holds a different type of information—an address. A pointer is a variable that contains the address in memory (RAM) where some data is stored, rather than the data itself. In other words, it *points* to the data, like an address book or an index.

A pointer is usually (but not necesarily) specific to some other data type. Consider the following declarations:

```
type
  Buffer  = string[255];
  BufPtr  = ^Buffer;
var
  Buf1    : Buffer;
  Buf2    : BufPtr;
```

The data type *Buffer* is now just another name for **string**[255], while the type *BufPtr* defines a pointer to a *Buffer*. The variable *Buf1* is of type *Buffer;* it takes up 256 bytes of memory. The variable *Buf2* is of type *BufPtr;* it contains a 32-bit address and takes up only 4 bytes of memory.

Where does *Buf2* point? Nowhere, currently. Before you can use *BufPtr*, you need to set aside (allocate) some memory and store its address in *Buf2*. You do that using the *New* procedure:

```
New(Buf2);
```

Since *Buf2* points to the type *Buffer*, this statement creates a 256-byte buffer somewhere in memory, then puts its address into *Buf2*.

How do you use the data pointed to by *Buf2*? Via the indirection operator ^. For example, suppose you want to store a string in both *Buf1* and the buffer pointed to by *Buf2*. Here's what the statements would look like:

```
Buf1 := 'This string gets stored in Buf1.'
Buf2^ := 'This string gets stored where Buf2 points.'
```

Note the difference between *Buf2* and *Buf2^*. *Buf2* refers to a 4-byte pointer variable; *Buf2^* refers to a 256-byte string variable whose address is stored in *Buf2*.

How do you free up the memory pointed to by *Buf2*? Using the *Dispose* procedure. *Dispose* makes the memory available for other uses. After you use *Dispose* on a pointer, it's good practice to assign the (predefined) value **nil** to that pointer. That lets you know that the pointer no longer points to anything:

```
Dispose(Buf2);
Buf2 := nil;
```

Note that you assign **nil** to *Buf2*, not to *Buf2^*.

This ends our brief discussion on pointers; a good Pascal text will tell you how and when they're useful.

Identifiers

Up until now, we've given names to variables without worrying about what restrictions there might be. Let's talk about those restrictions now.

The names you give to constants, data types, variables, and functions are known as *identifiers*. Some of the identifiers used so far include

integer, real, string	Predefined data types
Hello,DoSum,DoRatio	Programs
Name, A, B, Sum, Ratio	User-defined variables
Write,Writeln,Readln	Predeclared procedures

Turbo Pascal has a few rules about identifiers; here's a quick summary:

- All identifiers must start with a letter or underscore (*a...z*, *A...Z*, or _). The rest of an identifier can consist of letters, underscores, and/or digits (0...9); no other characters are allowed.
- Identifiers are *case-insensitive*, which means that lowercase letters (*a...z*) are considered the same as uppercase letters (*A...Z*). For example, the identifiers *indx*, *Indx*, and *INDX* are identical.
- Identifiers can be of any length, but only the first 63 characters are significant.

Operators

Once you get your data into the program (and into your variables), you'll probably want to manipulate it somehow, using the operators available to

you. There are eight operator types: assignment, arithmetic, bitwise, relational, logical, address, set, and string.

Most Pascal operators are *binary*, taking two operands; the rest are *unary*, taking only one operand. Binary operators use the usual algebraic form, for example, $a + b$. A unary operator always precedes its operand, for example, $-b$.

In more complex expressions, rules of precedence clarify the order in which operations are performed (see Table 3.3).

Table 3.3: Precedence of Operators

Operators	Precedence	Categories
@, not	First (high)	Unary operators
*, /, div, mod, and, shl, shr	Second	Multiplying operators
+,-, or, xor	Third	Adding operators
=, <>, <, >, <=, >=, in	Fourth (low)	Relational operators

Operations with equal precedence are normally performed from left to right, although the compiler may at times rearrange the operands to generate optimum code.

Sequences of operators of the same precedence are evaluated from left to right. Expressions within parentheses are evaluated first and independently of preceding or succeeding operators.

Assignment Operators

The most basic operation is *assignment* (that is, assigning a value to a variable), as in *Ratio* := *A* / *B*. In Pascal, the assignment symbol is a colon followed by an equal sign (:=). In the example given, the value of *A* / *B* on the right of the assignment symbol is assigned to the variable *Ratio* on the left.

Unary and Binary Arithmetic Operators

Pascal supports the usual set of binary arithmetic operators—they work with type integer and real values:

- Multiplication (*)
- Integer division (**div**)
- Real division (/)
- Modulus (**mod**)
- Addition (+)
- Subtraction (–)

Also, Turbo Pascal supports both *unary minus* (*a* + (-*b*)), which performs a *two's complement* evaluation, and *unary plus* (*a* + (+*b*)), which does nothing at all but is there for completeness.

Bitwise Operators

For bit-level operations, Pascal has the following operators:

- **shl** (shift left) — Shifts the bits left the indicated number of bits, filling at the right with 0's.
- **shr** (shift right) — Shifts the bits right the indicated number of bits, filling at the left with 0's.
- **and** — Performs a logical **and** on each corresponding pair of bits, returning 1 if both bits are 1, and 0 otherwise.
- **or** — Performs a logical **or** on each corresponding pair of bits, returning 0 if both bits are 0, and 1 otherwise.
- **xor** — Performs a logical, exclusive **or** on each corresponding pair of bits, returning 1 if the two bits are different from one another, and 0 otherwise.
- **not** — Performs a logical complement on each bit, changing each 0 to a 1, and vice versa.

These allow you to perform very low-level operations on type integer values.

Relational Operators

Relational operators allow you to compare two values, yielding a Boolean result of True or False. Here are the relational operators in Pascal:

Operator	Meaning
>	greater than
>=	greater than or equal to
<	less than
<=	less than or equal to
=	equal to
<>	not equal to
in	is a member of

Why would you want to know if something were True or False? Enter the following program:

```
program TestGreater;
var
  A,B   : integer;
  Test  : boolean;
begin
  Write('Enter two numbers:  ');
  Readln(A,B);
  Test := A > B;
  Writeln('A is greater than B', Test);
end.
```

This will print True if *A* is greater than *B* or False if *A* is less than or equal to *B*.

Logical Operators

There are four logical operators—**and**, **xor**, **or**, and **not**—which are similar to but not identical with the bitwise operators. These logical operators work with logical values (True and False), allowing you to combine relational expressions, Boolean variables, and Boolean expressions.

They differ from the corresponding bitwise operators in this manner:

- Logical operators always produce a result of either True or False (a Boolean value), while the bitwise operators do bit-by-bit operations on type integer values.
- You cannot combine boolean and integer-type expressions with these operators; in other words, the expression `Flag and Indx` is illegal if *Flag* is of type boolean, and *Indx* is of type integer (or vice versa).
- The logical operators **and** and **or** will short-circuit by default; **xor** and **not** will not. Suppose you have the expression `exp1 and exp2`. If *exp1* is False, then the entire expression is False, so *exp2* will never be evaluated. Likewise, given the expression `exp1 or exp2`, *exp2* will never be evaluated if *exp1* is True. You can force full Boolean expression using the {$B+} compiler directive or environment option.

Address Operators

Pascal supports two special address operators: the *address-of* operator (@) and the *indirection* operator (^).

The @ operator returns the address of a given variable; if *Sum* is a variable of type integer, then @Sum is the address (memory location) of that variable. Likewise, if *ChrPtr* is a pointer to type char, then *ChrPtr^* is the character to which *ChrPtr* points.

Set Operators

Set operators perform according to the rules of set logic. The set operators and operations include:

+	union
–	difference
*	intersection

String Operators

The only string operation is the + operator, which is used to concatenate two strings.

Output

It may seem funny to talk about output before input, but a program that doesn't output information isn't of much use. That output usually takes the form of information written to the screen (words and pictures), to a storage device (floppy or hard disk), or to an I/O port (serial or printer ports).

The Writeln Procedure

You've already used the most common output function in Pascal, the *Writeln* routine. The purpose of *Writeln* is to write information to the screen. Its format is both simple and flexible:

```
Writeln(item,item,...);
```

where each *item* is something you want to print to the screen. *item* can be a literal value, such as an integer or a real number (3, 42, –1732.3), a character ('a', 'Z'), a string ('Hello, world'), or a Boolean value (True). It can also be a

named constant, a variable, a dereferenced pointer, or a function call, as long as it yields a value that is of type integer, real, char, string, or boolean. All the items are printed on one line, in the order given. The cursor is then moved to the start of the next line. If you wish to leave the cursor after the last item on the same line, then use the statement

```
Write(item,item,...);
```

When the items in a *Writeln* statement are printed, blanks are *not* automatically inserted; if you want spaces between items, you'll have to put them there yourself, like this:

```
Writeln(item,' ',item,' ',...);
```

For example, the following statements produce the indicated output:

```
A := 1; B := 2; C := 3;
Name := 'Frank';
Writeln(A,B,C);                   123
Writeln(A,' ',B,' ',C);           1 2 3
Writeln('Hi',Name);               HiFrank
Writeln('Hi, ',Name,'.');         Hi, Frank.
```

You can also use *field-width specifiers* to define a field width for a given item. The format for this is

```
Writeln(item:width,...);
```

where *width* is an integer expression (literal, constant, variable, function call, or combination thereof) specifying the total width of the field in which *item* is written. For example, consider the following code and resulting output:

```
A := 10; B := 2; C := 100;
Writeln(A,B,C);                   102100
Writeln(A:2,B:2,C:2);             10 2100
Writeln(A:3,B:3,C:3);              10  2100
Writeln(A,B:2,C:4);               10 2 100
```

Note that the item is padded with leading blanks on the left to make it equal to the field width. The actual value is right-justified.

What if the field width is less than what is needed? In the second *Writeln* statement given earlier, C has a field width of 2 but has a value of 100 and needs a width of 3. As you can see by the output, Pascal simply expands the width to the minimum size needed.

This method works for all allowable items: integers, reals, characters, strings, and booleans. However, real numbers printed with the field-width specifier (or with none at all) come out in exponential form:

```
X := 421.53;
Writeln(X);                        4.2153000000E+02
Writeln(X:8);                      4.2E+02
```

Because of this, Pascal allows you to append a second field-width specifier: *item:width:digits*. This second value forces the real number to be printed out in fixed-point format and tells how many digits to place after the decimal point:

```
X := 421.53;
Writeln(X:6:2);                    421.53
Writeln(X:8:2);                      421.53
Writeln(X:8:4);                    421.5300
```

Input

Standard Pascal has two basic input functions, *Read* and *Readln,* which are used to read data from the keyboard. The general syntax is

```
Read(item,item,...);
```

or

```
Readln(item,item,...);
```

where each *item* is a variable of any integer, real, char, or string type. Numbers being input must be separated from other values by spaces or by pressing *Enter.*

Conditional Statements

There are times when you want to execute some portion of your program when a given condition is True or False, or when a particular value of a given expression is reached. Let's look at how to do this in Pascal.

The If Statement

Look again at the **if** statement in the previous examples; note that it can take the following generic format:

```
if expr
  then statement1
  else statement2
```

where *expr* is any Boolean expression (resolving to True or False), and *statement1* and *statement2* are legal Pascal statements. If *expr* is True, then *statement1* is executed; otherwise, *statement2* is executed.

We must explain two important points about **if/then/else** statements:

First, **else** *statement2* is optional; in other words, this is a valid **if** statement:

```
if expr
  then statement1
```

In this case, *statement1* is executed only if *expr* is True. If *expr* is False, then *statement1* is skipped, and the program continues.

Second, what if you want to execute more than one statement if a particular expression is True or False? You use a compound statement. A *compound statement* consists of the keyword **begin**, some number of statements separated by semicolons (;), and the keyword **end**.

The ratio example uses a single statement for the **if** clause

```
if B = 0.0 then
  Writeln('Division by zero is not allowed.')
```

and a compound statement for the **else** clause

```
else
begin
  Ratio = A / B;
  Writeln('The ratio is ',Ratio)
end;
```

You might also notice that the body of each program you've written is simply a compound statement followed by a period.

The Case Statement

This statement gives your program the power to choose from more than two alternatives without having to specify lots of **if** statements.

The **case** statement consists of an expression (the selector) and a list of statements, each preceded by a **case** label of the same type as the selector. It specifies that the one statement be executed whose **case** label is equal to the current value of the selector. If none of the **case** labels contain the value of the selector, then either no statement is executed or, optionally, the statements following the reserved word **else** are executed. (**else** is an extension to standard Pascal.)

A **case** label consists of any number of constants or subranges, separated by commas and followed by a colon; for example,

```
case BirdSight of
  'C', 'c' : Curlews := Curlews + 1;
  'H', 'h' : Herons  := Herons + 1;
  'E', 'e' : Egrets  := Egrets + 1;
  'T', 't' : Terns   := Terns + 1;
end;  { case }
```

A subrange is written as two constants separated by the subrange delimiter '..'. The constant type must match the selector type. The statement that follows the **case** label is executed if the selector's value equals one of the constants or if it lies within one of the subranges.

Loops

Just as there are statements (or groups of statements) that you want to execute conditionally, there are other statements that you may want to execute repeatedly. This kind of construct is known as a *loop*.

There are three basic kinds of loops: the **while** loop, the **repeat** loop, and the **for** loop. We'll cover them in that order.

The While Loop

You can use the **while** loop to test for something at the beginning of your loop. Enter the following program:

```
program Hello;
var
  Count : integer;
begin
  Count := 1;
  while Count <= 10 do
  begin
    Writeln('Hello and goodbye!');
    Inc(Count)
  end;
  Writeln('This is the end!')
end.
```

The first thing that happens when you run this program is that *Count* is set equal to 1, then you enter the **while** loop. This tests to see if *Count* is less than or equal to 10. *Count* is, so the loop's body (**begin..end**) is executed. This prints the message `Hello and goodbye!` to the screen, then increments *Count* by 1. *Count* is again tested, and the loop's body is executed once more. This continues as long as *Count* is less than or equal to 10 when it is

tested. Once *Count* reaches 11, the loop stops, and the string `This is the end!` is printed on the screen.

The format of the **while** statement is

```
while expr do statement
```

where *expr* is a Boolean expression, and *statement* is either a single or a compound statement.

The **while** loop evaluates *expr*. If it's True, then *statement* is executed, and *expr* is evaluated again. If *expr* is False, the **while** loop is finished and the program continues.

The Repeat..Until Loop

The second loop is the **repeat..until** loop, which we've seen in the program DORATIO.PAS:

```
program DoRatio;
var
  A,B   : integer;
  Ratio : real;
  Ans   : char;
begin
  repeat
    Write('Enter two numbers:  ');
    Readln(A,B);
    Ratio := A / B;
    Writeln('The ratio is ',Ratio);
    Write('Do it again? (Y/N)  ');
    Readln(Ans)
  until UpCase(Ans) = 'N'
end.
```

As described before, this program repeats until you answer *n* or *N* to the question `Do it again? (Y/N)`. In other words, everything between **repeat** and **until** is repeated until the expression following **until** is True.

Here's the generic format for the **repeat..until** loop:

```
repeat
  statement;
  statement;
  ...
  statement
until expr
```

There are three major differences between the **while** loop and the **repeat** loop. First, the statements in the **repeat** loop always execute at least once, because the test on *expr* is not made until after the **repeat** occurs. By contrast, the **while** loop will skip over its body if the expression is initially False.

Next, the **repeat** loop executes *until* the expression is True, where the **while** loop executes *while* the expression is True. This means that care must be taken in translating from one type of loop to the other. For example, here's the HELLO program rewritten using a **repeat** loop:

```
program Hello;
var
  Count : integer;
begin
  Count := 1;
  repeat
    Writeln('Hello and goodbye!');
    Inc(Count)
  until Count > 10;
  Writeln('This is the end!')
end.
```

Note that the test is now *Count > 10*, where for the *while* loop it was *Count <= 10*.

Finally, the **repeat** loop can hold multiple statements without using a compound statement. Notice that you didn't have to use **begin..end** in the preceding program, but you did for the earlier version using a **while** loop.

Again, be careful to note that the **repeat** loop will always execute at least once. A **while** loop may or may not execute, depending on the value of the expression.

The For Loop

The **for** loop is the one found in most major programming languages, including Pascal. However, the Pascal version is simultaneously limited and powerful.

Basically, the **for** loop executes a set of statements some fixed number of times while a variable (known as the *index variable*) steps through a range of values. To see how this works, modify the earlier HELLO program to read as follows:

```
program Hello;
var
  Count : integer;
```

```
begin
  for Count := 1 to 10 do
    Writeln('Hello and goodbye!');
  Writeln('This is the end!')
end.
```

When you run this program, you can see that the loop works the same as the **while** and **repeat** loops already shown and, in fact, is precisely equivalent to the **while** loop. Here's the generic format of the **for** loop statement:

```
for index := expr1 to expr2 do statement
```

where *index* is a variable of some scalar type (any integer type, char, boolean, any enumerated type), *expr1* and *expr2* are expressions of some type compatible with *index*, and *statement* is a single or compound statement. *Index* is incremented by one after each time through the loop.

You can also decrement the index variable instead of incrementing it by replacing the keyword **to** with the keyword **downto**.

The **for** loop is equivalent to the following code:

```
index := expr1;
while index <= expr2 do
begin
  statement;
  Inc(index)
end;
```

The main drawback of the **for** loop is that it only allows you to increment or decrement by one. Its main advantages are conciseness and the ability to use char and enumerated types in the range of values.

Procedures and Functions

You've learned how to execute code *conditionally* and *iteratively*. Now, what if you want to perform the same set of instructions on different sets of data or at different locations in your program? Well, you simply put those statements into a *subroutine*, which you can then call as needed.

In Pascal, there are two types of subroutines: *procedures* and *functions*. The main difference between the two is that a function returns a value and can be used in expressions, like this:

```
X := Sin(A);
```

while a procedure is called to perform one or more tasks:

```
Writeln('This is a test');
```

However, before you learn any more about procedures and functions, you need to understand Pascal program structure.

Program Structure

In Standard Pascal, programs adhere to a rigid format:

```
program ProgName;
label
  labels;
const
  constant declarations;
type
  data type definitions;
var
  variable declarations;
procedures and functions;
begin
  main body of program
end.
```

You do not have to have all five declaration sections—**label**, **const**, **type**, **var**, and **procedures** and **functions**—in every program. But in standard Pascal, if they do appear, they must be in that order, and each section can appear only once. The declaration section is followed by any procedures and functions you might have, then finally the main body of the program, consisting of some number of statements.

Turbo Pascal gives you tremendous flexibility in your program structure. All it requires is that your program statement (if you have one) be first and that your main program body be last. Between those two, you can have as many declaration sections as you want, in any order you want, with procedures and functions freely mixed in. But identifiers must be defined before they are used, or else a compile-time error will occur.

Procedure and Function Structure

As mentioned earlier, procedures and functions—known collectively as *subprograms*—appear anywhere before the main body of the program. Procedures use this format:

```
procedure ProcName(parameters);
label
  labels;
const
  constant declarations;
```

```
type
  data type definitions;
var
  variable declarations;
procedures and functions;
begin
  main body of procedure;
end;
```

Functions look just like procedures except that a function declaration starts with a **function** header and ends with a data type for the return value of the function:

```
function FuncName(parameters) : data type;
```

As you can see, there are only two differences between this and regular program structure: Procedures or functions start with a **procedure** or **function** header instead of a **program** header, and they end with a semicolon instead of a period. A procedure or function can have its own constants, data types, and variables, and even its own procedures and functions. What's more, all these items can only be used with the procedure or function in which they are declared.

Sample Program

Here's a version of the DORATIO program that uses a procedure to get the two values, then uses a function to calculate the ratio:

```
program DoRatio;
var
  A,B   : integer;
  Ratio : real;
procedure GetData(var X,Y : integer);
begin
  Write('Enter two numbers:  ');
  Readln(X,Y)
end;

function GetRatio(I,J : integer) : real;
begin
  GetRatio := I/J
end;

begin
  GetData(A,B);
  Ratio := GetRatio(A,B);
  Writeln('The ratio is ',Ratio)
end.
```

This isn't exactly an improvement on the original program, being both larger and slower, but it does illustrate how procedures and functions work.

When you compile and run this program, execution starts with the first statement in the main body of the program: `GetData(A,B)`. This type of statement is known as a *procedure call*. Your program handles this call by executing the statements in *GetData*, replacing *X* and *Y* (known as *formal parameters*) with *A* and *B* (known as *actual parameters*). The keyword **var** in front of *X* and *Y* in *GetData*'s procedure statement says that the actual parameters must be variables and that the variable values can be changed and passed back to the caller. So you can't pass literals, constants, expressions, and so on to *GetData*. Once *GetData* is finished, execution returns to the main body of the program and continues with the statement following the call to *GetData*.

That next statement is a function call to *GetRatio*. Note that there are some key differences here. First, *GetRatio* returns a value, which must then be used somehow; in this case, it's assigned to *Ratio*. Second, a value is assigned to *GetRatio* in its main body; this is how a function determines what value to return. Third, there is no **var** keyword in front of the formal parameters *I* and *J*. This means that the actual parameters could be any two integer expressions, such as *Ratio := GetRatio(A + B,300)*; and that even if you change the values of the formal parameters in the **function** body, the new values will not be passed back to the caller. This, by the way, is *not* a distinction between procedures and functions; you can use both types of parameters with either type of subprogram.

Program Comments

Sometimes you want to insert notes into your program to remind you (or inform someone else) of what certain variables mean, what certain functions or statements do, and so on. These notes are known as *comments*. Pascal, like most other programming languages, lets you put as many comments as you want into your program.

You can start a comment with the left curly brace ({), which signals to the compiler to ignore everything until after it sees the right curly brace (}).

Comments can even extend across multiple lines, like this:

```
{ This is a long
  comment, extending
  over several lines. }
```

Pascal also allows an alternative form of comment, beginning with a left parenthesis and an asterisk, (*, and ending with an asterisk and a right parenthesis, *). This allows for a limited form of comment nesting, because a comment beginning with (* ignores all curly braces, and vice versa.

Now that we've gotten you off to a fine start, we recommend that you buy a good tutorial on Turbo Pascal (for instance, *Turbo Pascal Tutor*), and start writing some practice programs of your own.

C H A P T E R

4

Units and Related Mysteries

In Chapter 3, you learned how to write standard Pascal programs. What about non-standard programming—more specifically, PC-style programming, with screen control, DOS calls, and graphics? To write such programs, you have to understand units or understand the PC hardware enough to do the work yourself. This chapter explains what a unit is, how you use it, what predefined units are available, how to go about writing your own units, and how to compile them.

What's a Unit, Anyway?

Turbo Pascal gives you access to a large number of predefined constants, data types, variables, procedures, and functions. Some are specific to Turbo Pascal; others are specific to the IBM PC (and compatibles) or to MS-DOS. There are dozens of them, but you seldom use them all in a given program. Because of this, they are split into related groups called *units*. You can then use only the units your program needs.

A *unit* is a collection of constants, data types, variables, procedures, and functions. Each unit is almost like a separate Pascal program: It can have a main body that is called before your program starts and does whatever initialization is necessary. In short, a unit is a library of declarations you can pull into your program that allows your program to be split up and separately compiled.

All the declarations within a unit are usually related to one another. For example, the *Crt* unit contains all the declarations for screen-oriented routines on your PC.

Turbo Pascal provides eight standard units for your use. Six of them—*System, Overlay, Graph, Dos, Crt,* and *Printer*—provide support for your regular Turbo Pascal programs. The other two—*Turbo3* and *Graph3*—are designed to help maintain compatibility with programs and data files created under version 3.0 of Turbo Pascal. All but *Graph, Graph3,* and *Turbo3* are stored in the file TURBO.TPL. Some of these are explained more fully in Chapter 12 of the *Reference Guide,* "Standard Units," but we'll look at each one here and explain its general function.

A Unit's Structure

A unit provides a set of capabilities through procedures and functions—with supporting constants, data types, and variables—but it hides how those capabilities are actually implemented by separating the unit into two sections: the *interface* and the *implementation*. When a program uses a unit, all the unit's declarations become available, as if they had been defined within the program itself.

A unit's structure is not unlike that of a program, but with some significant differences. Here's a unit, for example:

```
unit <identifier>;
interface
uses <list of units>;   { Optional }
  { public declarations }
implementation
uses <list of units>;   { Optional }
  { private declarations }
  { implementation of procedures and functions }
begin
  { initialization code }
end.
```

The unit header starts with the reserved word **unit**, followed by the unit's name (an identifier), much the way a program begins. The next item in a unit is the keyword **interface**. This signals the start of the interface section of the unit—the section visible to any other units or programs that use this unit.

A unit can use other units by specifying them in a **uses** clause. The **uses** clause can appear in two places. First, it can appear immediately after the keyword **interface**. In this case, any constants or data types declared in the interfaces of those units can be used in any of the declarations in this unit's interface section.

Second, it can appear immediately after the keyword **implementation**. In this case, any declarations from those units can be used only within the implementation section. This also allows for *circular unit references*; you'll learn how to use these later in this chapter.

Interface Section

The interface portion—the "public" part—of a unit starts at the reserved word **interface**, which appears after the unit header and ends when the reserved word **implementation** is encountered. The interface determines what is "visible" to any program (or other unit) using that unit; any program using the unit has access to these "visible" items.

In the unit interface, you can declare constants, data types, variables, procedures, and functions. As with a program, these can be arranged in any order, and sections can repeat themselves (for example, **type** ... **var** ... *<procs>* ... **const** ... **type** ... **const** ... **var**).

The procedures and functions visible to any program using the unit are declared here, but their actual bodies—implementations—are found in the implementation section. **forward** declarations are neither necessary nor allowed. The bodies of all the regular procedures and functions are held in the implementation section after all the procedure and function headers have been listed in the interface section.

A **uses** clause may appear in the implementation. If present **uses** must immediately follow the keyword **implementation**.

Implementation Section

The implementation section—the "private" part—starts at the reserved word **implementation**. Everything declared in the interface portion is visible in the implementation: constants, types, variables, procedures, and functions. Furthermore, the implementation can have additional declarations of its own, although these are not visible to any programs using the unit. The program doesn't know they exist and can't reference or call them. However, these hidden items can be (and usually are) used by the "visible" procedures and functions—those routines whose headers appear in the interface section.

A **uses** clause may appear in the implementation. If present, **uses** must immediately follow the keyword implementation.

If any procedures have been declared external, one or more {$L *filename*} directive(s) should appear anywhere in the source file before the final **end** of the unit.

The normal procedures and functions declared in the interface—those that are not inline—must reappear in the implementation. The **procedure/ function** header that appears in the implementation should either be identical to that which appears in the interface or should be in the short form. For the short form, type in the keyword (**procedure** or **function**), followed by the routine's name (identifier). The routine will then contain all its local declarations (labels, constants, types, variables, and nested procedures and functions), followed by the main body of the routine itself. Say the following declarations appear in the interface of your unit:

```
procedure ISwap(var V1,V2 : integer);
function IMax(V1,V2 : integer) : integer;
```

The implementation could look like this:

```
procedure ISwap;
var
  Temp : integer;
begin
   Temp := V1; V1 := V2; V2 := Temp
end;  { of proc ISwap }
function IMax(V1,V2 : integer) : integer;
begin
  if V1 > V2 then
    IMax := V1
  else IMax := V2
end;  { of func IMax }
```

Routines local to the implementation (that is, not declared in the interface section) must have their complete **procedure/function** header intact.

Initialization Section

The entire implementation portion of the unit is normally bracketed within the reserved words **implementation** and **end**. However, if you put the reserved word **begin** before **end**, with statements between the two, the resulting compound statement—looking very much like the main body of a program—becomes the *initialization* section of the unit.

The initialization section is where you initialize any data structures (variables) that the unit uses or makes available (through the interface) to the program using it. You can use it to open files for the program to use later. For example, the standard unit *Printer* uses its initialization section to

make all the calls to open (for output) the text file *Lst*, which you can then use in your program's *Write* and *Writeln* statements.

When a program using that unit is executed, the unit's initialization section is called before the program's main body is run. If the program uses more than one unit, each unit's initialization section is called (in the order specified in the program's **uses** statement) before the program's main body is executed.

How Are Units Used?

The units your program uses have already been compiled and stored as machine code, not Pascal source code; they are not Include files. Even the interface section is stored in the special binary symbol table format that Turbo Pascal uses. Furthermore, certain standard units are stored in a special file (TURBO.TPL) and are automatically loaded into memory along with Turbo Pascal itself.

As a result, using a unit or several units adds very little time (typically less than a second) to the length of your program's compilation. If the units are being loaded in from a separate disk file, a few additional seconds may be required because of the time it takes to read from the disk.

As stated earlier, to use a specific unit or collection of units, you must place a **uses** clause at the start of your program, followed by a list of the unit names you want to use, separated by commas:

```
program MyProg;
uses thisUnit,thatUnit,theOtherUnit;
```

When the compiler sees this **uses** clause, it adds the interface information in each unit to the symbol table and links the machine code that is the implementation to the program itself.

The ordering of units in the **uses** clause is not important. If *thisUnit* uses *thatUnit* or vice versa, you can declare them in either order, and the compiler will determine which unit must be linked into MyProg first. In fact, if *thisUnit* uses *thatUnit* but MyProg doesn't need to directly call any of the routines in *thatUnit*, you can "hide" the routines in *thatUnit* by omitting it from the **uses** clause:

```
unit thisUnit;
uses thatUnit;
...
program MyProg;
uses thisUnit, theOtherUnit;
...
```

In this example, *thisUnit* can call any of the routines in *thatUnit*, and MyProg can call any of the routines in *thisUnit* or *theOtherUnit*. MyProg cannot, however, call any of the routines in *thatUnit* because *thatUnit* does not appear in MyProg's **uses** clause.

If you don't put a **uses** clause in your program, Turbo Pascal links in the *System* standard unit anyway. This unit provides some of the standard Pascal routines as well as a number of Turbo Pascal-specific routines.

Referencing Unit Declarations

Once you include a unit in your program, all the constants, data types, variables, procedures, and functions declared in that unit's interface become available to you. For example, suppose the following unit existed:

```
unit MyStuff;
interface
const
  MyValue  = 915;
type
  MyStars  = (Deneb,Antares,Betelgeuse);
var
  MyWord  : string[20];
procedure SetMyWord(Star : MyStars);
function  TheAnswer : integer;
implementation
...
end.
```

What you see here is the unit's interface, the portion that is visible to (and used by) your program. Given this, you might write the following program:

```
program TestStuff;
uses MyStuff;
var
  I     : integer;
  AStar : MyStars;
begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I)
end.
```

Now that you have included the statement **uses** *MyStuff* in your program, you can refer to all the identifiers declared in the interface section in the interface of *MyStuff* (*MyWord*, *MyValue*, and so on). However, consider the following situation:

```
program TestStuff;
uses MyStuff;
const
  MyValue   = 22;
var
  I      : integer;
  AStar  : MyStars;
function TheAnswer : integer;
begin
  TheAnswer := -1
end;
begin
  Writeln(MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
  Writeln(MyWord);
  I := TheAnswer;
  Writeln(I)
end.
```

This program redefines some of the identifiers declared in *MyStuff*. It will compile and run, but will use its own definitions for *MyValue* and *The-Answer*, since those were declared more recently than the ones in *MyStuff*.

You're probably wondering whether there's some way in this situation to still refer to the identifiers in *MyStuff*? Yes, preface each one with the identifier *MyStuff* and a period (.). For example, here's yet another version of the earlier program:

```
program TestStuff;
uses MyStuff;
const
  MyValue = 22;
var
  I     : integer;
  AStar : MyStars;
function TheAnswer : integer;
begin
  TheAnswer := -1;
end;
begin
  Writeln(MyStuff.MyValue);
  AStar := Deneb;
  SetMyWord(AStar);
```

```
  Writeln(MyWord);
  I := MyStuff.TheAnswer
  Writeln(I)
end.
```

This program will give you the same answers as the first one, even though you've redefined *MyValue* and *TheAnswer*. Indeed, it would have been perfectly legal (although rather wordy) to write the first program as follows:

```
program TestStuff;
uses MyStuff;
var
  I     : integer;
  AStar : MyStuff.MyStars;
begin
  Writeln(MyStuff.MyValue);
  AStar := MyStuff.Deneb;
  MyStuff.SetMyWord(AStar);
  Writeln(MyStuff.MyWord);
  I := MyStuff.TheAnswer;
  Writeln(I)
end.
```

Note that you can preface any identifier—constant, data type, variable, or subprogram—with the unit name.

Implementation Section Uses Clause

As of version 5.0, Turbo Pascal allows you to place a **uses** clause in a unit's implementation section. If present, the **uses** clause must immediately follow the **implementation** keyword, just like a **uses** clause in the interface section must immediately follow the **interface** keyword.

A **uses** clause in the implementation section allows you to further hide the inner details of a unit, since units used in the **implementation** section are not visible to users of the unit. More importantly, however, it also enables you to construct mutually dependent units.

Since units in Turbo Pascal need not be strictly hierarchical, you can make circular unit references. The next section provides an example that demonstrates the usefulness of circular references.

Circular Unit References

The following program demonstrates how two units can "use" each other. The main program, *Circular*, uses a unit named *Display*. *Display* contains one routine in its interface section, *WriteXY*, which takes three parameters: an (*x*, *y*) coordinate pair and a text message to display. If the (*x*, *y*) coordinates are onscreen, *WriteXY* moves the cursor to (*x*, *y*) and displays the message there; otherwise, it calls a simple error-handling routine.

So far, there's nothing fancy here—*WriteXY* is taking the place of *Write*. Here's where the circular unit reference enters in: How is the error-handling routine going to display its error message? By using *WriteXY* again. Thus you have *WriteXY*, which calls the error-handling routine *ShowError*, which in turn calls *WriteXY* to put an error message onscreen. If your head is spinning in circles, let's look at the source code to this example, so you can see that it's really not that tricky.

The main program, *Circular*, clears the screen and makes three calls to *WriteXY*:

```
program Circular;
{ Display text using WriteXY }

uses
  Crt, Display;

begin
  ClrScr;
  WriteXY(1, 1, 'Upper left corner of screen');
  WriteXY(100, 100, 'Way off the screen');
  WriteXY(81 - Length('Back to reality'), 15, 'Back to reality');
end.
```

Look at the (*x*, *y*) coordinates of the second call to *WriteXY*. It's hard to display text at (100, 100) on an 80x25 line screen. Next, let's see how *WriteXY* works. Here's the source to the *Display* unit, which contains the *WriteXY* procedure. If the (*x*, *y*) coordinates are valid, it displays the message; otherwise, *WriteXY* displays an error message:

```
unit Display;
{ Contains a simple video display routine }

interface

procedure WriteXY(X, Y : integer; Message : string);

implementation
uses
  Crt, Error;

procedure WriteXY(X, Y : integer; Message : string);
```

```
begin
  if (X in [1..80]) and (Y in [1..25]) then
  begin
    GoToXY(X, Y);
    Write(Message);
  end
  else
    ShowError('Invalid WriteXY coordinates');
end;

end.
```

The *ShowError* procedure called by *WriteXY* is declared in the following code in the *Error* unit. *ShowError* always displays its error message on the 25th line of the screen:

```
unit Error;
{ Contains a simple error-reporting routine }

interface

procedure ShowError(ErrMsg : string);

implementation

uses
  Display;

procedure ShowError(ErrMsg : string);
begin
  WriteXY(1, 25, 'Error: ' + ErrMsg);
end;

end.
```

Notice that the **uses** clause in the **implementation** sections of both *Display* and *Error* refer to each other. These two units can refer to each other in their **implementation** sections because Turbo Pascal can compile complete **interface** sections for both. In other words, the Turbo Pascal compiler will accept a reference to partially compiled unit *A* in the **implementation** section of unit *B*, as long as both *A* and *B*'s **interface** sections do not depend upon each other (and thus follow Pascal's strict rules for declaration order).

Sharing Other Declarations

What if you want to modify *WriteXY* and *ShowError* to take an additional parameter that specifies a rectangular window onscreen:

```
procedure WriteXY(SomeWindow : WindRec;
                        X, Y : integer;
                     Message : string);
```

```
procedure ShowError(SomeWindow : WindRec;
                     ErrMsg : string);
```

Remember these two procedures are in separate units. Even if you declared *WindData* in the **interface** of one, there would be no legal way to make that declaration available to the **interface** of the other. The solution is to declare a third module that contains only the definition of the window record:

```
unit WindData;
interface
type
  WinDRec = record
              X1, Y1, X2, Y2 : integer;
              ForeColor,
              BackColor      : byte;
              Active         : boolean;
            end;
implementation
end.
```

In addition to modifying the code to *WriteXY* and *ShowError* to make use of the new parameter, the **interface** sections of both the *Display* and *Error* units can now "use" *WindData*. This approach is legal because unit *WindData* has no dependencies in its **uses** clause, and units *Display* and *Error* refer to each other only in their respective **implementation** sections.

TURBO.TPL

The file TURBO.TPL contains all the standard units except *Graph* and the compatibility units (*Graph3* and *Turbo3*): *System, Overlay, Crt, Dos,* and *Printer*. These are units loaded into memory with Turbo Pascal; they're always readily available to you. You will normally keep the file TURBO.TPL in the same directory as TURBO.EXE (or TPC.EXE).

System

Units used: ***none***

System contains all the standard and built-in procedures and functions of Turbo Pascal. Every Turbo Pascal routine that is *not* part of standard Pascal and that is *not* in one of the other units is in *System*. This unit is always linked into every program.

Dos

Units used: ***none***

Dos defines numerous Pascal procedures and functions that are equivalent to the most commonly used DOS calls, such as *GetTime*, *SetTime*, *DiskSize*, and so on. It also defines two low-level routines, *MsDos* and *Intr*, which allow you to directly invoke any MS-DOS call or system interrupt. *Registers* is the data type for the parameter to *MsDos* and *Intr*. Some other constants and data types are also defined.

Overlay

Units used: ***none***

Overlay provides support for 5.0's powerful overlay system. Overlays are discussed in detail in Chapter 13 of the *Reference Guide*.

Crt

Units used: ***none***

Crt provides a set of PC-specific declarations for input and output: constants, variables, and routines. You can use these to manipulate your text screen (do windowing, direct cursor addressing, text color and background). You can also do "raw" input from the keyboard and control the PC's sound chip.

Printer

Units used: ***none***

Printer declares the text-file variable *Lst* and connects it to a device driver that (you guessed it) allows you to send standard Pascal output to the printer using *Write* and *Writeln*. For example, once you include *Printer* in your program, you could do the following:

```
Write(Lst,'The sum of ',A:4,' and ',B:4,' is ');
C := A + B;
Writeln(Lst,C:8);
```

Graph **Units used:** ***none***

The *Graph* unit is not built into TURBO.TPL, but instead resides on the same disk with the .BGI and .CHR support files. Place GRAPH.TPU in the current directory or use the unit directory to specify the full path to GRAPH.TPU. (If you have a hard disk and you used the INSTALL program, your system is already set up so you can use *Graph*.)

Graph supplies a set of fast, powerful graphics routines that allow you to make full use of the graphics capabilities of your PC. It implements the device-independent Borland graphics handler, allowing support of CGA, EGA, Hercules, AT &T 400, MCGA, 3270 PC, and VGA and 8514 graphics.

Graph3 **Units used:** ***Crt***

Graph3 supports the full set of graphics routines—basic, advanced, and turtlegraphics—from version 3.0. They are identical in name, parameters, and function to those in version 3.0. This unit is provided for backward compatibility only.

Turbo3 **Units used:** ***Crt***

This unit contains two variables and several procedures that are no longer supported by Turbo Pascal. These include the predefined file variable *Kbd*, the Boolean variable *CBreak*, and the original integer versions of *MemAvail* and *MaxAvail* (which return paragraphs free instead of bytes free, as the current versions do). This unit is provided for backward compatibility only.

Now that you've been introduced to units, let's see about writing your own.

Writing Your Own Units

Say you've written a unit called *IntLib*, stored it in a file called INTLIB.PAS, and compiled it to disk; the resulting code file will be called INTLIB.TPU. To use it in your program, you must include a **uses** statement to tell the compiler you're using that unit. Your program might look like this:

```
program MyProg;
uses IntLib;
```

Note that Turbo Pascal expects the unit code file to have the same name (up to eight characters) of the unit itself. If your unit name is *MyUtilities*, then Turbo is going to look for a file called MYUTILIT.PAS.

Compiling a Unit

You compile a unit exactly the way you'd compile a program: Write it using the editor and select the **C**ompile/**C**ompile command (or press *Alt-C*). However, instead of creating an .EXE file, Turbo Pascal will create a .TPU (**T**urbo **P**ascal **U**nit) file. You can then leave this file as is or merge it into TURBO.TPL using TPUMOVER.EXE (see Appendix C).

In any case, you probably want to move your .TPU files (along with their source) to the unit directory you specified with the **O**/**D**/**U**nit Directories command. That way, you can reference those files without having to have them in the current directory or in TURBO.TPL. (The **U**nit Directories command lets you give multiple directories for the compiler to search for in unit files.)

You can only have one unit in a given source file; compilation stops when the final **end** statement is encountered.

An Example

Okay, now let's write a small unit. We'll call it *IntLib* and put in two simple integer routines—a procedure and a function:

```
unit IntLib;
interface
procedure ISwap(var I,J : integer);
function  IMax(I,J : integer) : integer;
implementation
procedure ISwap;
var
  Temp  : integer;
begin
  Temp := I; I := J; J := Temp
end;  { of proc ISwap }
function IMax;
begin
  if I > J then
    IMax := I
  else IMax := J
end;  { of func IMax }
end.  { of unit IntLib }
```

Type this in, save it as the file INTLIB.PAS, then compile it to disk. The resulting unit code file is INTLIB.TPU. Move it to your unit directory (whatever that might happen to be) or leave it in the same directory as the program that follows. This next program uses the unit *IntLib*:

```
program IntTest;
uses IntLib;
var
  A,B  : integer;
begin
  Write('Enter two integer values:  ');
  Readln(A,B);
  ISwap(A,B);
  Writeln('A = ',A,' B = ',B);
  Writeln('The max is ',IMax(A,B));
end.  { of program IntTest }
```

Congratulations! You've just created your first unit and written a program that uses it!

Units and Large Programs

Up until now, you've probably thought of units only as libraries—collections of useful routines to be shared by several programs. Another function of a unit, however, is to break up a large program into modules.

Two aspects of Turbo Pascal make this modular functionality of units work: (1) its tremendous speed in compiling and linking and (2) its ability to manage several code files simultaneously, such as a program and several units.

Typically, a large program is divided into units that group procedures by their function. For instance, an editor application could be divided into initialization, printing, reading and writing files, formatting, and so on. Also, there could be a "global" unit—one used by all other units, as well as the main program—that defines global constants, data types, variables, procedures, and functions.

The skeleton of a large program might look like this:

```
program Editor;
uses
  Dos,Crt,Printer                    { Standard units from TURBO.TPL }
  EditGlobals,                                   { User-written units }
  EditInit,
  EditPrint,
  EditRead,EditWrite,
  EditFormat;

{ Program's declarations, procedures, and functions }
begin  { main program }
end.  { of program Editor }
```

Note that the units in this program could either be in TURBO.TPL or in their own individual .TPU files. If the latter is true, then Turbo Pascal will manage your project for you. This means when you recompile program *Editor* using the compiler's built-in make facility, Turbo Pascal will compare the dates of each .PAS and .TPU file and recompile modules whose source has been modified.

Another reason to use units in large programs has to do with code segment limitations. The 8086 (and related) processors limit the size of a given chunk, or segment, of code to 64K. This means that the main program and any given segment cannot exceed a 64K size. Turbo Pascal handles this by making each unit a separate code segment. Your upper limit is the amount of memory the machine and operating system can support—640K on most PCs. Without units, you're limited to 64K of code for your program. (See Chapter 5, "Project Management," for more information about how to deal with large programs.)

Units as Overlays

Sometimes, even the ability to have multiple units loaded isn't enough to solve your memory problems. You might not have 640K to work with, or you may need to have large amounts of data in memory at the same time. In other words, you just can't fit your entire program into memory at once.

Turbo Pascal offers a solution: *overlays*. An overlay is a chunk of program that is loaded into memory when needed, and unloaded when not. This allows you to bring in sections of a program only when you need them.

Overlays in Turbo Pascal are based on units: The smallest chunk of code that can be loaded or unloaded is an entire unit. You can define complex sets of overlays, specifying which units can or cannot be in memory at the same time. Best of all, with Turbo Pascal's intelligent overlay manager, you don't have to worry about loading or unloading the overlays yourself—it's all done automatically.

You'll learn more about overlays and how to set them up and use them in Chapter 13 of the *Reference Guide*, "Overlays."

TPUMOVER

Suppose you want to add a well-designed and thoroughly debugged unit to the library of standard units (TURBO.TPL) so that it's automatically loaded into memory when you run the compiler. Is there any way to add to TURBO.TPL? Yes, by using the TPUMOVER.EXE utility.

You can also use TPUMOVER to remove units from the Turbo Pascal standard unit library file, reducing its size and the amount of memory it takes up when loaded. (More details on using TPUMOVER can be found in Appendix C, "Turbo Pascal Utilities.")

As you've seen, it's really quite simple to write your own units. A well-designed, well-implemented unit simplifies program development; you solve the problems only once, not for each new program. Best of all, a unit provides a clean, simple mechanism for writing very large programs.

C H A P T E R

5

Project Management

So far, you've learned how to write Turbo Pascal programs, how to use the predefined units, and how to write your own units. At this point, your program has the capability of becoming large and separated into multiple source files. How do you manage such a program? This chapter suggests how to organize your program into units, how to take advantage of the built-in Make and Build options, how to use the stand-alone Make utility, how to use conditional compilation within a source code file, and how to optimize your code for speed.

Program Organization

Turbo Pascal 5.0 allows you to divide your program into code segments. Your main program is a single code segment, which means that after compilation, it can have no more than 64K of machine code. However, you can exceed this limit by breaking your program up into units. Each unit can also contain up to 64K of machine code when compiled. The question is how should you organize your program into units?

The first step is to collect all your global definitions—constants, data types, and variables—into a single unit; let's call it *MyGlobals*. This is necessary if your other units reference those definitions. Unlike include files, units can't "see" any definitions made in your main program; they can only see what's in the interface section of their own unit and other units they use. Your units can use *MyGlobals* and thus reference all your global declarations.

A second possible unit is *MyUtils*. In this unit you could collect all the utility routines used by the rest of your program. These would have to be

routines that don't depend on any others (except possibly other routines in *MyUtils*).

Beyond that, you should collect procedures and functions into logical groups. In each group, you'll often find a few procedures and functions that are called by the rest of the program, and then several (or many) procedures/functions that are called by those few. A group like that makes a wonderful unit. Here's how to convert it over:

- Copy all those procedures and functions into a separate file and delete them from your main program.
- Open that file for editing.
- Type the following lines in front of those procedures and functions:

  ```
  unit unitname;
  interface
  uses MyGlobals;
  implementation
  ```

  where *unitname* is the name of your unit (and also the name of the file you're editing).
- Type **end.** at the very end of the file.
- Into the space between **interface** and **implementation**, copy the procedure and function headers of those routines called by the rest of the program. Those headers are simply the first line of each routine, the one that starts with **procedure** (or **function**).
- If this unit needs to use any others, type their names (separated by commas) between *MyGlobals* and the semicolon in the **uses** statement.
- Compile the unit you've created.
- Go back to your main program and add the unit's name to the **uses** statement at the start of the program.

Ideally, you want your program organized so that when you are working on a particular aspect of it, you are modifying and recompiling a single module (unit or main program). This minimizes compile time; more importantly, it lets you work with smaller, more manageable chunks of code.

Initialization

Remember in all this that each unit can (optionally) have its own initialization code. This code is automatically executed when the program is first loaded. If your program uses several units, then the initialization code for each unit is executed. The order of execution follows the order in

which the units are listed in your program's **uses** statement; thus, if your program had the statement

```
uses MyGlobals,MyUtils,EditLib,GraphLib;
```

then the initialization section (if any) of *MyGlobals* would be called first, followed by that of *MyUtils,* then *EditLib,* then *GraphLib.*

To create an initialization section for a unit, put the keyword **begin** above the **end** that ends the implementation section. This defines the initialization section of your unit, much as the **begin..end** pair defines the main body of a program, a procedure, or a function. You can then put any Pascal code you want in here. It can reference everything declared in that unit, in both the public (interface) and private (implementation) sections; it can also reference anything from the interface portions of any units that this unit uses.

The Build and Make Options

Turbo Pascal has an important feature to aid you in project management: a built-in Make utility. To discuss its significance, let's look at the previous example again.

Suppose you have a program, MYAPP.PAS, which uses four units: *MyGlobals, MyUtils, EditLib,* and *GraphLib.* Those four units are contained in the four text files MYGLOBAL.PAS, MYUTILS.PAS, EDITLIB.PAS, and GRAPHLIB.PAS, respectively. Furthermore, *MyUtils* uses *MyGlobals,* and *EditLib* and *GraphLib* use both *MyGlobals* and *MyUtils.*

When you compile MYAPP.PAS, it looks for the files MYGLOBAL.TPU, MYUTILS.TPU, EDITLIB.TPU, and GRAPHLIB.TPU, loads them into memory, links them with the code produced by compiling MYAPP.PAS, and writes everything out to MYAPP.EXE (if you're compiling to disk). So far, so good.

Suppose now you make some modifications to EDITLIB.PAS. In order to recreate MYAPP.EXE, you need to recompile both EDITLIB.PAS and MYAPP.PAS. A little tedious, but no big problem.

Now, let's suppose you modify the interface section of MYGLOBAL.PAS. To update MYAPP.EXE, you have to recompile all four units, as well as MYAPP.PAS. That means five separate compilations each time you make a change to MYGLOBAL.PAS—which could be enough to discourage you from using units at all. But wait...

The Make Option

As you probably guessed, Turbo Pascal offers a solution. By using the **Make** option (in the **Compile** menu), you can get Turbo Pascal to do all the work for you. The process is simple: After making any changes to any units and/or the main program, just **Make** the main program.

Turbo Pascal then makes three kinds of checks.

- First, it checks and compares the date and time of the .TPU file for each unit used by the main program against the unit's corresponding .PAS file. If the .PAS file has been modified since the .TPU file was created, Turbo Pascal recompiles the .PAS file, creating an updated .TPU file. So, as in the first example, if you modified EDITLIB.PAS and then recompiled MYAPP.PAS (using the **Make** option), Turbo Pascal would automatically recompile EDITLIB.PAS before compiling MYAPP.PAS.
- The second check is to see if you changed the interface portion of the modified unit. If you did, then Turbo Pascal recompiles all other units using that unit.

  As in the second example, if you modified the interface portion of MYGLOBAL.PAS and then recompiled MYAPP.PAS, Turbo Pascal would automatically recompile MYGLOBAL.PAS, MYUTILS.PAS, EDITLIB.PAS, and GRAPHLIB.PAS (in that order) before compiling MYAPP.PAS. However, if you only modified the implementation portion, then the other dependent units don't need to be recompiled, since (as far as they're concerned) you didn't change that unit.
- The third check is to see if you changed any Include or .OBJ files (containing assembly language routines) used by any units. If a given .TPU file is older than any of the Include or .OBJ files it links in, then that unit is recompiled. That way, if you modify and assemble some routines used by a unit, that unit is automatically recompiled the next time you compile a program using that unit.

To use the **Make** option under the integrated environment, either select the **Make** command from the **Compile** menu, or press *F9*. To invoke it with the command-line compiler, use the option */M*. Note that the **Make** option has no effect on units found in TURBO.TPL.

The Build Option

The **Build** option is a special case of the **Make** option. When you compile a program using **Build**, it automatically recompiles *all* units used by that

program (except, of course, those units in TURBO.TPL). This always brings everything up to date.

To use the **Build** option under the integrated environment, choose the **Build** command from the Compile menu. To invoke it with the command-line compiler, use the option */B*.

The Stand-Alone Make Utility

Turbo Pascal places a great deal of power and flexibility at your fingertips. You can use it to manage large, complex programs that are built from numerous unit, source, and object files. And it can automatically perform a Build or a Make operation, recompiling units as needed. Understandably, though, Turbo Pascal has no mechanism for recreating .OBJ files from assembly code routines (.ASM files) that have changed. To do that, you need to use a separate assembler. The question then becomes, how do you keep your .ASM and .OBJ files updated?

The answer is simple: You use the MAKE utility that's included with Turbo Pascal. MAKE is an intelligent program manager that—given the proper instructions—does all the work necessary to keep your program up to date. In fact, MAKE can do far more than that. It can make backups, pull files out of different subdirectories, and even automatically run your programs should the data files that they use be modified. As you use MAKE more and more, you'll see new and different ways it can help you to manage your program development.

MAKE is a stand-alone utility; it is different from the **Make** and **Build** options that are part of both the integrated environment and the command-line compiler. Full documentation of MAKE is given in Appendix C, but we'll give an example here to show how you might use it.

A Quick Example

Suppose you're writing some programs to help you display information about nearby star systems. You have one program—GETSTARS.PAS—that reads in a text file listing star systems, does some processing on it, then produces a binary data file with the resulting information in it.

GETSTARS.PAS uses three units: STARDEFS.TPU, which contains the global definitions; STARLIB.TPU, which has certain utility routines; and STARPROC.TPU, which does the bulk of the processing. The source code for these units are found in STARDEFS.PAS, STARLIB.PAS, and STARPROC.PAS, respectively.

The next issue is dependencies. STARDEFS.PAS doesn't use any other units; STARLIB.PAS uses STARDEFS; STARPROC.PAS uses STARDEFS and STARLIB; and GETSTARS.PAS uses STARDEFS, STARLIB, and STARPROC.

Given that, to produce GETSTARS.EXE you would simply "make" GETSTARS.PAS. Turbo Pascal (in either the integrated environment or the command-line version) would recompile the units as needed.

Suppose now that you convert a number of the routines in STARLIB.PAS into assembly language, creating the files SLIB1.ASM and SLIB2.ASM, then use Turbo Assembler to create SLIB1.OBJ and SLIB2.OBJ. Each time STARLIB.PAS is compiled, it links in those .OBJ files. And, in fact, Turbo Pascal is smart enough to recompile STARLIB.PAS if STARLIB.TPU is older than either of those .OBJ files.

However, what if either .OBJ file is older than the .ASM file upon which it depends? That means that the particular .ASM file needs to be re-assembled. Turbo Pascal can't assemble those files for you, so what do you do?

You create a *make file* and let MAKE do the work for you. A make file consists of *dependencies* and *commands*. The dependencies tell MAKE which files a given file depends upon; the commands tell MAKE how to create that given file from the other ones.

Creating a Makefile

Your makefile for this project might look like this:

```
getstars.exe: getstars.pas stardefs.pas starlib.pas slib1.asm \
              slib2.asm slib1.obj slib2.obj

tpc getstars /m

slib1.obj: slib1.asm
  TASM slib1.asm slib1.obj

slib2.obj: slib2.asm
  TASM slib2.asm slib2.obj
```

Okay, so this looks a bit cryptic. Here's an explanation:

- The first two lines tell MAKE that GETSTARS.EXE depends on three Pascal, two assembly language, and two .OBJ files (the backslash at the end of line 1 tells MAKE to ignore the line break and continue the dependency definition on the next line).

- The third line tells MAKE how to build a new GETSTARS.EXE. Notice that it simply invokes the command-line compiler on GETSTARS.PAS and uses the built-in Turbo Pascal Make facility (*/M* option).
- The next two lines (ignoring the blank line) tell MAKE that SLIB1.OBJ depends on SLIB1.ASM and show MAKE how to build a new SLIB1.OBJ.
- Similarly, the last two lines define the dependencies (only one file, actually) and MAKE procedures for the file SLIB2.OBJ.

Using MAKE

Let's suppose you've created this Make file using the editor in the Turbo Pascal integrated environment (or any other ASCII editor) and saved it as the file STARS.MAK. You would then use it by issuing the command

```
make -fstars.mak
```

where `-f` is an option telling MAKE which file to use. MAKE works from the bottom of the file to the top. First, it checks to see if SLIB2.OBJ is older than SLIB2.ASM. If it is, then MAKE issues the command

```
TASM SLIB2.asm SLIB2.obj
```

which assembles SLIB2.ASM, creating a new version of SLIB2.OBJ. It then makes the same check on SLIB1.ASM and issues the same command if needed. Finally, it checks all of the dependencies for GETSTARS.EXE and, if necessary, issues the command

```
tpc getstars /m
```

The */M* option tells Turbo Pascal to use its own internal MAKE routines, which will then resolve all unit dependencies, including recompiling STARLIB.PAS if either SLIB1.OBJ or SLIB2.OBJ is newer than STARLIB.TPU.

This is only a simple example using MAKE; more complete documentation can be found in Appendix C.

Conditional Compilation

To make your job easier, Turbo Pascal version 5.0 offers conditional compilation. This means that you can now decide what portions of your program to compile based on options or defined symbols. For a complete reference to conditional directives, refer to Appendix B, "Compiler Directives," in the *Reference Guide*.

The conditional directives are similar in format to the compiler directives that you're accustomed to; in other words, they take the format

```
{$directive arg}
```

where *directive* is the directive (such as DEFINE, IFDEF, and so on), and *arg* is the argument, if any. Note that there *must* be a separator (blank, tab) between *directive* and *arg*. Table 5.1 lists all the conditional directives, with their meanings.

Table 5.1: Summary of Compiler Directives

{$DEFINE symbol}	Defines symbol for other directives
{$UNDEF symbol}	Removes definition of symbol
{$IFDEF symbol}	Compiles following code if symbol is defined
{$IFNDEF symbol}	Compiles following code if symbol is not defined
{$IFOPT x+}	Compiles following code if directive *x* is enabled
{$IFOPT x-}	Compiles following code if directive *x* is disabled
{$ELSE}	Compiles following code if previous IF*xxx* is not True
{$ENDIF}	Marks end of IF*xxx* or ELSE section

The DEFINE and UNDEF Directives

The IFDEF and IFNDEF directives test to see if a given symbol is defined. These symbols are defined using the DEFINE directive and undefined UNDEF directives. (You can also define symbols on the command line and in the integrated environment.)

To define a symbol, insert the directive

```
{$DEFINE symbol}
```

into your program. *symbol* follows the usual rules for identifiers as far as length, characters allowed, and other specifications. For example, you might write

```
{$DEFINE debug}
```

This defines the symbol *debug* for the remainder of module being compiled, or until the statement

```
{$UNDEF debug}
```

is encountered. As you might guess, UNDEF "undefines" a symbol. If the symbol isn't defined, then UNDEF has no effect at all.

Defining at the Command Line

If you're using the command-line version of Turbo Pascal (TPC.EXE), you can define conditional symbols on the command line itself. TPC accepts a */D* option, followed by a list of symbols separated by semicolons:

```
tpc myprog /Ddebug;test;dump
```

This would define the symbols *debug, test,* and *dump* for the program MYPROG.PAS. Note that the */D* option is cumulative, so that the following command line is equivalent to the previous one:

```
tpc myprog /Ddebug /Dtest /Ddump
```

Defining in the Integrated Environment

Conditional symbols can be defined by using the **O/C/**Conditional defines option. Multiple symbols can be defined by entering them in the input box, separated by semicolons. The syntax is the same as that of the command-line version.

Predefined Symbols

In addition to any symbols you define, you also can test certain symbols that Turbo Pascal has defined. Table 5.2 lists these symbols; let's look at each in a little more detail.

Table 5.2: Predefined Conditional Symbols

VER50	Always defined (TP 4.0 has VER40 defined, etc.)
MSDOS	Always defined
CPU86	Always defined
CPU87	Defined if an 8087 is present at compile time

The VER50 Symbol

The symbol *VER50* is always defined for Turbo Pascal version 5.0. In a similar fashion, *VER40* is defined for version 4.0 of Turbo Pascal. Future versions will have corresponding predefined symbols; for example, version 5.1 would have *VER51* defined, version 6.0 would have *VER60* defined, and so on. This will allow you to create source code files that can use future enhancements while maintaining compatibility with version 5.0.

The MSDOS and CPU86 Symbols

These symbols are always defined (at least for Turbo Pascal version 5.0 running under MS-DOS). The *MSDOS* symbol indicates you are compiling under the MS-DOS operating system. The *CPU86* symbol means you are compiling on a computer using an Intel iAPx86 (8088, 8086, 80186, 80286, 80386) processor.

As future versions of Turbo Pascal for other operating systems and processors become available, they will have similar symbols indicating which operating system and/or processor is being used. Using these symbols, you can create a single source code file for more than one operating system or hardware configuration.

The CPU87 Symbol

Turbo Pascal 5.0 supports floating-point operations in two ways: hardware and software. If you have an 80x87 math coprocessor installed in your computer system, you can use the IEEE floating-point types (single, double, extended, comp), and Turbo Pascal will produce direct calls to the math chip. If you don't have an 8087, you can still use the IEEE types by instructing Turbo Pascal to emulate the 8087 in software. Otherwise, you can just use the standard floating-point type real (6 bytes in size), and Turbo Pascal will support all your operations with software routines. Use the *$N* and *$E* directives to indicate which you wish to use.

When you load the Turbo Pascal compiler, it checks to see if an 80x87 chip is installed. If it is, then the *CPU87* symbol is defined; otherwise, it's undefined. You might then have the following code at the start of your program:

```
{$N+}                          { Always use IEEE floating point }
{$IFNDEF CPU87}                  { If there's no 80x87 present }
{$E+}                     { No hardware: Use emulation library }
{$ENDIF}
```

The IFxxx, ELSE, and ENDIF Symbols

The idea behind conditional directives is that you want to select some amount of source code to be compiled if a particular symbol is (or is not) defined or if a particular option is (or is not) enabled. The general format follows:

```
{$IFxxx}
  source code
{$ENDIF}
```

where *IFxxx* is IFDEF, IFNDEF, or IFOPT, followed by the appropriate argument, and *source code* is any amount of Turbo Pascal source code. If the expression in the IF*xxx* directive is True, then *source code* is compiled; otherwise, it is ignored as if it had been commented out of your program.

Quite often you have alternate chunks of source code. If the expression is True, you want one chunk compiled, and if it's False, you want the other one compiled. Turbo Pascal lets you do this with the $ELSE directive:

```
{$IFxxx}
  source code A
{$ELSE}
  source code B
{$ENDIF}
```

If the expression in *IFxxx* is True, then *source code A* is compiled, else *source code B* is compiled.

Note that all IF*xxx* directives must be completed within the same source file, which means they cannot start in one source file and end in another. However, an IF*xxx* directive can encompass an include file:

```
{$IFxxx}
{$I file1.pas}
{$ELSE}
{$I file2.pas}
{$ENDIF}
```

That way, you can select alternate include files based on some condition.

You can nest *IFxxx..ENDIF* constructs so that you can have something like this:

```
{$IFxxx}                                    { First IF directive }
...
{$IFxxx}                                   { Second IF directive }
...
{$ENDIF}                        { Terminates second IF directive }
...
{$ENDIF}                         { Terminates first IF directive }
```

Let's look at each of the *IFxxx* directives in more detail.

The IFDEF and IFNDEF Directives

You've learned how to define a symbol, and also that there are some predefined symbols. The IFDEF and IFNDEF directives let you conditionally compile code based on whether those symbols are defined or undefined. You saw this example earlier:

```
{$IFDEF CPU87}                          { If there's an 80x87 present }
{$N+,E-}                                { Then use the inline 8087 code }
{$ELSE}
{$N+,E+}                                { Else use the emulation library }
{$ENDIF}
```

By putting this in your program, you can automatically select the *$N* option if an 8087 math coprocessor is present when your program is compiled. That's an important point: This is a compile-time option. If there is an 8087 coprocessor in your machine when you compile, then your program will be compiled with the *$N+* and *E-* compiler directives, selecting direct calls to the 8087. Otherwise, it will be compiled with the *$N+* and *$E+* directives, using the software 8087 emulation. If you compile this program on a machine with an 8087, you can't run the resulting .EXE file on a machine without an 8087. (Of course, a program compiled using {*$N+*,*E+*} will run on any system and use emulation only if no 8087 hardware is detected.)

It is also common to use the IFDEF and IFNDEF directives to insert debugging information into your compiled code. For example, if you put the following code at the start of each unit:

```
{$IFDEF debug}
{$D+,L+}
{$ELSE}
{$D-,L-}
{$ENDIF}
```

and the following directive at the start of your program:

```
{$DEFINE debug}
```

and compile your program, then complete debugging information will be generated by the compiler for use with the integrated debugger or the stand-alone Turbo Debugger. In a similar fashion, you may have sections of code that you want compiled only if you are debugging; in that case, you would write

```
{$IFDEF debug}
  source code
{$ENDIF}
```

where *source code* will be compiled only if *debug* is defined at that point.

The IFOPT Directive

You may want to include or exclude code, depending upon which compiler options (range-checking, I/O-checking, numeric-processing, and so on)

have been selected. Turbo Pascal lets you do that with the IFOPT directive, which takes two forms:

```
{$IFOPT x+}
```

and

```
{$IFOPT x-}
```

where *x* is one of the compiler options: *A, B, D, E, F, I, L, N, O, R, S*, or *V* (see Appendix B in the *Reference Guide*, "Compiler Directives," for a complete description). With the first form, the following code is compiled if the compiler option is currently enabled; with the second, the code is compiled if the option is currently disabled. So, as an example, you could have the following:

```
var
  {$IFOPT N+}
    Radius,Circ,Area   : double;
  {$ELSE}
    Radius,Circ,Area   : real;
  {$ENDIF}
```

This selects the data type for the listed variables based on whether or not 8087 support is enabled.

An alternate example might be

```
Assign(F,Filename);
Reset(F);
{$IFOPT I-}
IOCheck;
{$ENDIF}
```

where *IOCheck* is a user-written procedure that gets the value of *IOResult*, and prints out an error message as needed. There's no sense calling *IOCheck* if you've selected the {$*I*+} option since, if there's an error, your program will halt before it ever calls *IOCheck*.

Optimizing Code

A number of compiler options influence both the size and the speed of the code. This is because they insert error-checking and error-handling code into your program. It's best to enable them while you are developing your program, but you may want to disable them for your final version. Here are those options, with their settings for code optimization (the default settings are stated last):

- {$A+} enables word alignment of variables and type constants; this results in faster memory access on 80x86 systems. The default is {$A+}.
- {$B-} uses short-circuit Boolean evaluation. This produces code that can run faster, depending upon how you set up your Boolean expressions. The default is {$B-}.
- {$E-} disables linking with a run-time library that emulates an 8087 numeric coprocessor if one isn't present. This forces Turbo Pascal to use either 8087 hardware or the standard 6-byte type real, depending on the state of the $N numeric processing switch. (See the item describing {$N-}, listed shortly.) The default is {$E-}.
- {$I-} turns off I/O error-checking. By calling the predefined function *IOResult*, you can handle I/O errors yourself. The default is {$I+}.
- {$N-} generates code capable of performing all floating-point operations using the built-in 6-byte type real. When the $N switch is on, Turbo Pascal will use 8087 hardware or emulation in software instead. If you compile a program and all the units it uses with {$N-}, an 8087 run-time library is not required and Turbo Pascal ignores the emulation switch directive $E. The default is {$N-}.
- {$R-} turns off range-checking. This prevents code generation to check for array subscripting errors and assignment of out-of-range values. The default is {$R-}.
- {$S-} turns off stack-checking. This prevents code generation to ensure that there is enough space on the stack for each procedure or function call. The default is {$S+}.
- {$V-} turns off checking of **var** parameters that are strings. This lets you pass actual parameter strings that are of a different length than the type defined for the formal **var** parameter. The default is {$V+}.

See Appendix B of the *Reference Guide* for more information on compiler directives.

Optimizing your code using these options has two advantages. First, it usually makes your code smaller and faster. Second, it allows you to get away with something that you couldn't normally. However, they all have corresponding risks as well, so use them carefully, and reenable them if your program starts behaving strangely.

Note that besides embedding the compiler options in your source code directly, you can also set them using the Options/Compiler menu in the integrated environment or the /$X option in the command-line compiler (where *X* represents a letter for a compiler directive).

C H A P T E R

6

Debugging Your Turbo Pascal Programs

Turbo Pascal offers a superb development environment, with automatic project management, program modularity, high-speed compilation, and easy-to-use overlays. But with all that going for you, your program can still have *bugs*, or errors, that keep it from working correctly.

To help you with that, Turbo Pascal gives you the tools you need to *debug* your program, which means to remove all the errors and get it up and running. Turbo Pascal makes it easy to locate and fix compiler and run-time errors. And it lets you enable or disable automatic error-checking at run time.

But most important, Turbo Pascal comes with a powerful, flexible source-level debugger that allows you to execute your program one line at a time, viewing expressions and modifying variables as you go. This debugger is built into the Turbo Pascal integrated development environment (IDE); you can edit, compile, and debug without ever leaving Turbo Pascal. And for big or complex programs that require the full range of debugging support from machine language to evaluating Pascal expressions, Turbo Pascal fully supports Borland's stand-alone debugger, Turbo Debugger.

There are three basic types of program bugs: compile-time errors, run-time errors, and logic errors. Let's take a quick look at each.

Compile-Time Errors

A compile-time, or *syntax*, error occurs when you violate a rule of Pascal syntax: leave out a semicolon, forget to declare a variable, pass the wrong number of parameters to a procedure, assign a real value to an integer variable. What it really means is that you're writing Pascal statements that don't follow the rules of Pascal.

Pascal has strict rules, especially compared to many other programming languages, so cleaning up your syntax errors may take care of much of the debugging that needs to be done.

Turbo Pascal won't compile your program (generate machine code) until all your syntax errors are gone. If Turbo Pascal finds a syntax error while it is compiling your program, it stops compiling, goes into your source code, locates the error, positions the cursor there, and displays an error message in the Edit window. Once you've corrected it, you can start compiling again.

If you're using the command-line version (TPC.EXE), Turbo Pascal will print out the offending statement, along with the line number and the error message. You can then go into whatever editor you're using, find the given line, fix the problem, and recompile.

Run-Time Errors

Another type of error that can occur is a run-time, or *semantic*, error. This happens when you compile a syntactically legal program that does something illegal when it executes, such as opening a nonexistent file for input or dividing by 0. In that case, Turbo Pascal prints an error message to the screen that looks like this:

```
Run-time error ## at seg:ofs
```

and halts your program.

If you're running under the integrated environment, Turbo Pascal automatically finds the location of the run-time error, pulling in the appropriate source file.

If you ran your program from the MS-DOS prompt, you'll be returned to MS-DOS. You can load TURBO.EXE and use Compile/Find Error to locate the position in your source (make sure Destination is set to Disk). You can also use the command-line compiler (TPC.EXE) */F* option to find the error. (See Chapter 8, "Command-Line Reference," for a complete explanation and tour of using TPC.EXE to find run-time errors.)

Logic Errors

Finally, your program can have *logic* errors; simply put, this means that your program does what you *told* it to do instead of what you *want* it to do. A variable may not have been initialized; calculations may turn out wrong; pictures drawn onscreen don't look right; or the program might just skip doing what you think it should.

This can be the hardest error to find, and the one that the integrated debugger helps you with the most.

Turbo Pascal's Integrated Debugger

Some bugs are obscure and hard to track down. Others can be buried by subtle interactions between sections of a large program. In these cases, what you'd really like to do is to execute your program interactively, watching the values of certain variables or expressions. You'd like your program to stop when it reaches a certain place so that you can see just how it got there. You'd like to stop and change the values of some variables while the program is executing, to force a certain behavior or see how the program responds. And you'd like to do this in a setting where you can quickly edit, recompile, and run your program again.

Welcome to the Turbo Pascal integrated debugger, with all the capabilities just described and more. It is an integral part of the Turbo Pascal integrated development environment (IDE): Three of the main menus (**Run**, **Debug**, and **Break/Watch**) are devoted to its use; likewise, several hot keys are used for debugger commands.

Table 6.1 is a quick summary of all the debugging commands, including both hot keys and menu commands. See Table 2.1 on page 30 for a complete list of Turbo Pascal hot keys.

Table 6.1: Debugger Commands and Hot Keys

Menu Command	Hot Key	Function
Run/Run	*Ctrl-F9*	Runs your program to breakpoint, does a "make" first if necessary.
R/Program Reset	*Ctrl-F2*	Ends debugging session, releases allocated memory, and closes files in preparation for starting a new session.
R/Go to Cursor	*F4*	Runs program, stopping when and if line with cursor on it is executed. Will initiate a debugging session.
R/Trace Into	*F7*	Executes current line; if procedure or function call is online, traces into that procedure or function if possible. Will initiate a debugging session.
R/Step Over	*F8*	Executes current line; will not trace into a procedure or function. Will initiate a debugging session.
R/User Screen	*Alt-F5*	Toggles between the IDE and the User screen.
O/C/Debug Information		Enables source-level debugging. Turns the *$D* directive off and on; however, use of *$D* within the source file will override this setting.
O/C/Local Symbols		Enables evaluation of local symbols. Turns the *$L* directive off and on; however, use of *$L* within the source file will override this setting. This switch is ignored if no debug information is generated.
O/E/Zoom Windows	*F5*	Toggles between zoomed and non-zoomed display of the active window in the IDE.
Debug/Evaluate	*Ctrl-F4*	Brings up Evaluate box, allowing you to evaluate variables and expressions, and to modify variables.
D/Call Stack	*Ctrl-F3*	Shows current call stack; allows you to trace back through procedure and function calls. Valid only when debugging.
D/Find Procedure		Locates first line in a procedure or function declaration and displays it in Edit window. Works after a Compile/Make and while debugging.
D/Integrated Debugging		Enables debugging in the IDE. Turn this Off only if you need more memory to compile and run your program. When this switch is set to Off, you won't be able to debug your program in the IDE.

Table 6.1: Debugger Commands and Hot Keys (continued)

Menu Command	Hot Key	Function
D/Stand-alone Debugging		Enables debugging with the stand-alone Turbo Debugger by adding debug information to the end of the EXE.
D/Display Swapping		Chooses between three display-swapping settings: Smart, Always, and None. Normal position is Smart.
D/Refresh Display		Redraws the environment screen, cleaning up anything that might have been left onscreen.
Break/Watch/Add Watch	*Ctrl-F7*	Adds an expression to the Watch window. You can also do this by selecting the Watch window and pressing *Ins* or *Ctrl-N* (if you have Display Swapping set to None).
B/Delete Watch		Deletes the currently selected expression from the Watch window. You can also do this by selecting the Watch window, selecting the appropriate expression, and pressing *Del* or *Ctrl-Y*.
B/Edit Watch		Edits the currently selected expression. You can also do this by selecting the Watch window, selecting the appropriate expression, and pressing *Enter*.
B/Remove All Watches		Clears all expressions from the Watch window.
B/Toggle Breakpoint	*Ctrl-F8*	Sets or clears a breakpoint on the current line in the editor.
B/Clear All Breakpoints		Clears all breakpoints.
B/View Next Breakpoint		Displays (but does not execute to) the next breakpoint.
	F1	Calls up context-sensitive help.
	F6	Toggles between Edit and the other (Watch or Output) window in the IDE.
	F10	Toggles between main menu and active window.
	Alt-F6	Switches the contents of a window. If Edit window is selected, loads the previously loaded file into the Edit window. If the other window is selected, toggles between Watch and Output window.

A Quick Debugging Example

Get into Turbo Pascal and key in the following sample program:

```
{$D+,L+}      { Just to be sure complete debug information is being generated }
{$R-}                                { Just to be sure range-checking is off }
program RangeTest;
var
  List : array[1..10] of integer;
  Indx : integer;
begin
  for Indx := 1 to 10 do
    List[Indx] := Indx;
  Indx := 0;
  while (Indx < 11) do
  begin
    Indx := Indx + 1;
    if List[Indx] > 0 then
      List[Indx] := -List[Indx]
  end;
  for Indx := 1 to 10 do
    Writeln(List[Indx])
end.
```

Once you've typed in the program, save it to disk by pressing *F2*. When prompted for a new name, call it RANGE.PAS. Now, to start debugging, press *F7*. This is the "single step" command; you're asking Turbo Pascal to execute the first line in the main body of your program. Since your program hasn't been compiled yet, Turbo Pascal will do it for you automatically, and then prepare to single-step your program. Note that the execution bar is on the **begin** on line 7. Remember, the execution bar indicates the *next* line of the program to be run.

Now, press *F7* a few more times. The execution bar moves to `List[Indx] := Indx;`, and appears to stay there. What's happening is that this line is executing in a loop. So, let's see what's going on.

Choose the **B**reak/Watch/**A**dd Watch command (*Ctrl-F7*). A box labeled Add Watch appears in the middle of your screen. If the cursor is on the word *List*, then the word *List* will appear highlighted in the Add Watch box.

(What actually appears in the Add Watch box depends on where your cursor is positioned when you press *Ctrl-F7*. If you position the cursor on the first letter of any alphanumeric string, within it, or immediately following it, the string will be copied to the Add Watch box. So, if the cursor was instead positioned on *Indx, Indx* would appear in the box. If you want to change what's in the box, start typing and the original expression and the highlight will go away.)

Once the Add Watch box is displayed, regardless of its contents, you can add more to it by pressing the *Right arrow* key (which copies more text from the editor). Make sure *List* is the only text in the box and press *Enter*. A line like the following will appear in the Watch window at the bottom of your screen;

```
• List: (1,2,0,0,0,0,0,0,0,0)
```

Now, press *Ctrl-F7* again. When the Add Watch box appears, type `Indx` and press *Enter*. *Indx* is listed first in the Watch window, making it look something like this:

```
• Indx: 3
  List: (1,2,0,0,0,0,0,0,0,0)
```

Now press *F7* again, and you'll see the values of *Indx* and *List* change in the Watch window, reflecting what's happening in your program.

As you enter the **while** loop, you'll again see the values of *Indx* and *List* change, step by step. Note that the change in the Watch window reflects the actions of each line after you press *F7*.

Keep pressing *F7* until you're at the top of the **while** loop, with *Indx* equal to 10. This time through the loop, press *F7*, slowly watching how the values in the Watch window change. When you execute the statement

```
List[Indx] := -List[Indx]
```

the value of *Indx* changes to –11. If you continue to press *F7*, you'll find that what you have is an infinite loop.

If you type in this program, it will compile and run. And run. And run. It gets stuck in an infinite loop because the **while** loop executes 11 times, not 10, and the variable *Indx* has a value of 11 the last time through the loop. Since the array *List* only has 10 elements in it, *List[11]* points to some memory location outside of *List*. Because of the way variables are allocated, *List[11]* happens to occupy the same space in memory as the variable *Indx*. This means that when *Indx* = 11, the statement

```
List[Indx] := -List[Indx]
```

is equivalent to

```
Indx := -Indx
```

Since *Indx* equals 11, this statement sets *Indx* to –11, which starts the program through the loop again. That loop now changes additional bytes elsewhere, at the locations corresponding to *List[-11..0]*.

In other words, this program can really mess itself up. And because *Indx* never ends the loop at a value greater than or equal to 11, the loop never ends.

The important point is that, in just a few minutes and using only two keystrokes (*F7* and *Ctrl-F7*), you quickly and easily tracked down a subtle, nasty bug.

Why Use the Debugger?

Actually, the example we've presented probably gives you a pretty good idea why you'd want to use the debugger: to eliminate bugs in a quick, easy fashion. But what exactly does the debugger do that makes it so useful? Here's a quick run-down:

- Lets you trace the execution of your program. Sometimes simply seeing, line by line, how your program runs—which statements are executed, which aren't, and in what order—can help greatly in understanding your program. As a matter of fact, it's a tremendous aid to learning Pascal (and programming) in the first place, since you can write a program, then actually see how it runs from the "inside."
- Lets you trace your program's output line by line. You can have it swap screens as needed, or use dual monitors. You can also bring up part of the output screen in a window below the editor.
- Allows you to watch variables and expressions, monitoring their values as your program executes. As we showed you previously, this can help you track down bugs very quickly; it can also help you understand complex calculations, by letting you look at all the variables and subexpressions involved.
- Lets you change the values of variables, as well as elements of data structures. This provides an easy mechanism for testing how your code reacts to certain sets of values or conditions. It also lets you step in and adjust values while you're debugging, so that you don't have go back, fix problems, and recompile in order to continue a debugging session.
- Allows you to test and debug sections of code that might not execute during normal use, such as error-handling routines. You can also test boundary conditions, seeing how the program reacts to values that are right at the acceptable limits, or even out of limits.

What's amazing is that the debugger does all this in an uncomplicated manner. There are no special instructions inserted in your code, no increase in the size of your .EXE file, and no need to recompile to create a Stand-alone .EXE once you're finished debugging.

What if your program is divided into a number of units? No problem: The source code for each is automatically loaded into the editor as you trace execution.

What if you make use of overlays? No problem: The debugger handles overlays automatically, with no special effort on your part. And it does all this within the integrated development environment, smoothly switching back and forth between the compiler, the editor, and the debugger.

You've seen some of what the debugger can do; here's an overview of the debugger features.

Tracing

You can execute one line in your program, then pause to see the results. When procedures or functions within your program are called, you have the option of executing the call as a single step, or of tracing through that routine line by line.

Go to Cursor

You can move the cursor to a specific line in your program, then tell the debugger to execute your program until it reaches that line. This makes it easy to skip over loops and other tedious sections of code; it also lets you go right to the spot where you want to start debugging.

Breaking

You can mark lines in your program as *breakpoints*. When you run your program and it comes to a breakpoint, it stops and displays the source code with the breakpoint in the execution bar. You can then examine variables, start tracing, or run the program until another breakpoint is encountered. You can also break at any point during program execution by pressing *Ctrl-Break*. This has the effect of stopping at the next source line, as if a breakpoint had been set there.

Watching

You can set up a number of *watches* in the Watch window. Each one can be a variable, data structure, or expression. The watches change to reflect their current values as you step through your program.

Evaluating

You can bring up the Evaluate box, which lets you interactively examine the value of variables, data structures, and expressions.

Modifying

Using the Evaluate box, you can change the value of any variable, including strings, pointers, elements of an array, and fields of a record.

Navigating

You can quickly locate procedure or function declarations, even if your program is broken up into many modules. During a trace, you can quickly scroll back through the procedure or function call(s) that led to where you are and examine the parameters for each call.

Preparing to Use the Debugger

Before debugging a given program, you must instruct the Turbo Pascal compiler to generate the necessary symbol table and line-number information for that program. A *symbol table* is a small database of all the identifiers that are used—constants, types, variables, procedures, and line-number information.

The easiest way to do that is with the compiler directives {$*D*+} and {$*L*+}. The first, {$*D*+}, generates the general debug information pertaining to global identifiers. This has been used in previous versions of Turbo Pascal to generate information for Stand-alone debuggers; it's also required to set things up for the integrated debugger. Another way to enable this directive is to set the **Options/Compiler/Debug** Information command in the IDE to On. {$*D*+} generates line-number tables that map object code to source positions.

The directive {$*L*+} generates local debug information, which means it creates a list of the identifiers local to each procedure or function, so that the debugger can "remember" them while you're debugging. You can also enable this option using the **Options/Compiler/Local** Symbols command. (You'll notice that all the sample programs in this section have the `{$D+,L+}` directive as their first line. This isn't necessary if you always set **O/C/** Debug Information and Local Symbols to On. If you are going to use the

compiler directives, note that they are separated by a comma but *no spaces,* and *only the first directive* is preceded by a $.)

If your program includes units that you're going to be debugging as well, then you could put the {$*D*+} and {$*L*+} directives at the start of each unit's source code. You also may want to use the Compile/Primary File command to tell Turbo Pascal which file holds your main program. (Note that, by default, both these directives are set to On. In some circumstances, advanced programmers might want to turn these switches off to conserve memory or disk space during compilation.) For more information about these directives, see Appendix B of the *Reference Guide,* "Compiler Directives."

Finally, make sure that the IDE's debugging switch (Debug/Integrated Debugging) is set to On.

Before you start debugging, you should understand that the basic unit of execution in the debugger is a *line,* not a statement. More accurately, the *smallest* unit of execution is a line. If you have several Pascal statements on a single line, they will all be executed together with a single press of *F7.* If, on the other hand, you have a single statement spread out over several lines, then the entire statement will be executed by pressing *F7* once. All the execution commands are based on lines, including single-stepping and breakpoints; the line about to be executed is always shown in the execution bar.

For information on the way the Turbo Pascal integrated development environment works and how its components appear onscreen, refer to Chapter 7, "All About the Integrated Environment."

Your Display

The Turbo Pascal integrated development environment normally displays the main menu bar and two windows (Edit and Watch or Edit and Output). You can go to the User screen, zoom any of the windows to full size, toggle back and forth from one screen to another, and more. Here's what you have access to and what each displays.

- **User screen:** This screen disappears when you get into Turbo Pascal, but is displayed during normal execution of your program. The User screen is a DOS window that displays your program output; it reappears when you exit Turbo Pascal.
- **Edit window:** This displays the source code you're debugging. While you're debugging, though, you are free to use the standard editing and file commands to move around the source code, or even load in other

files. This lets you set breakpoints in different parts of the code, trace back through subroutine calls, or execute ahead to a certain spot.

- **Watch window:** This displays variables, data structures, and expressions that you've added to the Watch window. It starts out empty but will grow or shrink as expressions are added or removed. Press *F6* and switch to the Watch window. Now you can use the cursor keys to scroll up and down, *Enter* to edit the highlighted watch, *Ins* to add, and *Del* to delete. The familiar WordStar keys also work.
- **Output window:** This displays a copy of the User screen (in text mode only). It doesn't grow or shrink while you are in Turbo Pascal, but you can change its size using the TINST utility. When you choose it, you can scroll up and down in it using the keypad. You can also zoom the Output window to full screen.

While you're in Turbo Pascal, you can swap between the User screen and the integrated environment screen by choosing **R**un/**U**ser Screen (*Alt-F5*). This is true even when you're in the middle of debugging. If your program uses graphics, then you should be prepared for some momentary screen distortion as your computer switches back and forth between text and graphics.

In non-zoom mode, two windows are onscreen at the same time: the Edit window stacked above the Watch or Output window. Pressing *F6* still toggles between the two (Edit and the other), allowing you to select which one to work with. (In zoom mode, each window takes up the entire display; you toggle between them using *F6*.)

When you first enter Turbo Pascal, the IDE screen consists of the Edit and Watch windows. As you step through your program, Turbo Pascal will sometimes swap to the User screen, execute your code, then swap back to the integrated environment to await your next command. You can control when this screen swap occurs with the **D**ebug/**D**isplay Swapping command, which has three settings:

- **Smart:** This is the normal setting. The IDE only swaps to the User screen when a program line accesses video RAM or when a subroutine is stepped over.
- **Always:** The User screen is swapped with each step.
- **None:** No display swapping ever occurs, and the integrated environment remains visible at all times. If the program writes to the screen or if user input is required, the text will overwrite the integrated environment screen. You can have Turbo Pascal repaint its windows by choosing **D**ebug/**R**efresh Display.

Starting a Debugging Session

The quickest way to start debugging is to load in your program and choose **Run/Trace Into** (*F7*). Your program will be compiled. When it's finished, the editor will display the main body of your program, with the execution bar on the initial **begin**. You can continue to trace from there (using *F7* and *F8*), or you can use the other methods we describe here.

If you know where in the program you want to start debugging, you can have your program execute until it reaches that spot, then have it pause there. To do this, just bring up that section of code in the editor and move the cursor to the line where you want to stop. You can then do one of two things:

- You can choose **Run/Go to Cursor** (or press *F4*), which will execute your program until it reaches that point, then pause.
- You can set a breakpoint there (choose **Break/Watch/Toggle Breakpoint** or press *Ctrl-F8*), and then run your program (choose **Run/Run** or press *Ctrl-F9*); it will now stop *every* time it reaches that line. You can set several breakpoints, in which case your program will stop whenever it comes to any of the breakpoints.

Restarting a Debugging Session

If you're in the middle of debugging a program and want to start all over again, choose the **Program Reset** command from the **Run** menu. This reinitializes the debugging system so the next step command will take you to the first line in the main body of your program. At the same time, it closes any files your program may have opened, clears the stack of any nested subroutine calls, and releases any heap space being used. It does *not* reinitialize or otherwise modify any variables (Turbo Pascal never initializes variables automatically); typed constants, however, are restored to their original values.

Turbo Pascal will also offer a restart if you make any changes to the program itself while debugging. For example, if you modify any part of the program, then press any execution command (*F7*, *F8*, *F4*, *Ctrl-F9*, and so on), you'll get a box with the message `Source modified, rebuild? (Y/N)`. If you press *Y*, Turbo Pascal will re-make your program and start debugging from the beginning. If you press *N*, Turbo Pascal assumes you know what you're doing and continues the debug session in progress. (Any source code changes you made will *not* affect program execution until you recompile. If you added or deleted lines, the execution bar will *not* compensate for these changes and may appear to highlight the wrong line.)

Ending a Debugging Session

While you're debugging a program, Turbo Pascal keeps track of where you are and what you're doing. And since you can load and even edit different files while you're debugging, Turbo Pascal does not interpret loading a different file into the editor as "ending" a debugging session. So, if you want to run or debug a different program, let Turbo Pascal know by choosing the **Run/Program** Reset command (the hot key is *Ctrl-F2*).

Another reason for choosing **Run/Program** Reset is that, while your program is running, it may have used all the available memory, preventing the **File/OS** Shell command from working. To re-enable the command, use **Program** Reset, but note that this will require you to restart debugging. Since **Run/Program** Reset also allows you to start debugging your current program again, you may have to take additional steps to clear the last program. If you have set any breakpoints, you should clear them with the **Break/Watch/Clear** All Breakpoints command. If you don't clear them, these breakpoints might be invalid when you start debugging another program. If that happens, you will have a chance to clear old, invalid breakpoints before debugging your current program.

Stepping Through Your Program

Now let's look at how to actually use the debugger.

The simplest debugging technique is *single-step tracing*. You already tried this in an earlier example we gave you. If you have a program displayed in the Edit window, you must compile it first before beginning your debug session. You can do this by choosing Compile/**M**ake (*F9*) or **R**un/**T**race Into (*F7*). *F9* will make the program; *F7* will both make the program and place the execution bar on the first line in your program. You can then execute that line by choosing **R**un/**T**race Into (*F7*) or **R**un/**S**tep Over (*F8*).

The difference between **T**race Into (*F7*) and **S**tep Over (*F8*) is that *F7* traces into procedures and functions, while *F8* steps over those subroutine calls. These commands also have a special meaning at the **begin** statement of the main program if the program uses units with initialization code. In this case, *F7* will step into each unit's initialization code, allowing you to see how each unit sets itself up. *F8* will step over the initialization code, leaving the execution bar on the next executable line after the **begin**.

When you step using *F7* or *F8*, the User screen may be displayed, usually briefly (unless the program is waiting for input), and the program text is displayed again, with the execution bar on the next line to be executed.

When stepping into (with **Run/Trace** Into or *F7*) a procedure or function compiled with **Options/Compiler/Debug** Information set to On, Turbo Pascal will load the source code and position the execution bar at the beginning of the procedure or function.

If you encounter another procedure or function call, then you can trace into that routine, as far as you care to go. Once you reach the end of a routine, you are returned to the procedure that called it.

Consider the following example, TEST.PAS:

```
{$D+,L+}
program Test;

var
  X,Y : integer;

procedure Swap(var A,B : integer);
var
  T : integer;
begin
  T := A;
  A := B;
  B := T
end;
begin { main body of Test }
  Write('Enter two values:  ');
  Readln(X,Y);
  Swap(X,Y);
  Writeln('X = ',X,'  Y = ',Y)
end. { of program Test }
```

If you start debugging this program by pressing *F7*, the program will be compiled and the execution bar will highlight the second **begin** statement (on line 16).

Pressing *F7* continually will step you through the main body until you reach the line calling *Swap*. When you press *F7* again, the execution bar moves to the **begin** statement in *Swap*. From there, you step through each statement in *Swap* until the execution bar reaches the closing **end**. Another press of *F7* and you're back in the main body of *Test*, with the *Writeln* highlighted. Another *F7* executes that line and leaves you at the closing **end** of *Test*. One more *F7*, and the program is done. Notice the execution bar is gone.

You might wonder why you need to "execute" the **begin** and **end** statements of the main program. The explanation is that the execution bar only highlights lines that generate code. Executing the **begin** statement calls any initialization code, including initialization code from any units your program might be using. This means you can actually step through your

unit's initialization code (use *F7* to step into your unit's initialization code and *F8* to step over it.) Likewise, executing the final **end** statement causes your program to call the chain of exit procedures. If you've installed your own exit procedure(s), then they are called as well (although you must set breakpoints within them to debug them).

Let's look over TEST.PAS. Why did *F7* step *over* the *Write* and *Read* statements but step *into Swap*? Because Turbo Pascal could find the source code for *Swap*, but *Read* and *Write* are part of the run-time library and their source was not available (and, for that reason, we compiled them with Debug Information set to Off). Suppose you want to step through the main body of *Test*, but don't want to trace through *Swap*? Is there a way to force the debugger to treat *Swap* the same way it treated *Write*, *Readln*, and *Writeln*?

Yes, by using Run/Step Over (*F8*). This acts just like Run/Trace Into (*F7*), with one important difference: If the line contains any procedure or function calls, you don't trace into them; they are executed just as if they were system calls. This lets you avoid tedious tracing through subroutines that work just fine or that you want to avoid for now. Of course, if you have a breakpoint within that routine, the debugger will halt at that line.

If, while in TEST.PAS, you press *F8* when the execution bar is on the line containing the call to *Swap*, *Swap* is called and the execution bar is positioned on the next line (calling *Writeln*).

Now, consider the following (incomplete) sample program:

```
{$D+,L+}
program TestSort;
const
  NLMax = 100;
type
  NumList = array[1..NLMax] of integer;
var
  List : NumList;
  I,Count : word;

procedure Sort(var L : NumList; C : word);
begin
  { sort the list }
end; { of proc Sort }
```

```
begin
  Randomize;
  Count := NLMax;
  for I := 1 to Count do
    List[I] := Random(1000);
  Sort(List,Count);
  for I := 1 to Count do
    Write(List[I]:8);
  Readln
end. { of program TestSort }
```

Suppose you're debugging the *Sort* procedure. You want to trace your call to *Sort*, including checking the values within *List* before calling it. However, it gets tedious stepping through that first **for** loop 100 times as it initializes *List*. Is there some way you can get the loop to execute without having to single-step each line?

Yes. In fact, there are a few ways. First, you could put it in a separate procedure and press *F8* when you get to it, but that's a bit drastic. Second, you could set a *breakpoint* within your program. We'll explain what breakpoints are and how they work in just a minute.

Finally, you could use the **Run/Go** to Cursor command (hot key *F4*). Just move the cursor to the line calling *Sort*, then press *F4*. Your program will execute until it gets to the line containing the cursor. The execution bar will move to that line; you can then start tracing from there, in this case by pressing *F7* so that you can trace into *Sort*.

Run/Go to Cursor (*F4*) works through multiple levels of subroutine calls, even if the source code is in another file. For example, you could place the cursor somewhere within *Sort* and press *F4*; the program would execute until it reached that line within *Sort*. For that matter, *Sort* could be in a separate unit, and the debugger would still know when to stop and what to display.

There are three cases where **Go** to Cursor (*F4*) will not run to the line containing the cursor. The first is where you have the cursor positioned between two executable lines; for example, a blank line or a comment line within a code block. In this case, the program will run to the next line containing executable statements. The second case is when you have the cursor positioned outside the scope of a procedure block; for example, on the program statement or variable declarations. The debugger will tell you there is "no code generated for this line." The third case is when you position the cursor on a line that will never gain control; for example, the line *above* the execution bar (assuming you're not in a loop) or the **else** part of a conditional statement when the **if** expression is true. The debugger will behave as if you had chosen **Run/Run** (*Ctrl-F9*); your program will run until it terminates or until a breakpoint occurs.

Let's say that you trace through *Sort* for a while, then want the program to finish executing so you can see the output. How would you do this? First, you could move the cursor to the final **end** statement in the main body of the program, then choose **R**un/**G**o to Cursor (*F4*).

More simply, you could choose **R**un/**R**un (*Ctrl-F9*). This tells the debugger to let your program continue normal execution. Your program will then run until it ends or hits a breakpoint that you've set, or until you press *Ctrl-Break*.

Using Breakpoints

You've seen *breakpoints* mentioned a number of times; they're an important part of debugging. A breakpoint is like a stop sign embedded in your program. When your program encounters one, it stops execution and waits for further debugging instructions. Note that these breakpoints only exist during your debugging session; they aren't saved in your .EXE file if you compile your program to disk.

To set a breakpoint, use the regular editing commands to move the cursor to each line in your program where you want it to pause. The line on which a breakpoint is set should contain at least one executable statement. It should not be a blank line, a comment, or a compiler directive; a constant, type, label, or variable declaration; or a program, unit, procedure, or function header. To set the breakpoint, choose the **B**reak/Watch/**T**oggle Breakpoint command (*Ctrl-F8*). When a line has been set as a breakpoint, the line is highlighted.

You can have up to 21 breakpoints active at a time.

Once you've set your breakpoints, execute your program by choosing **R**un/**R**un (or pressing *Ctrl-F9*). Your program will begin executing normally. When a breakpoint is encountered, the program halts, the appropriate source code file (main program, unit, or Include file) is loaded in, and the Edit window is displayed with the execution bar on top of the breakpoint line. (Note that you cannot see the breakpoint highlight when the execution bar is on the breakpoint line.) If any variables or expressions have been added to the Watch window, they are also displayed with their current values.

At this point, you can use any of your debugging options. You can step through your code using **R**un/**T**race Into, **S**tep Over, or **G**o to Cursor (*F7*, *F8*, or *F4*). You can examine and modify variables. You can add or remove expressions from the Watch window. You can set or clear breakpoints. You can view program output with **R**un/**U**ser Screen (*Alt-F5*). You can re-start your program from the beginning (using **R**un/**P**rogram Reset and then a

step command). Or you can continue execution to the next breakpoint (or to the end of the program) by choosing **Run/Run** (*Ctrl-F9*).

To clear a breakpoint from a line, move the cursor to the line and choose **B**reak/Watch/**T**oggle Breakpoint (or press *Ctrl-F8*) again. This command toggles (turns On and Off) the breakpoint line; if you use it on a breakpoint line, that line returns to normal.

To clear all breakpoints, choose the **B**reak/Watch/**C**lear All Breakpoints command.

If you want to review the breakpoints that you've set in your program, choose the **B**reak/Watch/**V**iew Next Breakpoint command. This cycles you through all the breakpoints in your program.

Let's go back to the example you looked at earlier:

```
begin { main body of TestSort }
  Randomize;
  Count := NLMax;
  for I := 1 to Count do
    List[I] := Random(1000);
  Sort(List,Count);
  for I := 1 to Count do
    Write(List[I]:8);
  Readln
end. { of program TestSort }
```

As you recall, the idea was to skip over the initial loop and start tracing with the call to *Sort*. Your new solution is to move the cursor to that line and choose **B**reak/Watch/**T**oggle Breakpoint (*Ctrl-F8*), making it a breakpoint. Now, run to the breakpoint by choosing **R**un/**R**un (*Ctrl-F9*). When the program gets to that line, it will stop and allow you to begin debugging.

Using Ctrl-Break

In addition to any breakpoints you might set, you also have an "instant" breakpoint during execution: pressing *Ctrl-Break*. This means that, barring a major crash, you can interrupt your program at any time. When you press *Ctrl-Break*, you drop out of your program and back into the editor, with the execution bar on the next line and ready for single-stepping.

What actually happens is that the debugger hooks itself into DOS, the BIOS, and other services. In this way, it knows whether or not the code currently executing is a DOS routine, BIOS routine, or your program. When you press *Ctrl-Break*, the debugger waits until the program itself is executing. It then starts stepping every machine-level instruction until the next

instruction is at the beginning of a Pascal source code line. At that point, it breaks, moves the execution bar to that line and prompts you to press *Esc*.

If a second *Ctrl-Break* is detected before the debugger locates and displays the source code line, then the debugger terminates the program and doesn't try to find the source line. In such a case, the exit procedures are *not* executed, which means that files, video mode, and DOS memory allocations might not be completely cleaned up.

Watching Values

Program flow tells you a lot, but not as much as you'd like. What you really want to do is watch how variables change as your program executes. Consider, for example, the earlier sorting program. Suppose the *Sort* procedure for that program looked like this:

```
procedure Sort(var L : NumList; C : word);
var
  Top,Min,K : word;
  Temp : integer;
begin
  for Top := 1 to C-1 do
  begin
    Min := Top;
    for K := Top+1 to C do
      if L[K] < L[Min] then
        L[Min] := L[K];
    if Min <> Top then
    begin
      Temp := L[Top];
      L[Top] := L[Min];
      L[Min] := Temp
    end
  end
end; { of proc Sort }
```

There is a bug here, so step through it (using **Run/Trace** Into or *F7*) and watch the values of *L*, *Top*, *Min*, and *K*. (To make your job easier, you should probably change *NLMax* (in the program example on page 116) to 10, so that you're working with a smaller array.)

The debugger lets you set up *watches* to monitor values within your program as it executes. As you might guess, a *watch* is a variable, data structure, or expression placed in the Watch window. The current value of each watch is shown, updated as each line in the program executes.

Let's take the previous example. It's easy to set up your watches. Simply move the cursor to each identifier and choose **Break/Watch/Add Watch** (*Ctrl-F7*) to add each expression to the Watch window. This copies whatever identifier the cursor is positioned on into the Add Watch box; pressing *Enter* accepts that expression. The result might look like this:

```
• K: 21341
  Min: 51
  Top: 21383
  L: (163,143,454,622,476,161,850,402,375,34)
```

This presumes you've just stepped into *Sort* and the execution bar is on the initial **begin** statement. (If you haven't stepped into *Sort* yet, "unknown identifier" will be displayed next to each Watch expression until you do.) Note that *K*, *Min*, and *Top* just have random values, since they haven't been initialized yet. The values in *L* are supposed to be random; they won't look just like this when you run the program, but they will all be nonnegative values from 0 to 999.

Pressing *F7* four times will move you down to the line **if** `L[K] < L[Min]` **then**, where you'll notice that *K*, *Min*, and *Top* now have values of 2, 1, and 1, respectively. Keep pressing *F7* until you drop out of that inner **for** loop, through the **if** `Min <> Top` **then** line, back to the top of the outer loop, and down again to **if** `L[K] < L[Min]` **then**. At this point, the Watch window would look like this (given the previous values in *L*):

```
• K: 3
  Min: 2
  Top: 2
  L: (34,143,454,622,476,161,850,402,375,34)
```

By now, you may have noticed two things. First, the last value in *L* (34)—which also happens to be the lowest value—got copied into the first location in *L*, and the value that was there (163) has disappeared. Second, *Min* and *Top* were the same value all the way through. In fact, if you look closely, you'll notice something else: *Min* gets assigned the value of *Top*, but is never changed anywhere else. Yet the test at the bottom of the loop is **if** `Min <> Top` **then**. Either you have the wrong test, or there's something wacky between those two sections of code.

As it turns out, the bug is in the fifth line of code: It should read `Min := K;` instead of `L[Min] := L[K];`. Correct it, move the cursor to the initial **begin** in *Sort*, and choose **Run/Go to Cursor** (*F4*). Since you've changed the program, a box will appear with the question `Source modified, rebuild? (Y/N)`; press *Y*. Your program will recompile, start running, then pause at the initial **begin** in *Sort*. This time, the code works correctly: Instead of overwriting the first location with the lowest value, it swaps values, moving the value in the first location to the position where the lowest value

was previously. It then repeats the process with the second location, the third, and so on, until the list is completely sorted.

Scope and Qualification

Continue to step through *Sort* until it's done. If you get impatient, you can choose **Run/Run** (*Ctrl-F9*) or put the cursor on the final **end** in *Sort*, then press *F4*. Note that *L* is completely sorted, with the values ordered from lowest to highest. The Watch window will look something like this (the values of *L* will, of course, vary from execution to execution):

```
• K: 10
  Min: 9
  Top: 9
  L: (19,43,66,202,262,285,396,473,803,936)
```

Now press *F7* again, so that you return to the main body of *DoSort*. Your Watch window now looks like this:

```
• K: Unknown identifier
  Min: Unknown identifier
  Top: Unknown identifier
  L: Unknown identifier
```

What happened? A second ago, these expressions had values, and now they've disappeared altogether. Why? Because, as we mentioned earlier, they're all local to *Sort*, that is, they are only "visible" and usable while *Sort* is executing.

The area where an identifier (constant, type, variable, procedure, function) is visible and usable is its *scope*. The formal definition and rules of scope are given in Chapter 1 of the *Reference Guide*, "Tokens and Constants," but here's a simple explanation: An identifier's scope goes from the point of its declaration to the end of the program, procedure, or function in which it was declared. (The obvious exception is any identifier declared in the **interface** section of a unit. When a unit is used, its interfaced identifiers are "imported" into the local scope.)

Now what happened here becomes clear. The scope of the variables *Top*, *Min*, and *K*, as well as that of the dummy parameter *L*, is limited to the procedure *Sort*. Once you exit *Sort*, those identifiers are no longer visible, and the Watch window reflects that.

There is an important qualification to the scope explanation. Consider the following sample program:

```
{$D+,L+}
program TestScope;
```

```
var
  W,X,Y,Z : integer;

procedure Swap(var X,Y : integer);
var
  Z : integer;
begin
  Z := X;
  X := Y;
  Y := Z
end; { of proc Swap }

begin { main body of TestScope }
  W := 10; X := 20; Y := 30; Z := 40;
  Swap(X,W);
  Swap(Y,X);
  Swap(Z,Y);
  Writeln(W:8,X:8,Y:8,Z:8)
end. { of program TestScope }
```

Type in this program, save it as SCOPE.PAS, compile it using Compile/ Make (*F9*), then press *F7*. Once the Watch window appears, add *W*, *X*, *Y*, and Z to it with Break/Watch/Add Watch (*Ctrl-F7*). Press *F7* twice more; the execution bar should now be on the call to `Swap(X,W)`, and the Watch window should look like this:

```
• W: 10
  X: 20
  Y: 30
  Z: 40
```

Now press *F7* once more, so that you've traced into *Swap*. The Watch window now reads like this (Z is uninitialized, so its actual value will vary):

```
• W: 10
  X: 10
  Y: 20
  Z: 21326
```

Step through until you reach the closing **end** of *Swap*. The Watch window will now read

```
• W: 20
  X: 20
  Y: 10
  Z: 10
```

One more press of *F7* returns you to the main body of *TestScope*, and the Watch window contains

```
• W: 20
  X: 10
  Y: 30
  Z: 40
```

What's going on here? The values of *W*, *X*, *Y*,and *Z* appear to be varying widely for no reason at all. And yet it makes perfect sense once you understand this scope rule: When two identifiers (from different scopes) are identical, the "most local," or nested, identifier takes precedence.

Case in point: The procedure *Swap* redeclares three identifiers *X*, *Y* and *Z*. When you are stepping through the main body of *TestScope*, then the Watch window uses the global versions of *X*, *Y*, and *Z*. However, once you trace into *Swap*, the versions of *X*, *Y*, and *Z* local to *Swap* take precedence, and their values are displayed instead. Once you exit *Swap*, the global versions are used again.

Is there some way to help sort out this confusion? One way, of course, is to avoid using local identifiers that redeclare global identifiers. For example, you could rewrite *Swap* to look like this:

```
procedure Swap(var A,B : integer);
var
  T : integer;
begin
  T := A;
  A := B;
  B := T
end; { of proc Swap }
```

Now, if you trace through, the Watch window correctly displays the values of the global versions of *X*, *Y*, and *Z* all the way through, since there are no local versions in *Swap*.

There are times, though, when renaming isn't feasible or desirable. Can you somehow avoid the confusion anyway? Suppose you have the original version of *TestScope*, but you want to trace the global variables all the way through. Can it be done?

Yes, by *qualifying* the identifiers. You qualify a global identifier by explicitly preceding it with the program or unit name. Here's the format:

```
module.identifier
```

where *module* is the program or unit name, and *identifier* is the identifier name. For example, you might set up the Watch window this way:

```
• X:
  Y:
  Z:
  TestScope.W:
  TestScope.X:
  TestScope.Y:
  TestScope.Z:
```

This lets you trace all the variables and dummy parameters throughout the entire program. In fact, you could go one step farther and make the Watch window look like this:

```
• X:
  @X=@TestScope.X:
  Y:
  @Y=@TestScope.Y:
  Z:
  @Z=@TestScope.Z:
  TestScope.W:
  TestScope.X:
  TestScope.Y:
  TestScope.Z:
```

The first expression you added compares the address of *X* with the address of *TestScope.X*. When this expression is True, then you know that *X* refers to the global *X*; when it's False, then you know that *X* is some local variable or parameter. The other two expressions perform the same function for *Y* and *Z*.

You can also qualify local identifiers using a special syntax that specifies the full "procedural path" of that name. In the program *TestScope*, you could watch *Swap*'s X parameter by watching *TestScope.Swap.X*.

Types of Watch Expressions

So far, the only expressions you've seen in the Watch window are integer variables and arrays. You can actually put any kind of constant, variable, or data structure in as an expression; you can also put in Pascal expressions. Specifically, here's what you can add and how it will be displayed:

- **Integers**: Integers are displayed in decimal values; you can also display them in hexadecimal. Examples:

  ```
  -23   $10
  ```

- **Reals**: Reals are displayed without an exponent, if possible. Examples:

  ```
  38328.27   6.283e23   0.00299823532
  ```

- **Characters**: Printable characters (including the extended graphics characters) are displayed as themselves, within single quotes; control characters (ASCII codes 0..31) have their ASCII codes displayed. Examples:

  ```
  'b'   'Ø'   #4
  ```

  You can elect to have control characters be treated as "printable."
- **Booleans**: Booleans are displayed as either True or False.
- **Enumerated data types**: The data types are displayed as their actual named values (in all uppercase); for example,

  ```
  RED   JAN   WEDNESDAY
  ```

- **Pointers**: The current contents of a pointer are displayed as a pointer in (*segment:offset*) hexadecimal format. Examples:

  ```
  PTR($3632,$106)   PTR(DSEG,$AB)   PTR(CSEG,$220)
  ```

- **Strings**: The current contents of a string are displayed within single quotes. Example:

  ```
  'Droid'
  ```

- **Arrays**: The current contents of an array are displayed within parentheses, with elements separated by commas. Multidimensional arrays are displayed as nested lists. Examples:

  ```
  (-42,23,2292,0,684)   ((10,20)   (20,40)   (40,60))
  ```

- **Records**: The current contents of a record are displayed within parentheses, with fields separated by commas. Nested records are displayed as nested lists. Examples:

  ```
  (5,10,'Borland',RED,TRUE)   (5.6,7.8,(TRUE,'O',9),TUES)
  ```

- **Sets**: The current elements of a set are displayed within brackets, with expressions separated by commas; subranges are used when possible. Examples:

  ```
  [MON,WED,FRI]   ['0'..'9','A'..'F']
  ```

- **Files**: A file's current status is displayed in the format (*status,fname*), where *status* is CLOSED, OPEN, INPUT, or OUTPUT, and *fname* is the name of the disk file assigned to the file variable. Example:

  ```
  (OPEN,'BUDGET.DTA')   (INPUT,'INPUT.TXT')
  ```

Format Specifiers

To control exactly how information is displayed in the Watch window, Turbo Pascal allows you to add *format specifiers* to your Watch expressions.

A format specifier follows the Watch expression, separated from it by a single comma.

A format specifier consists of an optional *repeat count* (an integer), followed by zero or more format characters; no spaces are required between the repeat count and the format characters. Table 6.2 lists the available format specifiers, and describes their effects.

The repeat count is used to display consecutive variables, such as the elements of an array. For example, assuming *List* is an array of 10 integers, the Watch expression `List` would display:

```
List: (10,20,30,40,50,60,70,80,90,100)
```

If you want to look at a particular range of the array, you can specify the index of the first element, and add a repeat count:

```
List[6],3: 60,70,80
```

The above technique is particularly useful for dealing with arrays that are too large to be displayed completely on a single line.

Repeat counts aren't limited to arrays; any variable may be followed by a repeat count. The general syntax `var,x` simply displays *x* consecutive variables of the same type as `var`, starting at the address of `var`. Note however, that the repeat count is ignored if the Watch expression does not denote a variable. A good rule of thumb is that a given construct is a variable if it can legally appear on the left-hand side of an assignment statement, or be used as a **var** parameter to a procedure or function.

Table 6.2: Debug Expression Format Characters

Character	Function
$	Hexadecimal. Has the same effect as the **H** specifier.
C	Character. Shows special display characters for control characters (ASCII 0..31); by default, such characters are shown as ASCII values using the #*xx* syntax. Affects characters and strings.
D	Decimal. All integer values are displayed in decimal. Affects simple integer expressions as well as structures (arrays and records) containing integers.
F*n*	Floating-point. *n* is an integer between 2 and 18 specifying the number of significant digits to display. The default value is 11. Affects only floating-point values.
H	Hexadecimal. All integer values are displayed in hexadecimal with a preceding '$' character. Affects simple integer expressions as well as structures (arrays and records) containing integers.
M	Memory. Displays a memory dump of a variable. The expression must be a construct that would be valid on the left-hand side of an assignment statement (that is, a construct that denotes a memory address); otherwise, the **M** specifier is ignored. By default, each byte of the variable is shown as two hexadecimal digits. Adding a **D** specifier causes the bytes to be displayed in decimal, and adding an **H**, **$**, or **X** specifier causes the bytes to be displayed in hexadecimal with a preceding '$' character. A **C** or an **S** specifier causes the variable to be displayed as a string (with or without special characters). The default number of bytes displayed corresponds to the size of the variable, but a repeat count may be used to specify an exact number of bytes. **Note:** Because of the way INTEL architecture stores data, the memory dump may not look the way you expect it to. For example, integers are stored with the low byte first, then the high byte.
P	Pointer. Displays pointers in *seg : ofs* format rather than the default *Ptr(seg,ofs)* format. For example, displays 3EA0:0020 instead of *Ptr($3EA0,$20)*. Affects only pointer values.
R	Record. Displays record field names, such as (X:1;Y:10;Z:5) instead of (1,10,5). Affects only record variables.
S	String. Shows control characters (ASCII 0..31) as ASCII values using the #*xx* syntax. Since this is the default character and string display format, the **S** specifier is only useful in conjunction with the **M** specifier.
X	Hexadecimal. Has the same effect as the **H** specifier.

To demonstrate the use of format specifiers, assume that the following types and variables have been declared:

```
type
  NamePtr = ^NameRec;
  NameRec = record
              Next: NamePtr;
              Count: Integer;
              Name: string[31];
            end;
var
  List: array[1..10] of Integer;
  P: NamePtr;
```

Given the above declarations, the following Watch expressions can be constructed:

```
List: (10,20,30,40,50,60,70,80,90,100)
List[6],3H: $3C,$46,$50
P: PTR($3EA0,$C)
P,P: 3EA0:000C
P^: (PTR($3EF2,$2),412,'John')
P^,R$: (NEXT:PTR($3EF2,$2);COUNT:$19C;NAME:'John')
P^.Next^,R: (NEXT:NIL;COUNT:377;NAME:'Joe')
Mem[$40:0],10M: F8 03 F8 02 00 00 00 00 BC 03
Mem[$40:0],10MD: 248 3 248 2 0 0 0 0 188 3
```

Typecasting

Typecasting is another powerful feature you can use to modify how Watch expressions are displayed, letting you interpret data as a different type than it would normally be. This can be especially useful if you're working with an address or a generic pointer, and you want to view it as pointing to a particular data type.

Suppose your program has a variable *DFile* that is of type `file of MyRec`, and you execute the following sequence of code:

```
Assign(DFile,'INPUT.REC');
Reset(DFile);
```

If you add *DFile* as a watch, the corresponding line in the Watch window will look like this:

```
DFile: (OPEN,'INPUT.REC')
```

But you might want more information about the file record itself. If you change your program so that it uses the *Dos* unit, then you can modify the *DFile* watch to `FileRec(DFile),rh`, which means, "Display *DFile* as if it were a record of type `FileRec` (declared in the *Dos* unit), with all record fields

labeled and all integer values displayed in hexadecimal." The result in the Watch window might look something like this:

```
FileRec(DFile),rh: (HANDLE:$6;MODE:$D7B3;RECSIZE:$14;PRIVATE:($0,$0,...))
```

The record is too large to view at once; however, you can use the cursor movement keys to scroll the data not visible on the screen (see the section "Editing the Watch Window" on page 133).

With this typecasting, you can now watch specific fields of *DFile*. For example, you could view the *UserData* field by adding the expression `FileRec(DFile).UserData` to the Watch window:

```
FileRec(DFile).UserData: (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0)
```

You can apply the same technique to data structures and types of your own design. If they're declared in your program or units, you can typecast to them in the Watch window. The rules for typecasting are explained in Chapter 6 of the *Reference Guide*, "Expressions."

Expressions

As we mentioned earlier, you can use *expressions* as Watch expressions; you could have calculations, comparisons, address offsets, and other such expressions. Table 6.3 lists the kinds of features legal in a Watch expression, as well as acceptable values.

Table 6.3: Watch Expression Values

Legal in a Watch Expression	Acceptable Values
Literals and Constants	All normal types: boolean, byte, char, enumerated, integer, longint, real, shortint, string, and word.
Variables	All types, including user-defined types and elements of data structures:
integer-type	Any integer expression within the variable's range bounds.
floating-point	Any floating-point (or integer) expression within the variable's exponent range; excess significant digits are dropped.
char	Any character expression, including any printable character surrounded by single quotes; integer expressions typecast to char using *Chr* or *Char()*; ASCII constants (#, followed by any a value from 0 to 255).
Boolean	True and False; any Boolean expression.
enumerated data type	Any compatible enumerated constant; in-range integer expressions typecast to a compatible enumerated type.
pointer	Any compatible pointer; any compatible typecast expression; the function *Ptr* (with appropriate parameters).
string	Any string constant (text enclosed by single quotes); string variables; string expressions consisting of string constants and variables concatenated with the + operator.
set	Any set constant (compatible elements surrounded by square brackets); any compatible set expression, including the use of set operators +, -, *.
Typecasts	Following standard Pascal rules.
Operators	All normal Pascal operators, plus Turbo Pascal extensions such as **xor**, @, and so on.
Built-In Functions	*Abs, Addr, Chr, CSeg, DSeg, Hi, IOResult, Length, Lo, MaxAvail, MemAvail, Odd, Ofs, Ord, Pred, Ptr, Round, Seg, SizeOf, SPtr, SSeg, Succ, Swap,* and *Trunc.*
Arrays	*Mem, MemL, MemW*

In other words, the expression must be a normal, legal Pascal expression, and can use any or all of the features described in Table 6.3. See "Modification Issues" on page 136 for information on how to modify Watch expressions.

Editing the Watch Window

So far, you've only seen how to add watches using the **B**reak/**W**atch/**A**dd Watch command (*Ctrl-F7*). You can also remove them, of course, as well as edit ones already there.

The most straightforward approach is to use editing commands in the Watch window. Select the Watch window by pressing *F6*; it will have a double-lined border across the top, and the name of the window will be highlighted. (If the Output window is visible, press *Alt-F6*.)

Now that you're in the Watch window, use the *Up* and *Down arrow* keys to move from one Watch expression to another. You'll notice that using these keys changes which Watch expression is highlighted. The expression that's highlighted is the one you can edit or delete. You can always insert a new Watch above the highlight bar. You can scroll up or down through the Watches in this window if all the expressions are not visible at once.

If a Watch is longer than the window is wide (greater than 77 characters), move the highlight bar to that Watch and use the *Left* and *Right arrow* keys to scroll horizontally. You can use *Home* and *End* to move to the start and end of the line.

Press *Ins* (or *Ctrl-N*) to add a new Watch expression while in the Watch window. This brings up the Add Watch box, just like pressing *Ctrl-F7*. Once you've typed in a new expression, it's inserted above the currently selected watch.

To edit the currently selected Watch expression, just press *Enter*. An Edit Watch box will appear, with the Watch expression in it. If you want to totally replace the expression that's there, just begin typing and the highlight bar will disappear. Press *Right arrow*, *Left arrow*, *Home*, or *End* to edit the expression. If you want to replace the original expression with the edited version, press *Enter*. If you don't want to change the old one, press *Esc*. If you want to get the original version back, press *Ctrl-R*.

To delete a Watch expression, just choose it by moving up or down with the arrow keys, and then press *Del* (or *Ctrl-Y* or *Ctrl-G*). You can also choose the **B**reak/**W**atch/**R**emove All Watches command, which clears the Watch window completely.

Once you move to the Edit window by pressing *F6*, the current expression in the Watch window is no longer highlighted and instead has a bullet (•) in front of it. Any Watch operations performed from the menu refer to the current Watch expression.

Note that the Watch window is auto-tiled: It will shrink or grow automatically as you add or remove watches. In this way, no blank lines take up

space in the Watch window and most of the screen is used for displaying your source code. Once the Watch window grows to a maximum size (configurable using TINST), new Watches always appear on screen and lower ones will scroll off the bottom of the window. To see the watches that are not displayed, use the *Up* and *Down arrow* keys to scroll the window.

Evaluating and Modifying

The Watch window is wonderful for tracing values as you step through your program. However, there are times when it doesn't quite meet your needs. Often, there are variables and expressions that you only need to check at a certain point or points. You don't really want to have it sitting in the Watch window the whole time, but it can be a bit tedious to add it, then immediately delete it, especially if you have several such expressions to check at one time. You might also want to do more than just look at a variable; you might also want to change its value.

To accommodate these needs, the debugger offers the Evaluate window, shown in Figure 6.1 on page 136. To bring it up, choose the Debug/ Evaluate command (or press the hot key *Ctrl-F4*). This window has three boxes in it, labeled **Evaluate, Result,** and **New Value**.

As with the Add Watch box, the Evaluate box already contains the word found at the cursor; it's in highlight mode. Edit it as you would the Add Watch box and press *Enter* when you want to evaluate it. The current value of the constant, variable, or expression will then appear in the Result box.

The Evaluate box accepts exactly the same set of constants, variables, and expressions that the Watch window does. You have the same freedoms and restrictions we've already mentioned. You can also use the same format characters as you can for Watch expressions.

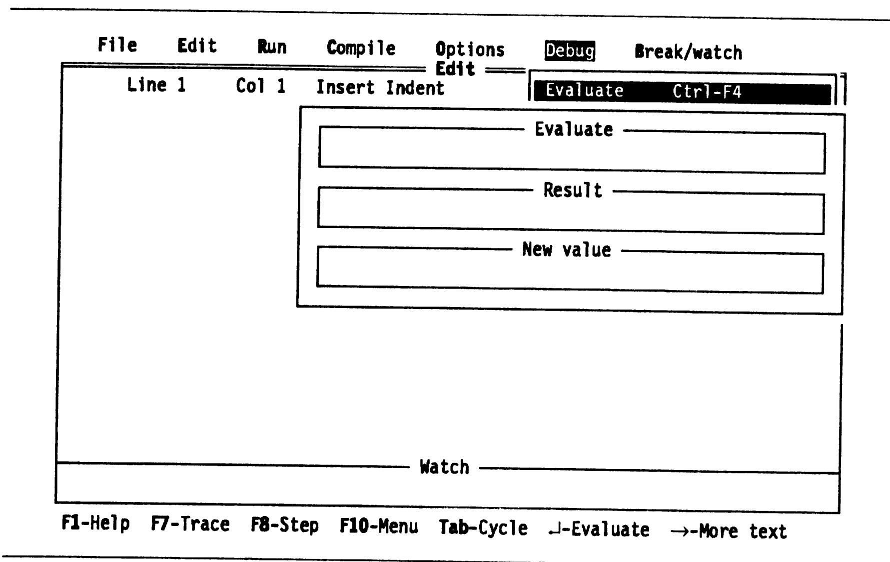


Figure 6.1: The Evaluate Window

When you press *Enter,* the identifier or expression in the Evaluate box is highlighted again, which means if you start typing a new name (without pressing *Ins* or an arrow key), it will replace the old one. This lets you quickly type in a series of variables and expressions.

The New Value box allows you to modify the value of the variables named in the Evaluate box. You can enter a constant value, the name of another variable, or even an expression. The resulting value must be of a type compatible with the variable in the Evaluate box. Therefore, if you have an expression in Evaluate that does not result in a memory location, then any value entered in New Value will result in the message `Cannot be modified`.

The Result box shows the current value of whatever is in the Evaluate box, using the same format as the Watch window. And, like the Watch window, the data will sometimes be too large to fit. In such cases, you can use the *Tab, Backtab, Left arrow, Right arrow, Home,* and *End* keys to scroll through the box.

In all cases, you can use the *Up* and *Down arrow* keys (or regular keyboard editing commands) to move between the three boxes. Once you have modified a box, press *Enter* to evaluate the input.

Modification Issues

The ability to modify a variable while the program is running is a tremendous help while debugging. It can also be dangerous, so you need to be sure you know the *do's* and *don'ts* of modification.

The simplest form of modification, of course, is to enter a variable name in the Evaluate box and a corresponding value in the New Value box. When you press *Enter* after typing in the new value, the variable's value is changed, and the Result box is updated to reflect that.

You are not limited to constant values, though. You can enter into the New Value box any variable or expression that you could enter into the Evaluate box, with one major qualification: It must be assignment-compatible with the variable or expression in the Evaluate box. In other words, if *expr1* represents what's currently in the Evaluate box, then you cannot legally enter the expression *expr2* into the New Value box if the statement

```
expr1 := expr2;
```

would cause a compiler or run-time error.

Note that the reverse is not necessarily true: There are cases when the statement

```
expr1 := expr2;
```

is legal, but you still cannot use *expr2* in the New Value box.

If the expression entered is an incompatible type—such as entering a floating-point value for an integer variable—then the Result box will instead display the message `Type mismatch`. To make the Result box redisplay the current value of the variable, move back up to the Evaluate box and press *Enter*.

If the expression entered yields an out-of-range value—such as entering 50,000 for a variable of type integer—the Result box will display the message `Constant out of range`. The same thing will occur if you type in an array element with an index that's out of range.

If the expression entered in the New Value box is one that can't be assigned, then the Result box will get the message `Cannot evaluate this expression`. Such expressions include arrays, records, sets, and files.

Likewise, if the variable or expression in the Evaluate box is one that can't be modified—a whole array, record, set, or a file—then attempting to assign a value to it will produce the message `Cannot be modified`.

What can you modify? Refer to Table 6.3 on page 131 for a list of what can be used in a Watch expression, along with acceptable values. Remember,

though, that expressions can only use the built-in functions listed as acceptable for Watch expressions in Table 6.3.

Other things to keep in mind:

- You can't modify entire arrays, entire records, or files; however, as mentioned, you can modify individual elements of arrays or records that resolve to one of the types listed in Table 6.3, provided they are not themselves arrays or records.
- You can't directly modify untyped parameters passed into a procedure or function. You can, however, typecast them to a given type, then modify them according to the restrictions we've just detailed.
- Be aware that there can be some real dangers in modifying variables. For example, if you change a pointer, you could end up making changes to memory that you didn't mean to, possibly even modifying other variables and data structures.

Navigation

When you are debugging a large program, especially one spread out over several units, you can actually get lost, or at least buried so deep that you can't figure out how best to get to where you want to go. To aid you with navigation, the debugger provides two mechanisms: the Debug/Call Stack and Find Procedure commands.

The Call Stack

Each time a procedure or function is called, Turbo Pascal remembers the call and the parameters passed to it; when you exit that procedure or function, then the call is "forgotten." This is known as the *call stack.*

Whenever your program pauses because of a breakpoint or a single-step command, you can ask to see the current call stack by using the Debug/ Call Stack command (*Ctrl-F3*). This pops up a window that displays the list of procedure/function calls currently active on the stack.

Suppose you have a mailing list program that reads in a list of names and addresses, then prints them out in double-column format. The program itself is named *Address.* The procedure that prints out the list of names is called *PrintList,* which in turn calls procedures named *PutLeft* and *PutRight* to write the data out to the two columns. If you set a breakpoint within *PutLeft,* run the program, then choose Debug/Call Stack once the program pauses, the Call Stack window might look like this:

```
─── Call Stack ───
PUTLEFT('Allan Jones')
PRINTLIST((...),36)
ADDRESS
```

This tells you that you're currently in the procedure *PutLeft*, which was passed (as a parameter) the string `'Allan Jones'`. *PutLeft* was called from *PrintList*, which was passed an array (arrays aren't shown, due to space considerations), and the value 36. *PrintList* itself was called from the main body of the program *Address*. (If your program doesn't have a name, then this entry will read `PROGRAM`.)

This is known as a call "stack" because it's set up as a stack—a last-in-first-out (LIFO) data structure that models how subprogram calls are handled. Each time you call a function or procedure, that call appears (or is *pushed*) on the top of the stack; when you exit that procedure or function, the corresponding call disappears (or is *popped*) from the top of the stack.

The call stack serves another important function: It allows you to look back through the sequence of calls. When you first bring up the call stack, the topmost call is highlighted. You can use the arrow keys to move up and down through the stack. If you press *Enter*, you will be taken to the last active point within that program or subprogram.

For example, consider the previously described stack. If you choose the entry for *PrintList* and press *Enter*, you'll be taken to the spot within *PrintList* where *PutLeft* was called. Likewise, if you choose the entry for *Address* and press *Enter*, you'll be shown the spot in the main body of the program where *PrintList* was called. Finally, if you choose the topmost entry and press *Enter*, you'll always be returned to the execution bar.

Moving around like this doesn't change where you're executing. If you press *F7* (to single step), you'll be right back in *PutLeft*, and the execution bar will highlight the next line in *PutLeft* just as if you'd never left. However, this does give you a very quick mechanism for getting out a series of function/procedure calls: Use the call stack to get back out to where you want to be, move the cursor to the line following the procedure/function call at that level, and press *F4*.

As a second example, consider the following small program:

```
program TestPower;

function Power(Base,Exp : word) : longint;
begin
  if Exp <= 0 then
    Power := 1
  else
    Power := Base * Power(Base,Exp-1)
```

```
end; { of func Power }

begin { main body of TestPower }
  Writeln('2^14 = ',Power(2,14))
end. { of program TestPower }
```

Type this in, compile it, and set a breakpoint on the second line of the procedure *Power* (the line `Power := 1`). Now run the program. When it pauses, choose the Debug/Call Stack command. The Call Stack window will look like this:

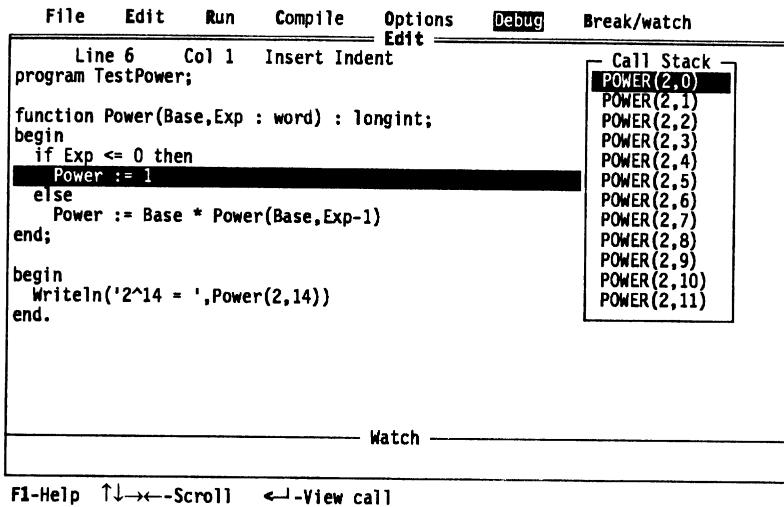


Figure 6.2: The Call Stack Window

Note that *TestPower* didn't show up at the bottom of the window. That's because the Call Stack window will only show nine procedure/function calls at once (12 on Hercules, EGA, and VGA). However, all the calls are still there; just use the *Up* and *Down arrow* keys to move through them. The call stack will track up to 128 nested calls.

Finding Procedures and Functions

Sometimes, in the middle of debugging, you want to find a particular procedure or function in order to set a breakpoint, execute to that point, check the parameter list, look at the variables, or any number of other reasons.

If your source code is spread out among multiple files, you'll love the **Debug/Find Procedure** command. When you choose this command, a small window pops up, asking you to enter the name of a procedure or function. After you type in an identifier and pres *Enter*, Turbo Pascal checks its internal tables to find where that subprogram is located, loads in the appropriate source file (if necessary), and puts you in the Edit window with the cursor positioned at the beginning of the procedure or function.

There are three important things to remember about using the **Debug/Find Procedure** command:

- Find Procedure does *not* affect your current debugging state. In other words, if you're paused at some point in your program, you are still paused there, and choosing **Run/Trace Into** (*F7*) will execute that line in your program, not the procedure or function you've just located.
- Find Procedure places the cursor at the first executable line of that procedure or function, rather than on the procedure or function header. This means you can choose **Run/Go to Cursor** (*F4*) to execute from your current position to the start of that procedure or function.
- You can only use this command if you have compiled your program and debug information is available for the procedure or function.

There is a possibility of ambiguity in giving the name of a procedure or function to find, since you could have routines with the same name in several different places in your program (in units, nested inside of other routines, and so on). You can *qualify* the routine's name by preceding it with the name of the unit or program containing it, as well as any procedures or functions that might enclose it; for example, *module.proc.proc.*<etc.>*.proc*. If you modify the source code and the file position (or even name) of a procedure or function is changed, the **Debug/Find Procedure** command won't know about any of those changes until you recompile. If you first compile program *TestPower* (see the section on "The Call Stack," page 136) and then delete the blank line above the declaration of function *Power*, **Debug/Find Procedure** will put the cursor on the **if...then** instead of the **begin**.

General Issues

You've learned all about how to use the debugger; let's cover some of the issues that might come up while you're actually using it.

How to Write Programs for Debugging

There are some simple things you can do to make your programs easier to debug. In most cases, don't put more than a single statement on a line. Since the debugger executes on a line-by-line basis, this ensures that no more than one statement will be executed each time you press *F7*.

At the same time, recognize that there are cases when you might want to put multiple statements per line. If there is a list of statements you have to step through, but which aren't really relevant to the debugging, feel free to bunch them up into one or two statements so that you can step through them more quickly. That's why in one of the earlier examples we wrote

```
W := 10; X := 20; Y := 30; Z := 40;
```

instead of

```
W := 10;
X := 20;
Y := 30;
Z := 40;
```

You can also organize your variable declarations so that the ones you are most likely to put in the Watch window are nearest the initial **begin** statement of the procedure or function. When you step into that procedure or function, you can quickly move the cursor through the list, using **B**reak/Watch/**A**dd Watch (*Ctrl-F7*) to add each variable as a watch.

In a similar fashion, if there are expressions that you commonly want to watch or evaluate at certain points in your program, insert them as comments. When you get to that point, you can move the cursor to the start of the expression and copy it into the Add Watch or Evaluate box. This is especially helpful if the expression is a complex one, involving typecasting, format characters, array elements, or record fields.

Finally, the best debugging is preventive debugging. A well-designed, clearly-written program will not only have fewer bugs, but it will make it easier for you to track down and fix what few bugs there are. Here are some basics to remember when you're writing your program:

- Program incrementally. When possible, code, test, and debug your program one (small) section at a time. Get each section working before moving on to the next section.
- Break your program into modules: units, procedures, functions. Avoid writing procedures or functions longer than about 25 lines; if one gets bigger than that, try breaking it up into a few smaller procedures and functions.

- When possible, pass information through parameters only, instead of referencing global variables inside procedures and functions. This avoids side effects and also makes the code easier to debug, since you can easily watch all information coming in and out of a given procedure or function.
- Don't get clever too quickly. Concentrate on making it work right before trying to make it work fast.

Memory Issues

It is possible to run out of memory while debugging a large program. After all, Turbo Pascal is holding the editor, compiler, debugger, current source code file, executable code, symbol tables, and any other debugging information in memory—all at the same time. You can monitor the amount of free memory with the Compile/Get Info command.

If you do run out of memory, however, there are a number of steps you can take to help reclaim some memory so that you can get your debugging done:

- If you have EMS memory, make sure that at least 64K bytes is available when you run TURBO.EXE; that way, Turbo Pascal can store the edit buffer in EMS and leave more main memory free for your program.
- Compile your program to disk instead of to memory (set the Compile/ Destination switch to Disk).
- Linking to disk will assist in preserving memory; just set Options/ Linker/Linker Buffer to Disk.
- If you don't use dynamic variables in your program, set the minimum heap size to zero using the Options/Compiler/Memory Sizes command or a $*M* compiler directive. Likewise, decrease the stack size if you need less than the default 16K bytes.
- Remove any RAM-resident programs, such as SideKick Plus or SuperKey.
- Remove seldom-used units from TURBO.TPL (using TPUMOVER.EXE). Units in TURBO.TPL take up memory even if they aren't used by the running application. It is recommended that you always leave the *System* unit in TURBO.TPL, since every program uses it.
- If you don't have EMS memory, use TINST to install a smaller edit buffer size. By default, the edit buffer takes up 64K bytes of memory, even though the largest file in your project may be substantially smaller.

- Place a {$*L*-} directive in units where you don't need local symbols. You will still be able to trace and single-step in such units, but you won't be able to examine local variables, constants, and parameters.
- Place a {$*D*-} directive in units that need no debugging. This suppresses the generation of all debug information; procedures and functions in a {$*D*-} unit will be treated just like standard Turbo Pascal routines—you won't be able to trace into them.
- Turn off any directives that generate error-checking code, such as {$*S*-} and {$*R*-}. This will reduce the size of your program somewhat.
- Organize your program into overlays, using the *Overlay* unit and $*O* compiler directives. By using overlays, you can debug very large applications from inside the Turbo Pascal environment. For more information on overlays, refer to Chapter 13 in the *Reference Guide*, "Overlays."
- Attempt to debug sections of code (such as units) apart from the rest of the program. In other words, make a copy of your program and cut away any sections not needed for your debugging efforts. This isn't always possible, but the attempt itself may result in a program with better, more modular design.

If you still run out of memory after taking the above steps, your program is simply too large to be debugged under the IDE. By turning off integrated debugging, using the Debug/Integrated Debugging switch, you can reclaim even more memory so you can run your program; to debug it, you should use the Turbo Debugger and turn Debug/Stand-alone Debugging to On.

Recursive Routines

Recursion is a programming technique where a procedure calls itself (directly or indirectly). For example, the function *Power* shown in an earlier example is recursive, because it calls itself to calculate the value it needs to return.

There are some considerations to keep in mind when debugging recursive code. First, deep levels of recursion can eat up lots of system stack space, which can have other side effects (such as your program halting or crashing due to stack overflow). This is a general danger of using recursion under any circumstance; just be aware that, if your program crashes while debugging, it may well be due to stack overflow rather than anything you did with the debugger.

Also, if you have deep levels of recursion, you may not be able to find your way out immediately with the call stack. That's because the call stack is limited to the last 128 function/procedure calls. You can, however, go to the bottom of the stack, use it to find the oldest call, pop out to that spot, then use the call stack again.

Each time a function is called recursively, it creates a new set of local variables and pass-by-value (non-**var**) parameters. If you have added these to the Watch window, be aware that these values will "float" to reflect the currently active local data.

Where Debugging Won't Go

There are some cases where you can't trace into a given function or procedure. This is usually—but not always—because the source isn't available. These situations include the following:

- Any **inline** procedure or function; that is, any procedure or function of type **inline**. That's because these aren't procedure or function calls at all; the associated machine language is inserted in place of the "call." Such a call is treated as a single statement.

  Note that you can trace into procedures and functions that happen to use **inline** statements. However, in that case, each **inline** statement is treated as a single line, no matter how many lines it occupies. This follows the same rule as other statements; that is, if a single statement takes up several lines, it is treated by **Run/Trace** Into and **Step** Over (*F7* and *F8*) as a single line.
- Any Turbo Pascal routine from one of the standard units (*Crt, Dos, Graph, Graph3, Overlay, Printer, System, Turbo3*).
- Any **external** procedure or function.
- Any **interrupt** procedure or function.

  Note: You cannot debug a program that takes over interrupt 9 in the integrated development environment. You can use Turbo Debugger, Borland's stand-alone debugger, instead. Also, interrupt service routines (ISRs) should not be debugged using the IDE.
- Any procedure, function, or initialization code contained in a unit that was not compiled with the {$*D*+} directive (or with **Options/Compiler/ Debug** Information set to **On** in the IDE).
- Any procedure, function, or initialization code contained in a unit whose source code cannot be found. If it's not in the current or the unit directory, or if its source code is in a file named something other than *unitname*.PAS (where *unitname* is the name of the unit as given in the

uses clause), the IDE will prompt you for the correct file name. If you enter a null file name, or if you press *Esc*, the debugger will move on as if debug information were not available.

- Any procedure set up as an exit procedure. If you step through your program with **Run/Trace Into** (*F7*), you'll never step into an *Exit* procedure. Note, however, that you can set a breakpoint in an *Exit* procedure, and the debugger will break appropriately when the execution bar arrives at your breakpoint.

Common Pitfalls

There are a few problems that you often run into while debugging. Here's a list of things to watch out for:

- Not generating the global and local debug information needed. By default, both of these switches are on. If you have problems stepping into a program or unit, put {$D+,L+} at the start of every program or unit you wish to debug.
- Starting to debug another program without clearing the breakpoints and Watch expressions from the previous one. Before loading in a new program to debug, you should always execute the following commands: **Run/Program Reset** (*Ctrl-F2*), **Break/Watch/Remove All Watches**, and **Break/Watch/Clear All Breakpoints**.
- Trying to compile and run another program when the previous one is still set up as the primary file. Use the **Compile/Primary File** command to clear out the previous name or set a new one.
- Press *N* when you get the `Source modified, rebuild? (Y/N)` prompt. This means that you've modified a source file while debugging, and the debugger's line-number tables may no longer be valid. This can throw off breakpoints, stepping, and other debugging activities. If you just accidentally typed a character and then deleted it, you're probably safe in pressing *N*; if you've inserted or deleted lines, though, you're better off pressing *Y*, because the machine code you're debugging doesn't match the source code you're looking at.

Error-Handling

In addition to the integrated debugger, Turbo Pascal provides several compiler directives and language features to help you trap programming errors. This section briefly describes some of those features.

You can insert run-time error checking for yourself by disabling the generation of automatic error-checking code and writing your own error-handling routines. Let's take a look at some examples.

Input/Output Error-Checking

If you ran this program, entered the values *45* and *8x* when prompted, and then pressed *Enter*, what would happen?

```
program DoSum;
var
  A,B,Sum : integer;
begin
  Write('Enter two numbers:  ');
  Readln(A,B);
  Sum := A + B;
  Writeln('The sum is ',Sum);
  Readln
end.
```

You'd get a run-time error (106, in fact) and the cursor would be positioned at the statement

```
Readln(A,B);
```

What happened? The program expected an integer value and you entered non-numeric data—*8x*—which generated a run-time error.

In a short program like this, such an error isn't a big bother. But what if you were entering a long list of numbers and had gotten through most of the list before making this mistake? You'd be forced to start all over again. Worse yet, what if you wrote the program for someone else to use, and *they* slipped up?

Turbo Pascal allows you to disable automatic I/O error-checking and test such errors for yourself within the program. To turn off I/O error-checking at some point in your program, include the compiler directive {$*I*-} in your program (or the **O/C/I/O**-Checking option). This instructs the compiler to prohibit the production of code that checks for I/O errors.

Let's revise *DoSum* so that it does its own I/O-checking:

```
program DoSum;
var
  A,B,Sum  : integer;
  IOCode   : integer;
begin
  repeat
```

```
    Write('Enter two numbers:  ');
    {$I-}                                 { Disable automatic I/O error-checking }
    Readln(A,B);
    {$I+}                                 { Enable automatic I/O error-checking }
    IOCode := IOResult;
    if IOCode <> 0 then
      Writeln('Bad data:  please enter again')
  until IOCode = 0;
  Sum := A + B;
  Writeln('The sum is ',Sum);
  Readln
end.
```

First, you disable automatic I/O error-checking with the {$*I*-} compiler directive. Then you put the input code into a **repeat..until** loop, because you're going to repeat the input until the user gets it right. The *Write* and *Readln* statements are the same, but after them comes the statement

```
IOCode := IOResult;
```

You've declared *IOCode* as a global variable, but what's *IOResult*? It's a predefined function that returns the error code from the last I/O operation; in this case, *Readln*. If no error occurred, then the value returned is 0; otherwise, a nonzero value is returned, indicating what happened. Once you've called *IOResult*, it "clears" itself and will return 0 until another I/O error occurs. This is why you assign *IOResult* to *IOCode*—so that you can test the result in both the **if** statement and the **until** clause.

A similar structure can be used for error-checking while opening files for input. Look at the following code sample:

```
var
  FileName  : string[40];
  F         : text;
begin
  Write('Enter file name: ');
  Readln(FileName);
  Assign(F,Filename);
  Reset(F);
  ...
```

This code fragment asks you to enter a file name, then tries to open that file for input. If the file you name doesn't exist, the program will halt with a run-time error (02). However, you can rewrite the code like this:

```
var
  FileName  : string[40];
  F         : text;
  IOCode    : integer;
begin
```

```
{$I-}
repeat
  Write('Enter file name: ');
  Readln(FileName);
  Assign(F,Filename);
  Reset(F);
  IOCode := IOResult;
  if IOCode <> 0 then
    Writeln('File ', FileName, ' does not exist, try again')
until IOCode = 0;
{$I+}
...
```

Using these and similar techniques, you can create a crash-proof program that lets you make mistakes without halting your program.

Range-Checking

Another common class of run-time errors involves out-of-range or out-of-bounds values. Some examples of how these can occur include assigning too large a value to an integer variable or trying to index an array beyond its bounds. If you want it to, Turbo Pascal will generate code to check for range errors. It makes your program larger and slower, but it can be invaluable in tracking down any range errors in your program.

Let's revisit an earlier example:

```
program RangeTest;
var
  List : array[1..10] of integer;
  Indx  : integer;

begin
  for Indx := 1 to 10 do
    List[Indx] := Indx;
  Indx := 0;
  while (Indx < 11) do
  begin
    Indx := Indx + 1;
    if List[Indx] > 0 then
      List[Indx] := -List[Indx]
  end;
  for Indx := 1 to 10 do
    Writeln(List[Indx])
end.
```

If you enter this program, then compile and run it, it will get stuck in an infinite loop. To see why, look carefully at this code: The **while** loop

executes 11 times, not 10, and the variable *Indx* has a value of 11 the last time through the loop. Since the array *List* only has 10 elements in it, *List[11]* points to some memory location outside of *List*. Because of the way variables are allocated, *List[11]* happens to occupy the same space in memory as the variable *Indx*. This means that when *Indx* = 11, the statement

```
List[Indx] := -List[Indx]
```

is equivalent to

```
Indx := -Indx
```

Since *Indx* equals 11, this statement sets *Indx* to –11, which starts the program through the loop again. That loop now changes additional bytes elsewhere, at the locations corresponding to *List[-11..0]*, which is undesirable. Also, since *Indx* never ends the loop at a value greater than or equal to 11, the loop will never end.

How do you check for things like this? You can insert {$*R*+} at the start of the program to turn range-checking on. Now, when you run it, the program will halt with run-time error 201 (out-of-range error, because the array index is out of bounds) as soon as you hit the statement *if List[Indx] > 0* with *Indx* = 11. If you were running under the integrated environment, it would automatically take you to that statement and display the error. (Range-checking is off by default; turning range-checking on makes your program larger and slower, but is strongly advised until your program is thoroughly debugged.)

There are some situations—usually in advanced programming—in which you may need to violate range bounds, most notably when working with dynamically allocated arrays or when using *Succ* and *Pred* with enumerated data types.

You can selectively implement range-checking by placing the {$*R*-} directive at the start of your program. For each section of code that needs range-checking, place the {$*R*+} directive at the start of it, and the {$*R*-} directive at the end. For example, you could have written the preceding loop like this:

```
while Indx < 11 do
begin
  Indx := Indx + 1;
  {$R+}                                    { Enable range-checking }
  if List[Indx] > 0 then
    List[Indx] := -List[Indx]
  {$R-}                                    { Disable range-checking }
end;
```

Range-checking will be performed only in the **if..then** statement and nowhere else, unless, of course, you have other {$*R*+} directives elsewhere.

Other Error-Handling Abilities

Turbo Pascal gives you the ability to perform other error-handling techniques, but because those techniques are described more fully in other parts of this manual, we'll only touch on them briefly in this section.

When your program terminates, either normally or through a run-time error, a standard exit procedure is called that's linked in with your program. Turbo Pascal lets you add in your own exit procedures, which are called *before* the standard exit procedure. In fact, each unit can have its own exit procedure, so that you can have automatic cleanup code, as well as the usual automatic initialization code. Exit procedures are described in more detail in Chapter 16 of the *Reference Guide*, "Turbo Pascal Reference Lookup."

If you try to allocate memory (through a call to *New* or *GetMem*) and there isn't sufficient memory on the heap, a heap error procedure is automatically called, which simply causes your program to exit with a run-time error. You can, however, install your own heap error procedure to handle things as you wish, like deallocating dynamic structures no longer needed or simply causing *New* or *GetMem* to return a **nil** pointer. Heap error procedures are described in more detail in Chapter 16 of the *Reference Guide*, "Turbo Pascal Reference Lookup."

If you're using the *Graph* unit, you can perform error-checking much as you do for I/O error-checking. One function in the unit, *GraphError*, returns an error result set by many of the graphics routines. Chapter 12 of the *Reference Guide*, "Standard Units," provides you with details on how to use this and the error codes that are generated.

The *Overlay* unit contains an integer variable, *OvrResult*, that stores the result code from the last operation performed by the overlay manager. Similarly, the *Dos* unit stores its result codes in the variable *DosError*.

Turbo Debugger

Turbo Pascal 5.0 includes full support for Borland's Turbo Debugger—a powerful, stand-alone debugger that works on Turbo Pascal, Turbo C, and Turbo Assembler .EXE files. Turbo Debugger includes both source- and machine-level debugging, powerful breakpoints (including breakpoints with conditionals or expressions attached to them), and it lets you debug

huge applications via virtual machine debugging on a 80386 or two-machine debugging (connected via the serial port).

Turbo Debugger is an industrial-strength debugger that includes scores of powerful features and—best of all—you already know how to use its windowed environment and hot keys because they're the same as Turbo Pascal IDE's.

Let's assume you have a Pascal program called MYPROG.PAS; here's how you'd generate MYPROG.EXE for use with Turbo Debugger.

If you're using the integrated development environment:

1. Load TURBO.EXE and pull MYPROG.PAS into the editor (Press *F3*, type `MYPROG` and press *Enter*).
2. Set **C**ompile/**D**estination to Disk.
3. Set **D**ebug/**S**tand-alone Debugging to On.
4. Compile your program (*F9*).

If you're using the command-line compiler, type the following DOS command and press Enter:

```
TPC /V MYPROG
```

(No matter which compiler you're using, make sure both the {$*D*} and {$*L*} compiler directives are on. {$*D*} controls generation of debug information and can be enabled by placing {$*D*+} in the source code, by setting **O**ptions/**C**ompiler/**D**ebug Information to On in the IDE, or by specifying /$*D* on the TPC command line. {$*L*} controls whether local symbol information is retained and can be enabled by placing {$*L*+} in the source code, by setting **O**ptions/**C**ompiler/**L**ocal Symbols to On in the IDE, or by specifying /$*L*+ on the TPC command line. They are both enabled by the default, so if you haven't changed the defaults and you haven't placed {$*D*} or {$*L*} directives in your source code, you're ready to go on.)

Assuming that your program compiles, MYPROG.EXE will now exist on disk. Don't be surprised if it's a little larger than you'd expect: The symbol and line-number information generated by the compiler for debugging purposes have been added to the end of the .EXE. To debug this .EXE using Turbo Debugger, just type the following DOS command and then press *Enter*:

```
TD MYPROG
```

TD is the Turbo Debugger, and *myprog* is your program's name. You'll need to be in the same directory as TD.EXE or have a path (or batch file) load it for you. If your program takes command-line parameters, you can simply specify them after the program name, like this:

```
TD MYPROG PARAM1 PARAM2
```

That's all there is to it! Between the built-in bug prevention of the Turbo Pascal language, the Pascal debug support provided by the integrated development environment, and the powerful combination of Turbo Pascal and Turbo Debugger—bugs don't stand a chance. Happy coding!

C H A P T E R

7

All About the Integrated Environment

This chapter is designed to help you learn how to use all the options available in the Turbo Pascal integrated development environment (IDE). You'll learn how to use the editor, move from window to window and menu to menu, and choose menu commands. Each menu item is then discussed in detail. If a menu item has a hot key or default setting, this is displayed onscreen.

To load the Turbo Pascal integrated development environment (IDE), type TURBO and press *Enter* at the DOS prompt.

Turbo Pascal Command-Line Switches

The Turbo Pascal integrated development environment accepts the following command-line switches:

- */C* causes a configuration file to be loaded. Enter the TURBO command, followed by */C* and the configuration file name, with no space between the two:

```
TURBO /CMYCONFIG.TP
```

- */B* causes Turbo Pascal to recompile and link all the files in your program, print the compiler messages to the standard output device, and then return to DOS. This switch allows you to invoke the IDE from a batch file, so you can automate project builds. Before the build, Turbo Pascal will load either a default configuration file or one given specially

with the */C* switch. Turbo Pascal determines what .EXE to build based on the primary file or the file currently loaded into the editor, in that order of precedence. Enter the `TURBO` command with either */B* alone or */C* and the configuration file name followed by */B*.

```
TURBO /CMYCONFIG.TP /B
TURBO /B
```

Unless the loaded configuration file has a primary file, you can specify the name of a program to be compiled and linked on the command line. Type in the program name after the `TURBO` command, followed by */B*.

```
TURBO MYFIRST /B
```

- */M* lets you do a make rather than a build (that is, only outdated source files in your project are recompiled and linked). Follow the instructions for the */B* switch, but use */M* instead.
- */D* causes Turbo Pascal to work in dual monitor mode, if it detects appropriate hardware. Otherwise, the */D* switch is ignored. Dual monitor mode is used when you are running or debugging a program, or shelling to DOS (**File/OS Shell**).

  You can type in the */D* switch on Turbo Pascal's command line to enable dual monitor mode, as long as the required hardware is present (for example, a monochrome card and a color card). If your system has two monitors, DOS treats one monitor as the active monitor. Use the DOS MODE command to switch between the two monitors (`MODE CO80`, for example, or `MODE MONO`). In dual monitor mode, the normal IDE screen will appear on the inactive monitor, and program output will go to the active monitor. Thus, when you type `TURBO /D` at the DOS prompt on one monitor, Turbo Pascal will come up on the other monitor. When you want to test your program on a particular monitor, you must exit the IDE, switch the active monitor to the one you want to test with, then issue the `TURBO /D` command again. Program output will then go to the monitor where you typed the `TURBO` command.

  Warning:

  - Do not change the active monitor (by using the DOS MODE command, for example) while you are in a DOS shell (**File/OS Shell**).
  - User programs which directly access ports on the inactive monitor's video card are not supported, and can cause unpredictable results.
  - When you run or debug programs that explicitly make use of dual monitors, do not use the dual monitor switch (*/D*).

  Note: The **Run/User Screen** command, which normally displays the program output screen, has no effect in dual monitor mode.

- /P controls palette-swapping on EGA video adapters. Type `TURBO /P MYFIRST` to restore the EGA palette each time the screen is swapped. In general, you don't need to use the /P switch unless your program modifies the EGA palette registers or uses the Borland Graphics Interface (BGI) to change the palette.

Menu Structure

Figure 7.1 (pages 156 and 157) show the complete structure of Turbo Pascal's main menu and its successive pull-down menus.

When you first get into Turbo Pascal's integrated environment, you're positioned at the main menu bar. You can choose a main menu item by pressing the key corresponding to the first letter of the menu name: **F**ile, **E**dit, **R**un, **C**ompile, **O**ptions, **D**ebug, and **B**reak/Watch.

All the commands except **E**dit open up pull-down menus with several other items. You can use the *Up* and *Down arrow]* keys on your keyboard to move the highlight bar up and down the list of commands, pressing *Enter* when the bar is on the command you want. To leave a menu, just press *Esc*. You can move from one pull-down menu to another by using the *Left arrow* and *Right arrow* keys or by pressing *Alt* and the highlighted letter of a menu item.

In general, when you choose a menu item, Turbo Pascal performs the action and puts the menus away, leaving you in the active window. Exceptions to this are when the menu item is a toggle (for example, **O/C/R**ange-Checking) or a setting (for example, **C/P**rimary File). Also, when an error occurs in the processing of a menu item, you remain in the menus so you can correct the problem and try the operation again.

The specific rules for when Turbo Pascal puts menus away are:

- toggles are "sticky" (choosing one leaves the menus displayed)
- string settings are "sticky"
- pressing *Esc* while in an input or dialog box leaves you in the menus
- pressing *Esc* from a pop-up menu takes you out one level of menus
- pressing *Esc* from a pull-down menu takes you out of the menus to the active window
- error conditions leave the menus displayed

If you're in any of the windows, you can get to the main menu by pressing *F10*, which toggles you back and forth from the active window to the main menu. If you're in the Edit window, you can also get to the main menu by pressing *Ctrl-K D* or *Ctrl-K Q*.

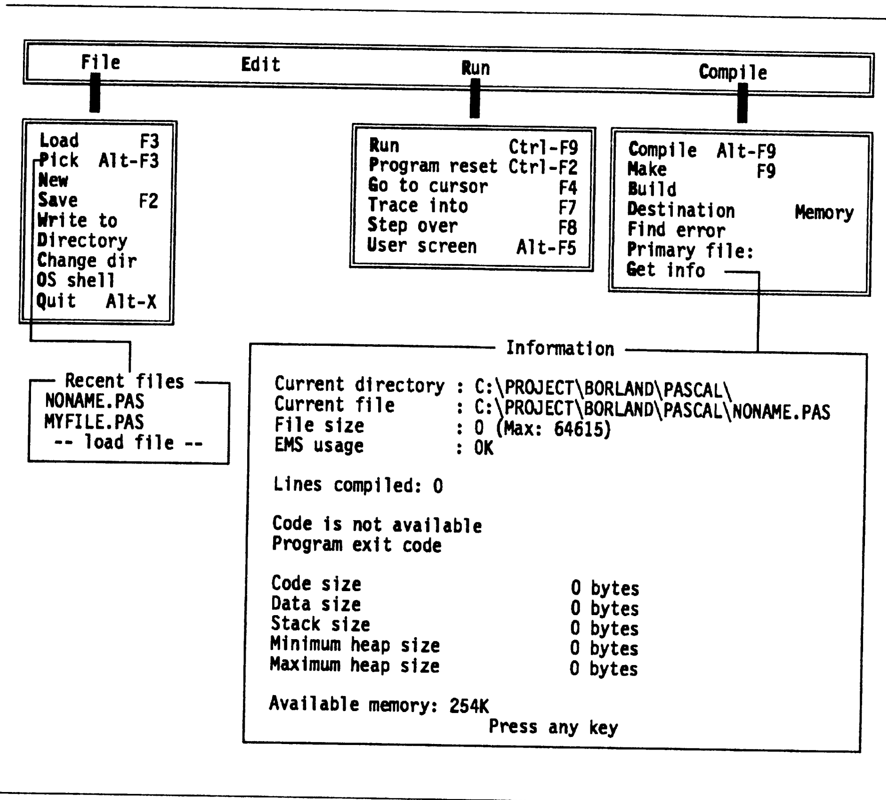


Figure 7.1: Turbo Pascal's Menu Structure

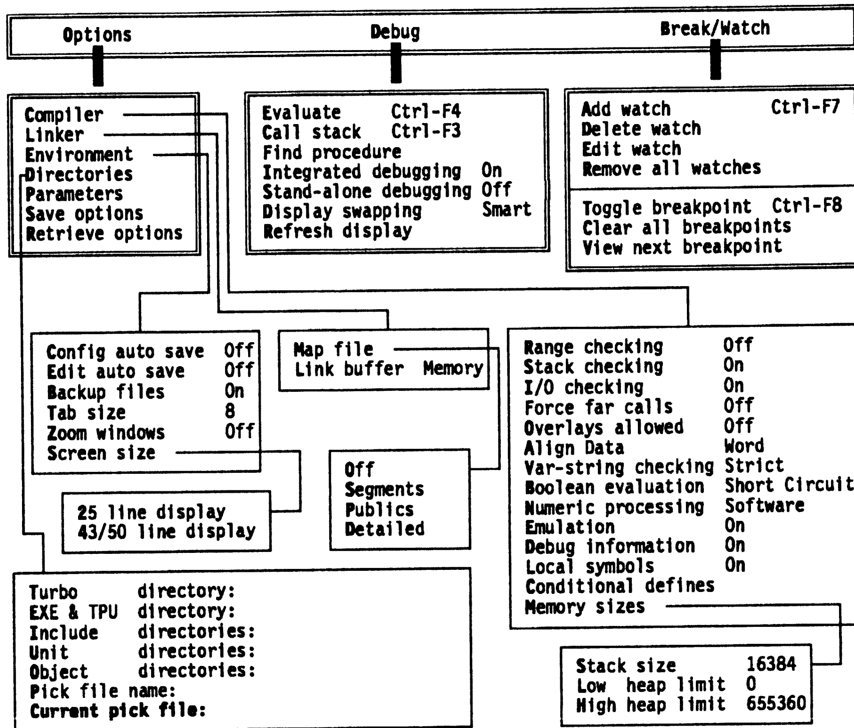


Figure 7.1: Turbo Pascal's Menu Structure, continued

You can also press an *Alt* key and the first letter of the main menu item you'd like to get to; for example, *Alt-O* will take you to the **O**ptions menu.

Here are the seven main menu commands:

File	Handles files (loading, saving, picking, creating, writing to disk) and manipulates directories (listing, changing), quits the program, and invokes DOS.
Edit	Lets you create and edit source files in the built-in text editor.
Run	Lets you execute and single-step through your program (using the integrated debugger), and view program output.
Compile	Compiles and makes your source code into compiled units and executable files, finds run-time errors, and reports system information.
Options	Allows you to specify new defaults for compiler options (such as range-checking, numeric coprocessor support, and memory sizes) and define an input string of command-line parameters. Lets you set linker options (disk-based or in memory) and generate .MAP files. Also records the **T**urbo, **E**XE & TPU, **I**nclude, **U**nit, and **O**bject directories, saves compiler options, and loads options from the configuration file.
Debug	Lets you examine and modify any variable, evaluate expressions, check the current call stack, quickly locate procedures and functions in your source text, and turn on and off debugging. Also lets you generate debug tables for use with the stand-alone Turbo Debugger.
Break/Watch	Allows you to add, edit, or delete items in the Watch window; lets you set, remove, or view breakpoints within your program.

There are three general types of Turbo Pascal menu items:

- **Commands** perform a task (running, compiling, quitting, storing options, and so on).
- **Toggles** let you switch a Turbo Pascal feature On or Off (**R**ange-Checking, **E**dit Auto Save, and so on) or cycle through and choose one of several options by repeatedly pressing the *Enter* key till you reach the item desired (such as **D**estination or **B**oolean Evaluation).

- **Settings** allow you to specify certain compile-time and run-time information to the compiler, such as directory locations, primary files, and so forth.

The Bottom Line

Whether you're in one of the windows or one of the menus, the line at the bottom of the screen provides at-a-glance function-key help for your current environment setting. When you're in the main menu or the Edit window, the bottom line looks like this:

```
F1-Help   F5-Zoom   F6-Switch   F7-Trace   F8-Step   F9-Make   F10-Menu
```

To see what *Alt*-key combinations are available at any time, hold down the *Alt* key for a few seconds. The bottom line will list what functions you can perform when you combine other keys with the *Alt* key.

```
Alt:   F1-Last help   F3-Pick   F6-Swap   F9-Compile   X-Exit
```

If you're in the Watch window, the bottom line looks like this:

```
F1-Help  F5-Zoom  F6-Switch  F10-Menu  Ins-Add  Del-Delete  ↵-Edit watch
```

The bottom line also changes when you're in input boxes, error boxes, and so on. For instance, when you're in the Evaluate window the line looks like this:

```
F1-Help  F7-Trace  F8-Step  F10-Menu  TAB-Cycle  ↵-Evaluate  →-More Text
```

The Edit Window

In this section, we describe the components of the Turbo Pascal Edit window and explain how to use the editor.

First, to get into the Edit window, press *Enter* when the highlight is positioned on the Edit option on the main menu (or press *E* from anywhere on the main menu), or press *F10* (a toggle between the active window and the main menu). To get into the Edit window from anywhere in the system, including the User screen, just press *Alt-E*. (Remember, *Alt-E* is just a shortcut for *F10-E*.) Once you're in the Edit window, notice that there are double lines at the top of it and its name is highlighted—this means it's the active window.

Besides the body of the Edit window, where you can see and edit several lines of your source file, the Turbo Pascal Edit screen has two information lines you should note: an Edit status line and the bottom line.

The status line at the top of the Edit window gives information about the file you are editing, where in the file the cursor is located, and which editing modes are activated:

```
Line n    Col n    Insert    Indent    Tab    Fill    Unindent * C:FILENAME.EXT
```

Line n Cursor is on file line number *n*.

Col n Cursor is on file column number *n*.

Insert Insert mode is On; toggle Insert mode On and Off with *Ins* or *Ctrl-V*.

Indent Autoindent mode is On. Toggle it Off and On with *Ctrl-O I*.

Tab Tab mode is On. Toggle it On and Off with *Ctrl-O T*.

Fill When Tab mode is On, the editor will fill the beginning of each line optimally with tabs and spaces. This option is toggled with *Ctrl-O F*.

Unindent The *Backspace* key will "outdent" a level whenever the cursor is on the first nonblank character of a line or on a blank line. This option is toggled with *Ctrl-O U*.

C:FILENAME.EXT The drive (C:), name (FILENAME), and extension (.EXT) of the file you are editing.

An asterisk (*) will appear in front of the file name if it has been modified since the last save. The line at the bottom of the screen displays available hot keys.

```
F1-Help   F5-Zoom   F6-Switch   F7-Trace   F8-Step   F9-Make   F10-Menu
```

To select one of these functions, press the listed key:

F1-Help Opens a Help window that provides information about the Turbo Pascal editor commands.

F5-Zoom Expands the active window to full screen. Toggle *F5* to get back to the split-screen environment.

F6-Switch Switches from one active window to another.

F7-Trace Lets you run your program one line at a time in debugging mode, tracing into procedures and functions as they are called.

F8-Step Lets you run your program one line at a time in debugging mode, stepping over procedure and function calls.

F9-Make Makes (compiles and links) your program.

F10-Main menu Toggles between the menus and the active window.

The editor uses a command structure similar to that of the SideKick NotePad and the original Turbo Pascal editor; Appendix B, "Using the Editor," describes the editor commands in detail.

If you're entering code in the editor, you can press *Enter* to end a line (the editor has no wordwrap). The maximum line width is 249 characters; you'll get a beep if you try to type past that. (Note that the compiler only recognizes characters out to column 126.) The Edit window is 77 columns wide. If you type past column 77, the text you've already entered moves to the left as you type. The Edit window's status line gives the cursor's location in the file by line and column.

After you've entered your code into the Edit window, press *F10* to invoke the main menu. (Press *F2* to save the file if you want.) Your file will remain onscreen; you need only press *E* (for Edit) at the main menu to return to it, or *Alt-E* from anywhere.

Working with Source Files

When you invoke the Edit window before loading a particular file, the Turbo Pascal editor automatically names the file NONAME.PAS. At this point you have all the features of the editor at your fingertips. You can

- create a new source file either as NONAME.PAS or another file name
- load and edit an existing file
- pick a file from a list of already edited files, and then load it into the Edit window
- save the file viewed in the Edit window
- write the file in the editor to a new file name

Creating a New Source File

To create a new file, select one of the following methods:

- At the main menu, choose File/New. This always opens the Edit window with a file named NONAME.PAS.
- At the main menu, choose File/Load (*F3*). The Load File Name prompt box opens; type in the name of your new source file.

Loading an Existing Source File

To load and edit an existing file, you can choose two options: File/Load or File/Pick.

If you choose File/Load (or press *F3*) at the main menu, you can either

- Type in the name of the file you want to edit; paths are accepted—for example,

  ```
  C:\TP\TESTFILE.PAS
  ```

- Or enter a mask in the Load File Name prompt box (using the DOS wildcards * and ?), and press *Enter*. Entering `*.*` will display all the files in the current directory as well as any other directories. Directory names are followed by a backslash (\). Selecting a directory displays the files in that directory. Entering `C:*.PAS`, for example, will bring up *only* the files with that extension in the root directory. You can change the wildcard mask by pressing *F4*. (For more on directories, look at Appendix E, "A DOS Primer."

  Press the *Up, Down, Left,* and *Right arrow* keys to highlight the file name you want to choose. Then press *Enter* to load the chosen file; you are placed in the Edit window. If you press *Enter* when you're positioned on a directory name, you'll get a new directory box.

Pick lets you quickly pick the name of a previously loaded file. So, if you choose File/Pick (*Alt-F3*) (see the discussion of the Pick command later in this chapter), you can do one of these two things:

1. Press *Alt-F* then *P* to bring up your pick list (or press the shortcut *Alt-F3*). Use the *Up* and *Down arrow* keys to move the selection bar to the file of your choice and press *Enter* to load it.
2. While the Edit window is active, press *Alt-F6*. This will load the last file that was in the editor.

Saving a Source File

In order to save a source file from the main menu, you can choose File/Save; you can use the File/Save hot key *F2* from anywhere in the system.

Writing an Output File

You can write the file in the editor to a new file or overwrite an existing file. You can write to the current (default) directory or specify a different drive and directory.

At the main menu, choose **File/Write** To. Then, in the New Name prompt box, enter the full path name of the new file name and press *Enter*:

```
C:\DIR\SUBDIR\FILENAME.EXT
```

where C: (optional) is the drive, \DIR\SUBDIR\ represents optional directories, and FILENAME.EXT is the name of the output file and its extension. (The extension .PAS is assumed; append a period (.) at the end of your file name if you don't want an extension name.)

If FILENAME.EXT already exists, the editor will verify that you want to overwrite the existing file before proceeding.

The Watch Window

The Watch window lists the Watch expressions from your program, and the current value of each expression. A Watch expression is reevaluated each time you Step or Run. The Watch window lets you track the value of important expressions at every step in your program.

As you add expressions to the Watch window, the window increases in size until it reaches the size specified by the TINST **Resize** Windows menu (see "The Resize Windows Menu" section on page 317). Once you've reached the maximum size, you can still add expressions, but you'll have to use the arrow keys to scroll through the window to see the rest of the expressions.

The current expression in the Watch window is marked by a highlight bar when the window is active; when the window is not active, the current expression has a bullet (•) in front of it.

The Output Window

The Output window contains the output generated by your program. At startup, it will display the last screen from DOS. You can pan through this window using the cursor keys, as well as the *Home, End, PgUp,* and *PgDn* keys. Use Zoom (*F5*) to enlarge this window to almost fill the screen.

The Output window has borders and is always shown in character mode. If your last screen was a graphics screen, or if you want to see all of the program's output screen with no borders, use the **User** Screen command (*Alt-F5*) to view the last User screen. This only works for text and CGA graphics.

The Integrated Debugger

The Turbo Pascal integrated environment includes a special built-in debugging feature called the integrated debugger to help you find errors ("bugs") in your programs. Since we're only going to cover the basic debugging menu here, take a look at Chapter 6, "Debugging Your Turbo Pascal Programs."

The debugger operates by allowing you to stop your program at any point as it is executing, so you can check or even alter the value of variables or other data items.

If **D**ebug/**I**ntegrated Debugging is set to On, then the IDE will invoke the debugger automatically when you run your program.

When you start a debugging session with **R**un/**R**un, Turbo Pascal compiles any out-of-date source files and prepares the program to run. Then it runs the program until it reaches either a breakpoint or the end of the program. Once the program has run, choose **R**un/**U**ser Screen (or press *Alt-F5*) to see your program's output.

To start a debugging session when no breakpoints have been set, choose **R**un/**S**tep Over (or press *F8*). The debugger will stop on the first executable statement.

Once Turbo Pascal has prepared the program to run, you are in a debugging session; now you can run your program using any of the following methods either singly or in any combination, and in any order:

- one line at a time, either skipping over procedure and function calls or stepping through them
- from your current position to a pre-established breakpoint, or until the end of the program
- from your current position to wherever you have positioned the cursor

It's generally unwise to continue running the program after you've modified any of the source files you are debugging. Instead, recompile your program by choosing **C**ompile/**M**ake (or pressing *F9*). In fact, if you have made changes to your source file while debugging, Turbo Pascal will ask if you want to rebuild your program each time you issue a run command like **S**tep Over or **T**race Into.

The Debugger Screen Display

In split-screen mode (*F5* toggles between split-screen and full-screen mode), the bottom window is the Watch window.

As Watch expressions are added to the Watch window, the window expands towards its maximum size (controlled by TINST's **Resize Windows** menu). Once the window is full-sized, new expressions are added at the top and older ones scroll off the bottom.

Your current position in the program is called the *execution position*. It is indicated onscreen by a highlight bar called the *execution bar*.

The File Menu

The File pull-down menu offers various choices for loading existing files, creating new files, and saving the file in the editor. When you load a file, it is read into the editor. When you finish with a file, you can save it to any directory or file name. In addition, from this pull-down you can change to another directory, temporarily go to the DOS shell, or exit Turbo Pascal.

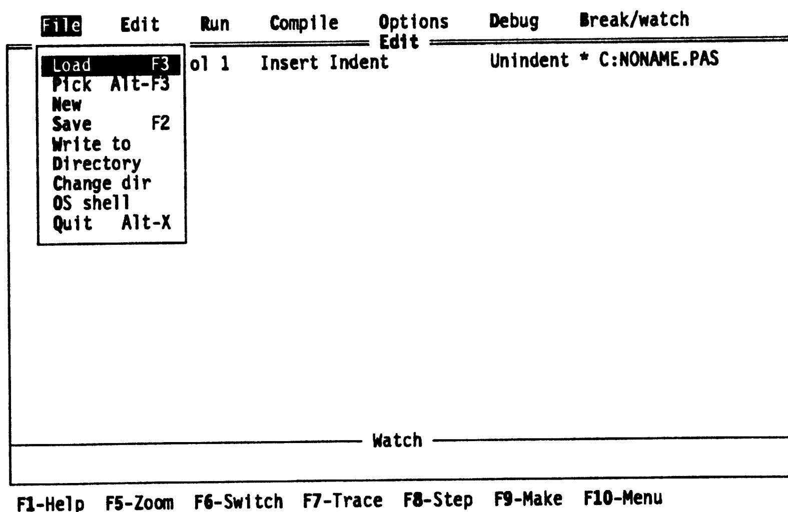


Figure 7.2: The File Menu

Load (F3)

Loads a file. You can use DOS-style masks to get a listing of file choices, or you can load a specific file. Simply type in the name of the file you want to load.

If you press *Enter* at the "Load File Name" prompt, or use a DOS-style mask to get a list of files, Turbo Pascal will display the directory box. You can move through the directory box by using the arrow keys or first-letter selection. Pressing the *B* key, for example, takes you to the first file name starting with *B*. Pressing *B* again takes you to the next file name, and so on. If there are no other file names beginning with the letter *B*, you will be taken back to the first one. If no file names start with the letter *B*, the cursor will not move. Holding down the *Shift* key and pressing *B* will take you to the first subdirectory that begins with the letter *B*.

Note: If you enter a nonexistent drive or directory, you'll get an error box onscreen. You'll get a verify box if you have an unsaved, modified file in the editor while you're trying to load another file. In either case, hot keys are disabled until you press the key specified in the error or verify box.

Pick (Alt-F3)

Lets you pick a file from a list of the previous eight files loaded into the Edit window. At the top of the list, you'll find the file currently in the editor. This provides an easy way to reload the current file if you wish to abandon changes. The file chosen is loaded into the Editor and the cursor is positioned where you last left the cursor. Note that the block markers and state are saved for each file, as are each of the four text markers and the find, replace, and search option strings. If you choose the "—load file—" item from the pick list, you'll get a Load File Name prompt box exactly as if you had chosen File/Load (or pressed *F3*). *Alt-F3* is the hot key for File/Pick. While in the editor, pressing *Alt-F6* will reload the last file you loaded.

You can define the pick file name with the **O/D/Pick** File Name menu command or from within Turbo Pascal's installation program (TINST—see page 307). This will have Turbo Pascal automatically save the current pick list when you exit Turbo Pascal and then reload that file upon reentering the program. For more information, see the **O/D/Pick** File Name command on page 185.

New

Places a new, empty file in the editor; by default, this file is called NONAME.PAS. (You can change this name later on when you save the file.)

Save (F2)

Saves the file in the editor to disk. If your file is named NONAME.PAS and you go to save it, the editor will ask if you want to rename it. From anywhere in the IDE, pressing *F2* will accomplish the same thing as **File/Save**.

Write To

Writes the file in the editor to a new name. If a file by that name already exists, you'll be asked to verify the overwrite.

Directory

Displays the directory and files you specify (to get a directory box displaying all files in the current directory, just press *Enter*). You can specify a mask using the DOS-style wildcards * and ?.

You can move through the directory box by using arrow keys or first-letter selection. Pressing the *B* key, for example, takes you to the first file name starting with *B*. Pressing *B* again takes you to the next file name, and so on. If there are no other file names beginning with the letter *B*, you will be taken back to the first one. If no file names start with the letter *B*, then the cursor will not move. Holding down the *Shift* key and pressing *B* will take you to the first subdirectory that begins with the letter *B*. When you are positioned on a file name, pressing *Enter* will load that file into the editor.

Change Dir

Displays the current directory and allows you to change to a specified drive and/or directory. Just type in the directory path, using any legal path name.

OS Shell

Leaves Turbo Pascal temporarily and takes you to the DOS prompt. To return to Turbo Pascal, type `EXIT`. This is useful when you want to run a DOS command without quitting Turbo Pascal. You can review the OS Shell screen by choosing the **Run/User Screen** command (*Alt-F5*). Normally, while running and debugging a program, there is not enough memory for this

operation. In this case, you can choose **Run/Program** Reset (hot key *Ctrl-F2*) to clear the running program's memory and then execute the OS Shell.

Note: In dual monitor mode, the OS shell will come up on the Turbo Pascal screen rather than the User screen. This allows you to shell to DOS without disturbing the output of your program. Since your program output is available on one monitor in the system, the **Run/User** Screen command (*Alt-F5*) is disabled.

Quit (Alt-X)

Quits Turbo Pascal and returns you to the DOS prompt in the currently active directory. If you've modified a work file without saving it, you'll see a prompt to do so now.

The Edit Command

The Edit command invokes the built-in screen editor.

You can invoke the main menu from the editor by pressing *F10*. Your source text remains displayed onscreen; you need only press *Esc* or *E* at the main menu to return to it (or *Alt-E* from anywhere).

The Run Menu

Use the items in the **Run** menu to **Run** a program, perform a **Program** Reset, Go to Cursor (execute to the current cursor location), Trace through your program (including into subroutine calls), Step Over any subroutine calls while tracing, or view the program output on the User Screen.

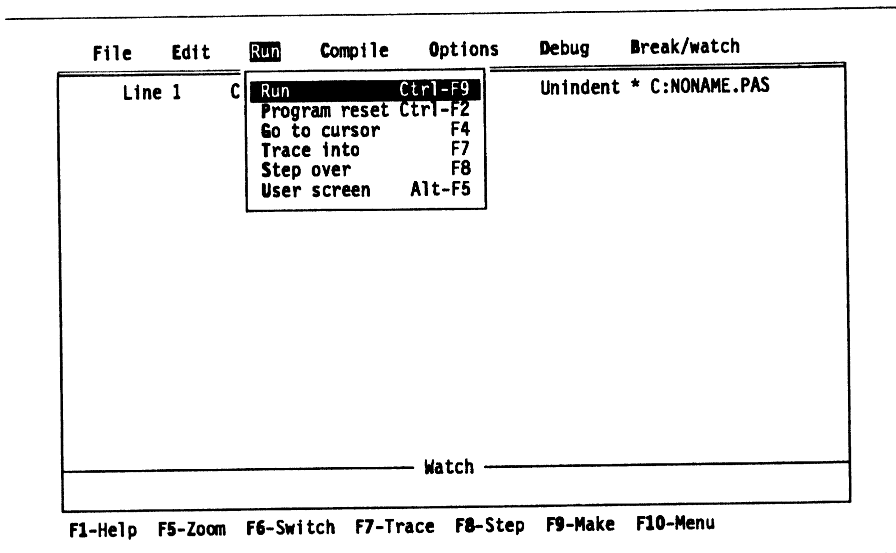


Figure 7.3: The Run Menu

Run (Ctrl-F9)

First, Run invokes the compiler (same as Compile/Make—*F9*) on the file you're currently editing. It then runs your program with the command-line arguments given in Options/Parameters. After your program's finished running, you'll be returned to your program. To see your program's output, choose Run/User Screen (or press *Alt-F5*).

Program Reset (Ctrl-F2)

Tells Turbo Pascal that you're finished with this particular debugging run and initializes the debugger in preparation for another run.

Cancels the current debugging session, takes you out of debugging mode and makes the current execution bar go away. It releases memory your program has allocated, and closes any open files. It does *not* reset any variable values.

Use Program Reset if you're finished debugging a program and want to free up memory so you can execute the File/OS Shell command.

Go to Cursor (F4)

Starts (or continues) execution of your program from the current execution position (the first line of your program if you haven't started debugging) to the line the Edit window cursor is on. This command presumes that you've gone into the Edit window and moved the cursor to some executable statement in your program. If the cursor is not at an executable location (for example, if it's at a blank line or comment line), then your program will run to the next executable line. However, if you position the cursor on a variable declaration or outside the scope of any procedure, then the debugger will tell you there is no code to run to on that line. Like Run/Trace Into, Run/Step Over, and Compile/Make (*F7*, *F8*, and *F9*), you can use Run/Go to Cursor (*F4*) to invoke a debugging session.

Use Go to Cursor to advance the execution bar to the part of your program you want to debug. If you want your program to stop at a certain statement every time it reaches that point, set a breakpoint with the Break/Watch menu.

Go to Cursor does not set a permanent breakpoint, but does allow the program to stop at a permanent breakpoint if it encounters one. If the program stops in this case, you must reposition the cursor on the line you want to run and reissue the Go to Cursor command.

Trace Into (F7)

Causes the next statement in your program to be executed. If it's a call to a subroutine, then tracing resumes with the first statement in that subroutine; Trace Into (*F7*) steps right across an Include or unit file boundary. If the source code exists for it, the Include file or unit will be loaded in.

Step Over (F8)

Works just like Trace Into, with one important exception: If the statement is a procedure or function call, the entire subroutine is executed in one step, and the debugger pauses at the statement following the subroutine call.

Step Over steps through your program as if there were no procedures or functions (it always advances to the next line of code). In contrast, Trace Into (*F7*) steps into all procedures and functions compiled with Options/Compiler/Debug Information set to On and whose source code can be found and loaded into the editor.

User Screen (Alt-F5)

Displays the program output (User) screen. This is the screen the IDE flips to when you are running and debugging your program. This screen is also used by the File/OS Shell command. At startup, the User screen contains the last screen from DOS.

Note: This command has no effect when you are in dual monitor mode.

The Compile Menu

Use the commands on the Compile menu to Compile, Make, or Build a program, to set the Destination of the object code (disk or memory), to find a run-time error (Find Error), to set a Primary File, or to Get Info about the current source file or the last compile.

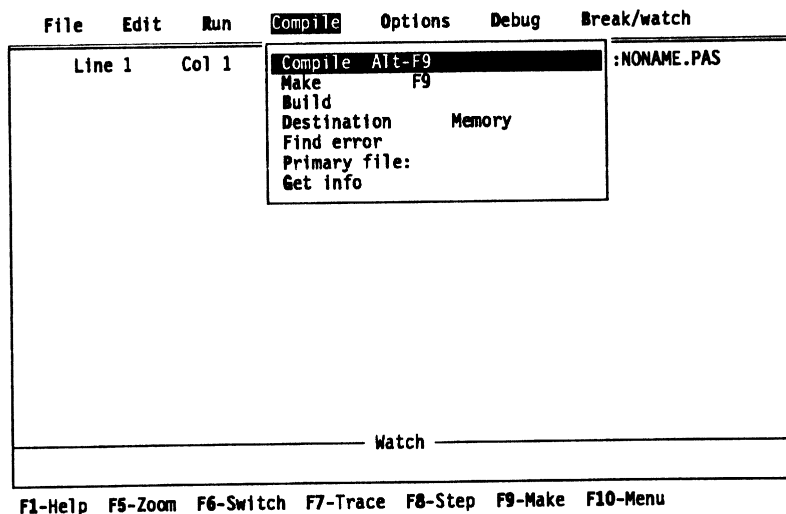


Figure 7.4: The Compile Menu

Compile (Alt-F9)

Compiles the last file you loaded into the editor. When Turbo Pascal is compiling, a window pops up to display compilation information: main file name, compiling file, lines compiled, available memory, and whether the

program compiled successfully. When the compilation is successful, press any key to remove this compiling window. If an error occurs, you are automatically placed in the Edit window at the error.

Make (F9)

Invokes Turbo Pascal's built-in Make sequence:

- If a primary file has been named, that file is compiled; otherwise, the last file loaded into the editor is compiled. Turbo Pascal checks all files upon which the file being compiled depends.
- If the source file for a given unit has been modified since the .TPU (object code) file was created, then that unit is recompiled.
- If the interface for a given unit has been changed, then all other units that depend upon it are recompiled.
- If a unit links in an .OBJ file (external routines), and the .OBJ file is newer than the unit's .TPU file, then the unit is recompiled.
- If a unit includes an Include file and the Include file is newer than that unit's .TPU file, then the unit is recompiled.

If the source to a unit (.TPU file) cannot be located, that unit is *not* compiled, but is used as is.

Build

Like Make, except Build recompiles all the files whether they are out of date or not. Build is unconditional; Make rebuilds only the .TPU files that aren't current.

If the source to a unit (.TPU file) cannot be located, that unit is *not* compiled, but is used as is.

Destination (Memory)

Use this option to specify whether the executable code will be stored on Disk (as an .EXE file) or whether it will just be stored in Memory (and thus lost when you exit from Turbo Pascal). Note that, even if Destination is set to Memory, any units recompiled during a Make or Build have their .TPU files updated on disk.

If Destination is set to Disk, then an .EXE file is created and its name is derived from one of two names, in the following order: the Primary File

name or, if none is specified, the name of the last file you loaded into the Edit window.

The resulting .EXE is stored in the same directory as the source file, or in the Options/Directories/EXE & TPU Directory, if one is specified.

Find Error

Finds the location of a run-time error. When a run-time error occurs, the address in memory of where it occurred is given in the format *seg:ofs*. When you return to the IDE, Turbo Pascal automatically locates the error for you. This command allows you to find the error again, given the *seg* and *ofs* values.

For Find Error to work, you must set the Debug/Integrated Debugging and Options/Compiler/Debug Information commands to On.

If run-time errors occur in a program running within the IDE, the default values for the error address are set automatically. This allows you to relocate the error after changing files. (Note that if you just move around in the same file, you can get back to the error location with the *Ctrl-Q W* command.)

When run-time errors occur under DOS, record the segment and offset displayed onscreen. Then load the main program into the editor or specify it as the Primary File. Be sure to set the Destination to Disk, then type in the segment offset value.

When you enter the error address, you must give it in hexadecimal segment and offset notation. The format is *"xxxx:yyyy"*; for example, "2BE0:FFD4."

Primary File

Specifies which .PAS file will be compiled when you use Compile/Make (*F9*) or Build (*Alt-C B*). Use this option when you are working on a program that uses several unit or Include files. No matter which file you've been editing, Compile/Make (or Build) will always operate on your primary file. If you've specified some other file as primary file but want to compile only the file in the editor, use Compile/Compile (*Alt-F9*).

Get Info

Brings up a window of information about the current .PAS file you're working with, the primary file name, the editor file name, the size (in bytes and lines) of the source code, the size (in bytes of code and data) of the resulting .EXE (or .TPU) file, stack size, available memory, minimum and maximum heap size, state of code, and error information.

The Options Menu

The Options menu contains settings that control how the integrated environment works. The settings affect things like compiler options, unit, object, and include directories, program run-time arguments, and so on. You can also save or load these option settings to and from disk.

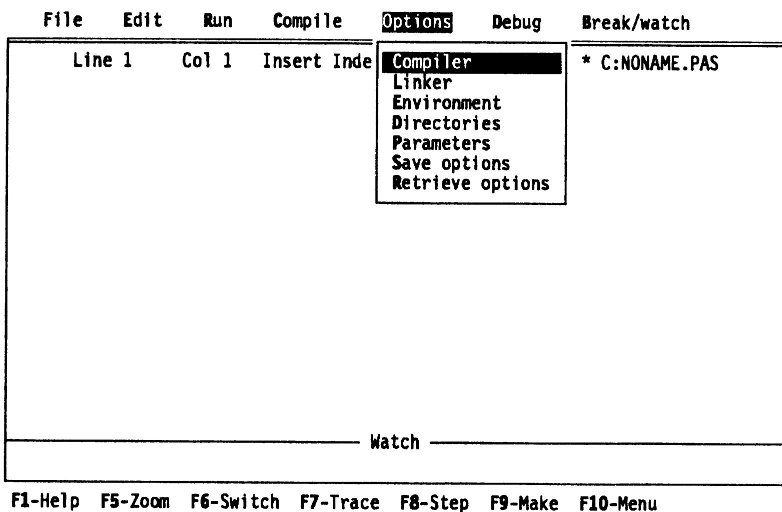


Figure 7.5: The Options Menu

Compiler

Specifies different default compiler options, including range-checking, stack-checking, I/O-checking, overlays allowed, and so on. These same options can also be specified directly in your source code using compiler directives (see Appendix B in the *Reference Guide*, "Compiler Directives").

Note that the first letter of each menu item corresponds to its equivalent compiler directive; for example, Range-Checking corresponds to $*R*. (The only exception is Conditional Defines, which is {$*DEFINE*xxx} for historical reasons.)

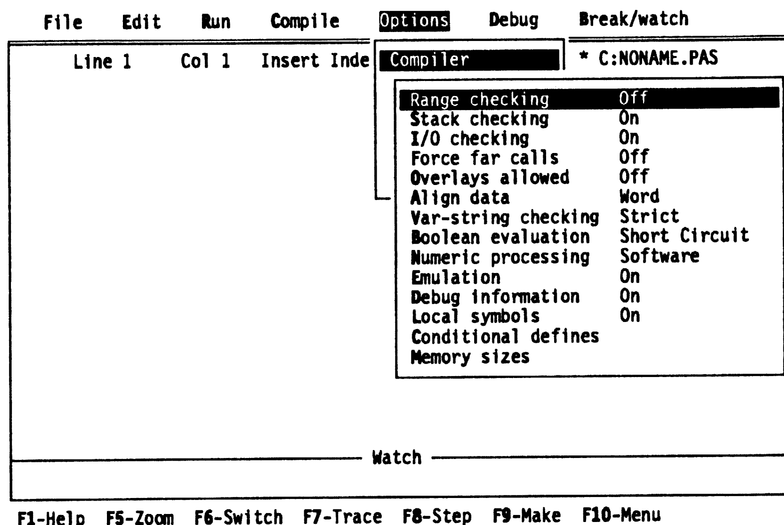


Figure 7.6: The Options/Compiler Menu

Range-Checking (Off)

Allows you to enable or disable range-checking. When this option is enabled, the compiler generates code to check that array and string subscripts are within bounds, and that assignments to scalar-type variables don't exceed their defined ranges. If the check fails, the program halts with a run-time error. When disabled, no such checking is done. This is equivalent to the $*R* compiler directive.

Stack-Checking (On)

Allows you to enable or disable stack-checking. When enabled, the compiler generates code to check that space is available for local variables on the stack before each call to a procedure or function. If the check fails, the program halts with a run-time error. When stack checking is disabled, no such checking is done. This is equivalent to the $*S* compiler directive.

I/O-Checking (On)

Allows you to enable or disable input/output (I/O) error-checking. When it is enabled, the compiler generates code to check for I/O errors after every I/O call. If the check fails, the program halts with a run-time error. When the option is disabled, no such checking is done; however, the user can test for I/O errors via the system function *IOResult*. I/O checking is equivalent to the *$I* compiler directive.

Force Far Calls (Off)

Allows you to force all procedures and functions to use the FAR call model. If the option is not enabled, the compiler will use the NEAR call models for any procedures or functions within the file being compiled. Force Far Calls is equivalent to the *$F* compiler directive.

Overlays Allowed (Off)

Enables or disables overlay code generation. Turbo Pascal allows a unit to be overlaid only if it was compiled with Overlays Allowed set to On (this is equivalent to the {*$O+*} compiler directive). In this state, the code generator takes special precautions passing string and set constant parameters from one overlaid procedure or function to another.

Setting Overlays Allowed to On does not force you to overlay that unit. It just instructs Turbo Pascal to ensure that the unit can be overlaid, if so desired. If you develop units that you plan to use in overlaid as well as non-overlaid applications, then compiling them with Overlays Allowed set to On ensures that you can indeed do both with just one version of the unit.

For further details on overlay code generation, refer to Chapter 13 of the *Reference Guide*, "Overlays."

Align Data (Word)

Switches between byte and word alignment of variables and typed constants. When you choose Word (the default), all variables and typed constants larger than 1 byte are aligned on a machine-word boundary (an even numbered address). If necessary, extra unused bytes are inserted between variables to achieve word alignment. When you choose Byte, no alignment measures are taken. Variables and typed constants are simply placed at the next available address, regardless of their size. This menu command is equivalent to the *$A* compiler directive.

Var-String Checking (Strict)

Allows you to choose between Strict or Relaxed string parameter error-checking. With Var-String Checking set to Strict, the compiler compares the declared type of a **var**-type string parameter with the type of the actual parameter being passed. If they are not identical, a compiler error occurs. With Var-String Checking set to Relaxed, no such type-checking is done. This option is equivalent to the $*V* compiler directive.

Boolean Evaluation (Short Circuit)

Allows you to choose Short Circuit or Complete Boolean expression evaluation. Set to Short Circuit, the compiler generates code to terminate evaluation of a Boolean expression as soon as possible; for example, in the expression `if False and MyFunc...`, the function *MyFunc* would never be called. If set to Complete, all terms in a Boolean expression are always evaluated. This option is equivalent to the $*B* compiler directive.

Numeric Processing (Software)

Allows for two options: 8087/80287 and Software. The 8087/80287 setting causes the compiler to generate direct 8087 inline code and allows the use of IEEE floating-point types (single, double, extended, comp), in addition to the standard Turbo Pascal 6-byte real data type. The Software setting allows only the 6-byte real data type.

This is equivalent to the $*N* compiler directive.

Emulation (On)

Enables or disables linking with a run-time library that will emulate the 8087 numeric coprocessor if the 8087 isn't present. When you compile a program with Emulation set to On, Turbo Pascal links with the full 8087 emulator. The resulting .EXE file can be used on any machine, regardless of whether an 8087 is present.

Used in a unit, the Emulation command has no effect; it applies only to the compilation of a program. Furthermore, if you compile a program with Emulation set to Off, and you compiled all the units used by the program with Emulation Off, then an 8087 run-time library isn't required, and the 8087 emulator is ignored.

This is equivalent to the $*E* compiler directive.

Debug Information (On)

Enables or disables the generation of debug information. This information consists of a line-number table for each procedure that maps object code addresses into source text numbers.

When you set Debug Information to On for a given program or unit, the IDE allows you to single-step and set breakpoints in that module. Also, when a run-time error occurs in a program or unit compiled with Debug Information set to On, Turbo Pascal can automatically take you to the statement that caused the error with Compile/Find Error.

Debug/Stand-alone Debugging and the Options/Linker/Map File commands produce complete information for a module only if you've compiled that module with Debug Information set to On.

For units, the debug information is recorded in the .TPU file, along with the unit's object code. Debug information increases the size of .TPU files, and takes up additional room when programs compile that use the unit, but it doesn't affect the size or speed of the executable program.

Those parts of your source code compiled and linked with Debug Information set to Off are not accessible to the debugger. If disk space is at a premium, set Debug Information to Off to create smaller .TPU files and use less memory during compilation and run time.

Debug Information is usually used in conjunction with the Options/Compiler/Local Symbols command, which enables and disables the generation of local symbol information for debugging. This is equivalent to the $*D* compiler directive.

Local Symbols (On)

Enables or disables the generation of local symbol information. Local symbol information consists of the names and types of all local variables and constants in a module; that is, the symbols in the module's implementation part, and the symbols within the module's procedures and functions.

When Local Symbols is set to On for a given program or unit, the IDE allows you to examine and modify the module's local variables. Also, calls to the module's procedures and functions can be examined with the Debug/Call Stack command.

The Options/Linker/Map File and Debug/Stand-alone Debugger commands will produce local symbol information for a module only if you've compiled the module with Local Symbols set to On.

For units, the local symbol information is recorded in the .TPU file along with the unit's object code. Local symbol information increases the size of .TPU files and takes up additional room when it is compiling programs that use the unit, but it doesn't affect the size or speed of the executable program.

You usually use **Local Symbols** in conjunction with **Debug Information**, which toggles the generation of line-number tables for debugging. Note that **Local Symbols** is ignored if **Debug Information** is set to Off. **Local Symbols** is equivalent to the *$L* compiler directive.

Conditional Defines

Defines symbols that you can reference in conditional compilation directives in your source code. Symbols are defined by typing in their name. Multiple symbols are separated by semicolons; for example, you may define the two symbols *Test* and *Debug* by entering `Test;Debug` in the Defined Symbols box.

Then, when the compiler runs across a sequence like

```
{$IFDEF Test}
Writeln("x =",x:1);
{$ENDIF}
```

the code for the *Writeln* will be generated. This is equivalent to defining symbols in the source using the {*$Define xxxx*} directive or on the command line with the */Dxxx* directive (using TPC.EXE).

Memory Sizes

Lets you configure the default memory requirements for a program. All three settings can be specified in your source code using the *$M* compiler directive. If you attempt to run your program and there is not enough heap space to satisfy the specified requirement, the program aborts with a runtime error.

Stack Size: Specifies the size (in bytes) of the stack segment. The default size is 16K, the maximum size is 64K.

Low Heap Limit: Specifies the minimum required heap size (in bytes). The default minimum size is 0K.

High Heap Limit: Specifies the maximum amount of memory (in bytes) to allocate to the heap. The default is 655360, which (on most systems) will allocate all available memory to the heap. This value must be greater than or equal to the smallest heap size.

Linker

Use the items on this menu for setting options for the built-in linker.

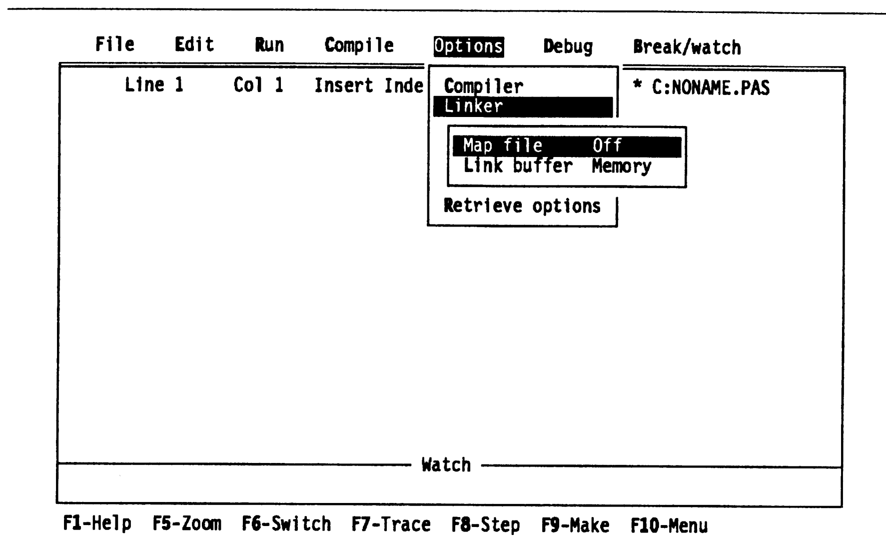


Figure 7.7: The Options/Linker Menu

Map File (Off)

Determines how much information goes into the map file to be produced. If you choose **Off**, no map file is created. When you choose a value other than **Off** from the Map File menu (Figure 7.8), the map file is placed in the same directory as the .EXE file with a .MAP file extension and contains:

- **Segments:** Only memory segment information (name, size, start and stop segments, and class).
- **Publics:** Segment information, all public or global symbol names and their addresses, and the program's entry point.
- **Detailed:** Segments, symbol and entry point information, and line-number and module tables.

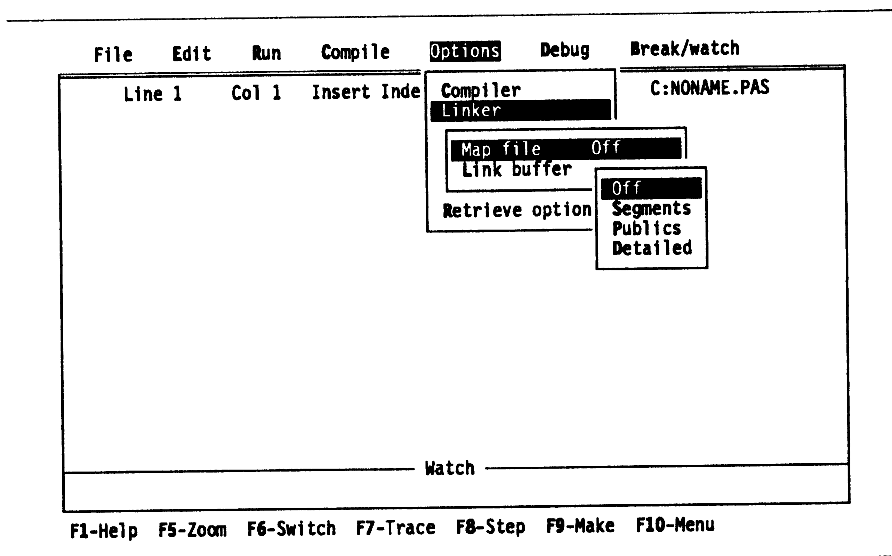


Figure 7.8: The Map File Menu

Link Buffer (Memory)

Tells Turbo Pascal to use Memory or Disk for the link buffer. When you set it to Memory, it speeds things up, but you may run out of memory for large programs. Setting Link Buffer to Disk frees up memory, but slows things down. This is equivalent to the */L* command-line option in TPC.EXE.

Environment

This menu's entries let you tailor the Turbo Pascal environment to suit your programming needs.

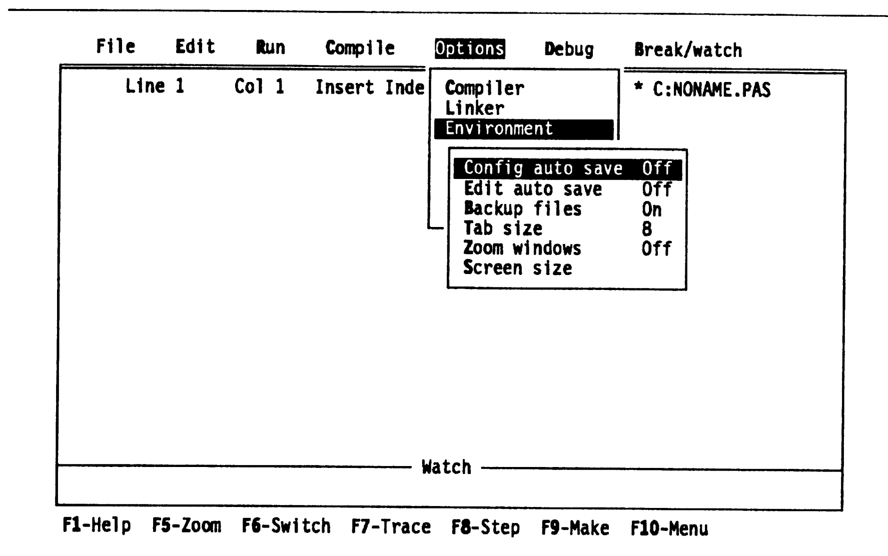


Figure 7.9: The Options/Environment Menu

Config Auto Save (Off)

A toggle to help prevent the loss of options you have changed, such as compiler settings or environment settings. Whenever you use File/OS Shell or issue a Run command like Step Over (*F8*) or Run (*Ctrl-F9*), the current configuration file is updated if it has been changed.

Edit Auto Save (Off)

A toggle to help prevent loss of your source file by automatically saving your edit file (if it's been modified) when you use Run commands like Step Over (*F8*) or Run (*Ctrl-F9*), or File/OS Shell.

Backup Files (On)

By default, Turbo Pascal automatically creates a backup of your source file when you do a Save. It saves the backup copy in a file with the same file name and a .BAK extension. This activity can be toggled Off and On with this option.

Tab Size (8)

Sets the hard tab size in the editor. The tab size can be set from 2 to 16; the default size is 8. Note that if Tab mode (*Ctrl-O T*) is set to Off, spaces are put into the text; if Tab mode is set to On, hard tabs are put into the text. (See page 160 for more information on Tab mode.)

Zoom Windows (Off)

A toggle that expands the Edit, Watch, and Output windows to full screen. You can still switch between them, but only one window at a time will be visible. Toggling this off returns you to the split-screen environment containing both the Edit and Watch (or Output) windows.

Screen Size (25-line)

Lets you choose between a 25-line standard display (25 Line Display) or a 43-line EGA display/50-line VGA display (43/50 Line Display). These options are only available on hardware that supports them.

Directories

Tells Turbo Pascal the configuration of your disk directories and chooses a pick file. (See Figure 7.10 on page 184.)

Turbo Directory

Turbo Pascal uses the Turbo Directory command to find TURBO.TPL, the configuration file (TURBO.TP), and the help file (TURBO.HLP). For Turbo Pascal to find any of these files at startup, if they are not in the current directory, you must install a path using TINST.EXE.

EXE & TPU Directory

.EXE and .TPU files are stored here. If the entry is blank, the files are stored in the directory where the source is found. .MAP files are also stored here if Options/Linker/Map File is set to anything besides Off.

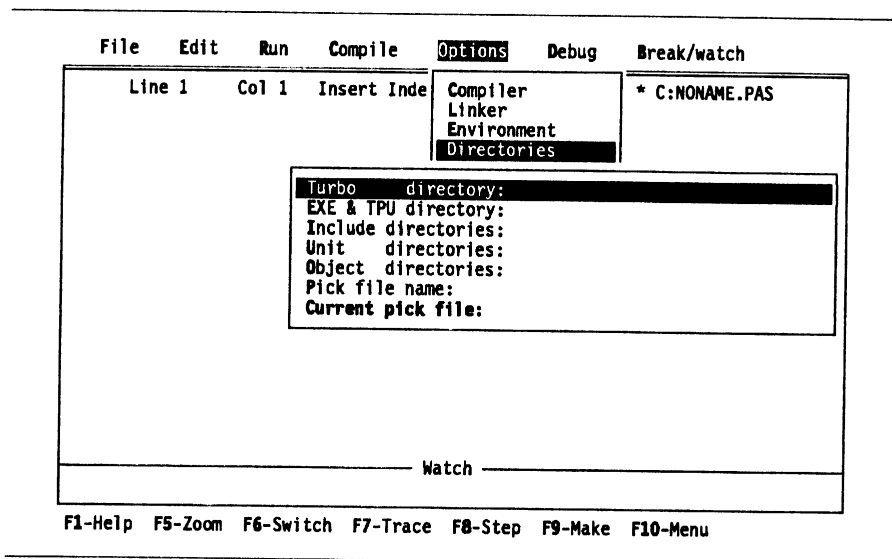


Figure 7.10: The Options/Directories Menu

Include Directories

Specifies the directories that contain your standard Include files. Include files are those specified with the *{$I filename}* compiler directive. Multiple directories are separated by semicolons (;), as in the DOS PATH command.

Unit Directories

Specifies the directories that contain your Turbo Pascal unit files. Multiple directories are separated by semicolons (;), as in the DOS PATH command.

To use the *Graph* unit, for example, you could create a directory \TURBO\GRAPH, copy the graphics files there (or at least GRAPH.TPU), and specify a Unit Directory of \TURBO\GRAPH. If you also wanted to keep other units in a \TURBO\UNITS directory, your Unit Directory would be \TURBO\UNITS;\TURBO\GRAPH.

Object Directories

Specifies the directories that contain .OBJ files (assembly language routines). When Turbo Pascal encounters a *{$L filename}* directive, it looks first in the current directory, then in the directories specified here. Multiple directories are separated by semicolons (;), as in the DOS PATH command.

Pick File Name

Defines the name and location of a pick file. When this field is defined, a pick file will always be written. If it is not defined, then a pick file is written only if the Current Pick File entry is nonblank. Since any name can be used, you must save the pick file name in your configuration file if it is not the default name of TURBO.PCK. For more information about this option, see the section entitled "About the Pick List and the Pick File" on page 191.

Current Pick File

Shows the file name and location of the current pick file, if any. This is where the current pick list information will be stored if the pick file name changes or if you exit the integrated environment. This item is always disabled and is for informational purposes only. For more information, see the section entitled "About the Pick List and the Pick File" on page 191.

Parameters

Specifies command-line parameters (or arguments) to the programs you run exactly as if you had typed them on the DOS command line (redirection is not supported). It is only necessary to give the arguments here (omit the program name).

Save Options

Saves all your chosen Compiler, Environment, and Directories options in a configuration file (the default file is TURBO.TP). At startup, Turbo Pascal looks in the current directory for TURBO.TP; if the file's not found, Turbo Pascal looks in the Turbo Directory for the same file. If the file isn't found there and you're running DOS 3.*x*, it will search the exec directory (the directory TURBO.EXE was started from).

Retrieve Options

Loads a configuration file (the default file is TURBO.TP) previously saved with the Save Options command.

The Debug Menu

Use the items in the Debug menu to Evaluate or modify a variable, to evaluate an expression, to show the current Call Stack, to find a given function or procedure in your program (Find Procedure), and to manage the video display swapping. Two switches in the Debug menu control whether you can debug programs in the IDE or by using Turbo Debugger (the stand-alone debugger).

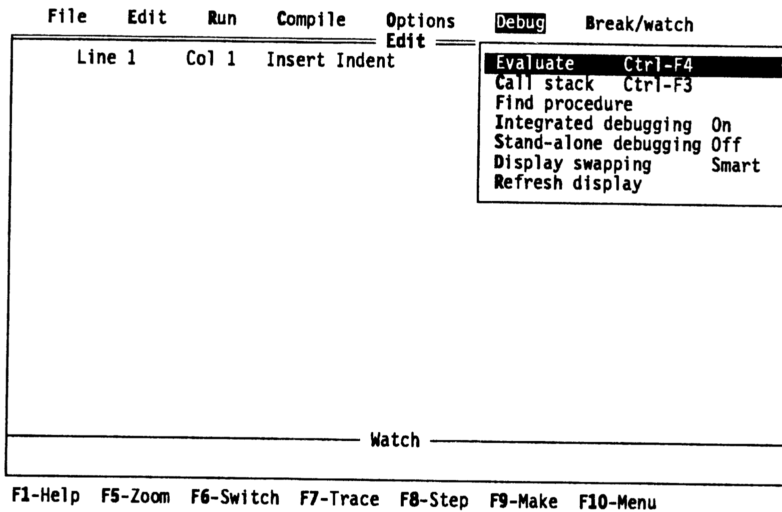


Figure 7.11: The Debug Menu

Evaluate (Ctrl-F4)

Brings up a window with three fields; the first lets you type in any variable name or expression, the second shows the current value of that variable or expression, and the third lets you type in a new value for the variable (see Figure 6.1 on page 135 of Chapter 6). *Ctrl-F4* is the shortcut.

The Evaluate box may be used whether or not you are debugging. If you are not debugging, no variables are available; however, you may use the Evaluate box like a calculator, entering constant expressions like *3*615*. When you are debugging, you may reference variables and expressions in your program. The default is taken from the current cursor position in the active window. Thus, you may position the cursor on the identifier you are

interested in before choosing Evaluate (*Ctrl-F4*); this saves you from having to type in the variable name.

Once you are in the Evaluate box, press the *Right arrow* key to extract more text from the cursor position. This is very useful when you are evaluating an expression involving more than one term.

You can modify the values of some expressions, and a number of format specifiers are supported. For an extensive look at this powerful tool, refer to "Format Specifiers" on page 127 of Chapter 6.

Call Stack (Ctrl-F3)

When debugging, pops up a window showing you the current call stack, which is the list of procedure and function calls that lead to your current location. Parameters for each call are also shown. You only have access to this function if you're debugging; otherwise, it is disabled. Refer to "The Call Stack" on page 137 of Chapter 6 for more information.

Find Procedure

Lets you type in the name of a procedure and function, then searches for it in your current program, loading Include files and units as needed. You only have access to this option once you've compiled your program; otherwise, the menu item is disabled. Refer to "Finding Procedures and Functions" on page 139 of Chapter 6 for more information.

Integrated Debugging (On)

When this toggle is set to On, debugging is enabled in the IDE. You can set breakpoints and step through code using the debug commands on the **Run** menu. Make sure the **Options/Compiler/Debug** Information command is set to *On* in order to step code (or use {$*D*+} in the source code).

When it is set to Off, debug tables are disposed before running the program; therefore, breakpoints and stepping code are *not* supported. Using this option when Compile/Destination is set to Disk makes more memory available to run your program inside the IDE.

Note: Turning Debug/Integrated Debugging Off also disables finding run-time errors in the integrated enivronment.

Stand-Alone Debugging (Off)

When this option is set to On and when Compile/Destination is set to Disk, debug information is appended to the .EXE file for use with the stand-alone Turbo Debugger.

When it is set to Off, or Compile/Destination is set to Memory, has no effect.

Display Swapping (Smart)

Display Swapping is a three-way toggle that determines when the screen is swapped from the IDE screen to the User screen during debugging (Smart, Always, or Never).

When you debug your program with Display Swapping set to Smart, the debugger looks at the code being executed to see if it affects the screen (that is, outputs to the screen). If it does (or if it calls a function), the screen is swapped from the IDE screen to the User screen long enough for output to take place, then is swapped back; otherwise, no swapping occurs.

Even if the current setting is Smart, the debugger swaps when it steps over a procedure or function call, even if the subroutine does no screen output.

In addition, in some situations the editor screen may be modified without being swapped; for example, if a timer interrupt routine writes to the screen.

Set to Always, Display Swapping causes the screen to be swapped every time a statement executes. Choose this setting any time the editor screen is likely to be overwritten by your running program.

Set to Never, Display Swapping tells the debugger not to do any screen swapping at all. It should be used for debugging sections of code that you are certain do not output to the screen. (See the explanation of Refresh Display, next.)

Note: If you are debugging in dual monitor mode (that is, you used the command-line */D* switch), you can see your program's output on one monitor and the Turbo Pascal screen on the other. Therefore, Turbo Pascal never swaps screens and the Debug/Display Swapping setting has no effect.

Refresh Display

You can use this option to restore the integrated development environment (IDE) screen. This is handy if your program has accidentally overwritten the IDE's screen and you need to restore it.

The Break/Watch Menu

When Debug/Integrated Debugging is set to On, you can use the commands on the Break/Watch menu to add and remove items from the Watch window, and to set and clear breakpoints. Specifically, you can add a watch, delete a watch, edit a watch, remove all watches, toggle a breakpoint, clear all breakpoints, or execute to the next breakpoint.

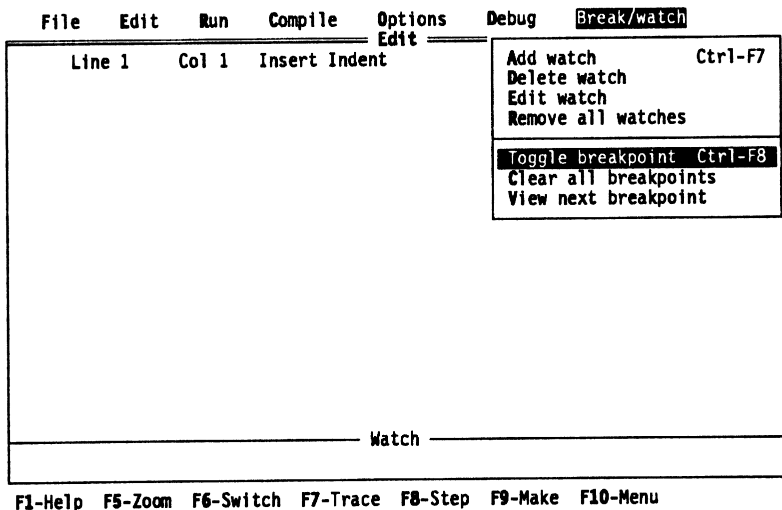


Figure 7.12: The Break/Watch Menu

Add Watch (Ctrl-F7)

Allows you to add an item—variable or expression—to the Watch window. Each new item added to the Watch window is inserted before the current Watch expression. As Watch expressions are added, the Watch window increases in size to the maximum set by the TINST Resize Windows

command. When the Watch window is at its maximum size, you can scroll through the Watch expression within the window.

Delete Watch

Deletes the current Watch expression *only* if the Watch window is visible. The current Watch expression is marked by a bullet (•) in the left margin of the Watch window, unless you are in the Watch window, in which case the current Watch expression is highlighted. Use *Del* or *Ctrl-Y* to delete Watches when you are in the Watch window.

Edit Watch

Edit Watch brings the current Watch expression into the Edit Watch box so you can modify it. If you want to completely replace the expression that's there, begin typing and the highlight bar will disappear. Press *Right arrow, Left arrow, Home,* or *End* to edit the expression. Pressing *Enter* replaces the original with the modified copy; *Esc* aborts the modification. If you want to get the original version back to edit again, press *Ctrl-R*.

Remove All Watches

Deletes all items in the Watch window, reducing it to its minimum size.

Toggle Breakpoint (Ctrl-F8)

Sets the current line as a breakpoint; if the current line is already a breakpoint, then it removes that breakpoint. Breakpoints in the editor are displayed with highlighted text.

Clear All Breakpoints

Clears all previously set breakpoints.

View Next Breakpoint

Moves the editing cursor to the next breakpoint. This doesn't execute any code; it just shows you where your breakpoints are. If there are no more breakpoints in the program, it takes you to the first one from the top.

Travels across file boundaries, loading a different file into the editor when necessary.

About the Pick List and Pick File

The pick list and pick file work together to save the state of your editing sessions. The pick list remembers recently loaded files while you are in the integrated environment, and the pick file remembers the same files after you have left the integrated environment or changed contexts within it.

The Pick List (Alt-F3)

The pick list is a pop-up menu located in the File menu. It provides a list of the eight most recent files that were loaded into the editor. Also, the first file in the list is the current file in the editor.

When you choose File/Pick, the highlight bar is placed on the second item in the menu (the last file loaded into the editor). By choosing this file or scrolling down and choosing one of the other files on the menu, you will load that file into the editor. At this point, the editor will position the cursor where you were last. In addition, any markers and marked blocks will be as you left them.

The pick list is a handy way to move back and forth from one file to another. The *Alt-F3* hot key takes you directly to the pick list, so pressing *Alt-F3* and *Enter* in succession swaps between two files. If the file you want is not on the pick list, you can choose the last entry on the pick list menu, which is "—load file—," or choose File/Load (*F3*) to load that file.

The Pick File

The pick file is used to store editor-related information, including the contents of the pick list. For each entry in the pick list, its file name, file position, marked block, and markers are stored. In addition to information about each file, the pick file contains data on the state of the editor when you last exited. This includes the last search-and-replace strings and search options.

To create a pick file, you must define a pick file name. This is done by entering a file name in the Pick File Name menu item found on the Options/Directories menu. When this field is defined, the pick list is updated on disk when you exit the integrated environment.

Loading a Pick File

If a pick file name is defined, the integrated environment will try to load it.

The pick file name can be defined in several ways. TINST—the Turbo Pascal installation program—can be used to install a permanent pick file name into the TURBO.EXE file. A configuration file can be loaded that contains a pick file name, or you can type in a pick file name. If a pick file name is defined but the integrated environment cannot find it, an error message is issued.

If no pick file name is defined, the integrated environment searches for the default pick file name, TURBO.PCK, first in the current directory, then in the Turbo directory, and, if you are running under DOS 3.x, it will then search the directory Turbo Pascal was loaded from.

Once a pick file is loaded, the integrated environment remembers the name and location of that file so that it can update that file when you exit.

Saving Pick Files

If a pick file has not been loaded and the **Pick File Name** option is blank, then the integrated environment will not save a pick file to disk when you exit.

Usually, pick files are saved only on exit from the integrated environment. However, there are certain times when the current pick file is updated and a new pick file is started (or restarted).

Whenever you change the pick file name, the integrated environment will cause the current pick list to be written to the last pick file and then the newly named pick file will be in effect.

Configuration Files and the Pick File

Since the pick file name is stored in the configuration file, it is possible to change pick files by loading a new configuration file. If the pick file name from the configuration file is different than the current pick file, then the current pick file is updated and the new pick file is loaded.

Note that two configuration files can easily use the same pick file; thus loading a configuration file with the same pick file name as the current one does not affect the pick file or the current pick list.

C H A P T E R

8

Command-Line Reference

Turbo Pascal 5.0 comes with a command-line version of the compiler so you can use compiler programs without entering the integrated environment (TURBO.EXE). This version of the compiler—identical to the one in TURBO.EXE—is called TPC.EXE and is found on your distribution disk.

Using the Compiler

You run TPC.EXE from the DOS prompt using a command line with the following syntax:

```
TPC [options] FILENAME [options]
```

where *filename* is the name of the source file to compile, and *options* are zero or more optional parameters that provide additional information to the compiler. If you omit both *filename* and *options,* TPC outputs a help screen that summarizes its syntax and command-line options.

If *filename* does not have an extension, TPC will assume .PAS. If you don't want the file you're compiling to have an extension, you must append a period (.) to the end of *filename.* If the source text contained in *filename* is a program, TPC creates an executable file named *filename*.EXE. If *filename* contains a unit, TPC creates a Turbo Pascal Unit file named *filename*.TPU.

You can specify a number of options for TPC. An option consists of a slash (/) immediately followed by an option letter. In some cases, the option letter is followed by additional information, such as a number, a symbol, or a directory name. Options can be given in any order and can come before and/or after the file name.

To find out about Turbo Pascal's memory-resident help utility for use with the command-line compiler (THELP), refer to Appendix C, "Turbo Pascal Utilities."

Compiler Options

The IDE (TURBO.EXE) allows you to set various options using the menus. TPC gives you access to most of these same options using the slash (/) command method described earlier. Alternately, you can precede options with a hyphen (-) instead of a slash (/). However, options that start with a hyphen must be separated from each other by blanks; those starting with a slash don't need to be separated, though it's legal to do so. So, for example, the following two command lines are equivalent and legal:

```
TPC -IC:\TP\INCLUDE -DDEBUG SORTNAME -$S- -$F+
TPC /IC:\TP\INCLUDE/DDEBUG SORTNAME /$S-/$F+
```

The first uses hyphens, so at least one blank separates options from each other; the second uses slashes, so no separation is needed.

Table 8.1 lists all the command-line options and gives their integrated environment equivalents. In some cases, a single command-line option corresponds to two or three menu commands.

Table 8.1: Command-Line Options

Command Line	Menu Command	Setting
/$A+	Options/Compiler/Align Data	Word
/$A-	O/C/Align Data	Byte
/$B+	O/C/Boolean Evaluation	Complete
/$B-	O/C/Boolean Evaluation	Short Circuit
/$D+	O/C/Debug Information	On
/$D-	O/C/Debug Information	Off
/$E+	O/C/Emulation	On
/$E-	O/C/Emulation	Off
/$F+	O/C/Force Far Calls	On
/$F-	O/C/Force Far Calls	Off
/$I+	O/C/I/O-Checking	On
/$I-	O/C/I/O-Checking	Off
/$L+	O/C/Local Symbols	On
/$L-	O/C/Local Symbols	Off
/$Msss,min,max	O/C/Memory Sizes	
/$N+	O/C/Numeric Processing	8087/80287
/$N-	O/C/Numeric Processing	Software
/$O+	O/C/Overlays Allowed	On
/$O-	O/C/Overlays Allowed	Off
/$R+	O/C/Range-Checking	On
/$R-	O/C/Range-Checking	Off
/$S+	O/C/Stack-Checking	On
/$S-	O/C/Stack-Checking	Off
/$V+	O/C/Var-String Checking	On
/$V-	O/C/Var-string Checking	Off
/B	Compile/Build	
/D*defines*	Options/Compiler/Conditional Defines	
/E*path*	Options/Directories/EXE & TPU Directory	
/F*seg:ofs*	Compile/Find Error	
/GS	Options/Linker/Map File	Segments
/GP	Options/Linker/Map File	Publics
/GD	Options/Linker/Map File	Detailed
/I*path*	Options/Directories/Include Directories	
/L	Options/Linker/Link Buffer	Disk
/M	Compile/Make	
/O*path*	Options/Directories/Object Directories	
/Q	(none)	
/T*path*	Options/Directories/Turbo Directory	
/U*path*	Options/Directories/Unit Directories	
/V	Debug/Stand-alone Debugging	On

Compiler Directive Options

Turbo Pascal supports several compiler directives, all of which are described in Appendix B of the *Reference Guide,* "Compiler Directives."

When embedded in the source code, these directives take one of the following forms:

```
{$directive+}
{$directive-}
{$directive info}
```

The /$ and /*D* command-line options allow you to change the default states of most compiler directives. Using /$ and /*D* on the command line is equivalent to inserting the corresponding compiler directive at the beginning of each source file compiled.

The Switch Directive (/$) Option

The /$ option allows you to change the default state of the following switch directives: *A, B, D, E, F, I, L, N, O, R, S,* and *V*. The syntax of a switch directive option is /$ followed by the directive letter, followed by a plus (+) or a minus (-). For example,

```
TPC MYSTUFF /$R-
```

would compile MYSTUFF.PAS with range-checking turned off, while

```
TPC MYSTUFF /$R+
```

would compile it with range-checking turned on. Note that if a {$*R*+} or {$*R*-} compiler directive appears in the source text, it overrides the /$*R* command-line option.

You can, of course, repeat the /$ option in order to specify multiple compiler directives:

```
TPC MYSTUFF /$R-/$I-/$V-/$F+
```

Remember, though, that if you use the hyphen instead of the slash, you must separate directives with at least one blank:

```
TPC MYSTUFF -$R- -$I- -$V- -$F+
```

Alternately, TPC allows you to write a list of directives (except for $*M*), separated by commas:

```
TPC MYSTUFF /$R-,I-,V-,F+
```

Note that only one dollar sign ($) is needed.

In addition to changing switch directives, /$ also allows you to specify a program's memory allocation parameters, using the following format:

```
/$MSTACK,HEAPMIN,HEAPMAX
```

where *stack* is the stack size, *heapmin* is the minimum heap size, and *heapmax* is the maximum heap size. All three values are in bytes, and each

is a decimal number unless it is preceded by a dollar sign ($), in which case it is assumed to be hexadecimal. So, for example, the following command lines are equivalent:

```
TPC MYSTUFF /$M16384,0,655360
TPC MYSTUFF /$M$4000,$0,$A0000
```

Note that, because of its format, you cannot use the *$M* option in a list of directives separated by commas.

The Conditional Defines (/D) Option

The */D* option lets you define conditional symbols, corresponding to the {*$DEFINE symbol*} compiler directive. The */D* option must be followed by one or more conditional symbols, separated by semicolons (;). For example, the following command line

```
TPC MYSTUFF /DIOCHECK;DEBUG;LIST
```

defines three conditional symbols, *iocheck, debug,* and *list,* for the compilation of MYSTUFF.PAS. This is equivalent to inserting

```
{$DEFINE IOCHECK}
{$DEFINE DEBUG}
{$DEFINE LIST}
```

at the beginning of MYSTUFF.PAS. If you specify multiple */D* directives, you can concatenate the symbol lists are concatenated. Thus

```
TPC MYSTUFF /DIOCHECK/DDEBUG/DLIST
```

is equivalent to the first example. The */D* option corresponds to the **O/C/** Conditional Defines command in TURBO.EXE.

Compiler Mode Options

A few options affect how the compiler itself functions. These are */M* (Make), */B* (Build), */F* (Find Error), */L* (Link Buffer) and */Q* (Quiet). As with the other options, you can use the hyphen format (remember to separate the options with at least one blank).

The Make (/M) Option

Just like the IDE, TPC has a built-in MAKE utility to aid in project maintenance. The */M* option instructs TPC to check all units upon which the file being compiled depends. If the source file for a given unit has been modified since the .TPU file was created, then that unit is recompiled.

Likewise, if a unit includes ($*I*) one or more Include files or links in ($*L*) one or more .OBJ files, and any of those files are newer than the unit's .TPU file, then that unit is recompiled. Finally, if the interface section of a given unit has been changed, then all other units that depend on it are recompiled. Units in TURBO.TPL are excluded from this process.

If you were applying this option to the previous example, the command would be

```
TPC MYSTUFF /M
```

This option is the same as the Compile/**M**ake command in the integrated development environment (IDE).

The Build All (/B) Option

What if you're unsure about what has or hasn't been updated? Instead of relying on the */M* option to determine what needs to be updated, you can tell TPC to update *all* units upon which your program depends. To do that, use the */B* option. Note that you can't use */M* and */B* at the same time.

If you were using this option in the previous example, the command would be

```
TPC MYSTUFF /B
```

This option is the same as the Compile/**B**uild command in the IDE.

The Find Error (/F) Option

When a program terminates due to a run-time error, it displays an error code and the address (*seg:ofs*) at which the error occurred. By specifying that address in a */Fseg:ofs* option, you can locate the statement in the source text that caused the error, provided your program and units were compiled with debug information enabled (via the $*D* compiler directive).

Suppose you have a file called TEST.PAS that contains the following program:

```
program Test;
var
  I : integer;
begin
  I := 0;
  I := I div I;                          { Force a divide by zero error }
end.
```

First, compile this program using the command-line compiler:

```
TPC TEST
```

If you do a DIR TEST.*, DOS lists two files:

```
TEST.PAS  -  your source code
TEST.EXE  -  executable file
```

Now, run TEST and get a run-time error:

```
C:\>TEST
Run-time error 200 at 0000:0011
```

Notice that you're given an error code (200) and the address (0000:0011 in hex) of the instruction pointer (CS:IP) where the error occurred. To figure out which line in your source caused the error, simply invoke the compiler, use the Find Error option, and specify the segment and offset as reported in the error message:

```
C:\>TPC TEST/F0:11
Turbo Pascal  Version 5.0  Copyright (c) 1983, 88 Borland International
TEST.PAS(7)
TEST.PAS(6): Target address found.
i:=i div i;
^
```

Note: In order for TPC to find the run-time error with /F, you must compile the program with the same command-line parameters you used the first time you compiled it.

The compiler now gives you the file name and line number, and points to the offending line number and text in your source code.

As mentioned previously, you *must* compile your program and units with debug information enabled for TPC to be able to find run-time errors. By default, all programs and units are compiled with debug information enabled, but if you turn it off, using a {$*D*-} compiler directive or a /$*D*- option, TPC will not be able to locate run-time errors.

The /*F* option corresponds to the **Compile/Find Error** command in the IDE.

The Link Buffer (/L) Option

The /*L* option disables buffering in memory when .TPU files are linked to create an .EXE file.

Turbo Pascal's built-in linker is two-pass. In the first pass through the .TPU files, the linker marks every procedure that gets called by other procedures. In the second pass, it generates an .EXE file by extracting the marked procedures from the .TPU files. By default, the .TPU files are kept in memory between the two passes; however, if the /*L* option is specified, they are reread during the second pass. The default method is faster but

requires more memory; for very large programs, you may have to specify /*L* to link successfully.

The /*L* option corresponds to the Disk setting of the **Options/Linker/Link Buffer** switch in the IDE.

The Quiet (/Q) Option

The quiet mode option suppresses the printing of file names and line numbers during compilation. When TPC is invoked with the quiet mode option

```
TPC MYSTUFF /Q
```

its output is limited to the sign-on message and the usual statistics at the end of compilation.

Directory Options

TPC supports several options that are equivalent to commands in the **Options/Directories** menu in the integrated environment. These options allow you to specify the five directory lists used by TPC: Turbo, EXE & TPU, Include, Unit, and Object.

The Turbo Directory (/T) Option

TPC looks for two files when it is executed: TPC.CFG, the configuration file, and TURBO.TPL, the resident library file. TPC automatically searches the current directory; if you're running under version 3.*x* (or later) of MS-DOS, then it also searches the directory containing TPC.EXE. The /*T* option lets you specify one other directory in which to search. For example, you could say

```
TPC /TC:\TP\BIN MYSTUFF
```

Note: If you want the /*T* option to affect the search for TPC.CFG, it must be the very first command-line argument, as in the previous example.

The /*T* option corresponds to the **Options/Directories/Turbo Directory** command in the IDE.

The EXE & TPU Directory (/E) Option

This option lets you tell TPC where to put the .EXE and .TPU files it creates. It takes a directory path as its argument:

```
TPC MYSTUFF /EC:\TP\BIN
```

If no such option is given, TPC creates the .EXE and .TPU files in the same directories as their corresponding source files. The /*E* option corresponds to the **O/D/EXE** & TPU Directory command in the IDE.

The Include Directories (/I) Option

Turbo Pascal supports Include files through the {$*I filename*} compiler directive. The /*I* option lets you specify a list of directories in which to search for Include files. Multiple directories are separated with semicolons (;). For example, the following command line causes TPC to search for Include files in C:\TP\INCLUDE and D:\INC *after* searching the current directory:

```
TPC MYSTUFF /IC:\TP\INCLUDE;D:\INC
```

If multiple /*I* directives are specified, the directory lists can be concatenated. Thus

```
TPC MYSTUFF /IC:\TP\INCLUDE/ID:\INC
```

is equivalent to the first example. The /*I* option corresponds to the **O/D/** Include Directories command in the IDE.

The Unit Directories (/U) Option

When you compile a program that uses units, TPC first attempts to find the units in TURBO.TPL (which is loaded along with TPC.EXE). If they cannot be found there, TPC searches for *unitname*.TPU in the current directory. The /*U* option lets you specify additional directories in which to search for units. As with the previous options, you can specify multiple directory paths as long as you separate them with semicolons (;). For example, the following command line causes TPC to look in C:\TP\UNITS and C:\LIBRARY for any units it doesn't find in TURBO.TPL or the current directory:

```
TPC MYSTUFF /UC:\TP\UNITS;C:\LIBRARY
```

As with the /*I* option, if multiple /*U* options are specified, the directory lists can be concatenated. The /*U* option corresponds to the **O/D/Unit** Directories command in the IDE.

The Object Directories (/O) Option

Using {$*L filename*} compiler directives, Turbo Pascal allows you to link in .OBJ files containing external assembly language routines, as explained in

Chapter 15, "Inside Turbo Pascal," in the *Reference Guide*. The /*O* option lets you specify a list of directories in which to search for such .OBJ files. Multiple directories are separated with semicolons (;). For example, the following command line causes TPC to search for .OBJ files in C:\TP\ASM and D:\OBJECT *after* searching the current directory:

```
TPC MYSTUFF /OC:\TP\ASM;D:\OBJECT
```

Like the /*I* option, if multiple /*O* options are specified, the directory lists can be concatenated. The /*O* option corresponds to the **O/D/O**bject Directories command in the IDE.

Debug Options

The IDE version of Turbo Pascal 5.0 features a built-in debugger to aid you in debugging your programs. Along the same lines, TPC has a number of command-line options that enable you to generate debugging information for stand-alone debuggers, including, of course, Borland's *Turbo Debugger*.

The Map File (/G) Option

The /*G* option instructs TPC to generate a .MAP file that shows the layout of the .EXE file. The /*G* option must be followed by a letter to indicate the desired level of information in the .MAP file: *S* (Segments only), *P* (Segments and Publics), or *D* (Detailed).

Unlike .EXE and .TPU files, which are in binary format, a .MAP file is a legible text file that can be output on a printer or loaded into TURBO's editor. A .MAP file is divided into three sections:

- Segment
- Publics
- Line Numbers

The /*GS* option outputs only the Segment section, /*GP* outputs the Segment and Publics section, and /*GD* outputs all three sections.

For modules (program and units) compiled in the {$*D*+,*L*+} state (the default), the Publics section shows all global variables, procedures, and functions, and the Line Numbers section shows line numbers for all procedures and functions in the module. In the {$*D*+,*L*-} state, only symbols defined in a unit's **interface** part are listed in the Publics section. **Note:** For modules compiled in the {$*D*-} state, there are no entries in the Line Numbers section.

The /*G* option corresponds to the **O/L/M**ap File command in the IDE.

The Stand-Alone Debugging (/V) Option

When the */V* option is specified on the command line, TPC appends *Turbo Debugger*-compatible debug information at the end of the .EXE file. Turbo Debugger (TD.EXE) is a powerful, stand-alone debugger that works on Turbo Pascal, Turbo C, and Turbo Assembler .EXE files. Turbo Debugger includes both source- and machine-level debugging, powerful breakpoints (including breakpoints with conditionals or expressions attached to them), and it lets you debug huge applications via virtual machine debugging on a 80386 or two-machine debugging (connected via the serial port).

Even though the debug information generated by */V* makes the resulting .EXE file larger, it does not affect the actual code in the .EXE file, and if it is executed from DOS, the .EXE file does not require additional memory.

The extent of debug information appended to the .EXE file depends on the setting of the *$D* and *$L* compiler directives in each of the modules (program and units) that make up the application. For modules compiled in the {*$D+,L+*} state, which is the default, *all* constant, variable, type, procedure, and function symbols become known to the debugger. In the {*$D+,L-*} state, only symbols defined in a unit's **interface** section become known to the debugger. In the {*$D-*} state, no line-number records are generated, so the debugger cannot display source lines when you debug the application.

The */V* option corresponds to the On setting of the Debug/Stand-alone Debugging command in the IDE.

The TPC.CFG File

You can set up a list of options in a configuration file called TPC.CFG, which will then be used in addition to the options entered on the command line. Each line in TPC.CFG corresponds to an extra command-line argument inserted before the actual command-line arguments. Thus, by creating a TPC.CFG file, you can change the default setting of any command-line option.

TPC allows you to enter the same command-line option several times, ignoring all but the last occurrence. This way, even though you've changed some settings with a TPC.CFG file, you can still override them on the command line.

When TPC starts, it looks for TPC.CFG in the current directory. If the file isn't found there, and if you are running DOS 3.*x*, TPC looks in the Turbo directory (where TPC.EXE resides). To force TPC to look in a specific list of

directories (in addition to the current directory), specify a */T* command-line option as the first option on the command line.

If TPC.CFG contains a line that does not start with a slash (/) or a hyphen (-), that line defines a default file name to compile. In that case, starting TPC with an empty command line (or with a command line consisting of command-line options only and no file name) will cause it to compile the default file name, instead of displaying a syntax summary.

Here's an example TPC.CFG file, defining some default directories for Include, object, and unit files, and changing the default states of the *$F* and *$S* compiler directives:

```
/IC:\TP\INC;C:\TP\SRC
/OC:\TP\ASM
/UC:\TP\UNIT
/$F+
/$S-
```

Now, if you type

```
TPC MYSTUFF
```

at the system prompt, TPC acts as if you had typed in the following:

```
TPC /IC:\TP\INC;C:\TP\SRC /OC:\TP\ASM /UC:\TP\UNIT /$F+ /$S- MYSTUFF
```

PART

2

Appendixes

A P P E N D I X

A

Differences Between Turbo Pascal 3.0, 4.0, and 5.0

This appendix lists the differences between Turbo Pascal versions 3.0, 4.0, and 5.0. Despite the many changes to the compiler and the introduction of many powerful features, version 5.0 is highly compatible with previous releases. As you will see by reading through this appendix, most of the differences are small. Where it's appropriate, we've made suggestions about how to make any necessary conversions, and we even provide a utility program to help you upgrade your 3.0 source code.

Note that the list of enhancements from 3.0 and 4.0 to 5.0 is extremely long and much of this manual is devoted to describing these features. The most important elements include

- integrated debugging in the IDE (see page 103)
- support for Borland's stand-alone Turbo Debugger (see page 150)
- overlays, including EMS support (see Chapter 13 in the *Reference Guide,* "Overlays")
- full 8087 emulation software (see page 177)
- smart linking which automatically strips unused code and data (see "Smart Linking" in Chapter 15 of the *Reference Guide*)
- procedural types, variables, and parameters (see Chapter 8 in the *Reference Guide,* "Procedures and Functions")

- constant expressions (see Chapter 1 in the *Reference Guide*, "Tokens and Constants")
- circular unit references (see "Implementation Section Uses Clause" on page 76)

This appendix is provided for the programmer who is porting 3.0 or 4.0 code to version 5.0. Therefore, it focuses on differences that affect backward compatibility.

How 4.0 and 5.0 Differ

There are very few language differences between version 4.0 and 5.0 that affect backward compatibility. The following (short) list details changes to 4.0 language features and compiler directives:

- A number of procedures and functions have been added to Turbo Pascal's run-time library, and two have been modified. Table A.1 lists them.

Table A.1: New and Modified Procedures and Functions

Name	Unit
DosVersion function	*Dos*
EnvCount function	*Dos*
EnvStr function	*Dos*
FExpand function	*Dos*
FillEllipse procedure	*Graph*
FSearch function	*Dos*
FSplit procedure	*Dos*
GetCBreak procedure	*Dos*
GetDefaultPalette function	*Graph*
GetDriverName function	*Graph*
GetEnv function	*Dos*
GetMaxMode function	*Graph*
GetModeName function	*Graph*
GetPaletteSize function	*Graph*
GetVerify procedure	*Dos*
InstallUserDriver function	*Graph*
InstallUserFont function	*Graph*
OvrClearBuf procedure	*Overlay*
OvrGetBuf function	*Overlay*
OvrInit procedure	*Overlay*
OvrInitEMS procedure	*Overlay*
OvrSetBuf procedure	*Overlay*
ParamStr function (modified)	*System*
RunError procedure	*Overlay*
Sector procedure	*Graph*
SetAspectRatio procedure	*Graph*
SetCBreak procedure	*Dos*
SetRGBPalette procedure	*Graph*
SetUserCharSize procedure (modified)	*Graph*
SetVerify procedure	*Dos*
SetWriteMode procedure	*Graph*
SwapVectors procedure	*Dos*

Refer to Chapter 16 of the *Reference Guide*, "Turbo Pascal Reference Lookup," for alphabetical entries on these functions and procedures.

- Turbo Pascal no longer reports the following compiler error messages, or has replaced them with new ones: 108, 109, 110, 111, 115, 119, 125. The following compiler error messages are new:
  - 133 Cannot evaluate this expression.
  - 134 Expression incorrectly terminated.
  - 135 Invalid format specifier.
  - 136 Invalid indirect reference.

- 137 Structured variables are not allowed here.
- 138 Cannot evaluate without *System* unit.
- 139 Cannot access this symbol.
- 140 Invalid floating-point operation.
- 141 Cannot compile overlays to memory.
- 142 Procedure or function variable expected.
- 143 Invalid procedure or function reference.
- 144 Cannot overlay this unit.
- 145 Too many nested scopes.

For detailed descriptions, refer to Appendix D in the *Reference Guide*, "Error Messages and Codes."

- The following run-time error messages are new:

  - 208 Overlay manager not installed.
  - 209 Overlay file read error.

  For detailed descriptions, refer to Appendix D in the *Reference Guide*, "Error Messages and Codes."

- Note that .TPU files from 4.0 are incompatible with .TPU files in 5.0. You'll need all the source code in order to rebuild your program and use it with 5.0. If you are using the integrated development environment, load the main program and choose the **C**ompile/**B**uild command. If you are using the command-line compiler, you must load your program with the Build option (*/B*—see page 198 for more information on the Build option). Just type

```
TPC /B YourProgramName
```

- There are new compiler directives in 5.0; they are described in Appendix B of the *Reference Guide*, "Compiler Directives." The following 4.0 directives are no longer supported:

4.0 Switch	Description
$*K*	Stack-checking
$*U*	Unit path (in source)
$*T*	Output .TPM file

  - Use the $*S* directive to control stack-checking. Use **O**ptions/**D**irectories/**U**nit Directories to specify a path for all the units used by your programs. The $*T* directive is obsolete because .TPM files are no longer emitted. (When **D**ebug/**S**tand-alone Debugging is set to On in the IDE, debug information is appended directly to the .EXE for use with the Turbo Debugger. You can generate .MAP files by setting the **O**ptions/**L**inker/**M**ap File command to On in the IDE.)

- The $L compiler directive, which controlled linking in 4.0, controls generation of local symbols in 5.0:

Switch	4.0 Function	5.0 Function
$L	Link in memory/disk	Local symbol information On/Off

- Use Options/Linker/Link Buffer to control linking (the default setting is Memory).

■ The command-line options /*R* and /*X*, used by TPC.EXE in version 4.0, are no longer supported in 5.0.

■ Version 5.0 introduces a word alignment option that switches between byte and word alignment of variables and typed constants. In the integrated environment, the menu command is Align Data on the Options/Compiler menu; the default setting is Word. The equivalent compiler switch directive is $*A*; the default is {$*A*+}.

■ Local variables in 5.0 procedures and functions are allocated on the stack in a different order than in 4.0. This will only affect routines that have inline statements that do not access the locals symbolically, but instead compute their BP offsets.

■ Function declarations in the **interface** and **implementation** sections of units must match exactly unless the **implementation** declaration declares only the identifier (no parameters or return value). Given the **interface** declaration

```
interface
  function MyFunc(X,Y : integer) : boolean;
```

version 4.0 allowed

```
implementation
  function MyFunc : boolean;
```

but this has been changed to be more consistent and fail-safe. Both of the following are allowed in 5.0's implementation:

```
implementation
  function MyFunc;
```

or

```
implementation
  function MyFunc(X,Y : integer) : boolean;
```

■ In version 4.0, when Compile/Destination was set to Disk and an Options/Directories/Executable Directory was specified, the .EXE file was output to the executable directory. In 5.0, Executable Directory is

renamed EXE & TPU Directory, and it also controls the output of .TPU files, whether the Destination is set to Disk or Memory.

- A new utility, TINSTXFR (TINST Transfer), lets you transfer your customized Turbo Pascal 4.0 TINST settings to 5.0 intact. Note that you don't have to use TINSTXFR; if you use the INSTALL program and choose the "Upgrade from 4.0 to 5.0" option, it will run TINSTXFR for you.

How 3.0 and 5.0 Differ

This section consists of two parts. The first half discusses programming changes from version 3.0 to 5.0, and briefly describes the 3.0 compatibility units (*Turbo3* and *Graph3*). Where appropriate, important enhancements made in version 5.0 have been noted. The second half introduces the UPGRADE program and describes how you can use it to port your 3.0 code to version 5.0.

Many of the changes are a result of 5.0's support for separate compilation using units. (If you aren't clear what units are, go back and read Chapter 4, "Units and Related Mysteries.") Any true conversion from 3.0 to 5.0 should include reorganization of your 3.0 code to take advantage of 5.0's units, which allows you to:

- Create tools that you can use in many different programs.
- Break up a large program into manageable modules by collecting related declarations, procedures and functions together. These modules can be separately compiled (once) and then quickly "included" in your program by Turbo Pascal's built-in linker.
- "Hide" declarations and routines that you don't need (or want) to be "visible" to the rest of the program.
- Break the 64K code barrier, since each unit can contain up to 64K of code.
- Build very large programs, since units can now be overlaid (see Chapter 13 of the *Reference Guide*, "Overlays," for more information).

Even with UPGRADE, *Turbo3*, and *Graph3* (discussed on page 227), you may still need to make changes to your 3.0 source code. The following section will look at what those changes are and how you should go about making them.

Programming Changes

Program Declarations

In version 3.0, the program name (the identifier given in the **program** statement) could be the same as another identifier in the program. In version 5.0, the program name must be unique—there cannot be a label, constant, data type, variable, procedure, function, or unit with the same name. That's because you can now refer to any identifier declared in your program as *progname.identifier*. This lets you resolve ambiguities in case you use a unit that also declares something named *identifier*. In that situation, you can refer to the unit's item as *unitname.identifier*.

In version 3.0, all Pascal items (constants, data types, variables, procedures, and functions) had to be compiled at the same time and were located either in your source file or in an Include file. In version 5.0, you can collect a group of constants, data types, variables, procedures and functions, compile them separately into a unit, and then use them in a series of programs.

In version 3.0, you could not have a program with more than 64K of code, and the compiler produced a .COM file. In version 5.0, your code size is limited only by the operating system (and your computer), since each unit itself can have up to 64K of code. The compiler produces an .EXE file.

In version 3.0, you used chaining and/or overlays to get around memory limitations. In 5.0, you can use overlays (based on units) to manage memory better. Overlay management is more sophisticated and more transparent. You can also use the *Exec* procedure (found in the *Dos* unit) to let one program execute others.

Compiler Directives

In version 3.0, you could embed a set of compiler directives in your code to set (or clear) certain options. In version 5.0, that set has been modified. It's a good idea to review *all* compiler directives. Of special note are *$B*, *$D*, and *$F*, since they are still valid but now have different meanings. Appendix B in the *Reference Guide*, "Compiler Directives," details all the directives.

Here is a list of the current compiler directives; see Appendix B in the *Reference Guide*, "Compiler Directives," for more details.

Directive	Description	Default
$A+/–	Align Data (+ = word, – = byte)	*$A+*
$B+/–	Boolean evaluation (+ = complete, – = short)	*$B-*
$D+/–	Debug information (+ = on, – = off)	*$D+*
$E+/–	Emulation of 8087 (+ = on, – = off)	*$E+*
$F+/–	Force FAR calls (+ = all FAR, – = as needed)	*$F-*
$I+/–	I/O error-checking (+ = on, – = off)	*$I+*
$I filename	Include file	
$L+/–	Local symbols (+ = on, – = off)	*$L+*
$L filename	Link object file	
$M s,l,h	Memory allocation (stack,minheap,maxheap) 16384,0,655360	
$N+/–	Numeric coprocessing (+ = 8087, – = software)	*$N-*
$O+/–	Overlays allowed (+ = on, – = off)	*$O-*
$O unitname	Overlay unit	
$R+/–	Range-checking (+ = on, – = off)	*$R-*
$S+/–	Stack-overflow checking (+ = on, – = off)	*$S+*
$V+/–	Var-string checking (+=on,-=off)	*$V+*

Note that:

- Range-checking is now off by default; if you want it on, place the {*$R+*} compiler directive at the start of your program. If you're unsure, omit it for now. If your program is halting with range-checking errors, and you don't want to address those problems immediately, you can always omit the {*$R+*} compiler directive for the time being.
- If an existing program doesn't work correctly, try setting Boolean evaluation to "complete" with the {*$B+*} compiler directive; {*$B-*} is the default setting. (The IDE equivalent of using the {*$B+*} compiler directive is setting the Options/Compiler/Boolean Evaluation menu command to Complete.)
- In version 3.0, the include option ({*$I filename*}) could be placed anywhere, and could simply contain executable statements. In version 5.0, the include option cannot be placed within a **begin/end** pair; if *filename* contains executable statements, they must be within a complete procedure or function. The main body of the program cannot appear in an Include file. It must reside in the main program file.
- In version 3.0, the include option ({*$I filename*}) did not require a space between *$I* and *filename*. In version 5.0, you must have a space after the *$I*.
- In version 3.0, you could not nest Include files; that is, if your program had the directive {*$I mystuff.pas*}, then MYSTUFF.PAS could not have any

$I (include) directives. In version 5.0, you can nest Include files and units up to 15 levels deep.

- You should review *all* use of error codes (for example, I/O error codes), especially when the check is more than simply zero or nonzero. Define all error codes as constants in a global location so you can deal more easily with future changes.

Input and Output

Turbo Pascal version 5.0 has made some significant changes in I/O handling, many of which are intended to increase ANSI compatibility.

If you are reading and writing real values with data files, be aware of the differences between the standard type real (6 bytes, compatible with version 3.0) and the IEEE floating-point types supported by the {$*N*+} or {$*E*+}.

The following discussion describes differences between 3.0's and 5.0's implementations of the *Read* and *Readln* standard procedures. Note that these comments apply only to *Read* and *Readln* from standard input (not from a disk file).

The *Read* and *Readln* procedures in version 3.0 handle input from the console in a non-standard way, whereas version 5.0 treats the console just like any other input device (such as a disk file).

In version 3.0, *Read* and *Readln* always wait for a line to be input. *Readln* echoes the *Enter* key at the end of the line, whereas *Read* does not (instead of moving to a new line, the cursor would stay at the end of the line). The input line is placed in an input buffer (without the CR/LF end-of-line marker), and the reading of variables then uses this buffer as the input source. If you specify more values on the input line than the number of variables in *Read* or *Readln*'s parameter list, version 3.0 ignores the remainder of the line. If you specify too few values on the input line, any excess character variables are set to *Ctrl-Z*, strings are empty, and numeric variables remain unchanged. The input buffer always clears upon a new *Read* or *Readln* request; no characters are carried over from a previous request.

In version 5.0, however, the *Read* and *Readln* procedures handle standard input just as if data were being read from a disk file. Every text file variable in version 5.0 has an input buffer, which by default holds up to 128 characters. Whenever *Read* or *Readln* needs to read a character from a disk file, and the input buffer is empty, a new block of up to 128 characters is read from the file and stored in the buffer. The characters are then picked

up one at a time from the buffer, until the buffer again becomes empty. For standard input, 5.0 behaves exactly the same, except that it reads a new line from the console whenever the input buffer becomes empty. The line is then stored in the input buffer, along with a CR/LF end-of-line marker.

As an example of the differences between 3.0 and 5.0, consider the following program:

```
program IntRead;
var
  I,J: integer;
begin
  Readln(I,J);
  Read(I); Readln(J);
  Read(Input,I); Read(Input,J); Readln(Input);
end.
```

In version 5.0, all three program lines behave identically, as the Pascal standards specify they should. The first line is simply a shortened form of the second, which is a shortened form of the third. In version 3.0, they all behave differently.

In version 5.0, when the *Readln* on the first line is executed, a line is input from the console and stored in the *Input* file's buffer. Assume for now that the following line is entered at that point:

```
10 20<Enter>
```

To *Read* an integer, version 5.0 skips leading spaces, CRs, and LFs, and then reads characters up until the next space, CR, or LF. These characters are then converted to an integer value. In this case, the string '10' is converted to an integer and stored in *I*; then the space is skipped, and the string '20' is converted to an integer and stored in *J*. Finally, the remainder of the line (which is empty) and the CR/LF sequence are skipped due to the *Readln*. Exactly the same would occur with the second and third program lines.

In version 3.0, each call to *Read* or *Readln* from standard input expects to input a line, so the second program line would require *Enter* to be pressed in between the values, and the third program line would require an additional *Enter* after the second value.

Getting back to version 5.0, the following input would also be acceptable:

```
10<Enter>
20<Enter>
```

The *Read(I)* causes the first line to be input, and processes the string '10'. The *Read(J)* then skips the CR/LF sequence that was left in the input buffer, inputs the second line, finds the string '20', and converts it to an integer. Finally, the *Readln* removes the last CR/LF sequence from the input buffer,

causing the next *Read* or *Readln* to input a new line. As you can see, *Read* will almost always leave further characters in the input buffer, possibly causing the next *Read* or *Readln* not to input a line. In particular, consider the following example:

```
program StrRead;
var
  S1,S2: string[79];
begin
  Read(S1);
  Read(S2);
end.
```

In version 3.0, each *Read* will input a string, without echoing the *Enter* key. Version 5.0 behaves quite differently. The first *Read* inputs a line, and stores it in the input buffer along with a CR/LF end- of-line marker. It then reads all characters up until the end-of-line marker into *S1*. The second *Read* sees the left over end-of-line marker, and cannot proceed further. As a result, *S1* is set to the entire input line, *S2* becomes empty, and a CR/LF sequence is left in the input buffer for the next *Read* or *Readln* to process.

With version 5.0, to avoid left over characters in the input buffer, always use *Readln* to input strings and numeric values. For example, the *StrRead* program above should be coded as follows:

```
program StrRead;
var
  S1,S2: string[79];
begin
  Readln(S1);
  Readln(S2);
end.
```

Data Types

Convert to new data 5.0 types where appropriate and practical. These include the scalar types shortint, longint, and word (to augment integer); pointer as a generic pointer type; and string, with an assumed maximum length of 255 characters.

The IEEE floating-point types single, double, extended, and comp are also supported (refer to the discussion of the *$N* and *$E* compiler directives in Chapter 14 of the *Reference Guide*, "Using the 8087," and in Appendix B of the *Reference Guide*, "Compiler Directives"). You may want to use these types to achieve more precision than the built-in type real provides; if your 3.0 program used 3.0's BCD data type or *Form* function, you might want to use an IEEE type (or longint) to simulate BCDs. (Also, see the sample

program, BCD.PAS, which came archived on your BGI/Demos/Doc/ Turbo3 distribution disk.)

Expression Evaluation Order

Even with Options/Compiler/Evaluation set to Complete, the 5.0 expression evaluation order may still be different than version 3.0:

```
{$B+}
function Func1 : boolean
begin
  Writeln('Func1');
  Func1 : = False
end;

function Func2 : boolean
begin
  Writeln('Func2');
  Func2 : = False;
end;

begin
  if Func1 and Func2 then;
end.
```

The 3.0 output is

```
Func1
Func2
```

but in 5.0 the output is

```
Func2
Func1
```

Predeclared Identifiers

In version 3.0, all predefined constants, data types, variables, procedures, and functions were always built into every program. In version 5.0, many of those predefined items are now located in one of the standard units (*Dos*, *Crt*, *Graph*, *Graph3*, *Overlay*, *Printer*, *System*, and *Turbo3*). In order to use those items, your program must have a **uses** statement listing the units to be used. For example,

```
uses Crt, Dos;
```

This table lists the 5.0 units and shows respective predeclared items in version 3.0:

5.0	**3.0**
Dos	*MsDos, Intr, Exec* (*Execute* in 3.0)
Crt	*KeyPressed, TextMode, Window, GotoXY, WhereX, WhereY, ClrScr, CrlEol, InsLine, DelLine, TextColor, TextBackground, LowVideo, NormVideo, Delay, Sound, NoSound,* and all the text mode and text color constants
Graph3	All the basic, advanced, and turtle graphics routines
Printer	*Lst*
Turbo3	*Kbd, CBreak, LongFileSize, LongFilePos, LongSeek*

Note: There are two additional standard units (*Overlay* and *Graph*) that contain routines not provided in 3.0.

In version 3.0, the following predefined items were available: *CrtExit, CrtInit, Aux, Con, Trm, Usr, ConInPtr, ConOutPtr, ConStPtr, LstOutPtr, UsrInPtr, UsrOutPtr, ErrorPtr.* In version 5.0, you can now write powerful I/O drivers instead.

In version 3.0, *CBreak* was an undocumented Boolean variable that let you enable or disable checking for program interruption via Ctrl-Break. In version 5.0, it is documented and has been renamed *CheckBreak; CBreak* is still available in the *Turbo3* unit.

In version 3.0, the *Execute* procedure was passed a file variable. In version 5.0, it has been renamed *Exec* (found in the *Dos* unit), and you pass it a program name and a command line (parameters).

In version 3.0, the predefined file variables *Aux, Con, Kbd, Lst, Trm,* and *Usr* were all available. In version 5.0, none of them are predefined; however, *Lst* is available by using the unit *Printer,* and *Kbd* is available in the unit *Turbo3*. More importantly, you can write your own device drivers.

In version 3.0, the functions *MemAvail* and *MaxAvail* were of type integer and returned the number of paragraphs (16-byte chunks) free. In version 5.0, those functions are of type longint and return the number of bytes free. Note that the original versions are available in the unit *Turbo3*.

In version 3.0, the *FileSize, FilePos,* and *FileSeek* functions returned a value of type integer. In version 5.0, they return a value of type longint (and these

can return values up to 2,147,483,647). In 5.0, *FileSize* of an untyped file will ignore a partial block when the block size is greater than 1.

In version 3.0, *MemW* returned an integer value. In version 5.0, it returns a value of type word.

In version 3.0, the *LongFile* functions (*LongFileSize*, *LongFilePos*, and *LongSeek*) returned a value of type real. In version 5.0, these functions are available only through the *Turbo3* unit, and they return a value of type real.

In version 3.0, the procedures *MsDos* and *Intr* both had an untyped parameter; you had to declare the appropriate register data structure and pass it in. In version 5.0, *MsDos* and *Intr* both require a parameter of type *Registers*, which is also defined in the *Dos* unit.

In version 3.0, the procedure *Intr* took a constant of type integer as its first parameter. In version 5.0, it accepts any expression (constant, variable, and so on), but the value must be of type byte.

In version 3.0, the function *IOResult* returned error codes specific to Turbo Pascal. In version 5.0, *IOResult* returns standard MS-DOS error codes. (*Turbo3* contains an *IOResult* function that maps 5.0 error codes to 3.0 values wherever possible.)

In version 3.0, the procedure *Seek* took a parameter of type integer for the record number. In version 5.0, that parameter is now of type longint.

In version 3.0, you could call *TextMode* without any parameters; this would restore the text mode to the last active mode before graphics. In version 5.0, *TextMode* must always have a parameter; however, since there is now the predefined variable of type word called *LastMode*, which holds the most recently selected video mode; therefore, 5.0's *TextMode(LastMode)* behaves exactly the same as 3.0's *TextMode*.

Other Additions and Improvements

In version 3.0, the *Addr* function returned the address of any variable. Even though *Addr* is supported in 5.0, you should now use the @ operator instead, so that *Ptr := Addr(Item)* becomes *Ptr := @Item*.

In version 3.0, assignment was allowed between types that were identical but defined separately:

```
var
  A: ^integer;
  B: ^integer;
begin
  ...
  A := B;
  ...
```

In version 5.0, stricter type-checking is enforced, and the preceding code would produce a compiler error. For variables to be assignment-compatible, they must either be declared together, like this:

```
var
  A,B: ^integer;
```

or they must be of the same defined data type:

```
type
  IntPtr  = ^integer;
var
  A: IntPtr;
  B: IntPtr;
```

In version 3.0, you could use a string variable of length 1 as a case selector in a **case** statement. In version 5.0, you can no longer do this, although you can use the individual characters.

In version 3.0, the type char was compatible with a string of length 1:

```
var
  Ch: char;
  S: string[10];
begin
  S  := 'a';
  Ch := S;
  ...
```

You could also use the function *Copy* in a similar fashion:

```
S  := 'abc';
Ch := Copy(S,2,1);
```

In version 5.0, neither is allowed. You can, however, still assign *Ch* to *S*, and you can always assign *S[1]* to *Ch*.

In version 3.0, you could call the procedure *Close* on a file that was already closed with no results. In version 5.0, this produces an I/O error, which you can detect by disabling I/O error-checking (via the {$*I*-} option) and testing the value returned by *IOResult*.

In version 3.0, you could use *CSeg* and *DSeg* in **absolute** statements:

```
var
  Parameters: string[127] absolute CSeg: $80;
```

In version 5.0, neither *CSeg* nor *DSeg* is allowed in **absolute** statements.

In version 3.0, there were no restrictions on where the control variable used in a **for** loop was declared. In version 5.0, the control variable must either be a global variable or, if the **for** loop is in a procedure or function, local to that procedure or function. The following code now results in a compiler error:

```
procedure Outer;
var
  I: integer;

procedure Inner;
begin
  for I := 1 to 10 do                          { I is declared in Outer }
    Writeln(I)
end;  { of proc Inner }

begin  { main body of Outer }
  Inner
end;  { of proc Outer }
```

In version 3.0, you could not assign –32768 directly to an integer variable; instead, you had to use the hex constant $8000. In version 5.0, you can now assign –32768 directly; the hex constant $8000 (which now equals +32768) is of type word and cannot be assigned to variables of type integer. You can also assign $FFFF8000 to a longint variable.

In version 3.0, you could declare labels in the **label** section without using the labels in your program. In version 5.0, if you declare a label and then don't use it, you'll get a compiler error. In that case, you need to either use the label in your code or remove it from the **label** declarations.

In version 3.0, you could (optionally) set the buffer size on a text file when you declared it:

```
var
  F: text[4096];                               { Buffer size of 4096 bytes }
```

In version 5.0, you now declare the text buffer as a data structure and assign it to the text file using the *SetTextBuf* procedure:

```
var
  F  : text;
  Buf: array[0..4095] of char;

begin
  Assign(F,'MyFile.TXT');
  SetTextBuf(F,Buf);
  Reset(F);
  ...
```

In version 3.0, you could use *Read(Kbd,Ch)* to do a direct, unechoed read from the keyboard. In version 5.0, the function *ReadKey* performs the same task and allows you to easily detect special keys (function keys, keypad keys, and so on):

```
Ch := ReadKey;
if Ch = #0 then                                    { Special key }
begin
  Ch := ReadKey;                                    { Read again }
  ...                                       { Handle special key }
end
else ...;                                   { Handle regular key }
```

(*Kbd* is still supported in the *Turbo3* unit; however, we strongly recommend that you switch to using *ReadKey*.)

In version 3.0, certain versions of the compiler supported the BCD (binary-coded decimal) data type. In version 5.0, you have no such data type. Consider using the longint (a 4-byte integer) type instead; or you can set the {$*N*+} compiler option and use the comp data type (an 8-byte integer). A sample program contained on the distribution disks demonstrates how to convert your BCD data for use with 5.0 data types (see the README file on the disk).

In version 3.0, if you had the following code:

```
var
  I: integer;

begin
  I := 30;
  Write('Enter I:  '); Readln(I);
  ...
```

and pressed *Enter* when asked to enter *I*, the program would continue and would leave *I* with a value of 30. In version 5.0, your program won't continue until you enter an integer value.

Refer to "Input and Output" on page 215 for information on changes in I/O handling in 5.0.

In version 3.0, typed constants resided in the code segment (CS). In version 5.0, they reside in the data segment.

In version 3.0, you had to use the *Move* procedure to copy data from one data structure to another if the structures were not assignment-compatible:

```
program Casting;
uses
  Crt;
type
  Buffer : array[0..3] of byte;
var
  BufPtr : ^Buffer;
  I      : integer;
  L      : longint;
begin
  ClrScr;
  New(BufPtr);
  for I := 0 to 3 do
  begin
    BufPtr^[I] := $FF;
    Writeln(BufPtr^[I];
  end;
  Writeln; Writeln;
  Move(BufPtr^,L,Sizeof(L)));
  Writeln(L);
end.
```

The exception was for ordinal data types (char, byte, integer, boolean, enumerated types), in which case you could use retyping (typecasting):

```
IntVar   := byte('a');
MonthVar := Month(3);
```

In version 5.0, typecasting has been extended to all types, with the requirement that the source and destination be exactly the same size (for example, in the following program, the array is 4 bytes long, as is the longint variable *L*):

```
program Casting;
uses
  Crt;
type
  Buffer : array[0..3] of byte;
var
  BufPtr : ^Buffer;
  I      : integer;
  L      : longint;
begin
  ClrScr;
```

```
    New(BufPtr);
    for I := 0 to 3 do
    begin
      BufPtr^[I] := $FF;
      Writeln(BufPtr^[I];
    end;
    Writeln; Writeln;
    L := longint(BufPtr^);
    Writeln(L);
  end.
```

In version 3.0, you were limited to the integer types byte (0..255, 1 byte) and integer (-32768..32767, 2 bytes). In version 5.0, you also have the types shortint (-128..127, 1 byte), word (0..65535, 2 bytes), and longint (-2147483648..2147483647, 4 bytes).

In version 3.0, you were limited to the floating-point type real. In version 5.0, you can set the {$*N*+} option and use three additional floating-point data types: single (4 bytes), double (8 bytes), and extended (10 bytes). You can also use the 8-byte integer type, comp. If you don't have an 8087, you can use these same types by turning 8087 software emulation on with the {$*N*+,*E*+} compiler directive.

In version 3.0, you had to give an explicit length to any string variable you declared; you also had to define your own type if you wanted to pass strings as parameters:

```
type
  BigStr = string[255];
var
  Name: string[20];
  S   : BigStr;
...
procedure Whatever(T : BigStr);
...
```

In version 5.0, you can now declare a variable to be of type **string**, which is equivalent to **string**[255]; you can declare formal parameters to be of type **string**:

```
var
  S: string;
...
procedure Whatever(T: string);
...
```

In version 3.0, all terms in a Boolean expression were evaluated, even if one term already ensured that the expression was True or False. Because of that, you couldn't write

```
if (B = 0) or (A/B = X) then ...
```

since the second term would be evaluated even if *B = 0*, which would produce a run-time error. In version 5.0, such expressions can be short-circuited. If *B = 0*, then the entire expression is True, and the expression *(A/B = X)* isn't evaluated. If you desire, you can force version 5.0 to evaluate all terms by using the {$*B*+} option.

In version 3.0, you used *ErrorPtr* to set up your own error handler. In version 5.0, *ErrorPtr* no longer exists; instead, you can handle both abnormal and normal termination of your program by installing an exit procedure (see *ExitProc* in Chapter 15 of the *Reference Guide*, "Inside Turbo Pascal.").

In version 3.0, you had to use assembly language coding to create an interrupt handler. In version 5.0, you can write interrupt handlers in Pascal by declaring procedures to be of type **interrupt**.

In version 3.0, you had the predefined identifiers *Mem* and *MemW* for direct memory access. In version 5.0, you also have *MemL*, which maps an array of type longint to memory.

In version 3.0, you could embed machine code within procedures and functions (or the main body of your program) using the **inline** statement. In version 5.0, you can also declare entire short procedures and functions to be of type **inline**; the machine code is then directly inserted (much like macro expansion) everywhere the procedure or function is called.

In version 3.0, external assembly language routines had to be in .BIN format and were declared in your program as offsets to the first routine in the file. In version 5.0, those routines can be in .OBJ format (like those produced by Turbo Assembler) and are simply declared external, with a {$*L files*} directive listing the .OBJ files to be linked in.

In version 3.0, all procedure and function calls were NEAR calls, which meant all code had to be in the same segment. In version 5.0, the compiler automatically generates NEAR or FAR calls as needed, and you can force all calls to be FAR using the {$*F*+} option.

Using Assembly Language

We still support inline in assembly language; it now includes the inline directive for procedure and function definitions, which defines an inline macro rather than a separate, callable routine. Refer to Chapter 15 of the *Reference Guide*, "Inside Turbo Pascal."

- For short assembly language code, consider using the inline directive (which differs from the **inline** statement). This generates actual inline macros in the resulting object code.
- Convert from inline to external subroutines where appropriate and practical; use inline only when necessary.
- The **inline** statement (within a subroutine) no longer allows references to the location counter (*), nor does it allow references to procedure and function identifiers. In order to refer to a procedure identifier, for example, declare a local pointer variable, assign it the address of the procedure (a procedure name), and refer to the pointer in the **inline** statement.
- External subroutines must be reassembled and incorporated in .OBJ format.
- Typed constants now reside in the data segment (DS) and so must be accessed differently by any external subroutines.
- **Inline/external** procedures and functions that use byte value parameters in version 3.0 often take advantage of the fact that the high byte of the word pushed on the stack is initialized to 0. This initialization is *not* done in version 5.0, so you'll need to make sure inline/external routines don't assume that the high byte is 0.

There are many changes to the conventions for passing parameters and function results on the stack. Refer to Chapter 15 of the *Reference Guide,* "Inside Turbo Pascal," for more information.

This list is not exhaustive. Many of your programs will run with little or no modification; others will work fine with the processing UPGRADE performs (see the section on UPGRADE at the end of this chapter). Likewise, this list doesn't cover all possible compatibility issues, since many Turbo Pascal programs take advantage of undocumented or unsupported features of version 3.0. Be sure to check the README file on your Turbo Pascal version 5.0 distribution disk for any additional conversion notes.

Converting from Turbo Pascal 3.0

Turbo Pascal 5.0 contains some exciting new features. This section discusses the tools we've provided to help you convert your 3.0 programs to 5.0. Note that in some cases, changes in your source code may be necessary (see the previous section).

We've provided a few upgrading tools: UPGRADE.EXE and two compatibility units, *Turbo3*, and *Graph3*. Note that UPGRADE should only be used

on 3.0 (or earlier) source code. Source code differences between 4.0 and 5.0 are minor and are easily fixed using the IDE.

- UPGRADE (UPGRADE.EXE) reads in a version 3.0 source code file and makes a series of changes to convert it for compilation under version 5.0. Some of these changes include commenting out obsolete buffer sizes, inserting appropriate **uses** statements, and optionally splitting large applications into separate units.
- The *Turbo3* unit restores some low-level I/O and system items found in version 3.0 but not in version 5.0. (Chapter 16 in the *Reference Guide*, "Turbo Pascal Reference Lookup," contains more details on all these items.)
- The *Graph3* unit supports the full set of graphics calls (basic, extended, turtlegraphics) from version 3.0. If you use *Graph3*, you'll have full access to all the constants, types, variables, procedures, and functions described in Chapter 7 of the *Reference Guide*, "Statements."

  Note: that a powerful new library of device-independent graphics routines is contained in the *Graph* standard unit. Unless you have programs that make extensive use of 3.0 graphics, you should use the new *Graph* unit instead.

In this section, we've also provided a checklist of conversion tasks that you may need to perform in addition to using these utilities. If you have a lot of code, don't worry—conversion usually goes very quickly, and the high speed of the version 5.0 compiler helps too.

Using UPGRADE

The UPGRADE program will aid in converting Turbo Pascal programs written for earlier versions of the compiler. UPGRADE scans the source code of an existing program, and performs the following actions:

- places warnings in the source where Turbo Pascal 5.0 differs in syntax or run-time behavior from version 3.0 of the compiler
- automatically fixes some constructions that have new syntactic requirements
- optionally writes a journal file that contains detailed warnings and advice for upgrading a program to 5.0
- automatically inserts a **uses** statement to pull in needed routines from the standard units
- optionally divides large programs into multiple units, to reorganize for overlaying or take advantage of separate compilation

In order to use UPGRADE, you need two files from your Turbo Pascal distribution disk. Copy the files UPGRADE.EXE and UPGRADE.DTA into your working drive and directory, or copy them into a subdirectory that is listed in the MS-DOS path.

UPGRADE is command-line driven; its format from the DOS prompt is

```
UPGRADE [options] filename
```

filename specifies the name of an existing Pascal 3.0 source file, which should be present in the current drive and directory. If no extension is specified, UPGRADE assumes .PAS as the file's extension.

If UPGRADE is executed with no command-line parameters (that is, with no options and no file name), it will write a brief help message and then halt.

The specified file must contain a complete Pascal 3.0 program, not just a fragment. If the file contains include directives, the specified Include files must also be present, either in the current directory or in another directory specified by the include directive.

The specified file must be a syntactically correct program, as determined by Turbo Pascal 3.0 or 2.0. UPGRADE does not perform a complete syntax check of source code—syntax errors in its input will cause unpredictable results. If you are uncertain whether a program contains syntax errors, compile it with Turbo Pascal 3.0 before proceeding with UPGRADE.

By default, UPGRADE will write a new version of the source code, overwriting the old version but saving it under a new name. Each old version saved will have the same name as the original, but with the extension .3TP attached. In the event that the extension .3TP would cause UPGRADE to overwrite an existing file, UPGRADE will try using the extensions .4TP, .5TP, and so on, until it finds a safe extension.

UPGRADE, by default, inserts comments into the source program; an example follows:

```
TextMode;
{! 20. ^ TextMode requires a parameter (Mode:word) in Turbo Pascal 5.0.}
```

In this example, `TextMode;` is a statement found in the program being upgraded. UPGRADE's comments always begin with `{!`, which makes it easy to find UPGRADE's warnings. UPGRADE numbers each comment with a sequential value, *20* in this example, which corresponds to the comments found in the optional journal file (described later). UPGRADE's comments contain a short statement describing the upgrade issue. UPGRADE inserts into the comment a caret (^) pointing to the exact location that triggered the warning in the preceding line of source code.

In a few cases, UPGRADE will make active changes to the source code; for example,

```
var
  f:text{[$1000]};
  {! 6. Use the new standard procedure SetTextBuf to set Text buffer size.}
```

This comment refers to the fact that Turbo Pascal 5.0 uses a different syntax to specify buffering of text files. Instead of the optional bracketed buffer size in the data declaration, Turbo Pascal 5.0 provides a new standard procedure, *SetTextBuf*, which can optionally be called to specify a buffer area and buffer size. Note that in this case UPGRADE automatically comments out the obsolete buffer size, and inserts a comment notifying you to call the *SetTextBuf* procedure at the appropriate location in your program.

UPGRADE accepts the following options on the command line:

/3	Use *Turbo3* compatibility unit when needed
/J	Write a detailed journal file
/N	No descriptive markup in source code
/O [*d:*][*path*]	Send output to *d:path*
/U	Unitize the program based on *.U* switches in source

A description of each option follows.

/3 Activate Turbo3 Unit

A special unit, *Turbo3*, is provided with the new compiler. This unit defines several variables and routines that cause new programs to mimic the behavior of Turbo Pascal 3.0 programs. The following identifiers defined within the *Turbo3* unit result in special handling by UPGRADE:

- *Kbd*
- *CBreak*
- *MemAvail*
- *MaxAvail*
- *LongFileSize*
- *LongFilePos*
- *LongSeek*

If your program uses any of these identifiers and you specify the /3 option, UPGRADE will insert the *Turbo3* unit name into the **uses** statement generated for the program.

Although the *Turbo3* unit and the /3 option can minimize the time required to convert an existing application, in the long run it may be better to make the (small) additional effort to use Turbo Pascal 5.0's new facilities. If you don't specify the /3 option, you will cause UPGRADE to generate warnings for each occurrence of the identifiers. With these warnings and the journal file (described next), you can achieve a complete upgrade in a short time.

/J Activate Journal File

When you specify the /J option, UPGRADE writes an additional file called the journal file. This file has the same name as your main program file but has the extension .JNL.

The journal file contains detailed descriptions of each warning UPGRADE produces, along with advice on how to go about upgrading your program. Here's an excerpt from a typical journal file:

```
4. MYPROG.PAS(6)
   s:byte absolute CSeg:$80;
                   ^
```

CSeg and *DSeg* can no longer be used in absolute statements.

Variables in Turbo Pascal 5.0 may be made absolute to other variables or typed constants (for example, *StrLen* : *byte* **absolute** *String1*), or to a fixed location in memory (for example, *KeyBoardFlag* : *byte* **absolute** $40:$17).

Given Turbo Pascal 5.0's separate compilation and smart linker, it is unlikely that variables absolute to *CSeg* or *DSeg* would have the intended effect.

Each journal entry begins with a numeric identifier, corresponding to the numbered comment inserted by UPGRADE into the actual source code. The journal file number is followed by the name of the original source file and the line number (within the original source file) of the statement that caused the warning. Note that the line number reported may be different than the line number in a marked-up or unitized source file. UPGRADE also inserts the actual source line and a pointer to the problem to make identification complete.

/N No Source Markup

Use this option if you don't want UPGRADE's comments inserted into your source code. UPGRADE will still perform any automatic fixes: the **uses** statement, Turbo Pascal 3.0 default compiler directives, mapping of compiler directives to Turbo Pascal 5.0 standards, and deactivation of *Overlay*, *OvrPath*, and text buffer sizes.

Generally, you should use the */N* option in combination with the */J* (journal file) or */U* (unitize) option.

/O [d:][path] Output Destination

Use this option to send UPGRADE's output to another drive or directory. When you activate this option, UPGRADE will not overwrite existing source files, nor will it rename them after processing. All UPGRADE output, including the journal file if any, will go to the drive and directory specified.

/U Unitize

The */U* option activates a second major function of UPGRADE. Using directives you place into your existing source code, UPGRADE will automatically split a large application into separate units.

You should use the */U* option only if your program is large enough to require overlays in Turbo Pascal 3.0, or if compilation times are long enough to be bothersome.

Before using the */U* option, you must make minor additions to your existing source program. These additions take the form of special comments that serve as directives to the UPGRADE utility. Each directive must have the following form:

```
{.U unitname}
```

unitname is a name that meets the following requirements:

- It is a legal Pascal identifier.
- It is a legal MS-DOS file name.
- It does not match the name of any existing identifier in the program being upgraded.

Additionally, legal unit name directives should begin with an alphabetic character and be limited to eight characters; here are some examples:

```
{.U UNIT1}
(*.U ScrnUnit *)
{.u   heapstuf}
```

Wherever UPGRADE encounters a unit name directive in your program's source code, it will route source code following that directive to the unit named. UPGRADE performs all necessary steps to prepare the unit source code for compilation, including

- inserting the **unit** and **uses** statements
- interfacing all global routines and data declarations
- implementing the source code
- generating an empty initialization block

Because the unitized program must fit the structure of Turbo Pascal 5.0's units, certain restrictions apply to the placement and use of unit name directives:

- Unit name directives can be placed only in the main file of a program, not within any Include file. This restriction avoids the need to split existing Include files into parts. In any case, Include files generally contain related routines that should reside within the same unit.
- Each unit name can be specified only once. This restriction prevents the generation of mutual dependencies between units, a feature of Turbo Pascal 5.0 that UPGRADE cannot automatically perform for you.
- A unit name directive must be placed outside of the scope of any procedure or function; that is, it must be placed at the global level of the program. This restriction enforces Turbo Pascal 5.0's definition of units as global entities.
- UPGRADE predefines one unit name, *Initial*. UPGRADE will automatically route to *Initial* any declarations or routines that precede the first unit name directive you place into your source code. UPGRADE defines the *Initial* unit so that later units will have access to any global identifiers defined prior to the first unit name. If you specify a unit name directive prior to any global declarations, *Initial* will be empty, and UPGRADE will delete it automatically.
- Each Turbo Pascal 5.0 unit is limited to at most 64K of code. You must place unit name directives so that this restriction is met.
- UPGRADE cannot deal effectively with global forward declarations. Instead, it places a warning in the source code whenever it encounters one. You must determine how to treat **forwards** and manually modify the source code after UPGRADE is finished. The best strategy is to minimize the use of **forwards** in the original program.

The */U* option automatically deletes any **overlay** keywords that appeared in the original source code.

When UPGRADE has unitized a program, the main module will be in the simplest possible form. It will contain the **program** statement, a **uses** statement that lists required system units and units you defined via unit name directives, and the original main block of code. All other procedures, functions, and data declarations will have been routed to other units.

UPGRADE "interfaces" user identifiers to the maximum extent possible. This means that all global procedures and functions will appear in the **interface** section of a unit, and that all global types, variables, and constants will appear in the **interface**. After your program is converted to the unit structure of Turbo Pascal 5.0, you may wish to hide selected global identifiers within the **implementation** sections of their units.

Although the */U* option of UPGRADE cannot deal with the more subtle issues of breaking a program into well-structured units, it does automate the otherwise time-consuming process of generating syntactically correct unit files.

What UPGRADE Can Detect

Here is a full list of the short warnings that UPGRADE generates:

- Use the new standard procedure *SetTextBuf* to set the text buffer size.
- New stack conventions require that many **inlines** be rewritten.
- Assure that *CSeg* refers to the intended segment.
- *CSeg* and *DSeg* can no longer be used in **absolute** statements.
- Restructure Chain and Execute programs to use units or *Exec*.
- Convert .BIN files to .OBJ files or convert them to typed constants (also, see BINOBJ.EXE).
- Use the new *ExitProc* facility to replace *ErrorPtr* references.
- Use new text file device drivers to replace I/O *Ptr* references.
- The Turbo Pascal 5.0 overlay system does not use the *Overlay* keyword.
- *OvrPath*, supported in 3.0, is replaced by *OvrInit* (see Chapter 16 of the *Reference Guide*, "Turbo Pascal Reference Lookup," for the *OvrInit* entry)
- The *Form* function (and BCD arithmetic) is not supported in Turbo Pascal 5.0.
- *BufLen* (for restricting *Readln*) is not supported in Turbo Pascal 5.0.
- The *TextMode* procedure requires a parameter (*Mode:word*) in Turbo Pascal 5.0.
- **interrupt**, **unit**, **interface**, **implementation**, and **uses** are now reserved words.
- *System*, *Dos*, and *Crt* are standard unit names in Turbo Pascal 5.0.
- Special file names INP:, OUT:, ERR: are not supported in Turbo Pascal 5.0.
- Assign unsigned values of $8000 or larger only to word or longint types.

- Use *Turbo3* unit in order for *MemAvail* and *MaxAvail* to return paragraphs.
- Use *Turbo3* unit to perform *LongFile* operations.
- *CBreak* has been renamed to *CheckBreak* in Turbo Pascal 5.0.
- *IOResult* now returns different values corresponding to DOS error codes.
- Use *Turbo3* unit for access to *Kbd*, or, better yet, use *Crt* and *ReadKey*.
- The $*I* Include file directive must now be followed by a space.
- Directives *A, B, C, D, F, G, P, U, W,* and *X* are obsolete or changed in meaning.
- The stack-checking directive *K* has been changed to *S* in Turbo Pascal 5.0.
- The effects of *HighVideo, LowVideo,* and *NormVideo* are different in Turbo Pascal 5.0.
- Special file name LST: is no longer supported; use *Printer Lst* file.
- Special file name KBD: is no longer supported; use *Turbo3 Kbd* file.
- Special file names CON:, TRM:, AUX:, USR: are not supported in Turbo Pascal 5.0.
- Special devices Con, Trm, Aux, and Usr are not supported in Turbo Pascal 5.0.
- An identifier duplicating a program/unit name is not allowed in Turbo Pascal 5.0.
- *Intr* and *MsDos* use the *Registers* type from the Turbo Pascal 5.0 *Dos* unit.
- **forwards** will require manual modification after unitizing.
- Include directives cannot be located within an executable block.
- The *CrtInit* and *CrtExit* procedures are not supported in Turbo Pascal 5.0.
- **for** loop counter variables must be local or global in Turbo Pascal 5.0.
- All defined labels within the current routine must be used.
- A parameter to *Intr* must be of the type *Registers* defined in the *Dos* unit.
- The * operator is not supported by Turbo Pascal 5.0 **inline** statements.
- Procedure and function names cannot be used in Turbo Pascal 5.0 inline statements.
- The state of directive *I, R, K,* or *V* differs from that at entry to include.
- The *System* unit now uses this name as a standard identifier.

What UPGRADE Cannot Detect

Here are descriptions of the various types of things that UPGRADE cannot detect in your source file:

- Mixing of **string** and char types in a way not allowed by Turbo Pascal 5.0; for example,

```
Ch:=Copy(S,1,1);
```

- Type mismatches due to Turbo Pascal 5.0's more stringent checking; for example,

```
var
  a : ^integer;
  b : ^integer;
begin
  a := b;                                    { Invalid assignment }
end.
```

- Unexpected run-time behavior due to side-effects of short-circuited Boolean expressions; for example,

```
{$B-}
if HaltOnError and (IoResult <> 0) then
  Halt;
```

  Turbo Pascal 3.0 calls the built-in *IOResult* function, and thus clears it to zero when the Boolean expression is evaluated. With short-circuiting activated in Turbo Pascal 5.0, the *IOResult* function will not be called if *HaltOnError* is False, and thus *IOResult* will potentially be left holding an error code from the previous I/O operation.

  Note that UPGRADE automatically inserts compiler directives that deactivate Boolean short-circuiting, thus avoiding problems like the one just described. Use caution before changing the Boolean evaluation directive.

An UPGRADE Checklist

Here is a summary of the basic steps for using UPGRADE:

1. Copy the files UPGRADE.EXE and UPGRADE.DTA from the compiler distribution disk to your current directory or to a directory in the DOS path.
2. If necessary, go to the directory where the 3.0 Pascal source files for the program you wish to upgrade are located.
3. Decide which UPGRADE options, if any, you wish to use.
4. If you decide to unitize the program, you must first edit the main source file to insert *{.U unitname}* directives, subject to the restrictions outlined previously.
5. From the DOS command line, enter the appropriate UPGRADE command, using the following syntax:

```
UPGRADE [options] filename
```

Examples of acceptable command lines follow:

```
upgrade MYPROG.PAS /J /3
UPGRADE bigprog /n /u /o c:\turbo5
```

6. UPGRADE will make two passes through the source code: one pass to detect areas of the program that may require modification, and a second pass to insert the appropriate **uses** statement, and optionally complete the process of unitization. At the end of the second pass, it will report the number of warnings it generated.
7. When UPGRADE is finished, change to the directory where output was sent (if it is other than the current directory). If you specified the */J* option, you may wish to browse through the journal file first to see the detailed explanations of UPGRADE's warnings. After doing so, use the Turbo Pascal editor to edit each source file that UPGRADE produced. Search for the string *{!*. Each match will display a warning produced by UPGRADE. In many cases, you will be able to change the source code immediately—when you do so, you may wish to delete UPGRADE's warning.
8. Once you have looked at all of UPGRADE's warnings and made appropriate changes to your source code, you are ready to compile with Turbo Pascal 5.0.

A P P E N D I X

B

Using the Editor

Turbo Pascal's built-in editor is specifically designed for creating program source text in the integrated environment. If you use the command-line version of the compiler, however, you'll be using another editor and can therefore skip this appendix.

The Turbo Pascal editor lets you enter up to 64K of text, 248 character lines, and any characters in the ASCII character set, extended character set, and control characters.

If you are familiar with WordStar, the version 3.0 Turbo Pascal editor, or the SideKick editor, you already know how to use the Turbo Pascal editor. At the end of this appendix, there's a summary of the few differences between Turbo Pascal's editor commands and the ever-familiar WordStar commands.

Quick In, Quick Out

To invoke the editor in the integrated environment, choose Edit from Turbo Pascal's main menu by pressing *E* from anywhere on the main menu or by using the arrow keys to move to the Edit command and then pressing *Enter*. The Edit window becomes the "active" window; meaning the Edit window's title is highlighted and has a double line at the top, and the cursor is positioned in the upper left-hand corner.

To enter text, just type as though you were using a typewriter. To end a line, press the *Enter* key.

To invoke the main menu from within the editor, press *F10*, *Ctrl-K D*, or *Ctrl-K Q*. The data in the Edit window remains on screen, but the menu bar now becomes active. To get back to editing, press *E* again.

The Edit Window Status Line

The status line at the top of the Edit window gives you information about the file you are editing: where in the file the cursor is located and which editing modes are activated:

```
Line n  Col n  Insert   Indent   Tab   C:FILENAME.TYP
```

`Line n` Cursor is on file line number *n*.

`Col n` Cursor is on file column number *n*.

`Insert` Tells you that the editor is in Insert mode; characters entered on the keyboard are inserted at the cursor position, and text to the right of the cursor is moved further right.

Use the *Ins* key or *Ctrl-V* to toggle the editor between Insert mode and Overwrite mode.

In Overwrite mode, text entered at the keyboard overwrites characters under the cursor instead of inserting them before existing text.

`Indent` Indicates the autoindent feature is on. You can toggle it off and on with the command *Ctrl-O I*.

`Tab` Indicates whether or not you can insert tabs; toggle it on or off with *Ctrl-O T*.

`C:FILENAME.EXT` Indicates the drive (C:), name (FILENAME), and extension (.EXT) of the file you are editing. If the file name and extension is NONAME.PAS, then you have not specified a file name yet. (NONAME.PAS is Turbo Pascal's default file name.)

Editor Commands

The editor uses approximately 50 commands to move the cursor around, page through text, find and replace strings, and so on. These commands can be grouped into four main categories:

- cursor movement commands (basic and extended)
- insert and delete commands
- block commands
- miscellaneous commands

Table B.1 summarizes the commands. Each entry in the table consists of a command definition, followed by the default keystrokes used to activate the command. The remainder of the appendix details each editor command.

Table B.1: Summary of Editor Commands

Basic Movement Commands	
Character left	*Ctrl-S* or *Left arrow*
Character right	*Ctrl-D* or *Right arrow*
Word left	*Ctrl-A* or *Ctrl-Left arrow*
Word right	*Ctrl-F* or *Ctrl-Right arrow*
Line up	*Ctrl-E* or *Up arrow*
Line down	*Ctrl-X* or *Down arrow*
Scroll up	*Ctrl-W*
Scroll down	*Ctrl-Z*
Page up	*Ctrl-R* or *PgUp*
Page down	*Ctrl-C* or *PgDn*
Extended Movement Commands	
Beginning of line	*Ctrl-Q S* or *Home*
End of line	*Ctrl-Q D* or *End*
Top of window	*Ctrl-Q E* or *Ctrl-Home*
Bottom of window	*Ctrl-Q X* or *Ctrl-End*
Beginning of file	*Ctrl-Q R* or *Ctrl-PgUp*
End of file	*Ctrl-Q C* or *Ctrl-PgDn*
Beginning of block	*Ctrl-Q B*
End of block	*Ctrl-Q K*
Last cursor position	*Ctrl-Q P*
Last error position	*Ctrl-Q W*
Insert and Delete Commands	
Insert compiler directives	*Ctrl-O O*
Insert line	*Ctrl-N*
Insert mode on/off	*Ctrl-V* or *Ins*
Delete block	*Ctrl-K Y*
Delete line	*Ctrl-Y*
Delete to end of line	*Ctrl-Q Y*
Delete character left of cursor	*Ctrl-H* or *Backspace*
Delete character under cursor	*Ctrl-G* or *Del*
Delete word right of cursor	*Ctrl-T*

Block Commands	
Mark block begin	*Ctrl-K B*
Mark block end	*Ctrl-K K*
Mark single word	*Ctrl-K T*
Print block	*Ctrl-K P*
Copy block	*Ctrl-K C*
Delete block	*Ctrl-K Y*
Hide/display block	*Ctrl-K H*
Move block	*Ctrl-K V*
Read block from disk	*Ctrl-K R*
Write block to disk	*Ctrl-K W*
Indent block	*Ctrl-K I*
Unindent block	*Ctrl-K U*
Miscellaneous Commands	
Abort operation	*Ctrl-U*
Autoindent mode on/off	*Ctrl-O I* or *Ctrl-Q I*
Control character prefix	*Ctrl-P*
Exit editor, no save	*Ctrl-K D* or *Ctrl-K Q*
Fill mode on/off	*Ctrl-O F*
Find	*Ctrl-Q F*
Find and replace	*Ctrl-Q A*
Find place marker	*Ctrl-Q n*
Go to last error position	*Ctrl-Q W*
Invoke main menu	*F10*
Language help	*Ctrl-F1*
Load file	*F3*
Pair matching forward	*Ctrl-Q [*
Pair matching backward	*Ctrl-Q]*
Repeat last find	*Ctrl-L*
Restore line	*Ctrl-Q L*
Save and remain in editor	*Ctrl-K S* or *F2*
Set place marker	*Ctrl-K n*
Tab	*Ctrl-I* or *Tab*
Tab mode on/off	*Ctrl-O T* or *Ctrl-Q T*
Unindent mode on/off	*Ctrl-O U*

Basic Movement Commands

The editor uses control-key commands to move the cursor up, down, right, and left on the screen (you can also use the arrow keys). To control cursor movement in the part of your file currently onscreen, use the sequences shown in Table B.2.

Table B.2: Control Cursor Sequences

When you press:	The cursor does this:
Ctrl-A or *Ctrl-Left arrow*	Moves to first letter in word to left of cursor
Ctrl-F or *Ctrl-Right arrow*	Moves to first letter in word to right of cursor
Ctrl-S	Moves to first position to left of cursor
Ctrl-D	Moves to first position to right of cursor
Ctrl-E or *Up arrow*	Moves up one line
Ctrl-X or *Down arrow*	Moves down one line
Ctrl-R	Moves up one full screen
Ctrl-C	Moves down one full screen
Ctrl-W	Scrolls screen down one line; cursor stays in line
Ctrl-Z	Scrolls screen up one line; cursor stays in line

Extended Movement Commands

The editor also provides six commands to move the cursor quickly to either ends of lines, to the beginning and end of the file, and to the last cursor position (see Table B.3).

Table B.3: Quick Movement Commands

When you press:	The cursor does this:
Ctrl-Q S or *Home*	Moves to column one of the current line
Ctrl-Q D or *End*	Moves to the end of the current line
Ctrl-Q E or *Ctrl-Home*	Moves to the top of the screen
Ctrl-Q X or *Ctrl-End*	Moves to the bottom of the screen
Ctrl-Q R or *Ctrl-PgUp*	Moves to the first character in the file
Ctrl-Q C or *Ctrl-PgDn*	Moves to the last character in the file

The *Ctrl-Q* prefix with a *B, K,* or *P* character allows you to jump to certain points in a document.

Beginning of block *Ctrl-Q B*
Moves the cursor to the block-begin marker set with *Ctrl-K B*. The command works even if the block is not displayed (see "Hide/display block" under "Block Commands") or if the block-end marker is not set.

End of block *Ctrl-Q K*
Moves the cursor to the block-end marker set with *Ctrl-K K*. The command works even if the block is not displayed (see "Hide/display block") or the block-begin marker is not set.

Last cursor position *Ctrl-Q P*
Moves to the last position of the cursor before the last command. This command is particularly useful after a Find or Find/replace operation has been executed and you'd like to return to the last position before its execution.

Last error position *Ctrl-Q W*
After the compiler has placed you in the editor with an error showing on the status line, you can later return to this position and redisplay the error by pressing *Ctrl-Q W*.

Insert and Delete Commands

To write a program, you need to know more than just how to move the cursor around. You also need to be able to insert and delete text. The following commands insert and delete characters, words, and lines.

Insert compiler directives *Ctrl-O O*
Inserts a string of compiler directive defaults (see Appendix B of the *Reference Guide*, "Compiler Directives," for more information) at the top of the current file, like this:

```
{$A, B-, D+, E+, F-, I+, L+, N-, O-, R-, S+, V+}
{$M 16384,0,655360}
```

Insert line *Ctrl-N*
Inserts a line break at the cursor position.

Insert mode on/off *Ctrl-V* or *Ins*
When entering text, you can choose between two basic entry modes: Insert and Overwrite. You can switch between these modes with the Insert mode toggle, *Ctrl-V* or *Ins*. The current mode is displayed in the status line at the top of the screen.

Insert mode is the Turbo Pascal editor's default; it lets you insert new characters into old text. Text to the right of the cursor moves further right as you enter new text.

Use Overwrite mode to replace old text with new; any characters entered replace existing characters under the cursor.

Delete block *Ctrl-K Y*
Deletes a previously marked block. There is no provision to restore a deleted block, so be careful with this command.

Delete line *Ctrl-Y*
Deletes the line containing the cursor and moves any lines below it one line up. There's no way to restore a deleted line, so use this command with care.

Delete to end of line *Ctrl-Q Y*
Deletes all text from the cursor position to the end of the line.

Delete character left of cursor *Ctrl-H* or *Backspace*
Moves one character to the left and deletes the character positioned there. Any characters to the right of the cursor move one position to the left. You can use this command to remove line breaks.

Delete character under cursor *Ctrl-G* or *Del*
Deletes the character under the cursor and moves any characters to the right of the cursor one position to the left. You can use this command to remove line breaks.

Delete word right of cursor *Ctrl-T*
Deletes the word to the right of the cursor. A word is defined as a sequence of characters delimited by one of the following characters:

space <> , ; . () [] ^ ' * + - / $

This command works across line breaks, and can be used to remove them.

Block Commands

The block commands also require a control-character command sequence. A block of text is any amount of text, from a single character to hundreds of lines, that has been surrounded with special block-marker characters. There can be only one block in a document at a time.

You mark a block by placing a block-begin marker before the first character and a block-end marker after the last character of the desired portion of text. Once marked, you can copy, move, or delete the block, or write it to a file.

Mark block begin *Ctrl-K B*
Marks the beginning of a block. The marker itself is not visible, and the block only becomes visible when the block-end marker is set. Marked text (a block) is displayed in a different intensity.

Mark block end *Ctrl-K K*
Marks the end of a block. The marker itself is invisible, and the block becomes visible only when the block-begin marker is also set.

Mark single word *Ctrl-K T*
Marks a single word as a block, replacing the block-begin/block-end sequence. If the cursor is placed within a word, then the word will be marked. If it is not within a word, then the word to the left of the cursor will be marked.

Print block *Ctrl-K P*
Prints the marked block. If no block is marked, Turbo Pascal prints the entire edit buffer.

Copy block *Ctrl-K C*
Copies a previously marked block to the current cursor position. The original block is unchanged, and the markers are placed around the new copy of the block. If no block is marked or the cursor is within the marked block, nothing happens.

Hide/display block *Ctrl-K H*
Causes the visual marking of a block to be alternately switched off and on. The block manipulation commands (copy, move, delete, and write to a file) work only when the block is displayed. Block-related cursor movements (jump to beginning/end of block) work whether the block is hidden or displayed.

Move block *Ctrl-K V*
Moves a previously marked block from its original position to the cursor position. The block disappears from its original position, and the markers remain around the block at its new position. If no block is marked, nothing happens.

Read block from disk *Ctrl-K R*
Reads a previously marked disk file into the current text at the cursor position, exactly as if it were a block. The text read is then marked as a block of different intensity.

When you issue this command, Turbo Pascal's editor prompts you for the name of the file to read. You can use DOS wildcards to select a file to read; a directory appears in a small window onscreen. The file specified can be any legal file name. If you don't specify a file type (.PAS, .TXT), the editor appends .PAS. To read a file without an extension, append a period to the file name.

Write block to disk *Ctrl-K W*
Writes a previously marked block to a file. The block is left unchanged in the current file, and the markers remain in place. If no block is marked, nothing happens.

When you issue this command, Turbo Pascal's editor prompts you for the name of the file to write to. To select a file to overwrite, use DOS wildcards; a directory appears in a small window onscreen. If the file specified already exists, the editor issues a warning and prompts for verification before overwriting the existing file. You can give the file any legal name (the default extension is .PAS). To write a file without an extension, append a period to the file name.

Indent block *Ctrl-K I*
Indents a selected block of text one column.

Unindent block *Ctrl-K U*
Unindents a selected block of text one column.

Miscellaneous Editing Commands

This section describes commands that do not fall into any of the categories already covered.

Abort operation *Ctrl-U*
Lets you abort any command in progress whenever it pauses for input, such as when Find/replace asks `Replace Y/N?` or when you are entering a search string or a file name (block read and write).

Autoindent mode on/off *Ctrl-O I* or *Ctrl-Q I*
Provides automatic indenting of successive lines. When autoindent is active, the cursor does not return to column one when you press *Enter;* instead, it returns to the starting column of the line you just terminated.

When you want to change the indentation, use the space bar and *Left* arrow key to select the new column. When autoindent is on, the message `Indent` shows up in the status line; when it is off, the message disappears. Autoindent is on by default. (When Tab is on, it works the same way, but it will use tabs if possible for indenting.)

Control character prefix *Ctrl-P*
Allows you to enter control characters into the file by prefixing the desired control character with a *Ctrl-P;* that is, first press *Ctrl-P,* then press the desired control character. Control characters will appear as low-intensity capital letters on the screen (or inverse, depending on your screen setup).

Exit editor, no save *Ctrl-K D* or *Ctrl-K Q*
Quits the editor and returns you to the main menu. You can save the edited file on disk either explicitly with the main menu's Save option under the File command or manually while you are still in the editor (*Ctrl-K S* or *F2*).

Fill mode on/off *Ctrl-O F*
Has no effect unless Tab mode is also on (use *Ctrl-O T*). When both these modes are enabled, the beginning of every autoindented and unindented line is filled optimally with tabs and spaces. This produces lines with a minimum number of characters.

Find *Ctrl-Q F*

Lets you search for a string of up to 30 characters. When you enter this command, the status line is cleared, and the editor prompts you for a search string. Enter the string you are looking for and then press *Enter*.

The search string can contain any characters, including control characters. You enter control characters in the search string with the *Ctrl-P* prefix. For example, enter a *Ctrl-T* by holding down the *Ctrl* key as you press *P*, and then press *T*. You can include a line break in a search string by specifying *Ctrl-M J* carriage return/line feed). Note that *Ctrl-A* has a special meaning: It matches any character and can be used as a wildcard in search strings.

You can edit search strings with the Character left, Character right, Word left, and Word right commands. Word right recalls the previous search string, which you can then edit. To abort (quit) the search operation, use the Abort command (*Ctrl-U*).

When you specify the search string, Turbo Pascal's editor asks which search options you'd like to use. The following options are available:

- **B** Searches backward from the current cursor position toward the beginning of the text.
- **L** Locally searches the marked block for the next occurrence of the string.
- **n** Where *n* equals a number, finds the *n*th occurrence of the search string, counted from the current cursor position.
- **N** Replaces without asking.
- **U** Ignores uppercase/lowercase distinctions.
- **W** Searches for whole words only, skipping matching patterns embedded in other words.

Examples:

- **W** Searches for whole words only. The search string *term* will match *term*, for example, but not *terminal*.
- **BU** Searches backward and ignores uppercase/lowercase differences. *Block* matches both *blockhead* and *BLOCKADE*, and so on.
- **125** Finds the 125th occurrence of the search string.

You end the list of find options (if any) by pressing *Enter*; the search starts. If the text contains a target matching the search string, the editor positions the cursor on the target. The search operation can be repeated with the Repeat last find command (*Ctrl-L*).

Find and replace *Ctrl-Q A*

This operation works identically to the Find command except that you can replace the "found" string with any other string of up to 30 characters. Note that *Ctrl-A* has a special meaning: It matches any character and can be used as a wildcard in search strings.

After you specify the search string, the editor asks you to enter a replacement string. Enter up to 30 characters; control-character entry and editing is performed as stated in the Find command. If you press *Enter*, the editor replaces the target with nothing, effectively deleting it.

Your options are the same as those in the Find command, with the addition of the following:

- **G** Globally searches the entire text, irrespective of the current cursor position, stopping at each occurrence of the string.
- **N** Replaces without asking; does not ask for confirmation of each occurrence of the search string.
- **n** Replaces the next *n* cases of the search string. When you're also using the *G* option, the search starts at the top of the file and ignores the *n*; otherwise it starts at the current cursor position.
- **L** Replaces only those strings local to a marked block.

Examples:

- **N10** Finds the next ten occurrences of the search string and replaces each without asking.
- **GW** Finds and replaces whole words in the entire text, ignoring uppercase/lowercase. It prompts for a replacement string.
- **GNU** Finds (throughout the file) uppercase and lowercase small, antelope-like creatures and replaces them without asking.

Again, you can end the option list (if any) by pressing *Enter*; the Find/replace operation starts. When the editor finds the item (and if the *N* option is not specified), it then positions the cursor at one end of the item, and asks `Replace (Y/N)?` in the prompt line at the top of the screen. You can abort the Find/replace operation at this point with the Abort command (*Ctrl-U*). You can repeat the Find/replace operation with the Repeat last find command (*Ctrl-L*).

Find place marker *Ctrl-Q n*

Finds up to four place markers (0-3) in text; *n* is a user-determined number from 0-3. Move the cursor to any previously set marker by pressing *Ctrl-Q* and the marker number, *n*.

Go to last error position *Ctrl-Q W*

Displays the last error message generated in the Edit window and places you in the editor at the point of error.

Invoke main menu *F10*

Press *F10* to call up the integrated environment main menu screen.

Language help *Ctrl-F1*

While in the editor and with the cursor positioned on a constant, variable, procedure, function, or unit, pressing *Ctrl-F1* will bring up help on the specified item.

Load file *F3*

Lets you edit an existing file or create a new file.

Pair matching *Ctrl-Q [* or *Ctrl-Q]*

Moves the cursor to a matching {, [, (*, ", ', <, >, *), }, or]. The cursor must be positioned on the character you want to match; in the case of (* or *), on the (or).

This command accounts for nested braces. If a match for the brace you are on cannot be found, then the cursor does not move. For (* *), { }, [ ], and < >, both *Ctrl-Q [* and *Ctrl-Q]* have the same effect. This is because the direction of the matching symbol can be determined. With " and ', the direction to search is determined by the key you select. Press *Ctrl-Q [* to find a match to the right (forward); press *Ctrl-Q]* to find a match to the left (backward).

Repeat last find *Ctrl-L*

Repeats the latest Find or Find/replace operation as if all information had been reentered.

Restore line *Ctrl-Q L*

Lets you undo changes made to a line, as long as you have not left the line. The line is restored to its original state regardless of any changes you have made.

Save and remain in editor *Ctrl-K S* or *F2*

Saves the file and remains in the editor.

Set place marker *Ctrl-K n*

You can mark up to four places in text; *n* is a user-determined number from 0-3. Press *Ctrl-K*, followed by a single digit *n* (0-3). After marking your location, you can work elsewhere in the file and then easily return to the marked location by using the *Ctrl-Q N* command.

Tab *Ctrl-I* or *Tab*

Tabs default to eight columns apart in the Turbo Pascal editor. You can change the tab size in the Options/Environment menu.

Tab mode on/off *Ctrl-O T* or *Ctrl-Q T*
With Tab mode on, a tab is placed in the text using a fixed tab stop of 8. Toggle it off, and it spaces to the beginning of the first letter of each word in the previous line.

Unindent mode on/off *Ctrl-O U*
Controls the effect of the *Backspace* key. If off, the *Backspace* key deletes the character to the left of the cursor. When it is On, pressing the *Backspace* key aligns the cursor with the first nonblank character in the first outdented line above the current or immediately preceding nonblank line.

The Turbo Pascal Editor versus WordStar

A few of the Turbo Pascal editor's commands are slightly different from WordStar. The Turbo Pascal editor contains only a subset of WordStar's commands, and several features not found in WordStar have been added to enhance program source-code editing. These differences are discussed here, in alphabetical order.

Autoindent
The Turbo Pascal editor's *Ctrl-O I* command toggles the autoindent feature on and off.

Cursor movement
Turbo Pascal's cursor movement controls—*Ctrl-S*, *Ctrl-D*, *Ctrl-E*, and *Ctrl-X*—move freely around on the screen without jumping to column one on empty lines. This does not mean that the screen is full of blanks; on the contrary, all trailing blanks are automatically removed. This way of moving the cursor is especially useful for program editing, for example, when matching indented statements.

Delete to left
The WordStar sequence *Ctrl-Q Del* (delete from cursor position to beginning of line) is not supported.

Del key
Using this key in the Turbo Pascal editor deletes the character at the cursor position rather than to the left of the cursor, as done in WordStar.

Mark word as block
Turbo Pascal allows you to mark a single word as a block using *Ctrl-K T*. This is more convenient than WordStar's two-step process of separately marking the beginning and the end of the word.

Movement across line breaks
Ctrl-S and *Ctrl-D* do not work across line breaks. To move from one line to another you must use *Ctrl-E, Ctrl-X, Ctrl-A,* or *Ctrl-F.*

Quit edit
Turbo Pascal's *Ctrl-K Q* does not resemble WordStar's *Ctrl-K Q* (quit edit) command. In Turbo Pascal, the changed text is not abandoned; it is left in memory, ready to be compiled and saved.

Undo
Turbo Pascal's *Ctrl-Q L* command restores a line to its pre-edit contents as long as the cursor has not left the line.

Updating disk file
Since editing in Turbo Pascal is done entirely in memory, the *Ctrl-K D* command does not change the file on disk as it does in WordStar. You must explicitly update the disk file with the Save option within the File menu or by using *Ctrl-K S* or *F2* within the editor.

A P P E N D I X

C

Turbo Pascal Utilities

This appendix describes the six stand-alone utility programs that come with Turbo Pascal: TPUMOVER, MAKE, THELP, TOUCH, GREP, and BINOBJ.

Using TPUMOVER, the Unit Mover

When you write units, you want to make them easily available to any programs that you develop. (Chapter 4, "Units and Related Mysteries," explains what a unit is and tells how to create your own units.) We'll now show you how to use TPUMOVER to *remove* seldom-used units from TURBO.TPL, and how to *insert* often-used units into TURBO.TPL.

A Review of Unit Files

There are two types of unit files: .TPU files and .TPL files. When you compile a unit, Turbo Pascal puts the resulting object code in a .TPU (Turbo Pascal Unit) file, which always contains exactly one unit.

A .TPL (Turbo Pascal Library) file, on the other hand, can contain multiple units. For example, all the units that come on your Turbo Pascal disk are in the file TURBO.TPL. The file TURBO.TPL is currently the only library file Turbo Pascal will load units from.

The naming distinction becomes important during compilation. If a particular unit used is not found in TURBO.TPL, then Turbo Pascal looks for the file *unitname*.TPU; if that file is not found, then compilation halts

with an error. If you are using the Build option (Compile/Build in the IDE or command-line option */B* with TPC), then Turbo Pascal first looks for *unitname*.PAS and recompiles it, using the resulting .TPU file. If you are using the Make option, then Turbo Pascal looks for both *unitname*.PAS and *unitname*.TPU, compares their latest modification dates and times, and recompiles the .PAS file if it has been modified since the .TPU file was created. Normally, when you write your own unit, it gets saved to a .TPU file; to use that unit, you must tell Turbo Pascal where to find it. If you're using the integrated environment, you must set the Unit directories option in the Options/Directories menu. (TURBO.TPL is loaded from the Turbo directory in the same menu.) If you're using the command-line environment, you must use the */U* option. (Use the */T* option to load the Turbo library from another subdirectory in the command-line compiler.)

You may have noticed, though, that you can use the standard Turbo Pascal units without giving a file name. That's because these units are stored in the Turbo Pascal standard unit file—TURBO.TPL on your distribution disk. Because the units are in that file, any program can use them without "knowing" their location.

Suppose you have a unit called TOOLS.TPU, and you use it in many different programs. Though adding *Tools* to TURBO.TPL takes up memory (TURBO.TPL is automatically loaded into memory by the compiler), adding it to the resident library makes "using" *Tools* faster because the unit is in memory instead of on disk.

There are five standard units already in TURBO.TPL: *System*, *Overlay*, *Printer*, *Crt*, and *Dos*.

Using TPUMOVER

TPUMOVER is a display-oriented program, much like the integrated environment. It shows you the units contained in two different files and allows you to copy units back and forth between them or to delete units from a given file. It's primarily used for moving files in and out of TURBO.TPL, but it has other useful functions.

Note that the TPUMOVER display consists of two side-by-side windows. The name of the file appears at the top of the window, followed by a list of the units in that file. Each line in a window gives information for a single unit: unit name, code size, data size, symbol table size, and the name(s) of any unit(s) that this unit uses. The sizes are all in bytes, and the unit names are all truncated to seven characters. If the list of units being used is too long to fit, it ends with three dots; press *F4* to see a pop-up window with the names of the other unit dependencies. Finally, two lines of information

appear in that window, giving (in bytes) the current size of that file and the amount of free space on the disk drive containing that file.

At any time, one of the two windows is the "active" window. This is indicated by a double line around the active window. The active window also contains a highlighted bar that appears within the list of units in that file; the bar can be moved up or down using the arrow keys. All commands apply to the active window; pressing *F6* switches you back and forth between the two windows.

To use TPUMOVER, simply type

```
TPUMOVER file1 file2
```

where *file1* and *file2* are .TPL or .TPU files. The extension .TPU is assumed, so you must explicitly add .TPL for .TPL files.

TPUMOVER loads and displays two windows—with *file1* in the left window of the display and *file2* in the right window. Note that both *file1* and *file2* are optional. If you only specify *file1*, then the right window has the default name NONAME.TPU. If you don't specify either file, TPUMOVER will attempt to load TURBO.TPL in the left window, with nothing in the right window. If that file cannot be found, TPUMOVER will display a directory of all files on the current disk that end in .TPL.

TPUMOVER Commands

The basic commands are listed at the bottom of the screen. Here's a brief description of each:

- *F1* brings up a help screen.
- *F2* saves the current file (the file associated with the active window) to disk.
- *F3* lets you select a new file for the active window.
- *F4* displays a pop-up window showing you all the unit dependencies for that unit. Only the first unit dependency is shown in the main window. If there are three dots following it, there are additional ones to be found by pressing *F4*.
- *F6* allows you to switch between the two windows, making the inactive window the active window (and vice versa).
- + (plus sign) marks a unit (for copying or deletion). You can have multiple units marked simultaneously; also, you can unmark a marked unit by pressing the + key again.
- *Ins* copies all marked units from the active window to the inactive window.

- *Del* deletes all marked units from the active window.
- *Esc* lets you exit from TPUMOVER. Note that this does not automatically save any changes that were made; you must explicitly use *F2* to save modifications before leaving TPUMOVER.

Moving Units into TURBO.TPL

Let's suppose you've created a unit *Tools*, which you've compiled into a file named TOOLS.TPU. You like this unit so much you want to put it into TURBO.TPL. How do you do this? To start, type the command

```
TPUMOVER TURBO TOOLS
```

This will bring up the TPUMOVER display with TURBO.TPL in the left window (the active one) and TOOLS.TPU in the right window. Note that this example assumes that TURBO.TPL and TOOLS.TPU are both in the current directory; if they are not, then you need to supply the appropriate path name for each.

Now perform the following steps:

1. Press *F6* to make the right window (TOOLS.TPU) active.
2. Press + to mark *IntLib* (the only unit in the right-hand window).
3. Press *Ins* to copy *IntLib* into TURBO.TPL.
4. Press *F6* to make the left window (TURBO.TPL) active.
5. Press *F2* to save the changes in TURBO.TPL to disk.
6. Press *Esc* to exit TPUMOVER.

The unit *Tools* is now part of TURBO.TPL and will be automatically loaded whenever you use Turbo Pascal.

If you want to add other units to TURBO.TPL, you can do so without exiting TPUMOVER. After pressing *F2* to save TURBO.TPL to disk, perform the following steps:

1. Press *F6* to make the right window active.
2. Press *F3* to select a new file for the right window.
3. Repeat the preceding steps two through five to mark the appropriate unit, copy it into TURBO.TPL, make the left window active, and save TURBO.TPL to disk.

You can repeat this as many times as desired in order to build up your library.

Deleting Units from TURBO.TPL

Let's assume that most of your programs do not use the *Overlay* or *Printer* units, and let's remove them from TURBO.TPL. To do this, first type

```
TPUMOVER TURBO
```

This brings up TPUMOVER with TURBO.TPL in the left window and NONAME.TPU (the default name) in the right. The left window is the active one, so do the following:

- Use the *Down arrow* key to move the highlighted bar over *Overlay*.
- Press + to select *Overlay*.
- Press *Del* to delete *Overlay*.
- Press *F2* to save the changes to TURBO.TPL.
- Press *Esc* to exit TPUMOVER.

You can repeat this procedure to remove *Printer*.

Moving Units Between .TPL Files

Suppose a friend has written a number of units and has given you the file (MYSTUFF.TPL) that contains them. You want to copy only the units *GameStuff* and *RandStuff* into TURBO.TPL. How do you do this? Your command line would read like this:

```
TPUMOVER MYSTUFF.TPL TURBO.TPL
```

This brings up TPUMOVER with MYSTUFF.TPL in the left (active) window and TURBO.TPL in the right window. Now use the following commands:

- Use the *Up arrow* and *Down arrow* keys to move the highlighted bar to *GameStuff*.
- Press + to select *GameStuff*.
- Use the *Up arrow* or the *Down arrow* key to move the highlighted bar to *RandStuff*.
- Press + to select *RandStuff*.
- Press *Ins* to copy *GameStuff* and *RandStuff* to TURBO.TPL.
- Press *F6* to make the TURBO.TPL window active.
- Press *F2* to save the changes made to TURBO.TPL.
- Press *Esc* to exit TPUMOVER.

Command-Line Shortcuts

You can use several command-line parameters that let you manipulate units quickly. The format for these parameters is

```
TPUMOVER TURBO /parameter unitname
```

where *parameter* is either +, –, or *.

These commands perform the following functions without displaying the side-by-side windows of the TPUMOVER program:

/+ Adds the named unit to TURBO.TPL

/– Deletes the named unit from TURBO.TPL

/* Extracts (copies) the named unit from TURBO.TPL and saves it in a file named *unitname*.TPU

/? Displays a small help window

The Stand-Alone MAKE Utility

This section contains complete documentation for creating makefiles and using MAKE.

Creating Makefiles

A makefile contains the definitions and relationships needed to help MAKE keep your program(s) up to date. You can create as many makefiles as you want and name them whatever you want. If you don't specify a makefile when you run MAKE (using the *-f* option), then MAKE looks for a file with the default name MAKEFILE.

You create a makefile with any ASCII text editor, such as Turbo Pascal's built-in interactive editor. All rules, definitions, and directives end with a carriage return; if a line is too long, you can continue it to the next line by placing a backslash (\) as the last character on the line.

Whitespace—spaces and tabs—is used to separate adjacent identifiers (such as dependencies) and to indent commands within a rule.

Creating a makefile is almost like writing a program—with definitions, commands, and directives. Here's a list of the constructs allowed in a makefile:

- comments
- explicit rules
- implicit rules
- macro definitions
- directives: file inclusion, conditional execution, error detection, macro undefinition

Let's look at each of these in more detail.

Comments

Comments begin with a number sign (#); the rest of the line following the # is ignored by MAKE. Comments can be placed anywhere and never have to start in a particular column.

A backslash (\) will *not* continue a comment onto the next line; instead, you must use a # on each line. In fact, you cannot use a backslash as a continuation character in a line that has a comment. That's because if the backslash precedes the #, it is no longer the last character on the line; if it follows the #, it is part of the comment itself.

Here are some examples of comments in a makefile:

```
# makefile for GETSTARS.EXE
# does complete project maintenance
# implicit rule
.asm.obj:                                  #.OBJ files depend on .ASM files
  tasm $*.asm,$*.obj;                          # command to create them
# unconditional rule
getstars.exe:                                # always create GETSTARS.EXE
  tpc getstars /m                                  # command to create it
# dependencies
slib2.obj: slib2.asm                       # uses the implicit rule above
slibl.obj: slibl.asm                         # recast as an explicit rule
  tasm slibl.asm,slibl.obj;
# end of makefile
```

Explicit Rules

Explicit rules take the form

```
target [target ... ]: [source source ... ]
   [command]
   [command]
   ...
```

where *target* is the file to be updated, *source* is a file upon which *target* depends, and *command* is any valid MS-DOS command (including invocation of .BAT files and execution of .COM and .EXE files).

Explicit rules define one or more target names, zero or more source files, and an optional list of commands to be performed. Target and source file names listed in explicit rules can contain normal MS-DOS drive and directory specifications, but they cannot contain wildcards.

Syntax here is important. *target* must be at the start of a line (in column 1), and each *command* must be indented (preceded by at least one space character or tab). As mentioned before, the backslash (\) can be used as a continuation character if the list of source files or a given command is too long for one line. Finally, both the source files and the commands are optional; it is possible to have an explicit rule consisting only of *target [target ...]* followed by a colon.

The idea behind an explicit rule is that the command or commands listed will create or update *target*, usually using the *source* files. When MAKE encounters an explicit rule, it first checks to see if any of the *source* files are target files elsewhere in the makefile. If so, those rules are evaluated first.

Once all the *source* files have been created or updated based on other explicit (or implicit) rules, MAKE checks to see if *target* exists. If not, each *command* is invoked in the order given. If *target* does exist, its time and date of last modification are compared against the time and date for each *source*. If any *source* has been modified more recently than *target*, the list of commands is executed.

A given file name can occur on the left side of an explicit rule only once in a given execution of MAKE.

Each command line in an explicit rule begins with whitespace. MAKE considers all lines following an explicit rule to be part of the command list for that rule, up to the next line that begins in column 1 (without any preceding whitespace) or up to the end of the file. Blank lines are ignored.

An explicit rule, with no command lines following it, is treated a little differently than an explicit rule with command lines.

- If an explicit rule exists for a target with commands, the only files that the target depends on are the ones listed in the explicit rule.
- If an explicit rule has no commands, the targets depend on the files given in the explicit rule, and they also depend on any file that matches an implicit rule for the target(s).

Here is a makefile with examples of explicit rules:

```
myutil.obj: myutil.asm
  tasm myutil.asm,myutil.obj;

myapp.exe:  myapp.pas myglobal.tpu myutils.tpu
  tpc myapp /Tc:\tp5\bin

myglobal.tpu: myglobal.pas
  tpc myglobal /Tc:\tp5\bin

myutils.tpu: myutils.pas myglobal.tpu myutil.obj
  tpc myutils /Tc:\tp5\bin
```

- The first explicit rule states that MYUTIL.OBJ depends upon MYUTIL.ASM, and that MYUTIL.OBJ is created by executing the given TASM command.
- The second rule states that MYAPP.EXE depends upon MYAPP.PAS, MYGLOBAL.TPU, and MYUTILS.TPU, and is created by the given TPC command. (The */T* plus path name in these examples will be explained later.)
- The third rule states that MYGLOBAL.TPU depends upon MYGLOBAL.PAS, and is created by the given TPC command.
- The last rule states that MYUTILS.TPU depends upon MYUTILS.PAS, MYGLOBAL.TPU, and MYUTIL.OBJ, and is created by the given TPC command.

If you reorder the rules so that the one for MYAPP.EXE comes first, followed by the others, MAKE will recompile (or reassemble) only the files that it has to in order to update everything correctly. This is because a MAKE with no target on the command line will try to execute the first explicit rule it finds in the makefile.

In practice, you would usually omit the last two explicit rules and simply append a */M* directive to the command under the explicit rule for MYAPP.EXE. However, you will need to add all the source dependencies from MYGLOBAL.TPU and MYUTILS.TPU to the source for MYAPP.EXE.

Implicit Rules

MAKE also allows you to define *implicit* rules, which are generalizations of explicit rules. Here's an example to illustrate the relationship between the two types. Consider this explicit rule from the previous sample program:

```
myutil.obj: myutil.asm
  tasm myutil.asm,myutil.obj;
```

This rule is a common one, because it follows a general principle: An .OBJ file is dependent on the .ASM file with the same file name and is created by executing TASM (Turbo Assember). In fact, you might have a makefile where you have several (or even several dozen) explicit rules following this same format.

By redefining the explicit rule as an implicit rule, you can eliminate all the explicit rules of the same form. As an implicit rule, it would look like this:

```
.asm.obj:
  tasm $*.asm,$*.obj;
```

This rule means, "any file ending with .OBJ depends on the file with the same name that ends in .ASM, and the .OBJ file is created using the command `tasm $*.asm,$*.obj`, where *$** represents the file's name with no extension." (The symbol *$** is a special macro and is discussed in the next section.)

The syntax for an implicit rule follows:

```
.source_extension.target_extension:
  {command}
  {command}
  ...
```

Note the commands are optional and must be indented. The *source_extension* (which must begin in column 1) is the extension of the source file, that is, it applies to any file having the format

```
fname.source_extension
```

Likewise, the *target_extension* refers to the the file

```
fname.target_extension
```

where *fname* is the same for both files. In other words, this implicit rule replaces all explicit rules having the format

```
fname.target_extension:fname.source_extension
  [command]
  [command]
  ...
```

for any *fname*.

Implicit rules are used if no explicit rule for a given target can be found or if an explicit rule with no commands exists for the target.

The extension of the file name in question is used to determine which implicit rule to use. The implicit rule is applied if a file is found with the same name as the target, but with the mentioned source extension. For

example, suppose you had a makefile (named MAKEFILE) whose contents were

```
.asm.obj:
  tasm $*.asm,$*.obj;
```

If you had an assembly language routine named RATIO.ASM that you wanted to compile to RATIO.OBJ, you could use the command

```
make ratio.obj
```

MAKE would take RATIO.OBJ to be the target. Since there is no explicit rule for creating RATIO.OBJ, MAKE applies the implicit rule and generates the command

```
tasm ratio.asm,ratio.obj;
```

which, of course, uses Turbo Assembler to create RATIO.OBJ.

Implicit rules are also used if an explicit rule is given with no commands. Suppose, as mentioned before, you had the following implicit rule at the start of your makefile:

```
.pas.tpu:
  tpc $<
```

You could then rewrite the last two explicit rules as follows:

```
myglobal.tpu: myglobal.pas
myutils.tpu: myutils.pas myglobal.tpu myutil.obj
```

Since you don't have explicit information on how to create these .TPU files, MAKE applies the implicit rule defined earlier.

Several implicit rules can be written with the same target extension, but only one such rule can apply at a time. If more than one implicit rule exists for a given target extension, each rule is checked in the order the rules appear in the makefile, until all applicable rules are checked.

MAKE uses the first implicit rule that it discovers for a file with the source extension. Even if the commands of that rule fail, no more implicit rules are checked.

All lines following an implicit rule are considered to be part of the command list for the rule, up to the next line that begins without whitespace or to the end of the file. Blank lines are ignored. The syntax for a command line is provided later in this appendix.

MAKE does not know the full file name with an implicit rule, as it does with explicit rules. For that reason, special macros are provided with MAKE that allow you to include the name of the file being built by the rule. (For details, see the discussion of macro definitions later in this appendix.)

Here are some examples of implicit rules:

```
.pas.exe:
  tpc $<

.pas.tpu:
  tpc $<

.asm.obj:
  tasm $* /mx;
```

In the first example, the target files are .EXE files and their source files are .PAS files. This example has one command line in the command list (command-line syntax is discussed later in this chapter). Likewise, the second implicit rule creates .TPU files from .PAS files.

The last example directs MAKE to assemble a given file from its .ASM source file, using TASM with the */mx* option.

Command Lists

We've talked about both explicit and implicit rules, and how they can have lists of commands. Let's talk about those commands and your options for setting them up.

Commands in a command list must be indented—that is, preceded by at least one space character or tab—and take the form

```
[ prefix ... ] command_body
```

Each command line in a command list consists of an (optional) list of prefixes, followed by a single command body.

The prefixes allowed in a command modify the treatment of these commands by MAKE. The prefix is either the at (@) sign or a hyphen (-) followed immediately by a number.

@ Keeps MAKE from displaying the command before executing it. The display is hidden even if the *–s* option was not given on the MAKE command line. This prefix applies only to the command on which it appears.

–num Affects how MAKE treats exit codes. If a number (*num*) is provided, then MAKE will abort processing only if the exit status exceeds the number given. In this example, MAKE will abort only if the exit status exceeds 4:

```
-4 myprog sample.x
```

If no *–num* prefix is given, MAKE checks the exit status for the command. If the status is nonzero, MAKE will stop and delete the current target file.

– With a hyphen but no number, MAKE will not check the exit status at all. Regardless of what the exit status was, MAKE will continue.

The *command body* is treated exactly as if it were entered as a line to COMMAND.COM, with the exception that redirection and pipes are not supported. MAKE executes the following built-in commands by invoking a copy of COMMAND.COM to perform them:

```
BREAK       CD          CHDIR       CLS         COPY
MD          MKDIR       PATH        PROMPT      REN
RENAME      SET         TIME        TYPE        VER
VERIFY      VOL
```

MAKE searches for any other command name using the MS-DOS search algorithm:

- The current directory is searched first, followed by each directory in the path.
- In each directory, first a file with the extension .COM is checked, then an .EXE file, and finally a .BAT.
- If a .BAT file is found, a copy of COMMAND.COM is invoked to execute the batch file.

Obviously, if an extension is supplied in the command line, MAKE searches only for that extension.

This command will cause COMMAND.COM to execute the Change directory command:

```
cd c:\include
```

This command will cause MYPROG.PAS to be searched for, using the full search algorithm:

```
tpc myprog.pas /$B+,R+,I+
```

This command will cause MYPROG.COM to be searched for, using only the .COM extension:

```
myprog.com geo.xyz
```

This command will be executed using the explicit file name provided:

```
c:\myprogs\fil.exe -r
```

Macros

Often certain commands, file names, or options are used again and again in your makefile. In an example earlier in this appendix, all the TPC commands used the switch */Tc:\tp5\bin,* which means that the files TPC.CFG and TURBO.TPL are in the subdirectory C:\TP5\BIN. Suppose you wanted to switch to another subdirectory for those files; what would you do? You could go through and modify all the */T* options, inserting the appropriate path name. Or, you could define a macro.

A *macro* is a name that represents some string of characters (letters and digits). A *macro definition* gives a macro name and the expansion text; thereafter, when MAKE encounters the macro name, it replaces the name with the expansion text.

Suppose you defined the following macro at the start of your makefile:

```
TURBO=c:\tp5\bin
```

You've defined the macro *TURBO,* which is equivalent to the string *c:\tp5\bin.* You could now rewrite the makefile as follows:

```
TURBO=c:\tp5\bin
myapp.exe:  myapp.pas myglobal.tpu myutils.tpu
  tpc myapp /T$(TURBO)

myutils.tpu: myutils.pas myglobal.tpu myutil.obj
  tpc myutils /T$(TURBO)

myglobal.tpu: myglobal.pas
  tpc myglobal /T$(TURBO)

myutil.obj: myutil.asm
  tasm myutil.asm,myutil.obj;
```

Everywhere the Turbo directory is specified, you use the macro invocation *$(TURBO).* When you run MAKE, *$(TURBO)* is replaced with its expansion text, `c:\TP5.BIN`. The result is the same set of commands you had before.

So what have you gained? Flexibility. By changing the first line to

```
TURBO=c:\tp5\project
```

you've changed all the commands to use the configuration and library files in a different subdirectory. In fact, if you leave out the first line altogether, you can specify which subdirectory you want each time you run MAKE, using the *-D* (Define) option:

```
make -DTURBO=c:\tp5\project
```

This tells MAKE to treat *TURBO* as a macro with the expansion text *c:\tp5\project.*

Macro definitions take the form

macro_name=expansion text

where *macro_name* is the name of a macro made up of a string of letters and digits with no whitespace in it, though you can have whitespace between *macro_name* and the equal sign (=). *expansion text* is any arbitrary string containing letters, digits, whitespace, and punctuation; it is ended by a carriage return.

If *macro_name* has previously been defined, either by a macro definition in the makefile or by the *-D* option on the MAKE command line, the new definition replaces the old.

Case is significant in macros; that is, the macros names *turbo, Turbo,* and *TURBO* are all considered to be different.

Macros are invoked in your makefile with the format

```
$(macro_name)
```

The parentheses are required for all invocation, even if the macro name is just one character, with the exception of six special predefined macros that we'll talk about in just a minute. This construct—*$(macro_name)*—is known as a *macro invocation.*

When MAKE encounters a macro invocation, it replaces the invocation with the macro's expansion text. If the macro is not defined, MAKE replaces it with the null string.

Macros in macros: Macros cannot be invoked on the left (*macro_name*) side of a macro definition. They can be used on the right (*expansion text*) side, but they are not expanded until the macro being defined is invoked. In other words, when a macro invocation is expanded, any macros embedded in its expansion text are also expanded.

Macros in rules: Macro invocations are expanded immediately in rule lines.

Macros in directives: Macro invocations are expanded immediately in *!if* and *!elif* directives. If the macro being invoked in an *!if* or *!elif* directive is not currently defined, it is expanded to the value 0 (False).

Macros in commands: Macro invocations in commands are expanded when the command is executed.

MAKE comes with several special predefined macros built-in: *$d, $*, $<, $:, $.,* and *$&.* The first is a defined test macro, used in the conditional directives *!if* and *!elif;* the others are file name macros, used in explicit and implicit rules. The various file name macros work in similar ways, expanding to some variation of the full path name of the file being built. In

addition, the current SET environment strings are automatically loaded as macros, and the macro *__MAKE__* is defined to be 1 (one).

Defined Test Macro ($d)

This macro expands to 1 if the given macro name is defined, or to 0 if it is not. The content of the macro's expansion text does not matter. This special macro is allowed only in *!if* and *!elif* directives. For example, if you wanted to modify your makefile so that it would use a particular Turbo Pascal directory if you didn't specify one, you could put this at the start of your makefile:

```
!if !$d(TURBO)                              # if TURBO is not defined
TURBO=c:\tp5\bin                            # define it to C:\TP5\BIN
!endif
```

If you invoke MAKE with the command line

```
make -DTURBO=c:\tp5\project
```

then *TURBO* is defined as *c:\tp5\project*. If, however, you just invoke MAKE by itself,

```
make
```

then *TURBO* is defined as `c:\tp5\bin`, your "default" subdirectory.

Base File Name Macro ($)*

This macro is allowed in the commands for an explicit or an implicit rule. The macro expands to the file name being built, excluding any extension, like this:

```
File name is A:\P\TESTFILE.PAS
$* expands to A:\P\TESTFILE
```

For example, you could modify the explicit MYAPP.EXE rule already given to look like this:

```
myapp.exe:  myapp.pas myglobal.tpu myutils.tpu
  tpc $* /T$(TURBO)
```

When the command in this rule is executed, the macro *$** is replaced by the target file name (without an extension), *myapp*. This macro is very useful for implicit rules. For example, an implicit rule for TPC might look like this (assuming that the macro *TURBO* has been or will be defined):

```
.pas.exe:
  tpc $* /T$(TURBO)
```

Full File Name Macro ($<)

The full file name macro ($<) is also used in the commands for an explicit or implicit rule. In an explicit rule, $< expands to the full target file name (including extension), like this:

```
File name is A:\P\TESTFILE.PAS
$< expands to A:\P\TESTFILE.PAS
```

For example, the rule

```
starlib.tpu: starlib.pas
  copy $< \oldtpus
  tpc $* /T$(TURBO)
```

will copy STARLIB.TPU to the directory \OLDTPUS before compiling STARLIB.PAS.

In an implicit rule, $< takes on the file name plus the source extension. For example, the previous implicit rule

```
.asm.obj:
  tasm $*.asm,$*.obj;
```

can be rewritten as

```
.asm.obj:
   tasm $<,$*.obj;
```

File Name Path Macro ($:)

This macro expands to the path name (without the file name), like this:

```
File name is A:\P\TESTFILE.PAS
$: expands to A:\P\
```

File Name and Extension Macro ($.)

This macro expands to the file name, with extension, like this:

```
File name is A:\P\TESTFILE.PAS
$. expands to TESTFILE.PAS
```

File Name Only Macro ($&)

This macro expands to the file name only, without path or extension, like this:

```
File name is A:\P\TESTFILE.PAS
$& expands to TESTFILE
```

Directives

The version of MAKE bundled with Turbo Pascal allows something that other versions of MAKE don't: conditional directives similiar to those allowed for Turbo Pascal. You can use these directives to include other makefiles, to make the rules and commands conditional, to print out error messages, and to "undefine" macros.

Directives in a makefile begin with an exclamation point (!). Here is the complete list of MAKE directives:

```
!include
!if
!else
!elif
!endif
!error
!undef
```

A *file-inclusion directive* (*!include*) specifies a file to be included into the makefile for interpretation at the point of the directive. It takes the following form:

```
!include "filename"
```

or

```
!include <filename>
```

These directives can be nested arbitrarily deep. If an include directive attempts to include a file that has already been included in some outer level of nesting (so that a nesting loop is about to start), the inner include directive is rejected as an error.

How do you use this directive? Suppose you created the file PATH.MAC so it contained the following:

```
!if !$d(TURBO)
TURBO=c:\tp5\bin
!endif
```

You could then make use of this conditional macro definition in any makefile by including the directive

```
!include "PATH.MAC"
```

When MAKE encounters the *!include* directive, it opens the specified file and reads the contents as if they were in the makefile itself.

Conditional directives (*!if*, *!elif*, *!else*, and *!endif*) give a programmer a measure of flexibility in constructing makefiles. Rules and macros can be

"conditionalized" so that a command-line macro definition (using the *-D* option) can enable or disable sections of the makefile.

The format of these directives parallels, but is more extensive than, the conditional directives allowed by Turbo Pascal:

```
!if expression
  [ lines ]
!endif

!if expression
  [ lines ]
!else
  [ lines ]
!endif

!if expression
  [ lines ]
!elif expression
  [ lines ]
!endif
```

Note: *[lines]* can be any of the following:

```
macro_definition
explicit_rule
implicit_rule
include_directive
if_group
error_directive
undef_directive
```

The conditional directives form a group, with at least an !if directive beginning the group and an *!endif* directive closing the group.

- One *!else* directive can appear in the group.
- *!elif* directives can appear between the *!if* and any *!else* directives.
- Rules, macros, and other directives can appear between the various conditional directives in any number. Note that complete rules, with their commands, cannot be split across conditional directives.
- Conditional directive groups can be nested arbitrarily deep.

Any rules, commands, or directives must be complete within a single source file.

Any *!if* directives must have matching *!endif* directives within the same source file. Thus the following include file is illegal regardless of what is contained in any file that might include it, because it does not have a matching *!endif* directive:

```
!if $(FILE_COUNT) > 5
```

```
some rules
!else
other rules
<end-of-file>
```

The expression allowed in an *!if* or an *!elif* directive uses a syntax similar to that found in the C programming language. The expression is evaluated as a simple 32-bit signed integer expression.

Numbers can be entered as decimal, octal, or hexadecimal constants. For example, these are legal constants in an expression:

```
4536                                        # decimal constant
0677                  # octal constant (note the leading zero)
0x23aF                                  # hexadecimal constant
```

and any of the following unary operators:

–	negation
~	bit complement
!	logical not

An expression can use any of the following binary operators:

+	addition
–	subtraction
*	multiplication
/	division
%	remainder
>>	right shift
<<	left shift
&	bitwise and
|	bitwise or
^	bitwise exclusive or
&&	logical and
||	logical or
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
==	equality
!=	inequality

An expression can contain the following ternary operator:

? : The operand before the ? is treated as a test.

If the value of that operand is nonzero, then the second operand (the part between the ? and the colon) is the result. If the value of

the first operand is zero, the value of the result is the value of the third operand (the part after the :).

Parentheses can be used to group operands in an expression. In the absence of parentheses, binary operators are grouped according to the same precedence given in the C language.

Grouping is from left to right for operators of equal precedence, except for the ternary operator (? :), which is right to left.

Macros can be invoked within an expression, and the special macro *$d()* is recognized. After all macros have been expanded, the expression must have proper syntax. Any words in the expanded expression are treated as errors.

The *error directive* (*!error*) causes MAKE to stop and print a fatal diagnostic containing the text after *!error*. It takes the format

```
!error any_text
```

This directive is designed to be included in conditional directives to allow a user-defined abort condition. For example, you could insert the following code in front of the first explicit rule:

```
!if !$d(TURBO)
# if TURBO is not defined
!error TURBO not defined
!endif
```

If you reach this spot without having defined *TURBO*, then MAKE will stop with this error message:

```
Fatal makefile 5: Error directive: TURBO not defined
```

The *undefine directive* (*!undef*) causes any definition for the named macro to be forgotten. If the macro is currently undefined, this directive has no effect. The syntax is

```
!undef macro_name
```

Using MAKE

You now know a lot about how to write makefiles; now's the time to learn how to use them with MAKE. The simplest way to use MAKE is to type the command

```
make
```

at the MS-DOS prompt. MAKE then looks for MAKEFILE; if it can't find it, it looks for MAKEFILE.MAK; if it can't find that, it halts with an error message.

What if you want to use a file with a name other than MAKEFILE or MAKEFILE.MAK? You give MAKE the file (*-f*) option, like this:

```
make -fstars.mak
```

The general syntax for MAKE is

```
make option option ... target target ...
```

where *option* is a MAKE option (discussed later) and *target* is the name of a target file to be handled by explicit rules.

Here are the syntax rules:

- The word *make* is followed by a space, then a list of make options.
- Each make option must be separated from its adjacent options by a space. Options can be placed in any order, and any number of these options can be entered (as long as there is room in the command line).
- After the list of make options comes a space, then an optional list of targets.
- Each target must also be separated from its adjacent targets by a space. MAKE evaluates the target files in the order listed, recompiling their constituents as necessary.

If the command line does not include any target names, MAKE uses the first target file mentioned in an explicit rule. If one or more targets are mentioned on the command line, they will be built as necessary.

Here are some more examples of MAKE command lines:

```
make -n -fstars.mak
make -s
make -Iinclude -DTURBO=c:\tp5\project
```

MAKE will stop if any command it has executed is aborted via a *Ctrl-Break*. Thus, a *Ctrl-C* will stop the currently executing command and MAKE as well.

The BUILTINS.MAK File

As you become familiar with MAKE, you will find that there are macros and rules (usually implicit ones) that you use again and again. You've got three ways of handling them. First, you can put them in every makefile you create. Second, you can put them all in one file and use the *!include*

directive in each makefile you create. Third, you can put them all in a file named BUILTINS.MAK.

Each time you run MAKE, it looks for a file named BUILTINS.MAK; if it finds the file, MAKE reads it in before handling MAKEFILE (or whichever makefile you want it to process).

The BUILTINS.MAK file is intended for any rules (usually implicit rules) or macros that will be commonly used in files anywhere on your computer.

There is no requirement that any BUILTINS.MAK file exist. If MAKE finds a BUILTINS.MAK file, it interprets that file first. If MAKE cannot find a BUILTINS.MAK file, it proceeds directly to interpreting MAKEFILE (or whatever makefile you specify).

How MAKE Searches for Files

MAKE will search for BUILTINS.MAK in the current directory or in the exec directory if your computer is running under DOS 3.x. You should place this file in the same directory as the MAKE.EXE file.

MAKE always searches for the makefile in the current directory only. This file contains the rules for the particular executable program file being built. The two files have identical syntax rules.

MAKE also searches for any *!include* files in the current directory. If you use the *–I* (Include) option, it will also search in the specified directory.

MAKE Command-Line Options

We've alluded to several of MAKE's command-line options; now we'll present a complete list of them. Note that case (upper or lower) *is* significant; the option *–d* is not a valid substitute for *–D*.

–D*identifier* Defines the named identifier to the string consisting of the single character 1.

–D*iden=string* Defines the named identifier *iden* to the string after the equal sign. The string cannot contain any spaces or tabs.

–I*directory* MAKE will search for include files in the indicated directory (as well as in the current directory).

–U*identifier* Undefines any previous definitions of the named identifier.

–s	Normally, MAKE prints each command as it is about to be executed. With the *–s* option, no commands are printed before execution.
–n	Causes MAKE to print the commands, but not actually perform them. This is useful for debugging a makefile.
–ffilename	Uses *filename* as the MAKE file. If *filename* does not exist and no extension is given, tries *filename*.MAK.
–? or **–h**	Prints help message.

MAKE Error Messages

MAKE diagnostic messages fall into two classes: fatals and errors. When a fatal error occurs, execution immediately stops. You must take appropriate action and then restart the execution. Errors will indicate some sort of syntax or semantic error in the source makefile. MAKE will complete interpreting the makefile and then stop.

Fatal Errors

Don't know how to make XXXXXXXX

This message is issued when MAKE encounters a nonexistent file name in the build sequence, and no rule exists that would allow the file name to be built.

Error directive: XXXX

This message is issued when MAKE processes an *#error* directive in the source file. The text of the directive is displayed in the message.

Incorrect command line argument: XXX

This error occurs if MAKE is executed with incorrect command-line arguments.

Not enough memory

This error occurs when the total working storage has been exhausted. You should try this on a machine with more memory. If you already have 640K in your machine, you may have to simplify the source file.

Unable to execute command

This message is issued after attempting to execute a command. This could be a result of the command file not being found, or because it was misspelled. A less likely possibility is that the command exists but is somehow corrupted.

Unable to open makefile

This message is issued when the current directory does not contain a file named MAKEFILE.

Errors

Bad file name format in include statement

Include file names must be surrounded by quotes or angle brackets. The file name was missing the opening quote or angle bracket.

Bad undef statement syntax

An *!undef* statement must contain a single identifier and nothing else as the body of the statement.

Character constant too long

Character constants can be only one or two characters long.

Command arguments too long

The arguments to a command executed by MAKE were more than 127 characters—a limit imposed by MS-DOS.

Command syntax error

This message occurs if

- the first rule line of the makefile contained any leading whitespace.
- an implicit rule did not consist of *.ext.ext:*.
- an explicit rule did not contain a name before the : character.
- a macro definition did not contain a name before the = character.

Division by zero

A divide or remainder in an *!if* statement has a zero divisor.

Expression syntax error in *!if* statement

The expression in an *!if* statement is badly formed—it contains a mismatched parenthesis, an extra or missing operator, or a missing or extra constant.

File name too long

The file name given in an *!include* directive was too long for MAKE to process. File path names in MS-DOS must be no more than 78 characters long.

Illegal character in constant expression X

MAKE encountered some character not allowed in a constant expression. If the character is a letter, this indicates a (probably) misspelled identifier.

Illegal octal digit
An octal constant was found containing a digit of 8 or 9.

Macro expansion too long
A macro cannot expand to more than 4096 characters. This error often occurs if a macro recursively expands itself. A macro cannot legally expand to itself.

Misplaced elif statement
An *!elif* directive was encountered without any matching *!if* directive.

Misplaced else statement
An *!else* directive was encountered without any matching *!if* directive.

Misplaced endif statement
An *!endif* directive was encountered without any matching *!if* directive.

No file name ending
The file name in an include statement was missing the correct closing quote or angle bracket.

Redefinition of target XXXXXXXX
The named file occurs on the left-hand side of more than one explicit rule.

Unable to open include file XXXXXXXXX.XXX
The named file could not be found. This could also be caused if an include file included itself. Check whether the named file exists.

Unexpected end of file in conditional started on line #
The source file ended before MAKE encountered an *!endif*. The *!endif* was either missing or misspelled.

Unknown preprocessor statement
A ! character was encountered at the beginning of a line, and the statement name following was not *error, undef, if, elif, include, else,* or *endif.*

THELP: The Online Help Utility

THELP.COM is a RAM-resident (TSR) utility that accesses Turbo Pascal's online help information for you if you are using an editor other than the one in Turbo Pascal's integrated development environment, or if you are using Turbo Debugger.

Installing THELP on Your System

If you used the INSTALL program to install Turbo Pascal on your system (see page 15), THELP.COM should already be in your main Turbo Pascal directory. If it is not there, copy it from the distribution disks into your main directory.

Make sure that TURBO.HLP, the text file containing the Turbo Pascal online help information, is in the same directory as THELP.COM. (If for some reason you wish to keep TURBO.HLP in another directory, THELP does have a special */F* command-line option, described shortly, that will enable THELP to find it.)

The amount of memory that THELP requires depends on whether you elect to use it with a swap file to conserve memory. If you elect to use the swap file, THELP occupies 8K of memory, with an additional 32K for the swap file only while THELP is active. If you use it without the swap file, THELP requires 40K of memory at all times.

Loading and Invoking THELP

To load THELP into memory so that you can invoke it from inside another application, just type `THELP` at the DOS command line before you go into your editor or Turbo Debugger.

Once you are in the other application, you can activate THELP at any time. Just position the cursor under the item you want information on, then press the THELP hot key. The default hot key is *5* on the numeric keypad (scan code 4ch, shift state 00h).

Warning: If you are going to have THELP resident in memory at the same time as SideKick 1.x or SideKick Plus, make sure you load THELP *before* you load SideKick.

For a brief introduction to Turbo Pascal's online help system in the context of the similarities and differences between using help in the integrated development environment and using THELP, refer to page 24 in Chapter 2, "Beginning Turbo Pascal."

The THELP Cursor Keys

Use the following keys to navigate through the help screens that THELP displays on your monitor:

Table C.1: THELP Command Keys

Command Key	Function
Arrow keys	Move the highlight from keyword to keyword within the current help screen.
PgUp/PgDn	Moves from screen to screen if additional screens are available.
Enter	Selects the entry for the keyword highlighted in the current help screen.
Esc	Ends help session.
F1	Displays the help index screen. Press *F1* in any help screen to call up the help index.
Alt-F1	Pressing *Alt-F1* repeatedly takes you in reverse order through the last 20 screens you have reviewed.
Ctrl-F1	Brings up the help screen for THELP's hot keys.
F	Lets you select a new help file to replace TURBO.HLP. Press *F* to call up a data entry box that allows you to change help files on the fly. Type in the complete path name of the new help file; the file will be read into memory and initialized as the help index of the new file. If the new help file you entered does not exist or is in an invalid format, THELP will beep twice, and return you to the original file.
I	Pastes the highlighted keyword into the file in your editor. Press *I* to insert the current highlighted keyword into the keyboard buffer and end the help session.
J	Jumps to the specified help page number. Press *J* to bring up a data entry box that allows you to enter the number of any particular page (1 to 9999) in the help file. The only editing key that you can use in the data entry box is *Backspace*. Press *Esc* to cancel the jump. Press *Enter* or type in four digits to execute the jump.
K	Searchs the help file for a specified keyword. Press *K* to bring up a data entry box in which you can enter a keyword (up to 40 characters), then press *Enter* to make THELP search the help file for a match. If there is no matching keyword in the current help file, THELP will beep twice and return to your previous help screen.
P	Pastes the entire help page into the file in your editor. Press *P* to insert the entire current help page (as it appears in the help window) into the current application. Pasting

Table C.1: THELP Command Keys (continued)

	can be interrupted with *Ctrl-C* or *Ctrl-Break*. (You can also adjust paste speed with a command-line option; see page 286 for information.)
S	Saves the current help screen to a disk file (THELP.SAV). Press *S* in any help screen to save the current help page to the ASCII file THELP.SAV, in the current directory. If the file already exists, the new help information is appended to the end.
5	If you load THELP with the */M+* option, you can press the THELP hot key (*5* on the numeric keypad is the default) twice to leave the current help screen displayed on the remote monitor and exit THELP. Your application regains control and you will see it displayed on the local monitor.

Summary of THELP Command-line Options

The basic syntax for loading THELP is

```
THELP [options]
```

Here is a summary of the THELP command-line options:

Table C.2: THELP Command-Line Options

Option	Function	
/B	Use BIOS for video	
/C#xx	Select color:	#=element number xx=hex color values
/Dname	Full path for disk swapping (implies /S1)	
/Fname	Full path and file name of help file	
/H, /?, ?	Display this help screen	
/Kxxyy	Change hot key:	xx=shift state(hex) yy=scan code(hex)
Lxx	Force number of rows on screen:	xx=25,43,50
/M+,/M-	Display help text:	on remote screen(+) on default screen(-)

Table C.2: THELP Command-Line Options (continued)

/Px	Pasting speed:*	0=slow 1=medium 2=fast (default)
/R	Send options to resident THELP	
/Sx	Default Swapping Mode:	1=Use Disk 2=Use EMS 3=No Swapping
/U	Remove THELP from memory	
/W	Write Options to THELP.COM and exit	
	*Paste speed: The default pasting speed is FAST. You'll have to experiment if it pastes too quickly for your editor. Note that you should turn off autoindent in the integrated environment before using the paste feature (*Ctrl-Q I* toggles autoindent).	

If you use more than one option, they must be separated by spaces.

The /B Option (Use BIOS for Video)

This option forces THELP to use Interrupt 10h BIOS video calls for all writing to/reading from the video display. Normally, THELP will write directly to video RAM.

Note that the use of this option negates the effect of the */M* switch described below; the alternate monitor may not be used if */B* is in effect. This option is enabled with */B+*, and disabled with */B-* (enabled is the default).

The /C#xx Option (Select Color)

This option lets you customize the background and foreground colors of various elements in a help screen. The */C* option is followed by the number of the element you want to customize and the hex color values for background and foreground respectively.

There are eight possible elements for which you can change the background and foreground colors. They are numbered as follows:

1 = Color Normal Text
2 = Monochrome Normal Text
3 = Color Possible Reference Pages (top/bottom description line)
4 = Monochrome Possible Reference Pages (top/bottom description line)
5 = Color Border Color
6 = Monochrome Border Color
7 = Color Current Reference Selection
8 = Monochrome Current Reference Selection

Any or all of these eight items may be specified on the command line.

The color values for a standard IBM-compatible Color Display are as follows:

Table C.3: Color Values for a Standard Color Display

First Digit (Background)		Second Digit (Foreground)	
0	Black	0	Black
1	Blue	1	Blue
2	Green	2	Green
3	Cyan	3	Cyan
4	Red	4	Red
5	Magenta	5	Magenta
6	Brown	6	Brown
7	Grey	7	Grey
		8	Intense Black
ORing the color value with Hex 80 produces a blinking color unless blinking has been disabled.		9	Intense Blue
		A	Intense Green
		B	Intense Cyan
		C	Intense Red
		D	Intense Magenta
		E	Intense Brown (Yellow)
		F	Intense Grey (White)

On monochrome monitors, the attribute values can differ widely, so some experimentation may be needed. Note that the monochrome attributes are used in only two cases; when the current video mode is 7, or when force mono is used (see "The */M* Option").

The /Dname Option (Full Path for Disk Swapping)

This option is used to override where THELP will place its swap files when it is swapping to disk. It assumes that the */S* option is set to 1 (that is, that swapping to disk will take place). A full path should be specified, but a

trailing '\' is not necessary. If no /D option is specified, under DOS 3.x swap files are placed in the directory where THELP.COM resides. Under DOS 2.x, swap files are placed by default in C:\.

Using this option also sets the flag that forces disk swapping instead of checking first for EMS.

The /Fname Option (Fill Path and File Name for Help File)

The name that follows the /F option should be the full drive/directory pathname of the help file to use; for example,

```
THELP /FC:\TP\TURBO.HLP
THELP /FC:\TPASCAL\TURBO.HLP
```

By default, THELP looks for the help file on the logged drive and directory.

The /H, /?, and ? Options (Display Help Screen)

Any of these options displays a summary of THELP's command-line options.

The /Kxxyy Option (Reassign Hot Key)

This option allows you to reassign a function to a new hot key. The option must be followed by the shift state (xx) and the scan code (yy) of the new key. Virtually any shift state/scan code combination may be selected. A quick summary of some common shift states and scan codes follows:

Shift States (may be OR'ed together)	
Right Shift	01h
Left Shift	02h
Ctrl	04h
Alt	08h

Scan Codes							
A	1eh	N	31h	0	0bh	F1	3bh
B	30h	O	18h	1	02h	F2	3ch
C	2eh	P	19h	2	03h	F3	3dh
D	20h	Q	10h	3	04h	F4	3eh
E	12h	R	13h	4	05h	F5	3fh
F	21h	S	1fh	5	06h	F6	40h
G	22h	T	14h	6	07h	F7	41h
H	23h	U	16h	7	08h	F8	42h
I	17h	V	2fh	8	09h	F9	43h
J	24h	W	11h	9	0ah	F10	44h
K	25h	X	2dh				
L	26h	Y	15h				
M	32h	Z	2ch				

Enhanced Keyboards only (may not work with all computers, keyboards)

F11	57h
F12	58h

The /Lxx Option (Force Number of Rows Onscreen)

This option allows you to specify the number of rows to be shown on the screen (for xx, enter 25, 43, or 50). This is sometimes necessary because some video adapters do not correctly store the number of video rows on the screen in the BIOS data location specified for the IBM-PC. This option forces THELP to use the given value, rather than the number the BIOS reports.

The /M+ and /M- Options (Display Help Text)

If you have a dual-monitor system, you can use this option to instruct THELP to bring up its display on your remote monitor, rather than on your local monitor. You can enable the remote display with */M+* and disable it with */M-*. Enabled is the default.

Note that */M* is incompatible with */B* (see above).

The /Px Option (Pasting Speed)

This option lets you specify the speed at which characters are pasted into the keyboard buffer (for x, substitute 0 for slow, 1 for medium, or 2 for fast).

Some editors do not allow characters to be pasted into the keyboard buffer as fast as THELP can put them there. By setting an appropriate paste speed, however, you can use virtually any editor configurations. Fast speed pastes as many characters as will fit on every timer tick; medium pastes up to four characters per tick; and slow pastes a single character into the buffer *only* when the buffer is empty.

The /R Option (Send Options to Resident THELP)

The */R* option is used to pass parameters (like new colors, or new hot keys) to the resident portion of THELP. All THELP command line options may be sent to the resident portion except for the swapping mode, which cannot be modified once THELP has been initialized.

In combination with these options, you can create a batch file that changes THELP's configuration as you change editors. For example,

`THELP /M /P0 /FC:\TP\TURBO.HLP /R`
: Use mono monitor, slow pasting, and the Turbo Pascal help file. Options are not saved to disk.

`THELP /P2 /FC:\TP\TURBO.HLP /R`
: Use default monitor, fast pasting, and the Turbo Pascal help file. Options are not saved to disk.

The /Sx Option (Default Swapping Mode)

This option lets you determine whether or not THELP will use a swap file, and where the swap file will be located (for x, substitute 1 for disk, 2 for EMS, or 3 no swap file).

If no */S* parameter is used, THELP first tests to see if Expanded Memory is available in the system. If it is, and if enough memory can be allocated, swapping is done to EMS. If EMS is not available, disk swapping is used. See the */D* parameter for information on where the swap file will be written if disk swapping is used.

The /U Option (Remove THELP from Memory)

This option is used to remove THELP from memory. If other TSRs have been loaded after THELP, make sure to remove them before removing THELP.

The /W Option (Write Options to THELP.COM and Exit)

The */W* parameter is used to create a new version of THELP that uses the options you desire as a default. All options, including */S* (but not */R*) may be specified and made permanent.

The TOUCH Utility

There are times when you want to force a particular target file to be recompiled or rebuilt, even though no changes have been made to its sources. One way to do this is to use the TOUCH utility included with Turbo Pascal. TOUCH changes the date and time of one or more files to the current date and time, making it "newer" than the files that depend on it.

To force a target file to be rebuilt, "touch" one of the files that target depends on. To touch a file (or files), enter

```
touch filename [ filename ... ]
```

at the DOS prompt. TOUCH will then update the file's creation date(s).

Once you do this, you can invoke MAKE to rebuild the touched target file(s).

You can use the DOS wildcards * and ? with TOUCH.

The GREP Utility

GREP is a powerful search utility that can look for text in several files at once. For example, if you had forgotten in which program you defined a procedure called *SetUpMyModem*, you could use GREP to search the contents of all the .PAS files in your directory to look for the string *SetUpMyModem*.

The command-line syntax for GREP follows:

```
GREP [options] searchstring [filespec ... ]
```

where *options* consists of one or more single characters preceded by a hyphen, *searchstring* defines the pattern to search for, and *filespec* is the file specification. *filespec* tells GREP which files (or groups of files) to search; it can be an explicit file name or a generic file name incorporating the DOS wildcards (? and *). You can also enter a path as part of *filespec;* if you use *filespec* without a path, GREP only searches the current directory. If you don't specify *filespec,* input to GREP must be specified by redirecting *stdin* or piping.

The GREP Switches

In the command line, *options* are one or more single characters preceded by a hyphen (-). Each individual character is a switch that you can turn on or off: type the plus symbol (+) after a character to turn the option on, or type a hyphen (-) after the character to turn the option off.

The default is on (the + is implied): for example, *–R* means the same thing as *–R+*. You can list multiple options individually like this: *–I –D –L*). Or you can combine them like this: *–ILD* or *–IL –D,* and so on). It's all the same to GREP.

Here is a list of the switches and their meanings:

–C *Count only*: Only a count of matching lines is printed. For each file that contains at least one matching line, GREP prints the file name and a count of the number of matching lines. Matching lines are not printed.

–D *Directories*: For each *filespec* specified on the command line, GREP searches for all files that match the file specification, both in the directory specified *and* in all subdirectories below the specified directory. If you give a *filespec* without a path, GREP assumes the files are in the current directory.

–I *Ignore case*: GREP ignores upper/lowercase differences (case folding). GREP treats all letters *a-z* as being identical to the corresponding letters *A-Z* in all situations.

–L *List match files*: Only the name of each file containing a match is printed. After GREP finds a match, it prints the file name and processing immediately moves on to the next file.

–N *Numbers*: Each matching line that GREP prints is preceded by its line number.

–O *UNIX output format*: Changes the output format of matching lines to support more easily the UNIX style of command-line piping. All lines of output are preceded by the name of the file which contained the matching line.

–R *Regular expression search*: The text defined by *searchstring* is treated as a regular expression instead of as a literal string.

–U *Update options*: GREP will combine the options given on the command line with its default options and write these to the GREP.COM file as the new defaults. (In other words, GREP is self-configuring.) This option allows you to tailor the default option settings to your own taste.

–V *Non-match*: Only non-matching lines are printed. Only lines that *do not* contain the search string are considered to be non-matching lines.

–W *Word search*: Text found which matches the regular expression will be considered a match only if the character immediately preceding and following cannot be part of a word. The default word character set includes *A-Z*, 9-0, and the underbar (_). An alternate form of this option allows you to specify the set of legal word characters. Its form is `-W[set]`, where `set` is any valid regular expression set definition. If alphabetic characters are used to define the set, the set will automatically be defined to contain both the upper and lower case values for each letter in the set, regardless of how it is typed, even if the search is case-sensitive. If the –*W* option is used in combination with the –*U* option, the new set of legal characters is saved as the default set.

–Z *Verbose*: GREP prints the file name of every file searched. Each matching line is preceded by its line number. A count of matching lines in each file is given, even if the count is zero.

Several of these options are in direct conflict with each other. In these cases, the following order applies (the first one is the one that takes precedence):

–*Z* –*L* –*C* –*N*

Each occurrence of an option overrides the previous definition: Its state reflects the way you last set it. At any given time, each option can only be on or off.

You can install your preferred default setting for each option in GREP.COM with the *-U* option. For example, if you want GREP to always do a verbose search (-Z on), you can install it with the following command:

```
GREP -U -Z
```

How to Search Using GREP

The value of *searchstring* defines the pattern GREP will search for. A search string can be either a (via the *-R* switch) or a literal string. In regular expressions, operators govern the search; literal strings have no operators.

You can enclose the search string in quotation marks to prevent spaces and tabs from being treated as delimiters. Matches will not cross line boundaries (a match must be contained in a single line).

When the *-R* switch is used, the search string is treated as a regular expression (as opposed to a literal expression), and the following symbols take on special meanings:

^ A caret at the start of the expression matches the start of a line.

$ A dollar sign at the end of the expression matches the end of a line.

. A period matches any character.

* An expression followed by an asterisk wildcard matches zero or more occurrences of that expression: *fo** matches *f*, *fo*, *foo*, etc.

+ An expression followed by a plus sign matches one or more occurrences of that expression: *fo+* matches *fo*, *foo*, etc., but not *f*.

[] A string enclosed in brackets matches any character in that string, but no others. If the first character in the string is a caret (^), the expression matches any character except the characters in the string. For example, *[xyz]* matches *x*, *y*, and *z*, while *[^xyz]* matches *a* and *b*, but not *x* or *y*. A range of characters can be specified by two characters separated by a hyphen (-). These can be combined to form expressions like *[?a-bd-z]* to match ? and any letter except *c*.

\ The backslash "escape character" tells GREP to seach for the literal character that follows it. For example, `\.` matches a period instead of any character.

Note: Four characters (?, +, *, and .) do not have any special meaning when used in a set. The character ^ is only treated specially if it immediately follows the beginning of the set (that is, immediately after the [).

Any ordinary character not mentioned in this list matches that character. A concatenation of regular expressions is a regular expression.

Examples Using GREP

The following examples assume all options default to off.

Search String `grep -n function dirdemo.pas`

Finds

```
File DIRDEMO.PAS:
46      LessFunc = function(X, Y: DirPtr): boolean;
55      function NumStr(N, D: integer): string;
68      function LessName(X, Y: DirPtr): boolean;
73      function LessSize(X, Y: DirPtr): boolean;
78      function LessTime(X, Y: DirPtr): boolean;
```

Remarks Finds all functions in the file DIRDEMO.PAS. The *-N* tells GREP to precede each matched line with its line number.

Search String `grep -n procedure dirdemo.pas`

Finds

```
File DIRDEMO.PAS:
85      procedure QuickSort(L, R: integer);
109     procedure GetCommand;
153     procedure FindFiles;
168     procedure SortFiles;
174     procedure PrintFiles;
```

Remarks Finds all procedures.

Search String `grep {\$ dirdemo.pas`

Finds

```
File DIRDEMO.PAS:
{$I-,S-}
{$M 8192,8192,655360}
{$F+}
{$F-}
```

Remarks Finds all compiler directives in DIRDEMO.PAS. The \ (backslash) preceding the $ is necessary. Without it, the $ would indicate the end of the line. All lines with { (curly bracket) as the last character would match this pattern and be printed out.

Search String **grep -n {\$IF *.pas**

Finds

```
File BGIDEMO.PAS:
74      {$IFDEF Use8514}        { check for Use8514 $DEFINE }
File HILB.PAS:
31      {$IFOPT N+}        { use extended type if using 80x87 }
File MCVARS.PAS:
10      {$IFOPT N+}
```

Remarks

Finds all compiler directives starting with *IF* in all files with the extension .PAS in the current directory. The \ (backslash) preceding the $ is necessary. If it were absent, the $ would be interpreted to mean end-of-line and all characters following it would be ignored. Thus the pattern *{$* would mean any line in which { is the last character.

Search String **grep -i "^ *function.*).*real" *.pas**

Finds

```
File MCPARSER.PAS:
function CellValue(Col, Row : word) : real;
function Parse(S : string; var Att : word) : real;
```

Remarks

Finds all lines that begin with zero or more spaces followed by the word **function**, followed by any string of zero or more characters, followed by a parenthesis, followed by another string of zero or more characters, followed by the word **real**, and ignores case. The net effect is to search for all functions returning a **real**. See if you can think of other ways to do this.

The double quotes are necessary because of the space in the pattern string. The quotes tell the DOS command-line processor to treat the intervening characters as a single argument. Without the quotes, DOS will think the search string is actually two arguments, and GREP will think that everything after ^ (the caret character) refers to file names, and will complain

```
No files matching: *FUNCTION.*).*.
```

Search String **grep \^ dirdemo.pas**

Finds

```
File DIRDEMO.PAS:
  DirPtr   = ^DirRec;
  LessName := X^.Name < Y^.Name;
```

```
LessSize := X^.Size < Y^.Size;
LessTime := X^.Time > Y^.Time;
  Move(F.Attr, Dir[Count]^, Length(F.Name) + 10);
with Dir[I]^ do
```

Remarks

Finds all pointers. The backslash is necessary. Without it the ^ (caret) would match the beginning of a line. GREP would print all lines in the file.

Search String `grep -IN count dirdemo.pas`

Finds

```
File DIRDEMO.PAS:
50        Count: integer;
122       for I := 1 to ParamCount do
157       Count := 0;
159       while (DosError = 0) and (Count < MaxDirSize) do
161         GetMem(Dir[Count], Length(F.Name) + 10);
162         Move(F.Attr, Dir[Count]^, Length(F.Name) + 10);
163         Inc(Count);
170       if (Count <> 0) and (@Less <> nil) then
171         QuickSort(0, Count - 1);
183       if Count = 0 then
189       for I := 0 to Count-1 do
228       if WideDir and (Count and 3 <> 0) then WriteLn;
229       WriteLn(Count, ' files, ', Total, ' bytes, ',
```

Remarks

Ignores case and prints line numbers while searching for *Count* in DIRDEMO.PAS.

Search String `grep : bgilink.mak`

Finds

```
File BGILINK.MAK:
bgilink.exe: drivers.tpu fonts.tpu
fonts.tpu: fonts.pas goth.obj litt.obj sans.obj trip.obj
goth.obj: goth.chr
litt.obj: litt.chr
sans.obj: sans.chr
trip.obj: trip.chr
drivers.tpu: drivers.pas cga.obj egavga.obj herc.obj
pc3270.obj att.obj
cga.obj: cga.bgi
egavga.obj: egavga.bgi
herc.obj: herc.bgi
pc3270.obj: pc3270.bgi
att.obj: att.bgi
```

Remarks Searches for all lines that contain colons in BGILINK.MAK. This finds all target-source dependencies in a make file.

Search String **grep := qsort.pas**

Finds

```
File QSORT.PAS:
  i:=1; j:=r; x:=a[(l+r) DIV 2];
    while a[i]<x do i:=i+1;
    while x<a[j] do j:=j-1;
      y:=a[i]; a[i]:=a[j]; a[j]:=y;
      i:=i+1; j:=j-1;
  for i:=1 to max do data[i]:=Random(30000);
  for i:=1 to 1000 do Write(data[i]:8);
```

Remarks Searches for all assignment statements in QSORT.PAS.

The BINOBJ Utility

A utility program called BINOBJ.EXE has been added to convert any file to an .OBJ file so it can be linked into a Turbo Pascal program as a "procedure." This is useful if you have a binary data file that must reside in the code segment or is too large to make into a typed constant array. For example, you can use BINOBJ with the *Graph* unit to link the graphics driver or font files directly into your .EXE file. Then, to use your graph program, you need only have the .EXE file (see the example BGILINK.PAS archived on Disk 2).

BINOBJ takes three parameters:

```
BINOBJ <source[.BIN]> <destination[.OBJ]> <public name>
```

where *source* is the binary file to convert, *destination* is the name of the .OBJ to be produced, and *public name* is the name of the procedure as it will be declared in your Turbo Pascal program.

The following example, the procedure *ShowScreen*, takes a pointer as a parameter and moves 4000 bytes of data to screen memory. The file called MENU.DTA contains the image of the main menu screen (80 * 25 * 2 = 4000 bytes).

Here's a simple (no error-checking) version of MYPROG.PAS:

```
program MyProg;

procedure ShowScreen(var ScreenData : pointer);
{ Display a screenful of data--no error-checking! }
```

```
var
  ScreenSegment: word;

begin
  if (Lo(LastMode) = 7) then                          { Mono? }
    ScreenSegment := $B000
  else
    ScreenSegment := $B800;
  Move(ScreenData^,                                   { From pointer }
       Ptr(ScreenSegment, 0)^,                        { To video memory }
       4000);                                         { 80 * 25 * 2 }
end;

var
  MenuP : pointer;
  MenuF : file;
begin
  Assign(MenuF, 'MENU.DTA');                          { Open screen data file }
  Reset(MenuF, 1);
  GetMem(MenuP, 4000);                                { Allocate buffer on heap }
  BlockRead(MenuF, MenuP^, 4000);                     { Read screen data }
  Close(MenuF);
  ShowScreen(MenuP);                                  { Display screen }
end.
```

The screen data file (MENU.DTA) is opened and then read into a buffer on the heap. Both MYPROG.EXE and MENU.DTA must be present at run-time for this program to work. You can use BINOBJ to convert MENU.DTA to an .OBJ file (MENUDTA.OBJ) and tell it to associate the data with a procedure called *MenuData*. Then you can declare the fake external procedure *MenuData*, which actually contains the screen data. Once you link in the .OBJ file with the *$L* compiler directive, *MenuData* will be 4000 bytes long and contain your screen data. First, run BINOBJ on MENU.DTA:

```
binobj MENU.DTA MENUDTA MenuData
```

The first parameter, MENU.DTA, shows a familiar file of screen data; the second, MENUDTA, is the name of the .OBJ file to be created (since you didn't specify an extension, .OBJ will be added). The last parameter, *MenuData*, is the name of the external procedure as it will be declared in your progam. Now that you've converted MENU.DTA to an .OBJ file, here's what the new MYPROG.PAS looks like:

```
program MyProg;

procedure ShowScreen(ScreenData : pointer);
{ Display a screenful of data--no error checking! }
var
  ScreenSegment: word;
begin
  if (Lo(LastMode) = 7) then                          { Mono? }
```

```
    ScreenSegment := $B000
  else
    ScreenSegment := $B800;
  Move(ScreenData^,                                    { From pointer }
       Ptr(ScreenSegment, 0)^,                         { To video memory }
       4000);                                          { 80 * 25 * 2 }
end;

procedure MenuData; external;
{$L MENUDTA.OBJ }
begin
  ShowScreen(@MenuData);                               { Display screen }
end.
```

Notice that *ShowScreen* didn't change at all, and that the ADDRESS of your procedure is passed using the @ operator.

The advantage of linking the screen data into the .EXE is apparent: You don't need any support files in order to run the program. In addition, you have the luxury of referring to your screen by name (*MenuData*). The disadvantages are

- Every time you modify the screen data file, you must reconvert it to an .OBJ file and recompile MYPROG.
- You have to have a separate .OBJ file (and **external** procedure) for each screen you want to display.

BINOBJ is especially useful when the binary file you wish to link in is fairly stable. One of the sample graphics programs uses BINOBJ to build two units that contain the driver and font files; refer to the extensive comment at the beginning of BGILINK.PAS on Disk 2.

A P P E N D I X

D

Customizing Turbo Pascal

This appendix explains how to customize Turbo Pascal and install your customizations in the TURBO.EXE file.

What Is TINST?

TINST is the Turbo Pascal installation program that you can use to customize TURBO.EXE (the integrated development environment version of Turbo Pascal). Through TINST, you can change various default settings in the Turbo Pascal operating environment, such as the screen size, editing commands, menu colors, and default directories. It directly modifies certain default values within your copy of TURBO.EXE.

With TINST, you can do any of the following:

- set up paths to the directories where your Include files, unit files, configuration files, help files, pick file, and executable files are located
- customize the editor command keys
- set up Turbo Pascal's editor defaults and onscreen appearance
- set up the default video display mode
- change screen colors
- resize Turbo Pascal's Edit and Output/Watch windows
- change the defaults of any of the settings accessible through the **Options/Compiler** menu or the **Options/Compiler/Memory Sizes** menu

- change the defaults of any of the settings accessible through the Options/Environment menu or the Options/Environment/Screen Size menu
- change the destination setting (IDE menu equivalent: Compile/Destination)
- choose default settings for the integrated debugger

Turbo Pascal comes ready to run; there is no installation per se. You can copy the files from the distribution disks to your working floppies (or hard disk) as described in Chapter 1, "Getting Started," then run Turbo Pascal.

You will also need to run TINST if you want to do any of the following:

- automatically load a configuration file (TURBO.TP) that does not reside in the current directory (see "The Directories Menu," page 306)
- change Turbo Pascal's default menu colors (see "The Set Colors Menu," page 316)
- force the display mode or snow-checking (see "The Mode for Display Menu," page 314)

Keeping Your Turbo Pascal 4.0 TINST Settings (TINSTXFR)

If you're upgrading from Turbo Pascal 4.0 to 5.0, you might want to preserve some of the features you customized in 4.0. Perhaps you configured Turbo Pascal 4.0 to preserve your directory settings for Include files, unit files, or configuration files. Other customizable features include editor command keys, screen colors, window sizes, compiler and environment options. Suppose you spent months customizing your Turbo Pascal editor keys and you want to avoid the agony of reentering all those keystrokes with TINST in 5.0. TINSTXFR lets you transfer your customized TINST settings to 5.0 intact.

Note: You would only want to use TINSTXFR if you left Turbo Pascal 4.0 on your hard disk after you installed Turbo Pascal 5.0. When you run the INSTALL program for Turbo Pascal 5.0 (as described in Chapter 1), installation option number two is

```
Update Turbo Pascal 4.0 to Turbo Pascal 5.0 on a Hard Drive
```

If you choose this option, TINSTXFR is done for you automatically, so you don't need to do it.

How to Use TINSTXFR

Let's suppose that you used INSTALL to copy your 5.0 files automatically to the default subdirectory C:/TP (see page 15 for details on INSTALL), and you want to move your TINST editor configuration there from Turbo Pascal 4.0, which you keep in a subdirectory called C:\TP4. Type

```
TINSTXFER \TP4\TURBO.EXE TURBO.EXE
```

(You must have TURBO.EXE for both Turbo Pascal 4.0 and 5.0.) The basic syntax is

```
TINSTXFR TP4_FILE TP5_FILE
```

where *TP4_FILE* is the full path name of TURBO.EXE for Turbo Pascal 4.0, and *TP5_FILE* is the full path name of TURBO.EXE for Turbo Pascal 5.0. The TINSTXFR process will modify your Turbo Pascal 5.0 TURBO.EXE file, so be sure to make a backup. TINSTXFR will warn you that the text block-marking functions *F7* and *F8* no longer exist: *F7* is now the hot key for the new **Run/Trace** Into command, and *F8* is now the hot key for the new **Run/Step** Over command. Other than this discrepancy, TINSTXFR will transfer your editor configuration and all other customized settings to Turbo Pascal 5.0 with no changes.

Running TINST

Normally, TINST comes up in color if it detects a color-type adapter in a color mode. You can override this default by using the */B* option, for instance, if you are using a composite monitor with a color-type adapter.

Here is the syntax for TINST:

```
TINST [option] [pathname]
```

Both *option* and *pathname* are optional. If you don't supply the *pathname,* TINST looks for TURBO.EXE in the current directory; otherwise it uses the given path name.

Note that you can use one version of TINST to customize several different copies of Turbo Pascal on your system. These various copies of TURBO.EXE can have different executable program names. Simply invoke TINST and give a (relative or absolute) path name to the copy of TURBO.EXE you're customizing; for example,

```
TINST c:\TURBOOO\TPOO.EXE
TINST ..\..\BWTP.EXE
TINST C:\BORLAND\COLORTP.EXE
```

In this way, you can customize the different copies of Turbo Pascal on your system to use different editor command keys, different menu colors, and so on, if you're so inclined.

Here's what the first screen of TINST will look like:

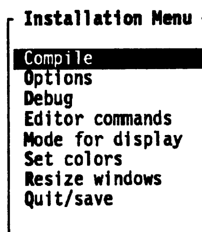


Figure D.1: The TINST Installation Menu

- The Compile command lets you specify a default name for the Primary File to be compiled and the Destination of the compile (Memory or Disk).
- The Options command gives you access to default settings for a great many features, including compiler, linker, and environment settings, path names to the directories, and parameters.
- The Debug command lets you set the Integrated Debugging, Stand-alone Debugging, and Display Swapping defaults in the integrated debugger.
- You can use Editor Commands to reconfigure (customize) the interactive editor's keystroke commands.
- With the Mode for Display command, you can specify the video display mode that Turbo Pascal will operate in, and tell TINST whether yours is a "snowy" video adapter.
- You can customize the colors of almost every part of Turbo Pascal's integrated environment with the Set Colors menu.
- The Resize Windows command allows you to change the sizes of the Edit and Output/Watch windows.
- The Quit/Save command lets you save the changes you have made to the integrated development environment, and returns you to system level.

To choose a menu command, just press the key for the highlighted capital letter of the command you want. For instance, press *S* to choose Set Colors. Or use the *Up* and *Down* arrow keys to move the highlight bar to the command you want, then press *Enter*.

Pressing *Esc* (more than once if necessary) returns you from a submenu to the main installation menu.

The Compile Menu

The Compile menu contains two commands: Destination and Primary File. You can set the Destination of the compile to either Memory or to Disk. If you choose Primary File, a prompt box will appear in which you can type the default name for the file.

The Options Menu

The Options menu lets you set up your environment any way you want with Compiler, Link, Environment, Directories, and Parameters settings; here's a brief description of each.

The Compiler Menu

Use this option to set up defaults for various features in the compiler.

Range-Checking (Off)

A toggle to enable or disable range-checking.

Stack-Checking (On)

A toggle to enable or disable stack-checking.

I/O-Checking (On)

A toggle to enable or disable input/output error-checking.

Force Far Calls (Off)

A toggle that lets you force procedures and functions to use the FAR call model. When disabled, the compiler automatically chooses NEAR or FAR appropriately.

Overlays Allowed (Off)

A toggle that specifies whether units can be overlaid.

Align Data (Word)

Let's you choose between byte or word alignment for variables and typed constants.

Var-String Checking (Strict)

Lets you choose between strict or relaxed string-checking.

Boolean Evaluation (Short Circuit)

Lets you choose between short circuit or complete evaluation.

Numeric Processing (Software)

Lets you choose between 8087/80287 and software.

Emulation (On)

Lets you use the IEEE floating-point types by linking in a library that emulates the 8087 or lets you call the 8087 directly if it is present.

Debug Information (On)

Enables or disables the generation of debug information. This information consists of a line-number table for each procedure that maps object code addresses into source text numbers for use with both Turbo Pascal and Turbo Debugger.

Local Symbols (On)

Toggle On or Off. Use this to control whether debug information is retained for formal parameters and local variables.

Conditional Defines

Brings up a prompt box in which you can define symbols.

Memory Sizes (16384, 0, 655360)

Lets you set the stack size, and minimum and maximum heap limits.

The Linker Menu

This menu lets you set defaults for the linker.

Map File (Off)

You can use this command to determine the default type for the map file. It can be set to **Off**, **Segments**, **Publics**, or **Detailed**.

Link Buffer (Memory)

You can use this command to set the link buffer to either Memory or Disk.

The Environment Menu

You can install several editor default modes of operation with this menu. The commands on the menu, and their significance, are described here.

First, take a look at the bottom status line for directions on how to choose these commands: Either use the arrow keys to move the highlight bar to the command and then press *Enter*, or press the key that corresponds to the highlighted capital letter of the command.

You can change the operating environment defaults to suit your preferences (and your monitor), then save them as part of Turbo Pascal. Of course, you'll still be able to change these settings from inside Turbo Pascal's editor.

Note: Any option you install with TINST that also appears as a menu command in TURBO.EXE (the IDE) will be overridden whenever you load a configuration file that contains a different setting for that option.

Config Auto Save (Off)

With **Config Auto Save** set to On, Turbo Pascal automatically saves the configuration file (if it's been modified since last saved) whenever you use a **Run** command (*Ctrl-F9*, *F7*, *F8*, or *F4*), **File/OS Shell**, or **File/Quit** (or *Alt X*). Which file it saves the current (recently modified) configuration to depends on three sets of factors.

Edit Auto Save (Off)

With **Edit Auto Save** set to On, Turbo Pascal automatically saves the file in the editor (if it's been modified since last saved) whenever you use **File/OS Shell** or a **Run** command like **Run** (*Ctrl-F9*) or **Step Over** (*F8*). This helps prevent the loss of your source files in the event of some calamity. With **Edit Auto Save** set to Off, no such automatic saving occurs.

Backup Files (On)

With **Backup Files** set to On, Turbo Pascal automatically creates a backup of your source file when you do a **File/Save**. It uses the same file name, and adds a .BAK extension: the backup file for FILENAME, FILENAME.C or FILENAME.XYZ would be FILENAME.BAK. With **Backup Files** set to Off, no .BAK file is created.

Zoom Windows (Off)

With **Zoom Windows** set to On, Turbo Pascal starts up with the Edit window occupying the full screen; when you switch to the Output window, it will also be full-screen. With Zoom Windows set to Off, the Edit window occupies the top portion of the screen, above the Output or Watch window. (You can resize the windows with the **Resize Windows** command from the main installation menu.)

Full Graphics Save (On)

To save graphics screens, Turbo Pascal reserves 8K of memory as a buffer for the palettes. If you're only using text screens, you make more memory available by turning this option off. You can only access this option through TINST.

Screen Size (25)

When you choose Screen Size, a menu pops up. The items in this menu allow you to set the Turbo Pascal integrated environment display to one of two sizes (25- or 43-/50-line). The available sizes depend on your hardware: 25-line mode is always available; 43-line/50-line mode is for systems with an EGA or VGA only.

Options for Editor

This command lets you set defaults for the editor:

Insert Mode (On)

With Insert Mode set to On, the editor inserts anything you enter from the keyboard at the cursor position, and pushes existing text to the right of the cursor even further right. Toggling Insert Mode to Off allows you to overwrite text at the cursor.

Autoindent Mode (On)

With Autoindent Mode set to On, the cursor returns to the starting column of the previous line when you press *Enter*. When Autoindent Mode is toggled to Off, the cursor always returns to column one.

Use Tabs (Off)

When you press the *Tab* key with Use Tabs set to On, the editor places a tab character (^I) in the text using the tab size specified with Tab Size. When you press the *Tab* key with Use Tabs set to Off, the editor inserts enough space characters to align the cursor with the first letter of each word in the previous line.

Optimal Fill (On)

Toggle On or Off. Optimal Fill mode has no effect unless Tab mode is also on. When both these modes are enabled, the beginning of every autoindented and unindented line is filled optimally with tabs and spaces. This produces lines with a minimum number of characters. The hot key for this command is *Ctrl-O F*.

Backspace Unindents (On)

Toggle On or Off. When it is On, this feature *outdents* the cursor; that is, it aligns the cursor with the first nonblank character in the first outdented line above the current or immediately preceding nonblank line. The hot key for this command is *Ctrl-O U*.

Tab Size (8)

When you choose this option, a prompt box appears in which you can enter the number of spaces you want to tab over at each tab command.

Editor Buffer Size (65,534)

If you normally write modular programs using small files, you can get extra memory for debugging by using a smaller editor buffer. You can set the editor buffer to any size between 20,000 and 65,534 bytes.

Make Use of EMS Memory (On)

The editor will normally use 64K of EMS memory for its text buffer if EMS is available. You can turn this Off, so Turbo will use no EMS.

Turbo Pascal's editor has expanded memory support on systems running EMS drivers conforming to the 3.2 (and above) Lotus/Intel/Microsoft Expanded Memory Specification (EMS). At startup, Turbo Pascal determines whether it can use EMS memory; if it can, it automatically places the editor's buffer into expanded memory. This frees up about 64K for compiling and running your programs. In the event that you don't want Turbo Pascal to use any available EMS memory that it finds, you can disable this feature using TINST.

The Directories Menu

With the Directories menu, you can specify a path to each of the TURBO.EXE default directories. These are the directories Turbo Pascal searches when looking for an alternate configuration file, the Help file, and the object, Include, and unit files, along with the directory where it will place your executable program.

When you choose Directories, TINST brings up a menu with the following items:

- Turbo Directory
- EXE & TPU Directory
- Include Directories
- Unit Directories
- Object Directories
- Pick File Name

Turbo Directory

Turbo Directory is where Turbo Pascal will look for the Help files, the default pick file, and TURBO.TP (the default configuration file) if they aren't located in the current directory.

For example, you could type the following path name at the Turbo Directory menu item:

```
C:\TURBO\CFGSNHLP
```

Then, if Turbo Pascal can't find the configuration and Help files in the current directory, it will look for them in the directory called TURBO\CFGSNHLP (off the root directory of the C drive).

EXE & TPU Directory

Use this option to name the default directory where the compiler will store the .EXE, .TPU, .MAP, and .OVR files it creates.

Include Directories, Unit Directories, and Object Directories

You can enter multiple directories in Include Directories, Unit Directories, and Object Directories. You must separate these "ganged" directory path names with a semicolon (;), and can enter a maximum of 127 characters for each path. You can enter absolute or relative directory names.

For example, if you have three directories of Include files, you could enter the following in the Include Directories pop-up input window:

```
C:\TURBO\INCLUDE;C:MYINCLD;A:..\..\INCLUDE2
```

If, in addition, you have divided your unit files among four different directories, and want Turbo Pascal to search each of those directories for units, you could enter the following in the Unit Directories pop-up input window:

```
C:\TURBO\STARTUPS;C:\TURBO\STDUNITS;C:..\MYUNITS2;A:NEWUNITS3
```

Turbo Pascal will also search the Object Directories in the same manner to find object files named in $L compiler directives.

Pick File Name

When you choose Pick File Name, an input window pops up. Type in the path name of the pick file you want Turbo Pascal to load or create. The default pick file name is TURBO.PCK.

After typing a path name (or names) for any of the Options/Directories menu commands, press *Enter* to accept, then press *Esc* to return to the main TINST installation menu. When you exit the program, TINST prompts whether you want to save the changes. Once you save the Turbo directory paths, the locations are written to disk and become part of TURBO.EXE's default settings.

The Parameters Setting

This setting lets you set default command-line arguments to pass to your running programs, exactly as if you had typed them on the DOS command line (redirection is not supported). It's only necessary to give the arguments here; the program name is omitted.

The Debug Menu

The items in the Debug menu let you set certain default settings for the Turbo Pascal integrated debugger.

Integrated Debugging (On)

This command controls integrated debugging in the IDE. Turning this switch to Off and compiling to disk leaves more room to run programs, if you are not debugging.

Stand-Alone Debugging (Off)

This controls whether debug information is placed in the .EXE file when you compile to disk. This information is needed if you want to use Turbo Debugger with your program.

Display Swapping (Smart)

This option allows you to set the default level of Display Swapping to Smart, Always, or Never.

When you run your program in debug mode with the default setting Smart, the Debugger looks at the code being executed to see whether the code will affect the screen (that is, output to the screen). If the code outputs to the screen, or if it calls a function, the screen is swapped from the Editor screen to the User screen long enough for output to take place, then is swapped back. Otherwise no swapping occurs. The Always setting causes the screen to be swapped every time a statement executes. The Never setting causes the debugger not to swap the screen at all.

The Editor Commands Menu

Turbo Pascal's interactive editor provides many editing functions, including commands for

- cursor movement
- text insertion and deletion
- block and file manipulation
- string search (plus search-and-replace)

These editing commands are assigned to certain keys or key combinations.

When you choose Editor Commands from TINST's main installation menu, the Install Editor screen comes up, displaying three columns of text:

- The left-hand column describes all the functions available in Turbo Pascal's interactive editor.
- The middle column lists the *Primary* keystrokes; what keys or special key combinations you press to invoke a particular editor command.
- The right-hand column lists the *Secondary* keystrokes; these are optional alternate keystrokes you can also press to invoke the same editor command.

Note: Secondary keystrokes always take precedence over primary keystrokes.

The bottom lines of text in the Install Editor screen summarize the keys you use to choose entries in the Primary and Secondary columns (see Table D.1).

Table D.1: Editor Screen Keystrokes

Key	Legend	What It Does
Left, Right, Up, and *Down arrow keys*	Select	Chooses the editor command you want to re-key
PgUp and *PgDn*	Page	Scrolls up or down one full screen page
Enter	Modify	Enters the keystroke-modifying mode
R	Restore factory defaults	Resets all editor commands to the factory default keystrokes
Esc	Exit	Leaves the Install Editor screen and returns to the main TINST installation menu
F4	Key Modes	Toggles between the three flavors of keystroke combinations

After you press *Enter* to enter the modify mode, a pop-up window appears on screen, listing the currently defined keystrokes for the chosen command (see Table D.2). The bottom lines of text in the Install Editor screen summarize the keys you use to change those keystrokes.

Table D.2: Editor Screen Keystrokes in Modify Mode

Key	Legend	What It Does
Backspace	Backspace	Deletes keystroke to left of cursor
Enter	Accept	Accepts newly defined keystrokes for chosen editor command
Esc	Abandon changes	Abandons changes to current choice, restoring the command's original keystrokes, and returns to the Install Editor screen (ready to choose another editor command)
F2	Restore	Abandons changes to current choice, restoring the command's original keystrokes, but keeps the current command chosen for redefinition
F3	Clear	Clears the current choice's keystroke definition, but keeps the current command chosen for redefinition
F4	Key Modes	Toggles between the three flavors of keystroke combinations: WordStar-like, Ignore Case, and Verbatim

Note: To enter the keys *F2, F3, F4,* or the backquote (`) character as part of an editor command key sequence, hold down the backquote key and press the appropriate function key.

Keystroke combinations come in three flavors: WordStar-like, Ignore Case, and Verbatim. These are listed on the bottom line of the screen; the highlighted one is the current choice.

WordStar-Like Selection

All commands must begin with a special key or a control key. Subsequent characters can be any key.

If you type a letter (or one of these five characters: [,], \, ^, –) in this mode, it will automatically be entered as a control-character combination. For example,

- typing *a* or *A* or *Ctrl-A* will yield *< Ctrl A >*
- typing *y* or *Y* or *Ctrl-y* will yield *< Ctrl Y >*
- typing *[* will yield *<Ctrl [>*

In the Turbo Pascal editor, you must then explicitly press the special key or the *Ctrl* key to enter the first keystroke of a command-key sequence, but for

the subsequent keystrokes of that command you can use a lowercase, uppercase, or control key.

For example, if you customize an editor command to be < *Ctrl A* > < *Ctrl B* > < *Ctrl C* > in WordStar-like mode, you can type any of the following in the Turbo Pascal editor to activate that command:

- < *Ctrl A* > < *Ctrl B* > < *Ctrl C* >
- < *Ctrl A* > < *Ctrl B* > < *C* >
- < *Ctrl A* > < *Ctrl B* > < *c* >
- < *Ctrl A* > < *B* > < *Ctrl C* >
- < *Ctrl A* > < *B* > < *C* >
- < *Ctrl A* > < *B* > < *c* >
- < *Ctrl A* > < *b* > < *Ctrl C* >
- < *Ctrl A* > < *b* > < *C* >
- < *Ctrl A* > < *b* > < *c* >

In WordStar-like keystrokes, any letter you type is converted to a control-uppercase-letter combination. Five other characters are also converted to control-character combinations:

- left square bracket ([)
- right square bracket (])
- backslash (\)
- caret (^, also known as *Shift 6*)
- minus (-)

Ignore Case Selection

In Ignore Case keystrokes, the only character conversions are from lowercase to uppercase (letters only). All commands must begin with a special key or a control key. Subsequent characters can be any key. In this mode all alpha (letter) keys you enter are converted to their uppercase equivalents. When you type a letter in this mode, it is *not* automatically entered as a control-character combination; if a keystroke is to be a control-letter combination, you must hold down the *Ctrl* key while typing the letter. For example:

- typing *a* or *A* will yield *A* (if this is the first keystroke, you'll get an error message)
- typing *Ctrl y* or *Ctrl Y* will yield < *Ctrl Y* >
- typing *Ctrl [* will yield < *Ctrl [* >

In Ignore Case keystrokes, the only character conversions are from lowercase to uppercase (letters only).

Verbatim Selection

These keystrokes must always begin explicitly with a character that is a special key or control key. If you type a letter in this mode, it will be entered exactly as you type it.

- typing *a* will yield *a* (if this is the first keystroke, you'll get an error message)
- typing *A* will yield *A* (if this is the first keystroke, you'll get an error message)
- typing *Ctrl Y* will yield *< Ctrl Y >*
- typing *Ctrl y* will yield *< Ctrl y >*
- typing *Ctrl [* will yield *< Ctrl [>*

In Verbatim keystrokes, what you enter in the Install Editor screen for a command's keystroke sequence is exactly what you must type in the Turbo Pascal editor when you want to invoke that command. If, for example, you enter *< Ctrl A > b* and *< Crtl H > B* as the Verbatim primary and secondary keystroke sequences for some editor command, you will only be able to type those exact key characters to invoke the command. Using the same letters but in different cases—*< Ctrl A > B* and *< Ctrl H > b*—won't work.

Allowed Keystrokes

Although TINST provides you with lots of flexibility for customizing the Turbo Pascal editor commands to your own taste, there are a few rules governing the keystroke sequences you can define. Some of the rules apply to any keystroke definition, while others only come into effect in certain keystroke modes.

Global Rules

1. You can enter a maximum of six keystrokes for any given editor command. Certain key combinations are equivalent to two keystrokes, such as *Alt* (*any valid key*), the cursor-movement keys (*Up arrow, PgDn, Del,* and so on) and all function keys or function key-combinations (*F4, Shift-F7, Alt-F8,* and so on).
2. The first keystroke must be a character that is non-alphanumeric, non-punctuation; in other words, it must be a control key or a special key.
3. To enter the *Esc* key as a command keystroke, type *Ctrl-[*.
4. To enter the *Backspace* key as a command keystroke, type *Ctrl-H*.

5. To enter the *Enter* key as a command keystroke, type *Ctrl-M*.
6. The Turbo Pascal predefined Help function keys (*F1* and *Alt F2*) can't be reassigned as Turbo Pascal editor command keys. Any other function key can, however. If you enter a hot key as part of an editor command key sequence, TINST will issue a warning that you are overriding a hot key in the editor and will verify whether you want to override that key.

Turbo Pascal Editor Keystrokes

Command name	Primary	Secondary
New Line	* *Ctrl-M*	• *Ctrl-M*
Cursor Left	* *Ctrl-S*	• *Lft*
Cursor Right	* *Ctrl-D*	• *Rgt*
Word Left	* *Ctrl-A*	• *Ctrl-Lft*
Word Right	* *Ctrl-F*	• *Ctrl-Rgt*
Cursor Up	* *Ctrl-E*	• *Up*
Cursor Down	* *Ctrl-X*	• *Dn*
Scroll Up	* *Ctrl-W*	•
Scroll Down	* *Ctrl-Z*	•
Page Up	* *Ctrl-R*	• *PgUp*
Page Down	* *Ctrl-C*	• *PgDn*
Left of Line	* *Ctrl-Q Ctrl-S*	• *Home*
Right of Line	* *Ctrl-Q Ctrl-D*	• *End*
Top of Screen	* *Ctrl-Q Ctrl-E*	• *Ctrl-Home*
Bottom of Screen	* *Ctrl-Q Ctrl-X*	• *Ctrl-End*
Top of File	* *Ctrl-Q Ctrl-R*	• *Ctrl-PgUp*
Bottom of File	* *Ctrl-Q Ctrl-C*	• *Ctrl-PgDn*
Move to Error Pos	* *Ctrl-Q Ctrl-W*	•
Move to Block Begin	* *Ctrl-Q Ctrl-B*	•
Move to Block End	* *Ctrl-Q Ctrl-K*	•
Move to Previous Pos	* *Ctrl-Q Ctrl-P*	•
Move to Marker 0	* *Ctrl-Q0*	•
Move to Marker 1	* *Ctrl-Q1*	•
Move to Marker 2	* *Ctrl-Q2*	•
Move to Marker 3	* *Ctrl-Q3*	•
Toggle Insert	* *Ctrl-V*	• *Ins*
Insert Line	* *Ctrl-N*	•
Delete Line	* *Ctrl-Y*	•
Delete to End of Line	* *Ctrl-Q Ctrl-Y*	•
Delete Word	* *Ctrl-T*	•
Delete Char	* *Ctrl-G*	• *Del*
Delete Char Left	* *Ctrl-BkSp*	• *Ctrl-H*
Set Block Begin	* *Ctrl-K Ctrl-B*	•
Set Block End	* *Ctrl-K Ctrl-K*	•

Command name	Primary	Secondary
Mark Word	* *Ctrl-K Ctrl-T*	•
Hide Block	* *Ctrl-K Ctrl-H*	•
Set Marker 0	* *Ctrl-K0*	•
Set Marker 1	* *Ctrl-K1*	•
Set Marker 2	* *Ctrl-K2*	•
Set Marker 3	* *Ctrl-K3*	•
Copy Block	* *Ctrl-K Ctrl-C*	•
Move Block	* *Ctrl-K Ctrl-V*	•
Delete Block	* *Ctrl-K Ctrl-Y*	•
Read Block	* *Ctrl-K Ctrl-R*	•
Write Block	* *Ctrl-K Ctrl-W*	•
Print Block	* *Ctrl-K Ctrl-P*	•
Exit Editor	* *Ctrl-K Ctrl-D*	• *Ctrl-KCtrl-Q*
Tab	* *Ctrl-I*	•
Toggle Autoindent	* *Ctrl-O Ctrl-I*	• *Ctrl-QCtrl-I*
Toggle Tabs	* *Ctrl-O Ctrl-T*	• *Ctrl-QCtrl-T*
Restore Line	* *Ctrl-Q Ctrl-L*	•
Find String	* *Ctrl-Q Ctrl-F*	•
Find and Replace	* *Ctrl-Q Ctrl-A*	•
Search Again	* *Ctrl-L*	•
Insert Control Char	* *Ctrl-P*	•
Save file	* *Ctrl-K Ctrl-S*	•
Match pair	* *Ctrl-Q Ctrl[*	•
Match pair backward	* *Ctrl-Q Ctrl]*	•
Language help	* *Ctrl-F1*	•
Insert options	* *Ctrl-O Ctrl-O*	•
Toggle optimal fill	* *Ctrl-O Ctrl-F*	•
Toggle unindent	* *Ctrl-O Ctrl-U*	•
Block indent	* *Ctrl-K Ctrl-I*	•
Block unindent	* *Ctrl-K Ctrl-U*	•

The Mode for Display Menu

Normally, Turbo Pascal will correctly detect your system's video mode. You should only change the Mode for Display option if

- you want to choose a mode other than the current video mode
- you have a Color/Graphics Adapter that doesn't "snow"
- you think Turbo Pascal is incorrectly detecting your hardware
- your system has a composite screen, which acts like a CGA with only one color—for this situation, choose Black and White
- you have a portable or laptop with an LCD screen and you don't know what to choose.

Press *M* to choose Mode for Display from the installation menu. From the Mode for Display menu, you can choose the screen mode Turbo Pascal will use during operation. Your options include

- Default
- Color
- Black and White
- LCD or Composite
- Monochrome

When you choose one of the first three options, the program conducts a video test on your screen; look at the bottom status line for instructions about what to do.

When you press any key, a window comes up with the query

```
Was there Snow on the screen?
```

You can choose

- Yes, the screen was "snowy"
- No, always turn off snow-checking
- Maybe, always check the hardware

Default

By default, Turbo Pascal always operates in the mode that is active when you load it, if a color adapter is detected.

Color

Turbo Pascal uses 80-column color mode if a color adapter is detected, no matter what mode is active when you load TURBO.EXE; it switches back to the previously active mode when you exit.

Black and White

Turbo Pascal uses 80-column black and white mode if a color adapter is detected, no matter what mode is active; it switches back to the previously active mode when you exit.

LCD or Composite

Turbo Pascal uses 80-column black and white mode if a color adapter is detected, no matter what mode is active; it switches back to the previously active mode when you exit.

If you are using a laptop computer and have difficulty reading text displayed in the integrated development environment, choose LCD or Composite.

Monochrome

Turbo Pascal uses monochrome mode if a monochrome adapter is detected, no matter what mode is active; it switches back to the previously active mode when you exit.

Look to the status line for more about Maybe. Press *Esc* to return to the main installation menu.

The Set Colors Menu

Pressing *S* from the main installation menu allows you to make extensive changes to the colors of your version of Turbo Pascal. After pressing *S*, you will see a menu with these options:

- Customize Colors
- Default Color Set
- Turquoise Color Set
- Version Color Set

Because there are nearly 50 different screen items that you can color-customize, you will probably find it easier to choose a *preset* set of colors. Three preset color sets are on disk.

Press *D*, *T*, or *V*, and scroll through the colors for the Turbo Pascal screen items using the *PgUp* and *PgDn* keys. If you don't like any of the preset color sets, you can design your own.

To make custom colors, press *C* to choose Customize Colors. Now you have a choice of 12 items that can be color-customized in Turbo Pascal; some of these are text items, some are screen lines and boxes. Choose one of these items by pressing a letter *A* through *L*.

Once you choose a screen item to color-customize, you will see a pop-up menu and a viewport. The viewport is an example of the screen item you chose, while the pop-up menu displays the components of that choice. The viewport also reflects the change in colors as you scroll through the color palette.

For example, if you chose **H** to customize the colors of Turbo Pascal's error boxes, you would see a new pop-up menu with the four different parts of an error box: **T**itle, **B**order, **N**ormal Text, and **I**nverse Text.

You must now choose one of the components from the pop-up menu. Type the appropriate highlighted letter, and you're treated to a color palette for the item you chose. Using the arrow keys, choose a color to your liking from the palette. Watch the viewport to see how that item looks in that color. Press *Enter* to record your choice.

Repeat this procedure for every screen item you want to color-customize. When you are finished, press *Esc* until you are back at the main installation menu.

Note: Turbo Pascal maintains three internal color tables: color, black and white, and monochrome. TINST only allows you to change the color table based on the setting of the **M**ode for Display command. For example, if you want to change the black and white color table, you would set **M**ode for Display to **B**lack and White, and then set the attributes for black and white mode.

The Resize Windows Menu

This option allows you to change the respective sizes of Turbo Pascal's Edit and Output/Watch windows. Press *R* to choose **R**esize Windows from the main installation menu.

Using the *Up* and *Down arrow* keys, you can move the bar dividing the Edit window from the Output/Watch window. When you have resized the windows to your liking, press *Enter*. You can discard your changes and return to the main installation menu by pressing *Esc*.

Note: If you are running Turbo Pascal in 43-/50-line mode, the ratio of the lines in 25-line mode will be used.

Quitting the Program

Once you have finished making all desired changes, choose Quit/Save at the main installation menu. The message

```
Save changes to TURBO.EXE? (Y/N)
```

will appear at the bottom of the screen.

- If you press *Y* (for Yes), all the changes you have made will be permanently installed in Turbo Pascal. (Of course, you can always run TINST again if you want to change them.)
- If you press *N* (for No), your changes will be ignored and you will be returned to the operating system prompt without changing Turbo Pascal's defaults or startup appearance. If you press *Esc*, you'll be returned to the menu.

If you decide you want to restore the original Turbo Pascal factory defaults, simply copy TURBO.EXE from your master disk onto your work disk. You can also restore the original editor commands by choosing the Editor Commands from the main menu, then pressing *R* (for Restore Factory Defaults) and *Esc*.

A P P E N D I X

E

A DOS Primer

If you are new to computers or to DOS, you may have trouble understanding certain terms used in this manual. This appendix provides you with a brief overview of the following DOS concepts and functions:

- what DOS is and does
- the proper way to load a program
- directories, subdirectories, and the path command
- using AUTOEXEC.BAT files

This information is by no means a complete explanation of the DOS operating system. If you need more details, please refer to the MS-DOS or PC-DOS user's manual that came with your computer system.

Turbo Pascal runs under the MS-DOS or PC-DOS operating system, version 2.0 or later.

What Is DOS?

DOS is shorthand for Disk Operating System. MS-DOS is Microsoft's version of DOS, while PC-DOS is IBM's. DOS is the traffic coordinator, manager, and operator for the transactions that occur between the parts of the computer system and the computer system and you. DOS operates in the background, taking care of may of the menial computer tasks you wouldn't want to have to think about—for instance, the flow of characters between your keyboard and the computer, between the computer and your printer, and between your disk(s) and internal memory (RAM).

Other transactions are initiated by entering commands on the DOS command line; in other words, immediately after the DOS prompt. Your DOS prompt probably looks like one of the following:

```
A>
B>
C>
```

The capital letter refers to the active disk drive (the one DOS and you are using right now). For instance, if the prompt is *A>*, it means you are working with the files on drive *A*, and that commands you give DOS will refer to that drive. When you want to switch to another disk, making it the active disk, all you do is type the letter of the disk, followed by a colon and press *Enter*. For instance, to switch to drive *B*, just type

`B:` *Enter*

There are a few commands you will use often with DOS, if you haven't already, such as

DEL or ERASE	To erase a file
DIR	To see a list of files on the logged disk
COPY	To copy files from one disk to another
TURBO	To load Turbo Pascal

DOS doesn't care whether you type in uppercase or lowercase letters, or a combination of both, so you can enter your commands however you like.

We'll assume you know how to use the first three commands listed; if you don't, refer to your DOS manual. We will explain the proper way to load a program like Turbo Pascal, which is the last command—*TURBO*.

How to Load a Program

On your distribution disk, you'll find the main Turbo Pascal program under the file name TURBO.EXE. This program file performs all Turbo Pascal operations, so you always need it when you start the program. A file name with the extension .COM or .EXE is a program file you can load and run (use, start) by typing its file name at the DOS prompt. To start Turbo Pascal, you simply type `TURBO` and press *Enter*, and Turbo Pascal will be loaded into your computer's memory.

There's one thing you need to remember about loading Turbo Pascal and other similar programs: *You must be logged onto the disk and directory where*

the program is located in order to load it; otherwise, unless you have set up a DOS path (described shortly), DOS won't know where to find the program.

For instance, if your distribution disk with the TURBO.EXE program is in drive *A* but the prompt you see on your screen is *B>*, DOS won't know what you're talking about if you type `TURBO` and press *Enter*. Instead of starting Turbo Pascal, it will give you the message `Bad command or file name`.

It's as if you were shuffling through the pet records file in your file cabinet looking for information about your home finances. You're in the wrong place. So if you happen to get that DOS message, check to make sure you are in the right drive and directory before you type `TURBO` and press *Enter* to load Turbo Pascal.

You can set up a path to the Turbo Pascal files, so that DOS can find them, with the DOS *path* command. See the section titled "The AUTOEXEC.BAT File" for more information.

Directories

A *directory* is a convenient way to organize your floppy or hard disk files. Directories allow you to subdivide your disk into sections, much the way you might put groups of manila file folders into separate file boxes. For instance, you might want to put all your file folders having to do with finance—a bank statement file, an income tax file, or the like—into a box labeled "Finances."

On your computer, it would be convenient to make a directory to hold all your Turbo Pascal files, another for your SideKick files, another for your letters, and so on. That way, when you type `DIR` on the DOS command line, you don't have to wade through hundreds of file names looking for the file you want. You'll get a listing of only the files on the directory you're currently logged into.

Although you can make directories on either floppy or hard disks, they are used most often on hard disks. Because hard disks can hold a greater volume of data, there is a greater need for organization and compartmentalization.

When you're at the DOS level, rather than in Turbo Pascal or another program, you can tell DOS to create directories, move files around between directories, and display which files are in a particular directory.

In the examples that follow, we assume you are using a hard disk system, and that you are logged onto the hard disk so that the prompt you see on

your screen is *C>*. If you want to create directories on your floppy disks, substitute *A* or *B* for *C* in the example.

To make a directory for your Turbo Pascal files, do the following:

1. At the *C>* prompt, type `MD TURBO` and press *Enter*. The MD (Make Directory) command tells DOS to make a directory called TURBO.
2. Type `CD TURBO` and press *Enter*. The CD (Change Directory) command tells DOS to move you into the TURBO directory.
3. Now, put the Turbo Pascal disk you want to copy *from* into one of your floppy drives—let's say *A* for this example—and type `COPY A:*.*` *Enter*. (The asterisks are *wildcards* that stand for all files and extensions.) The COPY command tells DOS to copy all files on the *A* drive to the TURBO directory on the *C* drive. As each file on the disk is copied, you will see it listed on the screen.

That's all there is to it. Treat a directory the same way you would a disk drive: To load Turbo Pascal, you must be in the TURBO directory before typing `TURBO` and pressing *Enter*, or DOS won't be able to find the program.

Subdirectories

If you really like organization, you can divide your directories into subdirectories. You can create as many directories and subdirectories as you like—just don't forget where you put your files!

A subdirectory is created the same way as a directory. To create a subdirectory from the TURBO directory (for instance, for storing your unit files), do the following:

1. Be sure you are in the TURBO directory.
2. Type `MD UNITS` *Enter*.
3. Type `CD UNITS`. You are now in the UNITS subdirectory.
4. Copy your unit files to the new subdirectory.

Where Am I? The $p $g Prompt

You've probably noticed when you change directories that you still see the C> prompt; there is no indication of the directory or subdirectory you are currently in. This can be confusing, especially if you leave your computer for a while. It's easy to forget where you were when you left.

DOS gives you an easy way to find out. Just type

```
PROMPT=$P $G
```

and from now on (until you turn your computer off or reboot), the prompt will show you exactly where you are. Try it. If you are still in the UNITS subdirectory, your DOS prompt will look like

```
C:\TURBO\UNITS >
```

The AUTOEXEC.BAT File

To avoid typing the prompt command (discussed in the previous section) to see where you are every time you turn on your computer, you can set up an AUTOEXEC.BAT file to do it for you.

The AUTOEXEC.BAT file is a useful tool that sets your computer to do things automatically when it starts up. There are many more things it can do, but rather than go into great detail here, we suggest referring to your DOS manual for more information. We will show you how to create an AUTOEXEC.BAT file that will automatically change your prompt so you know where you are in your directory structure, set a *path* to the TURBO directory, and then load Turbo Pascal.

The DOS *path* command tells your computer where to look for commands it doesn't recognize. DOS recognizes only the programs in the current (logged) directory, unless there is a path to the directory containing pertinent programs or files.

In the following example, we will set a path to the TURBO directory.

If you have an AUTOEXEC.BAT file in your root (main) directory, your computer will execute every command in that file when you first turn your computer on. (The root directory is where you see the C> or C:\ prompt, with no directory names following it.)

Here's how to create an AUTOEXEC.BAT file.

1. Type `CD \` to get to the root directory.
2. Type `COPY CON AUTOEXEC.BAT` *Enter.* This tells DOS to copy whatever you type next into a file called AUTOEXEC.BAT.
3. Type

   ```
   PROMPT=$P $G Enter
   PATH=C:\TURBO
   CD TURBO
   Ctrl-Z Enter
   ```

The *Ctrl-Z* sequence saves your commands in the AUTOEXEC.BAT file.

To test your new AUTOEXEC.BAT file, reboot your computer by holding down the *Ctrl* and *Alt* keys and then pressing *Del*. You should see C:\ TURBO>.

Changing Directories

How do you get from one directory to another? It depends on where you want to go. The basic DOS command for changing directories is CD. Use it like this:

- **To move from one directory to another**: For example, to change from the TURBO directory to one called SPRINT, type the following from the TURBO directory:

  `C:\TURBO> CD \SPRINT` *Enter*

  Notice the backslash (\) before the directory name. Whenever you are moving from one directory to another unrelated directory, type the name of the directory, preceded by a backslash.

- **To move from a directory to its subdirectory**: For example, to move from the TURBO directory to the UNITS subdirectory, type the following from the TP directory:

  `C:\TP> CD UNITS` *Enter*

  In this case, you did not need the backslash, because the UNITS directory is a direct offshoot of the TP directory. In fact, DOS would have misunderstood what you meant if you had used the backslash—DOS would have thought that UNITS was a directory off the main (root) directory.

- **To move from a subdirectory to its parent directory**: For example, to move from the UNITS subdirectory to the TP directory, type the following from the UNITS subdirectory:

  `C:\TP\UNITS> CD ..` *Enter*

  DOS will move you back to the TP directory. Anytime you want to move back to the parent directory, use a space followed by two periods after the CD command.

- **To move to the root directory**: The *root directory* is the original directory. It is the parent (or grandparent) of all directories. When you are in the root directory, you'll see this prompt: C:\ >.

To move to the root directory from any other directory, simply type

CD \ *Enter*

The backslash without a directory name signals DOS that you want to return to the root directory.

This appendix has presented only a quick look at DOS and some of its functions. Once you're familiar with the information given here, you may want to study your DOS manual and discover all the other things you can do with your computer's operating system. There are many DOS functions not mentioned here that can simplify and enhance your computer use.

A P P E N D I X

F

Glossary

absolute variable A variable declared to exist at a fixed location in memory rather than letting the compiler determine its location.

actual parameter A variable, expression, or constant that is substituted for a formal parameter in a procedure or function call.

address A specific location in memory.

algorithm A set of rules that defines the solution to a problem.

allocate To reserve memory space for a particular purpose, usually from the heap.

ANSI The acronym for the the American National Standards Institute, the organization that, among other things, describes the elements of so-called standard Pascal.

argument An alternative name for a parameter (see actual parameter).

array A sequential group of data elements of the same type that are arranged in a single data structure and are accessible by an index.

ASCII character set The American Standard Code for Information Interchange's standard set of numbers to represent the characters and control signals used by computers.

assembler A program that converts assembly-language programs into machine language (like TASM, the Turbo Assembler).

assembly language The first language level above machine language. Assembly language is specific to the microprocessor it is running on. The major difference between assembly language and machine language is that

assembly language uses mnemonics instead of opcodes, making it easier to read and write.

assignment operator The symbol :=, which transfers a value to a variable or function of the same type.

assignment statement A statement that assigns a specific value to an identifier.

base type The type of values in an array.

binary Base 2; a method of representing numbers using only two digits, 0 and 1.

bit A binary digit with a value of either 0 or 1. The smallest unit of data in a computer.

block The associated declaration and statement parts of a program or subprogram.

body The instructions pertaining to a program or a subprogram (a procedure or function).

boolean A data type that can have a value of True or False.

braces The characters { and }, used to delimit comments; sometimes called curly brackets.

brackets The characters [and]; sometimes called square brackets.

buffer An area of memory allocated as temporary storage.

bug An error in a program. Syntax errors refer to incorrect use of the rules of the programming language; logic errors refer to incorrect strategy in the program to accomplish the intended result.

build The process of recompiling all the units used by a program.

byte A sequence of 8 bits.

call To cause a subprogram (procedure or function) to execute by referring to its name.

case label A constant, or list of constants, that label a component statement in a **case** statement.

case selector An expression whose result is used to select which component statement of a **case** statement will be executed.

central processing unit (CPU) The "brain" of a computer system, which interprets and executes instructions and controls the other components of the system.

chaining The passing of control from one program to another.

char A Pascal type that represents a single character.

code Instructions to a computer. Code is made up of algorithms.

code segment A portion of a compiled program up to 32767 bytes in length.

comment An explanatory statement in the source code enclosed by the symbols (* *) or { }.

compiler A program that translates a program written in a high-level language into machine language.

compiler directive An instruction to the compiler that is embedded within the program; for example, {*$R+*} turns on range-checking.

compound statement A series of statements surrounded by a matching set of the reserved words **begin** and **end**.

concatenation The joining of two or more strings.

constant A fixed value in a program.

control character A special nonprinting character in the ASCII character set designed originally to control a printing device or communications link.

control structure A statement that manages the flow of execution of a program.

crash A sudden computer failure due to a hardware problem or program error.

data segment The segment in memory where the static global variables are stored.

data structures Areas of related items in memory, represented as arrays, records, or linked lists.

debugger A special program that provides capabilities to start and stop execution of a program at will, as well as analyze values that the program is manipulating. Turbo Pascal has a debugger integrated into its development environment. In addition, it provides full support for the stand-alone Turbo Debugger.

debugging The process of finding and removing bugs from programs.

decimal A method of representing numbers using base 10 notation, where legal digits range from 0 to 9.

declare To define explicitly the name and type of an identifier in a program.

definition part The part of a program where constants, labels, and structured types are defined.

delimiter A boundary marker; it can be a word, a character, or a symbol.

dereferencing The act of accessing a value pointed to by a pointer variable (rather than the pointer variable itself).

directory A work area on a disk or a listing of files (or directories) on a disk.

documentation A written explanation of a computer program. Documentation can vary from manuals hundreds of pages long to a one-line comment embedded in the program itself.

dynamic Changing while the program is running, as dynamic memory allocation.

dynamic allocation The allocation and deallocation of memory from the heap at run time.

dynamic variable A variable on the heap.

element One of the items in an array.

enumerated type A user-defined scalar type that consists of an arbitrary list of identifiers.

evaluate To compute the value of an expression.

execute To carry out the program's instructions.

expression Part of a statement that represents a value or can be used to calculate a value.

extension Any addition to the standard definition of a language. Also, the optional three-character ending (following the period) in a standard DOS file name.

external A file of one or more subprograms that have been written in assembly language and assembled to native executable code.

field list The field name and type definition of a record.

field width The number of place holders in an output statement.

file A collection of data that can be stored on and retrieved from a disk.

file pointer A pointer that tracks where the next object will be retrieved from within a file.

file variable An identifier in a program that represents a file.

fixed-point notation The representation of real numbers without decimal points.

flag A variable, usually of type integer or boolean, that changes value to indicate that an event has taken place.

floating-point notation The representation of real numbers using decimal points.

formal parameter An identifier in a procedure or function declaration heading that represents the arguments that will be passed to the subprogram when it is called.

forward declaration The declaration of a procedure or function and its parameters in advance of the actual definition of the subroutine.

function A subroutine that computes and returns a value.

global variable A variable declared in the main program block that can be accessed from anywhere within the program.

heap An area of memory reserved for the dynamic allocation of variables.

hexadecimal A method of representing numbers using base 16 notation, where legal digits range from 0 to 9 and *A* to *F*.

high-level language A programming language that more closely resembles human language than machine language. Pascal is a high-level language.

I/O Short for Input/Output. The process of receiving or sending data.

I/O error An error that occurs while the program is trying to input or output data.

I/O redirection The DOS ability to direct input/output to access devices other than the default DOS devices.

identifier A user-defined name for a specific item (a constant, type, variable, procedure, function, unit, program, and field). It must begin with a letter and cannot contain spaces.

implementation The particular embodiment of a programming language. Turbo Pascal is an implementation of standard Pascal for IBM-compatible computers.

increment To increase the value of a variable.

index A position within an array of elements.

index type The type of indexes in an array.

initialize The process of giving a known initial value to a variable or data structure.

input The information a program receives from some external device, such as a keyboard.

integer A numeric data type, a whole number from -32768 to +32767.

interactive A program that communicates with a user through some I/O device.

interpreter A program that sequentially converts each statement in a program into machine code and then immediately executes it.

interrupt The temporary halting of program execution in order to process an event of higher priority.

iteration The process of repetition or looping.

keyword A reserved word in Pascal. In this manual, keywords are shown in boldface type (for example, **begin**, **end**, **nil**).

label An identifier that marks a place in the program text for a **goto** statement. Labels have digit sequences whose values range from 0 to 9999.

level The depth of nesting procedures or control structures.

linked list A dynamic data structure made up of elements, each of which points to the next element in the list through a pointer variable.

literal An unnamed constant in a program.

local identifier An identifier declared within a procedure or a function.

local variable A variable declared within a procedure or a function.

long word A location in memory occupying 4 adjacent bytes; the storage required for a variable of type longint.

loop A set of statements that are executed repeatedly.

machine language A language consisting of strings of 0s and 1s that the computer interprets as instructions; compare the glossary entry for "assembly language."

main procedure The program part enclosed by the outermost **begin** and **end**.

main program The **begin**/**end** block terminated by a period that appears at the end of a program; also called the *statement part*.

make The process of recompiling only those units whose source code has been modified since the last compile. A program that manages this process.

memory The space within the computer for holding information and running programs.

module A self-contained routine or group of routines.

nesting The placement of one unit within another.

nil pointer A pointer having the special value **nil**; a **nil** pointer doesn't point to anything.

node An individual element of a tree or list.

object code The output of a compiler.

offset An index within a segment.

operand An argument that is operated upon by an operator.

operating system A program that manages all operations and resources of the computer.

operator A symbol, such as +, that operates upon an operand.

operator hierarchy The rules that determine the order in which operators in an expression are evaluated.

ordinal type An ordered range of values; same as scalar type.

output The information generated by running a program. Output can be sent to a printer, displayed on screen, or written to disk.

overflow The condition that results when an operation produces a value larger or smaller (more negative) than the computer can represent, given the allocated space for the value or expression.

parameter A variable or value that is passed to a procedure or function.

parameter list The list of value and variable parameters declared in the heading of a procedure or function declaration.

Pascal, Blaise A French mathematician and philosopher (1623-66) who built a mechanical adding machine, considered to be an early predecessor to calculators and computers.

pass To use as a parameter.

pointer A variable that points to (contains the address of) a specific memory location.

pop To remove the topmost element from a stack.

port An I/O device that can be accessed through the CPU's data bus.

precedence The order in which operators are executed.

predefined identifier A constant, type, file, logical device, procedure, or function that is available to the programmer without having to be defined or declared.

procedure A subprogram that can be called from various parts of a larger program.

procedure call The invocation of a procedure.

program A set of instructions for a computer to carry out.

prompt A string printed by a program to signal to the user that input is desired and (sometimes) what kind of input is expected.

push To add an element to the top of a stack.

random-access memory (RAM) Memory that can be read from and written to.

random access Directly accessing an element of a data structure without sequentially searching the entire structure for the element.

range-checking A Turbo Pascal feature that checks a value to make sure it is within the legal range defined.

read-only memory (ROM) The memory device from which data can be read but not written.

real number A number represented by decimal point and/or scientific notation.

record A structured data type referenced by one identifier that consists of several different fields.

recursion A programming technique in which a subprogram calls itself.

relational operator The operators =, <>, <, >, <=, >=, and **in**, all of which are used to form Boolean expressions.

reserved word An identifier reserved by the compiler. A word whose use and meaning is reserved for use only by the program. You cannot redefine the meaning of a reserved word.

result The value returned by a procedure, function, or program.

run time The time during which a program is executing.

scalar type Any Pascal type consisting of ordered components (for example, integer, char, longint, enumerated types, and so forth).

scientific notation A description of a number that uses a number between 1 and 10 (called the mantissa) multiplied by a power of 10 (called the exponent). Because computers cannot easily display exponents on the

screen, scientific notation on computers is usually written using an *E*, as in 24E15—which means 24 multiplied by 10 to the 15th power.

scope The visibility of an identifier within a program.

segment On 8088-based machines, RAM is divided into several segments, or parts, each made up of 64K of memory.

separate compilation The ability to break a large program into several discrete modules, compile each module separately, then link them into a large, executable (.EXE) file.

separator A blank or a comment.

sequential access The ordered access of each element of a data structure, starting at the first element of the structure.

set An unordered group of elements, all of the same scalar type.

set operator The symbols +, –, *, =, <=, >=, <>, and **in**, all of which return set-type results when used with set-type operands.

simple type A data type that contains only a single value.

source code The input to a compiler.

stack A data structure in which the last element stored is the first to be removed.

stack overflow An error condition that occurs when the amount of space allocated to the computer's stack is used up.

stack segment The segment in memory allocated as the program's stack.

statement The simplest unit in a program; statements are separated by semicolons.

static variable A variable with a lifetime that exists the entire length of the program. Memory for static variables is allocated in the data segment.

string A sequence of characters that can be treated as a single unit.

structured type One of the predefined types (array, set, record, file, or string) that are composed of structured data elements.

subprogram A procedure or function within a program; a subroutine.

subrange A continuous range of any scalar type.

subscript An identifier used to access a particular element of an array.

syntax error An error caused by violating the rules of a programming language.

terminal An I/O device for communication between a human being and a computer.

tracing Manually stepping through each statement in a program in order to understand the program's behavior—an important debugging technique.

transfer function A function that converts a value of one type to a value of another type.

tree A dynamic data structure in which a node (branch of a tree) may point to one or more other nodes.

type coercion Technique also known as typecasting in which a variable of one type is forced to be read as another type.

type conversion The reformulation of a value in another form, for example, the conversion of integer values to real.

type definition The specification of a non-predefined type. Defines the set of values a variable can have and the operations that can be performed on it.

typed constant A variable with a value that is defined at compile time but can be modified at run time. (Think of it as a preinitialized variable.)

underlying type The scalar type corresponding to a particular subrange.

unit A program module that makes it possible to do separate compilation. A unit can contain code, data, type and/or constant declarations. A unit can use other units, and is broken into interface (public) and implementation (private) sections.

untyped parameter A formal parameter that allows the actual parameter to be of any type.

value parameter A procedure or function parameter that is passed by value; that is, the value of the parameter is passed and cannot be changed.

vanilla Programmer's lingo for standard or basic.

variable declaration A declaration that consists of the variable and its associated type.

variable parameter A procedure or function parameter that is passed by reference; that is, the address of the parameter is passed so that the value of the parameter can be accessed and modified.

variant record A record in which some fields share the same area in memory.

word A location in memory occupying 2 adjacent bytes; the storage required for a variable of type integer.

Index

$ *See* compiler, directives
8087/80287/80387 coprocessor *See* numeric coprocessor
@ (address-of) operator 220
^ (indirection) operator 57
?: operator 272
makefile comment 259
! makefile directive 270

A

$A compiler directive 99, 176, 213
Abs function 132
active window 239
actual parameters, defined 67
Add Watch
  box 37, 107
  command 37, 121, 189
Addr function 132
  in version 3.0 220
address-of (@) operator 57, 220
address operators 57
Align Data command 176
  in TINST 302
aligning variable and typed constants 302
ANSI Pascal, compatibility with 5.0 215
.ARC files 13
  unpacking 16
arithmetic operators 54
.ASM files, MAKE utility and 91
assembly language 41, 226
  linking routines 90
  MAKE utility and 91
assignment, operators 54
AUTOEXEC.BAT file 323
  modifying 17
Autoindent mode 305
Autoindent Mode command (TINST) 305
automated build switch 153
AUX: (version 3.0) 235
Aux (version 3.0) 219, 235

B

/B command-line option
  in IDE 153
  in TINST or INSTALL 20
  in TPC 198
$B compiler directive 56, 100, 177, 213, 214, 226
Backspace Unindents command (TINST) 305
backup
  copies 11
  files, automatic 35
  source files option 5
Backup Files command 182
  in TINST 304
backward compatibility 208
.BAK files 5
  creating 304
base file name macro (MAKE) 268
basic editor commands 240
BCD arithmetic 223, 234
BGI.ARC 16
BGI example programs (BGIEXAMP.ARC), unpacking 16
BGILINK.PAS 294
.BIN files 226, 234
binary
  arithmetic operators 54
  floating-point arithmetic 48
  format 202
BINOBJ.EXE 294
BINOBJ utility 14, 253, 294, 294-296
bitwise operators 55
block commands 245
Boolean 47
  evaluation 100
  expressions 51, 225
  types 51
Boolean Evaluation command 177
  in TINST 302
bottom line 159
Break/Watch menu 37, 158, 189
breakpoints 110, 119-120, 164
  clearing 115
  commands 189
  instant 120
BufLen function (version 3.0) 234
Build command 3, 42, 43, 90, 172
build command-line option 3, 198
BUILTINS.MAK 274
byte data type 48

C

/C command-line switch 153
Call Stack
  command 137, 178, 187
  window 139
call stack 137
calls, tracking 139
case statements 60
CBreak variable 219, 230, 235
.CFG files 4
chain programs 213, 234
Change Dir command 167
char data types 49
  defined 47
CheckBreak variable 235
choosing menu commands
  IDE 26, 30
  TINST 300
Chr function 132
circular unit reference 77
Clear All Breakpoints command 115, 120, 190
clearing, breakpoints 115
Close procedure 221
ClrEol procedure 219
ClrScr procedure 219
code
  conditional execution 64
  iterative execution 64
  size 213
colors, customizing 316
.COM files 4
command-line
  compiler 2, 13, 14, 20
  compiler reference 193-204
  options 196-203
    /B 198
    /D 179, 197
    debug 202
    directory 200
    /E 200
    /F 198
    /G 202
    /GD 202
    /GP 202
    GREP 290
    /GS 202
    /I 201
    /L 181, 199
    /M 197
    mode 197
    /O 201
    /Q 200
    /R (version 4.0) 211
    switching directive defaults (/$) 196
    /T 200
    /U 201
    /V 203
    /X (version 4.0) 211
  parameters/arguments 185
    setting 307
  switches (IDE) 153-155
    automated build 153
    configuration file 153
    dual monitor mode 154
    make 154
    palette-swapping 154
command lists, makefile 264
commands *See also* individual listings
  breakpoint 189
  debugger 104
  tracing 170
comments 67
  makefile 259
  program 67
compilation 33, 42
  conditional 93
  separate 3
  unit 82
  window 171
Compile command 33, 40, 171
Compile menu 33, 158, 171
  in TINST 300, 301
compile-time
  error messages
    new and obsolete 209
compile-time errors 33, 102
compiler 41
  command-line *See* command-line, compiler
  directives 175, 213
    $D 178
    $A 99, 176, 213
    $B 56, 100, 177, 213, 214, 226

changed meanings 211
$D 111, 143, 198, 213
$DEFINE 94, 179, 197
$E 100, 177, 213, 225
$ELSE 94, 96, 97
emulation 100
$ENDIF 96
$F 176, 213, 226
$I 100, 176, 184, 213, 214, 221, 235
$IFDEF 94, 96, 97
$IFNDEF 94, 97
$IFOPT 94, 98
$IFOPT N+ 99
inserting 244
$K (version 3.0 and 4.0) 210
$L 71, 111, 142, 179, 184, 210, 213, 226
$M 179, 197, 213
makefile 270
$N 48, 98, 177, 213, 215, 223, 225
new and modified 210
$O 176, 213
obsolete 210
$R 100, 175, 213, 214
$S 100, 175, 210, 213
$T (version 4.0) 210
$U (version 4.0) 210
$UNDEF 94
$V 100, 177, 213
integrated environment version *See* integrated, development environment
mode, command-line options *See* command-line, options
options *See* command-line, options
Compiler menu 174
in TINST 301
compiling
for debugging 164
to disk 43, 142
to memory (RAM) 42
complete Boolean evaluation option 177
compound statements 60
CON: (version 3.0) 235
Con (version 3.0) 219
conditional
compilation 93
defines (command-line option) 197
directives, in makefiles 270
execution 47
statements 59
symbols 95
Conditional Defines command 179
in TINST 302
Config Auto Save command 182
in TINST 303
CONFIG.SYS file, modifying 17
configuration files 4
default 183, 306
menu commands 182, 185
pick file and 192
TPC.CFG 203
ConInPtr (version 3.0) 219
ConOutPtr (version 3.0) 219
constants, typed 223, 227
ConStPtr (version 3.0) 219
context-sensitive help 23
control characters 50
Copy function 221
CPU 41
symbols 96
Crt unit 69, 80, 234
CrtExit procedure (version 3.0) 219, 235
CrtInit procedure (version 3.0) 219, 235
CSeg function 221, 234
Current Pick File command 185
cursor movement commands 241, 243
customizing
colors 316
keystroke commands 300
Turbo Pascal with TINST *See* TINST

D

/D command-line
option 179, 197
switch 154
$D compiler directive 111, 143, 178, 198, 213

data 46
  defined 46
  types
    boolean 47, 51
    byte 48
    char 47, 49
    converting to 5.0 217
    defined 47
    integer 47
    longint 48
    pointer 47, 52
    real 49
    real numbers 47
    shortint 48
    string 51
    typecasting 224
    word 48
Debug Information command 178
  in TINST 302
Debug menu 158
  in TINST 300, 308
debugger, integrated *See* integrated, debugger; debugging
debugging 5, 101-152
  Add Watch box 107
  basic unit of execution 112
  cancelling 169
  commands 104, 186
  compile-time errors 102
  compiling a program for 164
  example 107
  global identifiers 111
  hot keys 104
  I/O error-checking 146
  IFDEF and 98
  IFNDEF and 98
  inability to trace 144
  information
    compiling to disk 308
    generating 178
    line-number 178
    suppressing 143
  integrated 308
    controlling with TINST 308
  interrupt service routines (ISRs) 144
  local identifiers 111
  memory 142
  navigation 137
  options, command-line 202
  pitfalls 145
  preventive 141
  range-checking 148
  recompiling during 164
  restarting 114
  run-time errors 102
  screen display 164
  stopping *See* Program Reset command
  syntax errors 102
  tracing 110
  Turbo Debugger and 150-152
declarations, unit 74
default
  directories
    changing 306
    installing 15
  settings
    changing 297
    command-line arguments 307
    configuration file 306
    overriding 4
    pick file name 307
    restoring 318
$DEFINE compiler directive 94, 179, 197
Defined Symbols box 179
Delay procedure 219
Delete Watch command 190
DelLine procedure 219
Destination command 35, 43, 172
  in TINST 301
destination default setting 172
directives *See* compiler, directives
directories
  changing 167
  command-line options 200
  configuration 183
  default settings 306
  DOS 321, 324
  saving command 185
Directories menu 184
  in TINST 306
Directory command 167
disks
  backup 11

distribution 11, 12
Display Swapping command 113, 188
in TINST 308
displaying source code 112
Dispose procedure 53
distribution disks 11, 12
div operator 49
DOS 235
basics 319
commands, MODE 154
directories in 321, 324
exiting to 27, 167, 168
returning from 167
symbol 96
Dos unit 69, 80, 219, 234
DSeg function 221, 234
dual monitor mode 154, 188
User Screen command and 171
dynamic variables 142

E

/E command-line option 200
$E compiler directive 100, 177, 213, 225
Edit
command 168
menu 158
status line 159
window 26, 31, 112, 159, 172, 239
creating source files in 161
status line 240
working in 161
Edit Auto Save command 182
in TINST 304
Edit Watch command 190
editor
commands 240
specifications 239
summary of 241
default modes, setting 303
Editor Buffer Size command (TINST) 305
Editor Commands menu (TINST) 300, 308
EGA 39, 183, 316
TINST option 304
$ELSE compiler directive 94, 96, 97
ELSE symbol 96
EMS
memory 142
editor buffer and 306
requirements 7
Emulation command 177
in TINST 302
$ENDIF compiler directive 96
ENDIF symbol 96
Environment menu 181
in TINST 303
environment options
Backup Files 182
Config Auto Save 182
Edit Auto Save 182
Screen Size 183
setting, in TINST 303
Tab Size 183
Zoom Windows 183
ERR: (version 3.0) 234
ErrorPtr (version 3.0) 219, 226, 234
errors 33
checking 177
compile-time 33, 102
converting from version 3.0 and 234
handling 99, 145, 226
I/O 146
messages 276
new and obsolete 209
out-of-memory/bounds 148
run-time *See* run-time errors
syntax 33, 102
Evaluate
command 134, 186
window 134, 186
evaluation, order of expressions 218
exclamation point (!), makefile directives 270
EXE & TPU Directory command 183
in TINST 307
EXE & TPU directory command-line option 200
.EXE files 4, 43
producing 43
storing 172, 183
Exec procedure 219, 234

executable
  code, storing 172
  directories command 183
Execute programs 234
execution
  bar 36, 107, 165
  position 165
ExitProc variable 226, 234
Expanded Memory Specification *See* EMS
expressions
  evaluation order 218
  Watch *See* Watch, expressions
extended
  memory support *See* EMS
  memory
  movement commands 243
external subroutines 227

F

/F command-line option 198
$F compiler directive 176, 213, 226
FAR call
  generation 226
  model, forcing use of 176
field-width specifiers 58
File menu 32, 158, 165
FilePos function 219
files
  .ARC 13
  .ARC, unpacking 16
  backup source 5
  .BAK, creating 304
  .BIN 226, 234
  commands 165
  configuration 4
  debugging 5
  default settings 4
  .EXE 43
    producing 43
    storing 172, 183
  extension names 4
  graphics, installing 20
  library 4
  .MAP 5, 180, 202
    storing 183
  name macros 268, 269
  .OBJ 43, 201, 226
    locating 184
  obsolete, deleting 20
  packed, unpacking 13
  .PAS 4, 13, 174
    searching contents of 287
  pick 5
  primary 42
  saving 167
  source code 4
  TPC.CFG 4
  .TPL 4, 253
  .TPU 4, 43, 82, 172, 253
    compatibility with version 4.0 210
    controlling output of 211
    debug information 178
    local symbol information 178
    storing 183
  Turbo Pascal 3.0 4
  TURBO.TPL 4
  unit 4
  unpacking 13
FileSeek function 219
FileSize function 219
Find Error command 34, 173, 199
find error command-line option 198
Find Procedure command 137, 139, 187
finding matching pairs 250
floating-point
  8087/80827 177
  numbers 47
  software 177
for
  loop counter variables 235
  statements, loop 63, 222
Force Far Calls command 176
  in TINST 301
Form function 234
formal parameters, defined 67
format specifiers 127
  repeat count 128
  using 130
forward declarations 233
full file name macro (MAKE) 269
Full Graphics Save command (TINST) 304

functions
  declarations 211
  defined 64
  finding 139
  new 209
  structure 65

G

/G command-line option 5, 202
/GD command-line option 202
generating line-number tables 111
Get Info command 142, 174
global
  declarations 233
  identifiers 111
glossary 327
Go to Cursor command 114, 170
GotoXY procedure 219
/GP command-line option 202
GRAPH3.TPU 14
Graph3 unit 69, 81, 212, 228
GRAPH.TPU 13
  compiling BGI programs and 20
Graph unit 40, 69, 81, 228
graphics 39
  example programs, unpacking 16
  files, installing into a separate subdirectory 20
  turtlegraphics 228
GREP.COM 14, 287, 290
GREP utility (file searcher) 14, 253, 287-294
  command-line
    options 288, 290
    syntax 288
  search string 290
  switches 288
/GS command-line option 202

H

HaltOnError 236
hardware requirements 7
heap management, sizes 142, 179, 196
help
  hot key 160
  online
    in command-line compiler 24
    in integrated environment 23
HELPME!.DOC 12, 16
hexadecimal constants 48
Hi function 132
high heap limit setting 179
HighVideo procedure 235
hot keys 27, 30
  debugger 104
  help 160
  zoom 160

I

/I command-line option 201
$I compiler directive 100, 176, 184, 213, 214, 221, 235
I/O
  defined 46
  error-checking 100, 146
    disabling 146
  handling 215
  real numbers and 215
I/O-Checking command 176
  in TINST 301
IDE *See* integrated, development environment
identifiers 53
  defined 53
  naming restrictions 53
  scope 123
  Turbo3 230
IEEE floating-point 96, 177, 215
if statements 59
IFDEF 97
$IFDEF compiler directive 94, 96, 97
IFNDEF 97
$IFNDEF compiler directive 94, 97
IFOPT 97
$IFOPT compiler directive 94, 98
IFxxx symbol 96
illustration, menu system 156, 157
implementation sections 78
  uses clauses in 76
include compiler directive 213, 215
  in version 3.0 214
Include Directories command 184
  in TINST 307

include directories command-line option 201
Include files 201, 213, 214, 233
index variable 63
indirection (^) operator 57
infinite loop 40, 108
InitGraph procedure
  installing graphics files into a separate subdirectory and 20
Initial unit (UPGRADE) 233
initialization
  units 88
  variables 72
inline
  directives 227
  statements 226, 227
INP: (version 3.0) 234
input 46
  defined 46
  functions 59
insert and delete commands 244
Insert mode 305
Insert Mode command (TINST) 305
InsLine procedure 219
Install Editor screen 309
INSTALL.EXE 13, 15
INSTALL utility 15-20
  /B command-line option 20
  LCD or composite screen display
    adjusting 20
installation, Turbo Pascal 15-20
  floppy disk 18
  hard disk 15
Installation menu (TINST) 300
integers
  defined 47
  types 47
integrated
  debugger 36, 164, 308, *See also* debugging
  development environment 12, 20
    commands *See* individual listings
    compiling in 33
    context-sensitive help 23
    Edit window *See* Edit, window
    editor 239
    Graph unit and 40
    graphics 39
    hot keys in 27, 160
    loading 31, 153
    machine code in 41
    main screen 25
    menus *See also* menus
      choosing commands from 26, 30
      main 158
    .OBJ files in 43
    quitting 27
    reference 153-192
    saving files in 32
    statements 32
    TINST and 28
    .TPU files in 43
    tutorial 31
    using 2, 23
    variables 32
    windows 26, *See also* windows
Integrated Debugging command 187
  in TINST 308
interface sections 79
interrupt handlers 226
  Turbo Debugger and 144
interrupt service routines (ISRs)
  debugging 144
Intr procedure 219, 220
IOResult function 132, 176, 220, 235, 236
ISRs (interrupt service routines), debugging 144

J

journal file (UPGRADE) 231

K

$K compiler directive (version 3.0 and 4.0) 210
Kbd 219, 223, 230, 235
KBD: (version 3.0) 235
KeyPressed function 219
keystrokes
  allowed 312
  customizing 300
    commands 308
    control keys 310

function keys 310
keywords 24

L

/L command-line option 181, 199
$L compiler directive 71, 111, 142, 179, 184, 210, 213, 226
labels 222
language help, online 250
laptop computers *See* LCD mode
large programs, managing 87
LastMode variable 220
LCD mode 315, 316
IDE display, adjusting 26
TINST or INSTALL display, adjusting 20
Length function 132
libraries
files 4
line-number tables 178
Link Buffer command 181, 200, 211
in TINST 303
Linker menu 180
in TINST 303
linking
buffer option 199
defaults, setting 303
$L compiler directive 71
with 8087 run-time library 177
literal strings 290
Lo function 132
Load command 161, 165
loading
options 165, 185
pick files 192
programs in DOS 320
Turbo Pascal 31
local
identifiers 111
symbol information, generating 178, 210
Local Symbols command 178
in TINST 302
logic errors 103
logical operators 56
LongFile functions (version 3.0) 220, 235
LongFilePos function (version 3.0) 219, 220, 230
LongFileSize function (version 3.0) 219, 220, 230
longint data type 48
LongSeek function (version 3.0) 220, 230
loops
defined 47
for 63
repeat..until 62
while 61
low heap limit setting 179
low-level operations 55
LowVideo procedure 219, 235
LST: (version 3.0) 235
Lst variable (version 3.0) 219
LstOutPtr (version 3.0) 219

M

/M command-line
option 197
switch 154
$M compiler directive 142, 179, 197, 213
machine code 41
macros, makefile 266
main menu 155
Make command 3, 42, 43, 90, 164, 172
make command-line option 197
Make Use of EMS Memory command (TINST) 306
MAKE utility 3, 14, 253, 258
.ASM files and 91
command-line options 92, 258, 275
error messages 276
syntax 274
using 273
makefiles, creating 258
Map File command 178, 180, 183, 202, 210
in TINST 303
map file command-line option 202
.MAP files 5, 180, 202
generating 210
storing 183
matching pairs 250

math coprocessor *See* numeric coprocessor
MaxAvail function 132, 219, 230, 235
MaxInt 47
Mem array 132, 226
MemAvail function 132, 219, 230, 235
MemL array 132, 226
memory 142
  access 226
  allocation 196
  conserving 142
  defaults, configuring 179
  EMS 142, 306
  menu command 179
  setting size for editor buffer 305
Memory Sizes command 142, 179
  in TINST 303
MemW array 132, 220, 226
MENU.DTA 295
menus
  "sticky" 155
  Break/Watch 158, 189
  choosing commands 30
  commands *See* individual listings
  Compile 33, 158, 171
  Compiler 174
  Debug 158, 186
  Directories 184
  Edit 158
  Environment 181
  File 32, 158, 165
  illustration 156, 157
  Linker 180
  main 155
    structure of 155
  Options 158, 174
  Pick List 191
  Run 34, 158, 168
  settings 159
  structure of 156, 157
  TINST *See* individual listings and TINST, menus
  toggles 158
MicroCalc 13
Mode for Display menu (TINST) 300, 314
modes, editing 160
modifying expressions and variables 136
Move procedure 224
MsDos procedure 219, 220

N

$N compiler directive 48, 98, 177, 213, 215, 223, 225
NEAR calls 226
nesting files 214
New command 161, 166
NormVideo procedure 219, 235
NoSound procedure 219
numbers, counting 47
numeric
  constants 40
  coprocessor 48, 96, 98
    emulating 177
      with 8087 run-time library 177
Numeric Processing command 177
  in TINST 302

O

/O command-line option 201
$O compiler directive 176, 213
.OBJ files 43, 201, 226
  locating 184
  MAKE utility and 91
Object Directories command 184, 202
  in TINST 307
object directories command-line option 201
Ofs function 132
online help 250
operating system 27
operations 46
  defined 46
  low-level 55
operators
  ^ (indirection) 57
  address 57
  address-of (@) 57, 220
  arithmetic 54
  assigment 54
  binary 54
  bitwise 55
  defined 53

div 49
logical 56
makefile directives 270
precedence of 54
relational 55
set 57
string 57
unary 54
Optimal Fill command (TINST) 305
optimization of code 99
Options for Editor command (TINST) 304
Options menu 158, 174
in TINST 300, 301
Ord function 132
OS Shell command 35, 154, 167
OUT: (version 3.0) 234
out-of-memory/bounds errors 148
output
defined 46
devices 57
files 162
Writeln 57
Output window 113, 163
Overlay unit 69, 80, 231
overlays 213, 232, 233
enabling 176
Overlays Allowed command 176
in TINST 302
OvrPath (version 3.0) 231, 234

P

/P command-line switch 154
pair matching 250
palette-swapping, on EGA 154
Parameters command 185
.PAS files 4, 13, 174
searching contents of 287
PC-DOS 319
.PCK file 5
pick
files 5, 185, 191
default name, setting 307
defining 191
list 4
loading 192
locating 185
saving 192
list 191
saving 166
Pick command 162, 166
Pick File Name command 185
in TINST 307
Pick List menu 191
pointers 52
defined 47
Pred function 132
Primary File command 42, 173
in TINST 301
Printer unit 69, 80
PrinterLst file 235
procedures
defined 64
finding 139
new and modified 209
structure 65
Program Reset command 114, 169
programming, elements of 46
programs
comments 67
compiling 33, 40
debugging *See* debugging
declarations 213
editing 31
reinitializing 114
running 34
saving 32
stepping through 37
structure of 65, 87, 213
tracing 110
updating 33
project management 87
Ptr function 132

Q

/Q command-line option 200
quiet mode command-line option 200
Quit command 168
Quit/Save command
in TINST 300
quitting
debugging *See* Program Reset command
Turbo Pascal 27, 168

R

/R command-line option
  $R and 196
  in version 4.0 211
$R compiler directive 100, 175, 213, 214
range-checking 100, 175, 196, 214
  errors 148
  selectively implementing 149
Range-Checking command 175
  in TINST 301
Read procedure
  implementing 215-217
  text files 59
ReadKey function 235
Readln procedure 59, 234
  implementing 215-217
README 12
README.COM 14
real numbers 47, 48
recursion 143
Refresh Display command 113, 189
reinitializing a program 114
relational operators 55
relaxed error-checking 177
Remove All Watches command 190
repeat..until loop 62
repeat count 128
requirements
  minimum system 7
reserved words 70, 71, 234
Resize Windows menu (TINST) 300, 317
restarting a debugging session 114
restoring default settings 318
Retrieve Options command 185
Round function 132
routines, recursive 143
rules
  explicit 259
  implicit 261
  makefile 259, 261
Run command 36, 42, 169
Run menu 34, 158, 168
run-time errors 36, 102, 175
  Debug Information command and 173
  Find Error command and 173
  finding 198
  messages, added in 5.0 210
running programs 34

S

$S compiler directive 100, 175, 210, 213
Save (directories) command 185
Save (file) command 167
saving
  files 167
  pick files 192
  programs 32
scope 123
screen
  data file 295
  editor 159, 168
  size 304
  swapping 113, 188
Screen Size command 183
  in TINST 304
search utility 14
searching *See* GREP utility (file searcher)
Seek procedure 220
Seg function 132
semantic errors 102
separate compilation 3, 69, 253
Set Colors menu (TINST) 300, 316
sets, operators 57
SetTextBuf procedure 222, 234
setting breakpoints 114
short-circuit Boolean
  evaluation 226
  expressions 177
shortint data type 48
significant digits, defined 48
single-step tracing 115
SizeOf function 132
smart screen swapping 188
software numeric processing *See* Numeric Processing command
Sound procedure 219
source
  code, displaying 112
  files 4
    backing up 182

creating 161
in Edit window 161
loading 162
saving 162
working with 161
SPtr function 132
SSeg function 132
stack
checking 100, 175
controlling 210
size 179
decreasing 142
Stack-Checking command 175
in TINST 301
Stand-alone Debugging command 178, 188, 203
in TINST 308
stand-alone debugging command-line option 203
standard units *See* units, standard
statements 32
case 60
compound 60
conditional 59
if 59
uses 70, 73
status line 159, 240
status window *See* compilation window
Step Over command 164, 170
stepping through a program 37
strict error-checking 177
strings 51
operators 57
subdirectories, DOS 322
subroutines 47, 64
Succ function 132
Swap function 132
swapping screens 113
switches
command-line (IDE) 153-155
automated build 153
configuration file 153
dual monitor mode 154
make 154
palette-swapping 154
GREP 288
symbols
definition 179
table 111
syntax
errors 102
MAKE 274
system requirements 7
System unit 69, 74, 79, 234

T

/T command-line option 200
$T compiler directive (version 4.0) 210
Tab mode 183
Tab Size command 183
in TINST 305
tabs
default 305
setting 183
TD.EXE 203
technical support 8, 17
TEST.PAS 116
TextBackground procedure 219
TextColor procedure 219
TextMode procedure 219, 220, 234
THELP.COM 13
THELP utility 13, 24
TINST 14, 28, 45, 297-318
/B command-line option 20
commands *See* individual listings
LCD or composite screen display
adjusting 20
menus
choosing commands from 300
Compile 300, 301
Compiler 301
Debug 300, 308
Directories 306
Editor Commands 300, 308
Environment 303
exiting 301
Installation 300
Linker 303
Mode for Display 300, 314
Options 300, 301
Resize Windows 300, 317
Set Colors 300, 316

transferring settings from 4.0 to 5.0 298
TINSTXFR.EXE 14
TINSTXFR utility 212, 298
INSTALL and 17
Toggle Breakpoint command 114, 119, 190
toggles 158
TOUCH utility 14, 253, 287
.TP files 4
TPC.CFG file 4, 200, 203
sample 204
TPC.EXE 2, 13, 20, 193
TPCONFIG.EXE 14
.TPL files 4, 253
.TPU files 4, 43, 82, 172, 253
compatibility with version 4.0 210
controlling output of 211
debug information 178
local symbol information 178
storing 183
TPUMOVER.EXE 13, 82, 84
TPUMOVER utility 13, 84, 253
Trace Into command 36, 170
tracing
commands 170
programs 110
single-step 115
trapping, I/O errors 146
TRM: (version 3.0) 235
Trm (version 3.0) 219
Trunc function 132
TURBO3.TPU 14
Turbo3 unit 69, 81, 212, 228, 230, 235
Turbo Assembler 226, 327
Turbo Debugger 150, 203, 308, 329, *See also* debugging
interrupt 9 handlers and 144
THELP utility and 24
Turbo Directory command 183, 200
in TINST 306
Turbo directory command-line option 200
TURBO.EXE 2, 12, 31, 297
TURBO.HLP 13, 183
Turbo Pascal
installation
floppy disk 18
hard disk 15
Turbo Pascal 3.0 14, 70
conversion 21, 207, 227
ANSI compatibility 215
assembly language 226
BCD arithmetic 223, 234
.BIN files 226, 234
Boolean expressions 225, 236
CBreak variable 219, 230
Close procedure 221
code size 213
Copy function 221
CSeg function 221, 234
DSeg function 221, 234
ERR: 234
ErrorPtr 226
Execute procedure 219
ExitProc variable 226
external procedures/functions 227
$F compiler directive 226
FAR calls 226
file names and 229
FilePos function 219
FileSeek function 219
FileSize function 219
for loop, control variables in 222
forward declarations 233
global declarations 233
GotoXY procedure 219
Graph3 unit 81, 228
HaltOnError 236
I/O errors 215, 220
include directive 214, 215
Include files 214, 233
Initial unit (UPGRADE) 233
inline
directives 227
procedures/functions 227
statements 226, 227
interrupt handler 226
Intr procedure 220
IOResult function 220
journal file 231
Kbd 219, 223, 230
KBD: 235
KeyPressed function 219
$L compiler directive 226

labels 222
LastMode variable 220
LongFile functions 220
LongFilePos function 220, 230
LongFileSize function 220, 230
LongSeek function 220, 230
LowVideo procedure 219
Lst variable 219
MaxAvail function 219, 230
Mem array 226
MemAvail function 219, 230
MemL array 226
memory access 226
MemW array 226
mixing of types 235
Move procedure 224
MsDos procedure 220
$N compiler directive 223
NEAR calls 226
nesting files 214
.OBJ files 226
overlays 213
predeclared identifiers 218, 219
program
declarations 213
structure 213
range-checking 214
Seek procedure 220
SetTextBuf procedure 222
short-circuit Boolean evaluation 214, 226, 236
TextMode procedure 220
Turbo3 unit 81, 220, 228, 230
type
checking 221
mismatches 236
typecasting 224
typed constants 223, 227
units 213
UPGRADE 228
UPGRADE.DTA 229
UPGRADE.EXE 229
versus 5.0 207, 212
Turbo Pascal 4.0
backward compatibility 208
conversion 21, 207
$K compiler directive 210
$T compiler directive 210
$U compiler directive 210
versus 5.0 207, 208
TURBO.PCK 4, 185, 307
TURBO.TP 45, 185, 298, 306
TURBO.TPL 4, 12, 70, 79, 142, 200, 201, 253
turtlegraphics 219, 228
tutorial 23, 45
two's complement 55
type-checking 221
typecasting 130, 224
typed constants 223, 227
types *See* data, types

U

/U command-line option 201
GRAPH.TPU file and 20
$U compiler directive (version 4.0) 210
unary
minus 55
plus 55
$UNDEF compiler directive 94
Unindent mode 305
Unit Directories command 184, 201
GRAPH.TPU file and 20
in TINST 307
units 3, 13, 40, 43, 69
Build option 90
circular reference 77
compiling 43, 82, 90
converting from 3.0 and 213, 228, 232
declarations 74
definition 69
forward declarations and 71
global 87
large programs and 83, 87
implementation section 71
initialization section 72
initializing 88
inserting 256
interface section 71
Turbo Pascal 3.0 and 233
large programs and 83
Make option 90
merging 82

mover utility 253
overlays and 232, 233
recompiling 42
removing 257
standard 234
Crt 69, 80
Dos 69, 80
Graph 69, 81, 228
Graph3 69, 81, 228
Overlay 69, 80
Printer 69, 80
System 69, 74, 79
Turbo3 69, 81, 228, 230
structure 70
.TPL files 253
.TPU files 82, 253
TPUMOVER 84
TURBO.TPL file 70, 79, 82, 84
inserting into 256
removing from 257
Unit Directories command 82
Unit directories option 201, 254
uses statement 70, 73
use of 73
writing 81
UPGRADE.DTA 14, 229, 236
UPGRADE.EXE 14, 227, 236
UPGRADE program 14, 212, 228
comments 229, 231
journal file 231
options 230
using 228
warnings 231, 234
Use Tabs command (TINST) 305
User screen 34, 112
displaying 171
User Screen command 34, 171
uses
clause 43
in an implementation section 76
statement 70, 73
USR: (version 3.0) 235
Usr (version 3.0) 219
UsrInPtr (version 3.0) 219
UsrOutPtr (version 3.0) 219
utilities
BINOBJ *See* BINOBJ utility
GREP *See* GREP utility (file searcher)
INSTALL *See* INSTALL utility
MAKE *See* MAKE utility
stand-alone 253, 287
TINST *See* TINST
TOUCH *See* TOUCH utility
TPCONFIG 14
TPUMOVER *See* TPUMOVER utility
UPGRADE *See* UPGRADE program

V

/V command-line option 203
$V compiler directive 100, 177, 213
var parameters
checking 100
Var-String Checking command 177
in TINST 302
variables 32
dynamic 142
index 63
initializing 72
local, stack allocation 211
modifying 136
VER50 95
VGA
display 183
TINST option 304
video modes
black and white 316
changing 314
color 315
LCD 316
monochrome 316
View Next Breakpoint command 120, 190

W

Watch
expressions 163
acceptable values 131
adding 133, 189
built-in functions 132

deleting 133, 190
display 126
editing 133
format specifiers 127
repeat count 128
using 130
modifying 136
typecasting 130
types 126
arrays 127
Booleans 127
characters 126
enumerated data types 127
files 127
integers 126
pointers 127
reals 126
records 127
sets 127
strings 127
window 26, 113, 163
editing 133
watches 110, 121
commands 189
setting up 121
while (syntax)
loop 61
Window procedure 219
windows 26
active 239
Call Stack 139
compilation 171
Edit *See* Edit, window
Evaluate 134
evaluate 186
repainting 113
resizing 317
switching 113
Watch *See* Watch, window
zoom 183
word data type 48
WordStar, Turbo Pascal Editor versus 251
Write To command 36, 162, 167
Writeln procedure 57
field-width specifiers and 58

X

/X command-line option (version 4.0) 211

Z

zoom hot key 160
Zoom Windows command 37, 183
in TINST 304
zooming between windows 113