User's Guide

VERSION
4.1

Borland® Turbo Assembler®
for OS/2®

User's Guide

Borland® Turbo Assembler® for OS/2®

Version 4.1

Borland International, Inc.
100 Borland Way, Scotts Valley, CA 95066-3249

PRINTED IN THE UNITED STATES OF AMERICA

1E0R0294
9495969798-987654321
H1

Contents

Introduction 1
Hardware and software requirements 2
About the manuals 2
Notational conventions 4
Contacting Borland 4
Borland Assist plans 4

Chapter 1 Getting started with Turbo Assembler 7
Installing Turbo Assembler 7
Utility and example programs 8
Online help 8
Writing your first Turbo Assembler program 8
Assembling your first program 9
Linking your first program 11
Recommended reading 11

Chapter 2 Using directives and switches 13
Starting Turbo Assembler 13
Command-line options 16
/a 16
/b 17
/c 17
/d 17
/e 18
/h or /? 18
/i 19
/j 19
/kh 20
/l 20
/la 21
/m 21
/ml 21
/mu 22
/mv# 22
/mx 23
/n 23
/o 24
/oi 24
/op 24
/os 24
/p 25
/q 25
/r 25
/s 26
/t 26
/u 27
/v 27
/w 27
/x 28
/z 29
/zd 29
/zi 29
/zn 30
Indirect command files 30
The configuration file 31

Chapter 3 General programming concepts 33
Turbo Assembler Ideal mode 33
Why use Ideal mode? 34
Entering and leaving Ideal mode 34
MASM and Ideal mode differences 35
Expressions and operands 36
Operators 36
Suppressed fixups 36
Operand for BOUND instruction 37
Segments and groups 37
Accessing data in a segment belonging to a group 37
Commenting the program 40
Comments at the end of the line 40
The COMMENT directive 40
Extending the line 40
Using INCLUDE files 41
Predefined symbols 42
Assigning values to symbols 43
General module structure 44
The VERSION directive 44
The NAME directive 45
The END directive 45
Displaying a message during assembly 46
Displaying warning messages 46
Multiple error-message reporting 47

Chapter 4 Creating object-oriented programs 49
Terminology 49
Why use objects in Turbo Assembler? 50
What is an object? 50
A sample object 51
Declaring objects 51
Declaring a base object 52

Declaring a derived object 54
Declaring a method procedure 55
The virtual method table 56
Initializing the virtual method table 56
Calling an object method 57
Calling a static method 57
Calling a virtual method 58
Calling ancestor virtual methods 60
More on calling methods 61
Creating an instance of an object 62
Programming form for objects 62

Chapter 5 Using expressions and symbol values 65
Constants 65
Numeric constants 65
Changing the default radix 66
String constants 66
Symbols 67
Symbol names 67
Symbol types 67
Simple address subtypes 68
Describing a complex address subtype 68
Expressions 69
Expression precision 70
Constants in expressions 70
Symbols in expressions 70
Registers 70
Standard symbol values 71
Simple symbol values 71
The LENGTH unary operator 72
The SIZE unary operator 72
The WIDTH unary operator 73
The MASK unary operator 73
General arithmetic operators 74
Simple arithmetic operators 74
Logical arithmetic operators 74
Bit shift operators 75
Comparison operators 75
Setting the address subtype of an expression 75
Obtaining the type of an expression 76
Overriding the segment part of an address expression 77
Obtaining the segment and offset of an address expression 77
Creating an address expression using the location counter 78
Determining the characteristics of an expression 79
Referencing structure, union, and table member offsets 79
Describing the contents of an address 80
Implied addition 80
Obtaining the high or low byte values of an expression 80
Specifying a 16- or 32-bit expression 81

Chapter 6 Choosing processor directives and symbols 83
iAPx86 processor directives 83
Predefined symbols 85
@Cpu 85
@WordSize 86
8087 coprocessor directives 86
Coprocessor emulation directives 87

Chapter 7 Using program models and segmentation 89
The **MODEL** directive 90
Symbols created by the MODEL directive ... 93
The @Model symbol 93
The @32Bit symbol 93
The @CodeSize symbol 93
The @DataSize symbol 93
The @Interface symbol 93
Simplified segment directives 94
Symbols created by the simplified segment directives 95
The STARTUPCODE directive 95
The @Startup symbol 96
The EXITCODE directive 96
Defining generic segments and groups 96
The SEGMENT directive 96
Segment combination attribute 97
Segment class attribute 98
Segment alignment attribute 98
Segment size attribute 98
Segment access attribute 99
The ENDS directive 99
The GROUP directive 99
The ASSUME directive 100
Segment ordering 101
Changing a module's segment ordering .. 101
DOS ordering of segments: the DOSSEG directive 102
Changing the size of the stack 103

Chapter 8 Defining data types 105
Defining enumerated data types 105

Defining bit-field records 106
Defining structures and unions 108
Opening a structure or union definition 108
Specifying structure and union members 108
Defining structure member labels with LABEL 109
Aligning structure members 109
Closing a structure or union definition 109
Nesting structures and unions 110
Including one named structure within another 111
Using structure names in expressions 113
Defining tables 113
Overriding table members 115
Defining a named type 115
Defining a procedure type 115
Defining an object 116
The TBLPTR directive 118
Symbols defined by the extended STRUC directive 118

Chapter 9 Setting and using the location counter 119
The $ location counter symbol 119
Location counter directives 120
The ORG directive 120
The EVEN and EVENDATA directives 122
The ALIGN directive 122
Defining labels 123
The : operator 123
The LABEL directive 124
The :: directive 124

Chapter 10 Declaring procedures 125
Procedure definition syntax 125
Declaring NEAR or FAR procedures 126
Declaring a procedure language 128
Specifying a language modifier 129
Defining arguments and local variables 131
ARG and LOCAL syntax 131
The scope of ARG and LOCAL variable names 133
Preserving registers 134
Defining procedures using procedure types 134
Nested procedures and scope rules 135
Declaring method procedures for objects 136
Using procedure prototypes 137

Chapter 11 Controlling the scope of symbols 141
Redefinable symbols 141
Block scoping 142
The LOCALS and NOLOCALS directives 142
MASM block scoping 143
MASM-style local labels 143

Chapter 12 Allocating data 145
Simple data directives 146
Creating an instance of a structure or union 149
Initializing union or structure instances 150
Creating an instance of a record 153
Initializing record instances 153
Creating an instance of an enumerated data type 154
Initializing enumerated data type instances 155
Creating an instance of a table 155
Initializing table instances 155
Creating and initializing a named-type instance 156
Creating an instance of an object 157
Creating an instance of an object's virtual method table 157

Chapter 13 Advanced coding instructions 159
Intelligent code generation: SMART and NOSMART 159
Extended jumps 160
Additional 80386 LOOP instructions 161
Additional 80386 ENTER and LEAVE instructions 161
Additional return instructions 161
Additional IRET instructions 162
Extended PUSH and POP instructions 162
Multiple PUSH and POPs 162
Pointer PUSH and POPs 163
PUSHing constants on the 8086 processor 163
Additional PUSHA, POPA, PUSHF and POPF instructions 163
The PUSHSTATE and POPSTATE instructions 164
Extended shifts 165
Forced segment overrides: SEGxx instructions 166
Additional smart flag instructions 166
Additional field value manipulation instructions 167
The SETFIELD instruction 167
The GETFIELD instruction 168

Additional fast immediate multiply instruction . . . 169
Extensions to necessary instructions for the 80386 processor . . . 169
Calling procedures with stack frames . . . 170
Calling procedures that contain RETURNS . 172
Calling procedures that have been prototyped . . . 172
Calling method procedures for objects: CALL..METHOD . . . 172
Tail recursion for object methods: JMP..METHOD . . . 173
Additional instruction for object-oriented programming . . . 174

Chapter 14 Using macros 175
Text macros . . . 175
Defining text macros with the EQU directive . . . 175
String macro manipulation directives . . . 176
The CATSTR directive . . . 176
The SUBSTR directive . . . 176
The INSTR directive . . . 177
The SIZESTR directive . . . 177
Text macro manipulation examples . . . 177
Multiline macros . . . 177
The multiline macro body . . . 178
Using & in macros . . . 178
Including comments in macro bodies . . . 179
Local dummy arguments . . . 180
The EXITM directive . . . 180
Tags and the GOTO directive . . . 180
General multiline macros . . . 181
Invoking a general multiline macro . . . 182
Redefining a general multiline macro . . . 184
Deleting a general multiline macro: The PURGE directive . . . 184
Defining nested and recursive macros . . . 185
The count repeat macro . . . 186
The WHILE directive . . . 187
String repeat macros . . . 187
The % immediate macro directive . . . 188
Including multiline macro expansions in the list file . . . 189
Saving the current operating state . . . 189

Chapter 15 Using conditional directives 193
General conditional directives syntax . . . 193
IFxxx conditional assembly directives . . . 193
ELSEIFxxx conditional assembly directives . 195
ERRxxx error-generation directives . . . 195
Specific directive descriptions . . . 196
Unconditional error-generation directives . . 196
Expression-conditional directives . . . 196
Symbol-definition conditional directives . . . 197
Text-string conditional directives . . . 198
Assembler-pass conditionals . . . 200
Including conditionals in the list file . . . 201

Chapter 16 Interfacing with the linker 203
Publishing symbols externally . . . 203
Conventions for a particular language . . . 203
Declaring public symbols . . . 204
Declaring library symbols . . . 204
Defining external symbols . . . 205
Defining global symbols . . . 205
Publishing a procedure prototype . . . 205
Defining communal variables . . . 206
Including a library . . . 207
The ALIAS directive . . . 208

Chapter 17 Generating a listing 209
Listing format . . . 209
General list directives . . . 210
Include file list directives . . . 211
Conditional list directives . . . 212
Macro list directives . . . 213
Cross-reference list directives . . . 213
Changing list format parameters . . . 215

Chapter 18 Interfacing Turbo Assembler with Borland C++ 219
Calling Turbo Assembler functions from Borland C++ . . . 219
The framework . . . 221
Linking assembly language modules with C++ . . . 221
Memory models and segments . . . 223
Publics and externals . . . 229
Linker command line . . . 233
Parameter passing . . . 233
Preserving registers . . . 240
Returning values . . . 241
Calling an assembler function from C++ . . . 242
Writing C++ member functions in assembly language . . . 245
Pascal calling conventions . . . 248
Calling Borland C++ from Turbo Assembler . . 249
Link in the C++ startup code . . . 249

The segment setup 250
Performing the call 250
Calling a Borland C++ function from Turbo Assembler 251

Appendix A Program blueprints 255
Simplified segmentation segment description . 255
OS/2 programs 256
OS/2 flat-model program blueprint 256

Appendix B Turbo Assembler syntax summary 259
Lexical grammar 259
MASM mode expression grammar 261
Ideal mode expression grammar 263
Keyword precedence 266
Ideal mode precedence 266
MASM mode precedence 266
Keywords and predefined symbols 267
Directive keywords 267

Appendix C Compatibility issues 273
One-pass versus two-pass assembly 273
Environment variables 274
Microsoft binary floating-point format 274

Appendix D Error messages 275
Information messages 275
Warning and error messages 276

Index 299

Tables

4.1 Object-oriented programming terminology . 49
4.2 Symbols defined for objects 50
5.1 Radixes 65
5.2 Characters determining radixes 66
5.3 Numeric constants 66
5.4 Symbol types 67
5.5 Address subtypes 68
5.6 Complex address subtypes 69
5.7 Distance syntax 69
5.8 Simple expressions 70
5.9 Standard symbols 71
5.10 Values of symbols used by themselves 71
5.11 LENGTH operator return values 72
5.12 SIZE values 73
5.13 WIDTH values 73
5.14 MASK return values 74
5.15 Simple arithmetic operators 74
5.16 Logical arithmetic operators 74
5.17 Bit shift operators 75
5.18 Comparison operators 75
5.19 Type override operators 75
5.20 TYPE values 76
5.21 Bit fields from SYMTYPE and .TYPE 79
6.1 Processor directives 83
6.2 8087 coprocessor directives 86
7.1 Standard memory models 91
7.2 Model modifiers 92
7.3 Model modifiers 94
7.4 Simplified segment directives 94
7.5 Symbols from simplified segment directives 95
7.6 Segment combination attribute 97
7.7 Segment alignment attribute 98
7.8 Segment size attribute values 98
7.9 Segment access attribute 99
7.10 Stack size modification directives 103
8.1 STRUC, UNION, and ENDS directives 110
8.2 Block members 111
8.3 Available modifiers 117
8.4 Symbols used or defined by STRUC 118
12.1 Data size directives 146
13.1 Intelligent code generation directives 159
13.2 Return instructions 162
13.3 Segment override instructions 166
13.4 Smart flag instructions 167
13.5 Instructions for setting and retrieving values 167
13.6 Instructions affected by SMALL and LARGE 170
14.1 Dummy argument types 182
14.2 Uses for the ! character 184
15.1 Conditional assembly directives using expressions 197
15.2 Error-generation directives using expressions 197
15.3 Evaluation of defined and undefined symbol 197
15.4 Symbol-expression directives using symbol_expr 198
15.5 Error-generation directives 198
15.6 Conditional assembly directives using text_strings 199
15.7 Error-generation directives using text_strings 199
18.1 Register settings when Borland C++ enters assembler 227
A.1 Default segments and types for TINY memory model 255
A.2 Default segments and types for SMALL memory model 255
A.3 Default segments and types for MEDIUM memory model 255
A.4 Default segments and types for COMPACT memory model 256
A.5 Default segments and types for LARGE or HUGE memory model 256
A.6 Default segments and types for Borland C++ HUGE (TCHUGE) memory model 256
B.1 Turbo Assember V1.0 (VERSION T100) keywords 267
B.2 Turbo Assembler V2.0 (VERSION T200) new keywords 270
B.3 Turbo Assembler V2.5 (VERSION T250) new keywords 270
B.4 Turbo Assembler V3.0 (VERSION T300) new keywords 271
B.5 Turbo Assembler V3.1 (VERSION T310) new keywords 271

B.6 Turbo Assembler V3.2 (VERSION T320) new keywords 271
B.7 Turbo Assembler V4.0 (VERSION T400) new keywords 271

Figures

1.1 The edit, assemble, link, and run cycle 10
2.1 Turbo Assembler command line 13
10.1 How language affects procedures 129
18.1 Compile, assemble, and link with Borland C++, Turbo Assembler, and TLINK 220
18.2 State of the stack just before executing Test's first instruction 235
18.3 State of the stack after PUSH and MOV .. 236
18.4 State of the stack after PUSH, MOV, and SUB 237
18.5 State of the stack immediately after MOV BP, SP 249

Introduction

Welcome to Borland's Turbo Assembler, a multi-pass assembler with forward-reference resolution, assembly speeds of up to 48,000 lines per minute (on an IBM PS/2 model 60), Microsoft Macro Assembler (MASM) compatibility, and an optional Ideal mode extended syntax. Whether you're a novice or an experienced programmer, you'll appreciate these features and others we've provided to make programming in assembly language easier. Here are the highlights—we'll describe them in detail later:

OOP

- Object-oriented programming capabilities
- 32-bit model and stack frame support
- Full 386, i486, and Pentium support
- Simplified segmentation directives
- Table support
- Enumerations
- Smart flag instructions
- Fast immediate multiply operation
- Multiline definition support
- **VERSION** specification directive
- Nested directives
- Quirks mode to emulate MASM
- Full source debugging output
- Cross-reference utility (TCREF)
- Configuration and command files
- File converter utility (converts C .h files to TASM .ash files)
- Procedure prototyping and argument checking capabilities
- Alias support

Turbo Assembler is a powerful command-line assembler that takes your source (.ASM) files and produces object (.OBJ) modules. You then use TLINK.EXE, Borland's high-speed linker program, to link your object modules and create executable (.EXE) files.

Hardware and software requirements

➡ Turbo Assembler runs on the IBM PC family of computers, including the XT, AT, and PS/2, along with all true compatibles.

Turbo Assembler generates instructions for the 8086, 80186, 80286, 386, i486, and Pentium processors. It also generates floating-point instructions for the 8087, 80287, and 387 numeric coprocessors. (For more information about the instruction sets of the 80x86/80x87 families, consult the Intel data books.)

About the manuals

Turbo Assembler comes with the *Turbo Assembler User's Guide* (this book) and the *Turbo Assembler Quick Reference Guide*. The *User's Guide* provides basic instructions for using Turbo Assembler, explores how to interface Turbo Assembler with other languages, and describes in detail the operators, predefined symbols, and directives Turbo Assembler uses. The *Quick Reference Guide* is a handy guide to directives and processor and coprocessor instructions.

Here's a more detailed look at what the *User's Guide* contains.

Chapter 1: Getting started with Turbo Assembler tells you how to install Turbo Assembler on your system

Chapter 2: Using directives and switches describes how you can control the way the assembler runs when you use directives and switches.

Chapter 3: General programming concepts discusses the differences between Ideal and MASM modes, how to use predefined symbols, using comment characters, and so forth.

OOP

Chapter 4: Creating object-oriented programs describes how you can use object-oriented programming techniques in assembly language.

Chapter 5: Using expressions and symbol values talks about evaluating and defining expressions and operators.

Chapter 6: Choosing processor directives and symbols tells you how to generate code for particular processors.

Chapter 7: Using program models and segmentation talks about program models, creating symbols, simplified segments, and ordering of segments.

Chapter 8: Defining data types explains how to define structures, unions, tables, bit-field records, and objects.

Chapter 9: Setting and using the location counter describes how and why you'd want to use the location counter, as well as how to define labels.

Chapter 10: Declaring procedures examines how to use various types of procedures, and how to define and use arguments and local variables.

Chapter 11: Controlling the scope of symbols discusses how you can limit or expand the area in which a symbol has a particular value.

Chapter 12: Allocating data describes simple data directives, and how to create instances of structures, unions, records, enumerated data types, tables, and objects.

Chapter 13: Advanced coding instructions covers Turbo Assembler's extended instructions, including prototyping and calling language procedures.

Chapter 14: Using macros tells you how to use macros in your code.

Chapter 15: Using conditional directives talks about the directives that let you execute your code conditionally.

Chapter 16: Interfacing with the linker describes how you can include libraries and publish symbols as you link your code.

Chapter 17: Generating a listing talks about Turbo Assembler listing files and how to use them.

Chapter 18: Interfacing Turbo Assembler with Borland C++ explains how to use Borland's line of C++ compilers with assembly language.

Appendix A: Program blueprints contains examples of different types of program structures.

Appendix B: Turbo Assembler syntax summary illustrates Turbo Assembler expressions (both MASM and Ideal modes) in modified Backus-Naur form (BNF).

Appendix C: Compatibility issues covers the differences between MASM and Turbo Assembler MASM mode.

Appendix D: Error messages describes all the error messages that can be generated when using Turbo Assembler: information messages, fatal error messages, warning messages, and error messages.

Notational conventions

When we talk about IBM PCs or compatibles, we're referring to any computer that uses the 8088, 8086, 80186, 80286, 386, and i486 chips (all of these chips are commonly referred to as 80x86).

All typefaces were produced by Borland's Sprint: The Professional Word Processor, output on a PostScript printer. The different typefaces displayed are used for the following purposes:

Italics	In text, italics represent labels, placeholders, variables, and arrays. In syntax expressions, placeholders are set in italics to indicate they are user-defined.
Boldface	Boldface is used in text for directives, instructions, symbols, and operators, as well as for command-line options.
CAPITALS	In text, capital letters are used to represent instructions, directives, registers, and operators.
`Monospace`	Monospace type is used to display any sample code or text that appears on your screen, and any text that you must actually type to assemble, link, and run a program.
Keycaps	In text, keycaps indicate a key on your keyboard. It is often used when describing a key you must press to perform a particular function; for example, "Press *Enter* after typing your program name at the prompt."

Contacting Borland

The Borland Assist program offers a range of services to fit the different needs of individuals, consultants, large corporations, and developers. To receive help with your questions about our products, send in the registration card. North American customers can register by phone 24 hours a day by calling 1-800-845-0147.

Borland Assist plans

Borland Assist is made up of three levels of support:

- Standard Assist gives all registered users assistance with installation and configuration, and offers automated and online services to answer other product questions (see the following table).
- Enhanced Assist plans are designed for individuals who need unlimited support on a toll-free number or priority hotline access.

- Premium Assist plans are designed to support large corporations and software developers.

Available at no charge, Standard Assist offers all registered users the following services:

Service	How to contact	Cost	Available	Description
Installation hotline	408-461-9133	The cost of the phone call	6:00am – 5:00pm PST Monday – Friday	Provides assistance on product installation and configuration.
Automated support	Voice: 1-800-524-8420 Modem: 408-431-5250	Free The cost of the phone call	24 hours daily	Provides answers to common questions. Requires a Touch-Tone phone or modem.
TechFax	1-800-822-4269 (voice)	Free	24 hours daily	Sends technical information to your fax machine (up to 3 documents per call). Requires a Touch-Tone phone. Document #1 is the catalog of available catalogs.
Online services				
Borland Download BBS	408-431-5096	The cost of the phone call	24 hours daily	Sends sample files, applications, and technical information via your modem. Requires a modem (up to 9600 baud).
CompuServe	Type `GO BORLAND`. Address messages to Sysop or All.	Your online charges	24 hours daily; 1-working-day response time	Sends answers to technical questions via your modem. Messages are public.
BIX	Type `JOIN BORLAND`. Address messages to Sysop or All.	Your online charges	24 hours daily; 1-working-day response time	Sends answers to technical questions via your modem. Messages are public.
GEnie	Type `BORLAND`. Address messages to All.	Your online charges	24 hours daily; 1-working-day response time	Sends answers to technical questions via your modem. Messages are public.

For additional details on these and other Borland services, see the *Borland Assist Support and Services Guide* included with your product.

CHAPTER 1

Getting started with Turbo Assembler

You might have heard that programming in assembly language is a black art suited only to hackers and wizards. However, assembly language is nothing more than the human form of the language of the computer. And, as you'd expect, the computer's language is highly logical. As you might also expect, assembly language is very powerful—in fact, assembly language is the only way to tap the full power of the Intel 80x86 family, the processors at the heart of the IBM PC family and compatibles.

You can write whole programs using nothing but assembly language or you can mix assembly language with programs written in high-level languages such as Borland C++ and Borland Pascal. Either way, assembly language lets you write small and blindingly fast programs. In addition to the advantage of speed, assembly language gives you the ability to control every aspect of your computer's operation, all the way down to the last tick of the computer's system clock.

Installing Turbo Assembler

The Turbo Assembler package consists of a set of executable programs, utilities, and example programs. In addition, the package includes a *Quick Reference Guide* and this *User's Guide*.

For instructions on installing Turbo Assembler, refer to the `INSTALL.TXT` file on your installation disk:

1. Insert the TASM Install disk in drive A of your computer.
2. User your text editor to open INSTALL.TXT, or issue the following command at the command line:

```
TYPE A:INSTALL.TXT | MORE
```

Utility and example programs

The Turbo Assembler package includes several utility programs to help you build assembly programs. For a complete list of the utilities included with Turbo Assembler, refer to the online text file INSTALL.TXT. For instructions on using the utilities, refer to the text file UTILS.TSM.

To get you started writing assembler programs, the Turbo Assembler package includes various example programs that demonstrate different assembler programming techniques. For a complete listing of the example programs, refer to the online text file INSTALL.TXT.

Online help

You can get online help for Turbo Assembler using the OS/2 help facility. During the Turbo Assembler installation, the installation program creates a folder for the Turbo Assembler help file. You can access this folder from any of three locations:

- If you're in an OS/2 window, open the folder. Click the icon labeled "TASM Reference" to access the Turbo Assembler help file.
- If you're running OS/2 in full-screen mode, press *ALT+ESC*. The Turbo Assembler folder appears. Open the folder, and click the TASM Reference icon.
- If you're in the Borland C++ IDE, open the Turbo Assembler folder, and click the TASM Reference icon.

Writing your first Turbo Assembler program

If you have not yet written an assembly program, the following "Greetings, World!" program is a good place to start. To begin writing this program, open your favorite program editor and enter the following lines of code to create the HELLO.ASM program:

```
ideal
p386

model flat

codeseg
extrn DOSEXIT:near,DOSWRITE:near

stack 800h

dataseg
```

```
; Handle for printing to the console
HANDLE_CON    = 1

; The message to print, and its length
message       db 13,10,'Greetings, World!',12,10
messagelength = $-message-1

; Storage for number of characters written
written dd 0

codeseg
start:

        ; To do output, use the file handle to the console,
        ; and send the output there.

        ; Note that for OS/2 flat model, you don't need to
        ; do anything with segment registers. Just push
        ; the offset of the items.

        ; The OS/2 system calls are C-style, so the caller
        ; must clean up the stack after the call.

        ; DOSWRITE (FileHandle, pBufferArea, ulBufferlength,
        ; pBytesWritten)

        call DOSWRITE C,        \
                HANDLE_CON,     \ ; Handle is a doubleword
                offset message, \ ; Location of message to print
                messagelength,  \ ; Doubleword length of message
                offset written    ; Storage for # chars written.

        ; Exit the program now

        call    DOSEXIT C,0,1

end start
```

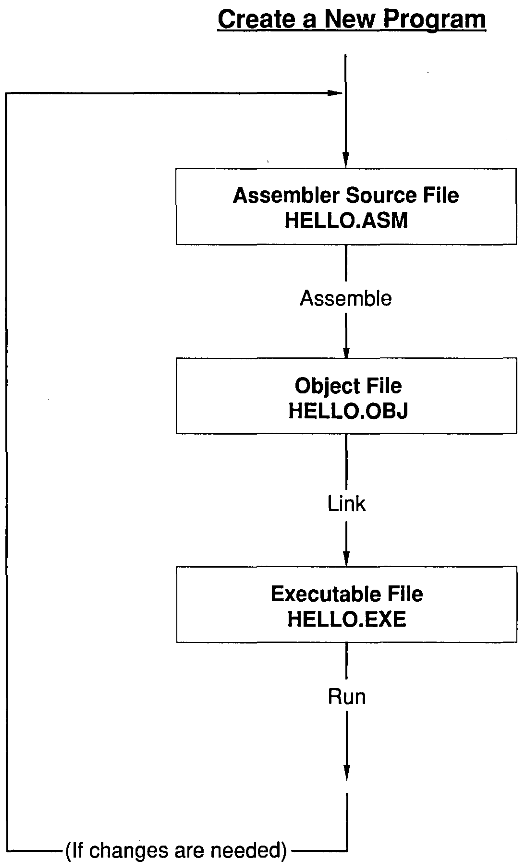
After you've entered the preceding program, save it to disk as HELLO.ASM.

If you're familiar with high-level languages (such as C, C++, or Pascal), you might think that HELLO.ASM is a bit long for a "Greetings, World!" program. Indeed, assembler programs tend to be much longer than high-level language programs because each high-level language statement actually breaks down to form many assembler instructions. However, assembly language gives you complete freedom over the actual instructions that are given to the computer's CPU. With assembly language, you can write programs that tell the computer to do *anything* that it's capable of doing.

Assembling your first program

Now that you've saved HELLO.ASM, you'll want to run it. However, before you can run it, you'll have to assemble it into an .OBJ file, and then link the file to form an executable program. This program development cycle is shown in Figure 1.1.

Figure 1.1
The edit, assemble, link, and run cycle



The assembly step turns your source code into an intermediate form called an *object module*, and the linking step combines one or more object modules into an executable program. You can do your assembling and linking from the command line.

To assemble HELLO.ASM, type the following line at the DOS command line:

```
TASM hello
```

Unless you specify another file name, HELLO.ASM will be assembled to form the object file HELLO.OBJ. (Note that you don't need to type in the file extension name; Turbo Assembler assumes all source files end with .ASM.) If you entered the HELLO.ASM program correctly, you'll see a listing similar to the following one displayed onscreen:

```
Turbo Assembler Version 4.1 Copyright (c) 1992 by Borland International, Inc.

Assembling file:  HELLO.ASM
Error messages:   None
Warning messages: None
Passes: 1
Remaining memory: 266K
```

If you get warnings or errors, they are displayed with the program line numbers to indicate where they occurred. If you do get errors, edit HELLO.ASM make sure it's precisely the same as the program shown above. After editing the program, reassemble it with the `TASM hello` command.

Linking your first program

After you've successfully assembled HELLO.ASM, you'll need to link the program using TLINK. At the command line, type:

```
TLINK hello,,,OS2.LIB;
```

If no errors or warnings are reported, an executable file is created, named HELLO.EXE. To run this program, enter the command `HELLO` from the DOS command line.

Errors can occur during the linking process, although it's unlikely with this example program. If you do receive linker errors, modify your code to exactly match the code shown here, then assemble and link again.

Recommended reading

Although HELLO.ASM is a good program for testing TASM.EXE and TLINK.EXE, the example is of little use if you're trying to learn assembly language. However, many books are available that teach both the fundamentals and the advanced features of assembly language. To help you get started with assembly language, refer to one or more of the following book titles:

- Duntemann, Jeff. *Assembly Language from Square One: For the PC AT and Compatibles*. Glenview, IL: Scott, Foresman and Company, 1990
- Hummel, Robert L. *Programmers Technical Reference: Processor and coprocessor*. Emeryville, CA: Ziff Davis press, 1992

- Mischel, Jim. *Macro Magic with Turbo Assembler*. New York, NY: John Wiley & Sons, 1993
- Swan, Tom. *Mastering Turbo Assembler*. Carmel, IN: Howard W. Sams and Co., 1989.
- Syck, Gary. *The Waite Group's Turbo Assembler Bible.* Carmel, IN: Howard W. Sams and Co., 1990.

In addition to these books, Intel Corporation offers fact sheets and reference manuals on the workings of their processor products. For more information, contact Intel at the following address:

Intel Literature Sales
P.O. Box 7641
Mount Prospect, IL 60056-7641

C H A P T E R 2

Using directives and switches

This chapter is dedicated to familiarizing you with Turbo Assembler's command-line options. We'll describe each of the command-line options you can use to alter the assembler's behavior, and then show how and when to use command files. We'll also describe the configuration file, and how you can control the display of warning and error messages.

Starting Turbo Assembler

If you start Turbo Assembler from your operating system command line without giving it any arguments, like this,

```
TASM
```

you'll get a screenful of help describing many of the command-line options, and the syntax for specifying the files you want to assemble. Figure 2.1 shows you how this looks.

Figure 2.1
Turbo Assembler command line

```
Turbo Assembler  Version 4.1  Copyright (c) 1988, 1993 Borland International
Syntax:  TASM [options] source [,object] [,listing] [,xref]
/a,/s            Alphabetic or Source-code segment ordering
/c               Generate cross-reference in listing
/dSYM[=VAL]      Define symbol SYM = 0, or = value VAL
/e,/r            Emulated or Real floating-point instructions
/h,/?            Display this help screen
/iPATH           Search PATH for include files
/jCMD            Jam in an assembler directive CMD (e.g. /jIDEAL)
/kh#             Hash table capacity # symbols
/l,/la           Generate listing: l=normal listing, la=expanded listing
/ml,/mx,/mu      Case sensitivity on symbols: ml=all, mx=globals, mu=none
/mv#             Set maximum valid length for symbols
/m#              Allow # multiple passes to resolve forward references
/n               Suppress symbol tables in listing
/os,/o,/op,/oi   Object code: standard, standard w/overlays, Phar Lap, or IBM
/p               Check for code segment overrides in protected mode
/q               Suppress OBJ records not needed for linking
/t               Suppress messages if successful assembly
/uxxxx           Set version emulation, version xxxx
```

```
/w0,/w1,/w2       Set warning level: w0=none, w1=w2=warnings on
/w-xxx,/w+xxx     Disable (-) or enable (+) warning xxx
/x                Include false conditionals in listing
/z                Display source line with error message
/zi,/zd,/zn       Debug info: zi=full, zd=line numbers only, zn=none
```

With the command-line options, you can specify the name of one or more files that you want to assemble, as well as any options that control how the files get assembled.

The general form of the command line looks like this:

```
TASM fileset [; fileset]...
```

The semicolon (;) after the left bracket ([) lets you assemble multiple groups of files on one command line by separating the file groups. If you prefer, you can set different options for each set of files; for example,

```
TASM /e FILE1; /a FILE2
```

assembles FILE1.ASM with the **/e** command-line option and assembles file FILE2.ASM with the **/a** command-line option.

In the general form of the command line, *fileset* can be

```
[option]...sourcefile [[+] sourcefile]...
           [,[objfile] [, [listfile] [, [xreffile]]]]
```

This syntax shows that a group of files can start off with any options you want to apply to those files, followed by the files you want to assemble. A file name can be a single file name, or it can use the normal wildcard characters * and ? to specify multiple files to assemble. If your file name does not have an extension, Turbo Assembler adds the .ASM extension. For example, to assemble all the .ASM files in the current directory, you would type

```
TASM *
```

If you want to assemble multiple files, you can separate their names with the plus sign (+):

```
TASM MYFILE1 + MYFILE2
```

You can follow the file name you want to assemble by an optional object file name, listing file name, and a cross-reference file name. If you do not specify an object file or listing file, Turbo Assembler creates an object file with the same name as the source file and an extension of .OBJ.

A listing file is not generated unless you explicitly request one. To request one, place a comma after the object file name, followed by a listing file name. If you don't explicitly provide a listing file name, Turbo Assembler creates a listing file with the same name as the source file and the extension .LST. If you supply a listing file name without an extension, .LST is appended to it.

A cross-reference file is not generated unless you explicitly request one. To request one, place a comma after the listing file name, followed by a cross-reference file name. If you don't explicitly provide a cross-reference file name, Turbo Assembler creates a cross-reference file with the same name as the source file and the extension .XRF. If you supply a cross-reference file name without an extension, .XRF is appended to it. (TCREF, a cross-reference utility, is described on disk.)

If you want to accept the default object file name and also request a listing file, you must supply the comma that separates the object file name from the listing file name:

```
TASM FILE1,,TEST
```

This assembles FILE1.ASM to FILE1.OBJ and creates a listing file named TEST.LST.

If you want to accept the default object and listing file names and also request a cross-reference file, you must supply the commas that separate the file names:

```
TASM MYFILE,,,MYXREF
```

This assembles file MYFILE.ASM to MYFILE.OBJ, with a listing in file MYFILE.LST and a cross-reference in MYXREF.XRF.

If you use wildcards to specify the source files to assemble, you can also use wildcards to indicate the object and listing file names. For example, if your current directory contains XX1.ASM and XX2.ASM, the command line

```
TASM XX*,YY*
```

assembles all the files that start with *XX*, generates object files that start with *YY*, and derives the remainder of the name from the source file name. The resulting object files are therefore called YY1.OBJ and YY2.OBJ.

If you don't want an object file but you do want a listing file, or if you want a cross-reference file but don't want a listing file or object file, you can specify the null device (NUL) as the file name. For example,

```
TASM FILE1,,NUL,
```

assembles file FILE1.ASM to object file FILE1.OBJ, doesn't produce a listing file, and creates a cross-reference file FILE1.XRF.

Command-line options

The command-line options let you control the behavior of the assembler, and how it outputs information to the screen, listing, and object file. Turbo Assembler provides you with some options that produce no action, but are accepted for compatibility with the current and previous versions of MASM:

/b Sets buffer size
/v Displays extra statistics

You can enter options using any combination of uppercase and lowercase letters. You can also enter your options in any order except where you have multiple **/i** or **/j** options; these are processed in sequence. When using the **/d** option, you must also be careful to define symbols before using them in subsequent **/d** options.

➡ You can override command-line options by using conflicting directives in your source code.

Figure 2.1 on page 13 summarizes the Turbo Assembler command-line options; here's a detailed description of each option.

/a

Function Specifies alphabetical segment-ordering

Syntax

```
/a
```

Remarks The **/a** option tells Turbo Assembler to place segments in the object file in alphabetical order. This is the same as using the **.ALPHA** directive in your source file.

You usually only have to use this option if you want to assemble a source file that was written for very early versions of the IBM or Microsoft assemblers.

The **/s** option reverses the effect of this option by returning to the default sequential segment-ordering.

If you specify sequential segment-ordering with the **.SEQ** directive in your source file, it will override any **/a** you provide on the command line.

Example

```
TASM /a TEST1
```

This command line creates an object file, TEST1.OBJ, that has its segments in alphabetical order.

/b

Syntax

```
/b
```

Remarks

The **/b** option is included for compatibility. It performs no action and has no effect on the assembly.

/c

Function

Enables cross-reference in listing file

Syntax

```
/c
```

Remarks

The **/c** option enables cross-reference information in the listing file. Turbo Assembler adds the cross-reference information to the symbol table at the end of the listing file. This means that, in order to see the cross-reference information, you must either explicitly specify a listing file on the command line or use the **/l** option to enable the listing file.

For each symbol, the cross-reference shows the line on which it is defined and all lines that refer to it.

Example

```
TASM /l /c TEST1
```

This code creates a listing file that also has cross-reference information in the symbol table.

/d

Function

Defines a symbol

Syntax

/d*symbol*[=*value* or *expression*]

Remarks

The **/d** option defines a symbol for your source file, exactly as if it were defined on the first line of your file with the **=** directive. You can use this option as many times as you want on the command line.

You can only define a symbol as being equal to another symbol or a constant value. You can't use an expression with operators to the right of the equal sign (=). For example, **/d***X=9* and **/d***X=Y* are allowed, but **/d***X=Y-4* is not.

Example

```
TASM /dMAX=10 /dMIN=2 TEST1
```

This command line defines two symbols, **MAX** and **MIN**, that other statements in the source file TEST1.ASM can refer to.

/e

Function Generates floating-point emulator instructions

Syntax `/e`

Remarks The **/e** option tells Turbo Assembler to generate floating-point instructions that will be executed by a software floating-point emulator. Use this option if your program contains a floating-point emulation library that mimics the functions of the 80x87 numeric coprocessor.

Normally, you would only use this option if your assembler module is part of a program written in a high-level language that uses a floating-point emulation library. (Borland's line of C++ compilers, Borland Pascal, Turbo Basic, and Turbo Prolog all support floating-point emulation.) You can't just link an assembler program with the emulation library, since the library expects to have been initialized by the compiler's startup code.

The **/r** option reverses the effect of this option by enabling the assembly of real floating-point instructions that can only be executed by a numeric coprocessor.

If you use the **NOEMUL** directive in your source file, it will override the **/e** option on the command line.

The **/e** command-line option has the same effect as using the **EMUL** directive at the start of your source file, and is also the same as using the **/jEMUL** command-line option.

Example

```
TASM /e SECANT
TCC -f TRIG.C SECANT.OBJ
```

The first command line assembles a module with emulated floating-point instructions. The second command line compiles a C source module with floating-point emulation and then links it with the object file from the assembler.

/h or /?

Function Displays a help screen

Syntax `/h` or `/?`

Remarks The **/h** option tells Turbo Assembler to display a help screen that describes the command-line syntax. This includes a list of the options, as well as the various file names you can supply. The **/?** option does the same thing.

Example

```
TASM /h
```

/i

Function Sets an include file path

Syntax `/iPATH`

Remarks The **/i** option lets you tell Turbo Assembler where to look for files that are included in your source file by using the **INCLUDE** directive. You can place more than one **/i** option on the command line (the number is only limited by RAM).

When Turbo Assembler encounters an **INCLUDE** directive, the location where it searches for the include file is determined by whether the file name in the **INCLUDE** directive has a directory path or is just a simple file name.

If you supply a directory path as part of the file name, that path is tried first, then Turbo Assembler searches the directories specified by **/i** command-line options in the order they appear on the command line. It then looks in any directories specified by **/i** options in a configuration file.

If you don't supply a directory path as part of the file name, Turbo Assembler searches first in the directories specified by **/i** command-line options, then it looks in any directories specified by **/i** options in a configuration file, and finally it looks in the current directory.

Example

```
TASM /i\INCLUDE /iD:\INCLUDE TEST1
```

If the source file contains the statement

```
INCLUDE MYMACS.INC
```

Turbo Assembler will first look for \INCLUDE\MYMACS.INC, then it will look for D:\INCLUDE\MYMACS.INC. If it still hasn't found the file, it will look for MYMACS.INC in the current directory. If the statement in your source file had been

```
INCLUDE INCS\MYMACS.INC
```

Turbo Assembler would first look for INCS\MYMACS.INC and then it would look for \INCLUDE\MYMACS.INC, and finally for D:\INCLUDE\MYMACS.INC.

/j

Function Defines an assembler startup directive

Syntax `/jdirective`

Remarks

The **/j** option lets you specify a directive that will be assembled before the first line of the source file. *directive* can be any Turbo Assembler directive that does not take any arguments, such as **.286, IDEAL, %MACS, NOJUMPS,** and so on.

You can put more than one **/j** option on the command line; they are processed from left to right across the command line.

Example

```
TASM /j.286 /jIDEAL TEST1
```

This code assembles the file TEST1.ASM with 80286 instructions enabled and Ideal mode expression-parsing enabled.

/kh

Function

Sets the maximum number of symbols allowed

Syntax

/kh*nsymbols*

Remarks

The **/kh** option sets the maximum number of symbols that your program can contain. If you don't use this option, your program can only have a maximum of 8,192 symbols; using this option increases the number of symbols to *nsymbols*, up to a maximum of 32,768.

Use this option if you get the `Out of hash space` message when assembling your program.

You can also use this option to reduce the total number of symbols below the default 8,192. This releases some memory that can be used when you are trying to assemble a program but don't have enough available memory.

Example

```
TASM /kh10000 BIGFILE
```

This command tells Turbo Assembler to reserve space for 10,000 symbols when assembling the file BIGFILE.

/l

Function

Generates a listing file

Syntax

/l

Remarks

The **/l** option indicates that you want a listing file, even if you did not explicitly specify it on the command line. The listing file will have the same name as the source file, with an extension of .LST.

Example

```
TASM /l TEST1
```

This command line requests a listing file that will be named TEST1.LST.

/la

Function Shows high-level interface code in listing file

Syntax

```
/la
```

Remarks The **/la** option tells Turbo Assembler to show all generated code in the listing file, including the code that gets generated as a result of the high-level language interface **.MODEL** directive.

Example

```
TASM /la FILE1
```

/m

Function Sets the maximum number of assembly passes

Syntax

```
/m[npasses]
```

Remarks Normally, Turbo Assembler functions as a single-pass assembler. The **/m** option lets you specify the maximum number of passes the assembler should make during the assembly process. TASM automatically decides whether it can perform less than the number of passes specified. If you select the **/m** option, but don't specify *n*passes, a default of five is used.

You might want to specify multiple passes either if you want Turbo Assembler to remove **NOP** instructions added because of forward references or if you are assembling a module containing instructions that require two passes. If multiple passes are not enabled, such a module will produce at least one "Pass-dependent construction encountered" warning. If the **/m** option is enabled, Turbo Assembler assembles this module correctly but will not optimize the code by removing **NOP**s, no matter how many passes are allowed. The warning "Module is pass dependent—compatibility pass was done" is displayed if this occurs.

Example

```
TASM /M2 TEST1
```

This tells Turbo Assembler to use up to two passes when assembling TEST1.

/ml

Function Treats symbols as case-sensitive

Syntax

```
/ml
```

Remarks

The **/ml** option tells Turbo Assembler to treat all symbol names as case-sensitive. Normally, uppercase and lowercase letters are considered equivalent so that the names *ABCxyz*, *abcxyz*, and *ABCXYZ* would all refer to the same symbol. If you specify the **/ml** option, these three symbols will be treated as distinct. Even when you specify **/ml**, you can still enter any assembler keyword in uppercase or lowercase. Keywords are the symbols built into the assembler that have special meanings, such as instruction mnemonics, directives, and operators.

Example

```
TASM /ml TEST1
```

where TEST1.ASM contains the following statements:

```
abc DW 0
ABC DW 1            ;not a duplicate symbol
    Mov Ax,[Bp]     ;mixed case OK in keywords
```

The **/ml** switch used together with **/mx** has a special meaning for Pascal symbols. See the **/mx** section for further details.

/mu

Function

Converts symbols to uppercase

Syntax

```
/mu
```

Remarks

The **/mu** option tells Turbo Assembler to ignore the case of all symbols. By default, Turbo Assembler specifies that any lowercase letters in symbols will be converted to uppercase unless you change it by using the **/ml** directive.

Example

```
TASM /mu TEST1
```

makes sure that all symbols are converted to uppercase (which is the default):

```
EXTRN myfunc:NEAR
call myfunc         ;don't know if declared as
                    ; MYFUNC, Myfunc,...
```

/mv#

Function

Sets the maximum length of symbols.

Syntax

```
/mv#
```

Remarks

The **/mv#** option sets the maximum length of symbols that TASM will distinguish between. For example, if you set **/mv12**, TASM will see *ABCDEFGHIJKLM* and *ABCDEFGHIJIKLL* as the same symbol, but not *ABCDEFGHIJKL*. Note that the minimum number you can have here is 12.

/mx

Function

Makes public and external symbols case-sensitive

Syntax

```
/mx
```

Remarks

The **/mx** option tells Turbo Assembler to treat only external and public symbols as case-sensitive. All other symbols used (within the source file) are treated as uppercase.

You should use this directive when you call routines in other modules that were compiled or assembled so that case-sensitivity is preserved; for example, modules compiled by one of Borland's line of C++ compilers.

Example

```
TASM /mx TEST1;
```

where TEST1.ASM contains the following source lines:

```
EXTRN Cfunc:NEAR
myproc PROC NEAR
call  Cfunc
⋮
```

Note: using the **/mx** and **/ml** options together has a special meaning for symbols declared as Pascal; if you use these symbols together, the symbols will be published as all uppercase to the linker.

/n

Function

Suppresses symbol table in listing file

Syntax

```
/n
```

Remarks

The **/n** option indicates that you don't want the usual symbol table at the end of the listing file. Normally, a complete symbol table listing appears at the end of the file, showing all symbols, their types, and their values.

You must specify a listing file, either explicitly on the command line or by using the **/l** option; otherwise, **/n** has no effect.

Example

```
TASM /l /n TEST1
```

This code generates a listing file showing the generated code only, and not the value of your symbols.

/o

Function Generates overlay code for TLINK

Syntax `/o`

Remarks Specifying the **/o** switch on the command line causes overlay-compatible fixups to be generated. When this switch is used, 386 references to **USE32** segments should not be made since they won't link properly.

/oi

Function Generates overlay code for the IBM linker

Syntax `/oi`

Remarks Specifying the **/oi** switch on the command will generate overlay-compatible fixups for the IBM linker. The resulting object file will not be compatible with TLINK, Borland's linker.

/op

Function Generates overlay code for the Phar Lap linker

Syntax `/op`

Remarks Specifying the **/op** switch on the command will generate overlay-compatible fixups for the Phar Lap linker. The resulting object file will not be compatible with TLINK, Borland's linker.

/os

Function Outputs TLINK-compatible objects without overlay support. This is the default selection.

Syntax `/os`

Remarks

Specifying the **/os** switch on the command will generate objects without overlay support for use with TLINK.

/p

Function

Checks for impure code in protected mode

Syntax

```
/p
```

Remarks

The **/p** option specifies that you want to be warned about any instructions that generate "impure" code in protected mode. Instructions that move data into memory by using a **CS:** override in protected mode are considered impure because they might not work correctly unless you take special measures.

You only need to use this option if you are writing a program that runs in protected mode on the 80286, 386, or i486.

Example

```
TASM /p TEST1
```

where TEST1.ASM contains the following statements:

```
     .286P
CODE SEGMENT
temp DW  ?
     mov CS:temp,0    ;impure in protected mode
```

/q

Function

Suppresses .OBJ records not needed for linking

Syntax

```
/q
```

Remarks

The **/q** option removes the copyright and file dependency records from the resulting .OBJ files, making it smaller. Don't use this option if you are using MAKE or a similar program that relies on the dependency records.

/r

Function

Generates real floating-point instructions

Syntax

```
/r
```

Remarks

The **/r** option tells Turbo Assembler to generate real floating-point instructions (instead of generating emulated floating-point instructions).

Use this option if your program is going to run on machines equipped with an 80x87 numeric coprocessor.

The **/e** option reverses the effect of this option in generating emulated floating-point instructions.

If you use the **EMUL** directive in your source file, it will override the **/r** option on the command line.

The **/r** command-line option has the same effect as using the **NOEMUL** directive at the start of your source file, and is also the same as using the **/jNOEMUL** command-line option.

Example

```
TASM /r SECANT
TPC /$N+ /$E- TRIG.PAS
```

The first command line assembles a module with real floating-point instructions. The second compiles a Pascal source module with real floating-point instructions that links in the object file from the assembler.

/s

Function Specifies sequential segment-ordering

Syntax `/s`

Remarks The **/s** option tells Turbo Assembler to place segments in the object file in the order in which they were encountered in the source file. By default, Turbo Assembler uses segment-ordering, unless you change it by placing an **/a** option in the configuration file.

If you specify alphabetical segment-ordering in your source file with the **.ALPHA** directive, it will override **/s** on the command line.

Example

```
TASM /s TEST1
```

This code creates an object file (TEST1.OBJ) that has its segments ordered exactly as they were specified in the source file.

/t

Function Suppresses messages on successful assembly

Syntax `/t`

Remarks The **/t** option stops any display by Turbo Assembler unless warning or error messages result from the assembly.

You can use this option when you are assembling many modules, and you only want warning or error messages to be displayed onscreen.

Example

```
TASM /t TEST1
```

/u

Function
Sets version ID in command line

Syntax

```
/u version
```

Remarks
The **/u** option lets you specify which version of Turbo Assembler or MASM you want to use to run your modules. This is the command-line version of the **VERSION** directive.

/v

Syntax

```
/v
```

Remarks
The **/v** option is included for compatibility. It performs no action and has no effect on the assembly.

/w

Function
Controls the generation of warning messages

Syntax

```
/w
w-[warnclass]
w+[warnclass]
```

Remarks
The **/w** option controls which warning messages are emitted by Turbo Assembler.

If you specify **/w** by itself, "mild" warnings are enabled. Mild warnings merely indicate that you can improve some aspect of your code's efficiency.

If you specify **/w-** without *warnclass*, all warnings are disabled. If you follow **/w-** with *warnclass*, only that warning is disabled. Each warning message has a three-letter identifier:

ALN	Segment alignment
ASS	Assuming segment is 16-bit
BRK	Brackets needed
GTP	Global type doesn't match symbol type
ICG	Inefficient code generation

INT	INT 3 generation
LCO	Location counter overflow
MCP	MASM compatibility pass
OPI	Open IF conditional
OPP	Open procedure
OPS	Open segment
OVF	Arithmetic overflow
PDC	Pass-dependent construction
PQK	Assuming constant for [const] warning
PRO	Write-to memory in protected mode needs CS override
RES	Reserved word warning
TPI	Borland Pascal illegal warning
UNI	For turning off uninitialized segment warning

If you specify **/w+** without *warnclass*, all warnings are enabled. If you specify **/w+** with *warnclass* from the preceding list, only that warning will be enabled.

By default, Turbo Assembler first starts assembling your file with all warnings enabled except the inefficient code-generation (ICG) and the write-to-memory in protected mode (PRO) warnings.

You can use the **WARN** and **NOWARN** directives within your source file to control whether a particular warning is allowed for a certain range of source lines. These directives are described later in this chapter.

Example

```
TASM /w TEST1
```

The following statement in TEST1.ASM issues a warning message that would not have appeared without the **/w** option:

```
mov bx,ABC    ;inefficient code generation warning
ABC = 1
```

With the command line

```
TASM /w-OVF TEST2
```

no warnings are generated if TEST2.ASM contains

```
dw  1000h * 20h
```

/x

Function Includes false conditionals in listing

Syntax

```
/x
```

Remarks

If a conditional **IF, IFNDEF, IFDEF,** and so forth evaluates to False, the **/x** option causes the statements inside the conditional block to appear in the listing file. This option also causes the conditional directives themselves to be listed; normally they are not.

You must specify a listing file on the command line or use the **/l** option, otherwise **/x** has no effect.

You can use the **.LFCOND**, **.SFCOND**, and **.TFCOND** directives to override the effects of the **/x** option.

Example

```
TASM /x TEST1
```

/z

Function

Displays source lines along with error messages

Syntax

```
/z
```

Remarks

The **/z** option tells Turbo Assembler to display the corresponding line from the source file when an error message is generated. The line that caused the error is displayed before the error message. With this option disabled, Turbo Assembler just displays a message that describes the error.

Example

```
TASM /z TEST1
```

/zd

Function

Enables line-number information in object files

Syntax

```
/zd
```

Remarks

The **/zd** option causes Turbo Assembler to place line-number information in the object file. This lets the debugger display the current location in your source code, but does not put the information in the object file that would allow the debugger to access your data items.

If you run out of memory when trying to debug your program, you can use **/zd** for some modules and **/zi** for others.

Example

```
TASM /zd TEST1
```

/zi

Function

Enables debug information in object file

Syntax `/zi`

Remarks The **/zi** option tells Turbo Assembler to output complete debugging information to the object file. This includes line-number records to synchronize source code display and data type information to let you examine and modify your program's data.

The **/zi** option lets you use all the features of the debugger to step through your program and examine or change your data items. You can use **/zi** on all your program's modules, or just on those you're interested in debugging. Since the **/zi** switch adds information to the object and executable programs, you might not want to use it on all your modules if you run out of memory when running a program under the debugger.

Example

```
TASM /zi TEST1
```

/zn

Function Disables debug information in object file

Syntax `/zn`

Remarks The **/zn** option tells Turbo Assembler to disable the output of debugging information to the object file. It's useful for overriding any prevailing **/zi** switch in a configuration file.

Indirect command files

At any point when entering a command line, Turbo Assembler lets you specify an *indirect* command file by preceding its name with an "at" sign (@). For example,

```
TASM /dTESTMODE @MYPROJ.TA
```

causes the contents of the file MYPROJ.TA to become part of the command line, exactly as if you had typed in its contents directly.

This useful feature lets you put your most frequently used command lines and file lists in a separate file. And you don't have to place your entire command line in one indirect file, since you can use more than one indirect file on the command line and can also mix indirect command files with normal arguments. For example,

```
TASM @MYFILES @IOLIBS /dBUF=1024
```

This way you can keep long lists of standard files and options in files, so that you can quickly and easily alter the behavior of an individual assembly run.

You can either put all your file names and options on a single line in the command file, or you can split them across as many lines as you want.

The configuration file

Turbo Assembler also lets you put your most frequently used options into a configuration file in the current directory. This way, when you run Turbo Assembler, it looks for a file called TASM.CFG in your current directory. If Turbo Assembler finds the file, it treats it as an indirect file and processes it before anything else on the command line.

This is helpful when you have all the source files for a project in a single directory, and you know that, for example, you always want to assemble with emulated floating-point instructions (the **/e** option). You can place that option in the TASM.CFG file, so you don't have to specify that option each time you start Turbo Assembler.

The contents of the configuration file have exactly the same format as an indirect file. The file can contain any valid command-line options, on as many lines as you want. The options are treated as if they all appeared on one line.

The contents of the configuration file are processed before any arguments on the command line. This lets you override any options set in the configuration file by simply placing an option with the opposite effect on the command line. For example, if your configuration file contains

```
/a /e
```

and you invoke Turbo Assembler with

```
TASM /s /r MYFILE
```

MYFILE is your program file, and your file will be assembled with sequential segment-ordering (**/s**) and real floating-point instructions (**/r**), even though the configuration file contained the **/a** and **/e** options that specified alphabetical segment-ordering and emulated floating-point instructions.

C H A P T E R 3

General programming concepts

This chapter introduces you to the basic concepts of Turbo Assembler. We'll look at Ideal mode versus MASM mode, commenting your programs and extending lines of code, including files, using predefined symbols, and using several important directives that produce module information. Although this is a lot of ground to cover, it will give you a good idea of what assembly language is all about.

Turbo Assembler Ideal mode

For those of you struggling to make MASM do your bidding, this may be the most important chapter in the manual. In addition to near-perfect compatibility with MASM syntax, Turbo Assembler smooths the rough areas of assembly language programming with a MASM derivative we call Ideal mode.

Among other things, Ideal mode lets you know solely by looking at the source text exactly how an expression or instruction operand will behave. There's no need to memorize all of MASM's many quirks and tricks. Instead, with Ideal mode, you write clear, concise expressions that do exactly what you want.

Ideal mode uses nearly all MASM's same keywords, operators, and statement constructions. This means you can explore Ideal mode's features one at a time without having to learn a large number of new rules or keywords.

Ideal mode adds strict type checking to expressions. Strict type checking helps reduce errors caused by assigning values of wrong types to registers and variables, and by using constructions that appear correct in the source text, but are assembled differently than you expect. Instead of playing guessing games with values and expressions, you can use Ideal mode to write code that makes logical and aesthetic sense.

With strict type checking, Ideal mode expressions are both easier to understand and less prone to producing unexpected results. And, as a result, many of the MASM idiosyncrasies we warn you about in other chapters disappear.

Ideal mode also has a number of features that make programming easier for novices and experts alike. These features include the following:

- duplicate member names among multiple structures
- complex HIGH and LOW expressions
- predictable **EQU** processing
- correct handling of grouped data segments
- improved consistency among directives
- sensible bracketed expressions

Why use Ideal mode?

There are many good reasons why you should use Turbo Assembler's Ideal mode. If you are just learning assembly language, you can easily construct Ideal mode expressions and statements that have the effects you desire. You don't have to experiment trying different things until you get an instruction that does what you want. If you are an experienced assembly language programmer, you can use Ideal mode features to write complex programs using language extensions such as nestable structures and unions.

As a direct benefit of cleaner syntax, Ideal mode assembles files 30% faster than MASM mode. The larger your projects and files, the more savings in assembly time you'll gain by switching to Ideal mode.

Strong type-checking rules, enforced by Ideal mode, let Turbo Assembler catch errors that you would otherwise have to find at run time or by debugging your code. This is similar to the way high-level language compilers point out questionable constructions and mismatched data sizes.

Although Ideal mode uses a different syntax for some expressions, you can still write programs that assemble equally well in both MASM and Ideal modes. You can also switch between MASM and Ideal modes as often as necessary within the same source file. This is especially helpful when you're experimenting with Ideal mode features, or when you're converting existing programs written in the MASM syntax. You can switch to Ideal mode for new code that you add to your source files and maintain full MASM compatibility for other portions of your program.

Entering and leaving Ideal mode

Use the **IDEAL** and **MASM** directives to switch between Ideal and MASM modes. Turbo Assembler always starts assembling a source file in MASM mode. To switch to Ideal mode, include the **IDEAL** directive in your source file before using any Ideal mode capabilities. From then on, or until the next **MASM** directive, all statements behave as described in this chapter.

You can switch back and forth between MASM and Ideal modes in a source file as many times as you wish and at any place. Here's a sample:

```
DATA SEGMENT        ;start in MASM mode
abc  LABEL BYTE     ;abc addresses xyz as a byte
xyz  DW   0         ;define a word at label xyz
DATA ENDS           ;end of data segment

     IDEAL          ;switch to Ideal mode

SEGMENT CODE        ;segment keyword now comes first
PROC MyProc         ;proc keyword comes first, too
:
                    ;Ideal mode programming goes here
ENDP MyProc         ;repeating MyProc label is optional
ENDS                ;repeating segment name not required

     MASM           ;switch back to MASM mode

CODE SEGMENT        ;name now required before segment keyword
Func2 PROC          ;name now comes before proc keyword, too
:
                    ;MASM-mode programming goes here
     IDEAL          ;switch to Ideal mode again!
:
                    ;do some programming in Ideal mode
     MASM           ;back to MASM mode. Getting dizzy?

Func2 ENDP          ;name again required before keyword
CODE ENDS           ;name again required here
```

In Ideal mode, directive keywords such as **PROC** and **SEGMENT** appear before the identifying symbol names, which is the reverse of MASM's order. You also have the option of repeating a segment or procedure name after the **ENDP** and **ENDS** directives. Adding the name can help clarify the program by identifying the segment or procedure that is ending. This is a good idea, especially in programs that nest multiple segments and procedures. You don't have to include the symbol name after **ENDP** and **ENDS**, however.

MASM and Ideal mode differences

This section describes the main differences between Ideal and MASM modes. If you know MASM, you might want to experiment with individual features by converting small sections of your existing programs to Ideal mode. Further details of these differences are in Chapter 5, "Using expressions and symbol values."

Expressions and operands

The biggest difference between Ideal and MASM mode expressions is the way square brackets function. In Ideal mode, square brackets always refer to the contents of the enclosed quantity. Brackets never cause implied additions to occur. Many standard MASM constructions, therefore, are not permitted by Ideal mode.

In Ideal mode, square brackets must be used in order to get the contents of an item. For example,

```
mov  ax,wordptr
```

displays a warning message. You're trying to load a pointer (*wordptr*) into a register (AX). The correct form is

```
mov  ax,[wordptr]
```

Using Ideal mode, it's clear you are loading the contents of the location addressed by *wordptr* (in the current data segment at DS) into AX.

If you wish to refer to the offset of a symbol within a segment, you must explicitly use the **OFFSET** operator, as in this example:

```
mov  ax,OFFSET wordptr
```

Operators

The changes made to the expression operators in Ideal mode increase the power and flexibility of some operators while leaving unchanged the overall behavior of expressions. The precedence levels of some operators have been changed to facilitate common operator combinations.

The period (.) structure member operator is far more strict in Ideal mode when accurately specifying the structure members you're referring to. The expression to the left of a period *must* be a structure pointer. The expression to the right *must* be a member name in that structure. Here's an example of loading registers with the values of specific structure members:

```
;Declare variables using the structure types
S_Stuff SomeStuff <>
O_Stuff OtherStuff <>
mov  ax,[S_Stuff.Amount]     ;load word value
mov  bl,[O_Stuff.Amount]     ;load byte value
```

Suppressed fixups

Turbo Assembler in Ideal mode does not generate segment-relative fixups for private segments that are page- or paragraph-aligned. Because the linker does not require such fixups, assembling programs in Ideal mode can result in smaller object files that also link more quickly than object files generated by MASM mode. The following demonstrates how superfluous fixups occur in MASM but not in Ideal mode:

This difference has no effect on code that you write. The documentation here is simply for your information.

```
SEGMENT DATA PRIVATE PARA
VAR1 DB   0
VAR2 DW   0
ENDS
SEGMENT CODE
     ASSUME ds:DATA
     mov  ax,VAR2                ;no fixup needed
ENDS
```

Operand for BOUND instruction

The **BOUND** instruction expects a **WORD** operand, not a **DWORD**. This lets you define the lower and upper bounds as two constant words, eliminating the need to convert the operand to a **DWORD** with an explicit **DWORD PTR**. In MASM mode, you must write

```
BOUNDS    DW   1,4               ;lower and upper bounds
BOUND     AX, DWORD PTR BOUNDS       ;required for MASM mode
```

but in Ideal mode, you need only write

```
BOUNDS    DW   1,4               ;lower and upper bounds
   BOUND  AX,    [BOUNDS]          ;legal in Ideal mode
```

Segments and groups

The way Turbo Assembler handles segments and groups in Ideal mode can make a difference in getting a program up and running. If you're like most people, you probably shudder at the thought of dealing with a bug that has anything to do with the interaction of segments and groups.

Much of the difficulty in this process stems from the arbitrary way that MASM and, therefore, Turbo Assembler's MASM mode, makes assumptions about references to data or code within a group. Fortunately, Ideal mode alleviates some of the more nagging problems caused by MASM segment and group directives, as you'll see in the information that follows.

Accessing data in a segment belonging to a group

In Ideal mode, any data item in a segment that is part of a group is considered to be principally a member of the group, not of the segment. An explicit segment override must be used for Turbo Assembler to recognize the data item as a member of the segment.

MASM mode handles this differently; sometimes a symbol is considered to be part of the segment instead of the group. In particular, MASM mode treats a symbol as part of a segment when the symbol is used with the **OFFSET** operator, but as part of a group when the symbol is used as a pointer in a data allocation. This can be confusing because when you directly access the data without **OFFSET**, MASM incorrectly generates the reference relative to the segment instead of the group.

Here's an example of how easily you can get into trouble with MASM's addressing quirks. Consider the following incomplete MASM program, which declares three data segments:

```
dseg1   SEGMENT PARA PUBLIC 'data'
v1      DB      0
dseg1   ENDS

dseg2   SEGMENT PARA PUBLIC 'data'
v2      DB      0
dseg2   ENDS

dseg3   SEGMENT PARA PUBLIC 'data'
v3      DB      0
dseg3   ENDS

DGROUP  GROUP   dseg1,dseg2,dseg3
cseg    SEGMENT PARA PUBLIC 'code'

        ASSUME  cs:cseg,ds:DGROUP

start:
        mov     ax,OFFSET v1
        mov     bx,OFFSET v2
        mov     cx,OFFSET v3
cseg    ENDS
        END     start
```

The three segments, *dseg1*, *dseg2*, and *dseg3*, are grouped under one name, **DGROUP**. As a result, all the variables in the individual segments are stored together in memory. In the program source text, each of the individual segments declares a **BYTE** variable, labeled *v1*, *v2*, and *v3*.

In the code portion of this MASM program, the offset addresses of the three variables are loaded into registers AX, BX, and CX. Because of the earlier **ASSUME** directive and because the data segments were grouped together, you might think that MASM would calculate the offsets to the variables relative to the entire group in which the variables are eventually stored in memory.

But this is not what happens. Despite your intentions, MASM calculates the offsets of the variables relative to the individual segments, *dseg1*, *dseg2*, and *dseg3*. It does this even though the three segments are combined into one data segment in memory, addressed here by register DS. It makes no sense to take the offsets of variables relative to individual segments in the program text when those segments are combined into a single segment in memory. The only way to address such variables is to refer to their offsets relative to the entire group.

To fix the problem in MASM, you must specify the group name along with the **OFFSET** keyword:

```
mov   ax,OFFSET DGROUP:v1
mov   bx,OFFSET DGROUP:v2
mov   cx,OFFSET DGROUP:v3
```

Although this now assembles correctly and loads the offsets of *v1*, *v2*, and *v3* relative to **DGROUP** (which collects the individual segments), you might easily forget to specify the **DGROUP** qualifier. If you make this mistake, the offset values will not correctly locate the variables in memory and you'll receive no indication from MASM that anything is amiss. In Ideal mode, there's no need to go to all this trouble:

```
        IDEAL
SEGMENT dseg1   PARA PUBLIC 'data'
v1      DB      0
ENDS

SEGMENT dseg2   PARA PUBLIC 'data'
v2      DB      0
ENDS

SEGMENT dseg3   PARA PUBLIC 'data'
v3      DB      0
ENDS

GROUP   DGROUP  dseg1,dseg2,dseg3
SEGMENT cseg    PARA PUBLIC 'code'

        ASSUME  cs:cseg, ds:DGROUP

start:
        mov     ax,OFFSET v1
        mov     ax,OFFSET v2
        mov     ax,OFFSET v3

ENDS
        END     start
```

The offsets to *v1*, *v2*, and *v3* are correctly calculated relative to the group that collects the individual segments to which the variables belong. Ideal mode does not require the **DGROUP** qualifier to refer to variables in grouped segments. MASM mode does require the qualifier and, even worse, gives no warning of a serious problem should you forget to specify the group name in every single reference.

Commenting the program

Commenting your code is a great way to help you (or anyone who has to maintain your code in the future) quickly understand how it functions. Using comments is good programming practice in any language. They can describe the semantic as opposed to syntactic function of your code. We recommend that you use comments liberally in your Turbo Assembler code, and this section describes how you can do so.

Comments at the end of the line

There are several ways to comment assembler code. One approach is to add a comment at the end of a line using the semicolon (;), such as

```
mov [bx],al                        ;store the modified character
```

Another way to comment assembler code is to use the line continuation character (\) as a comment character. See the section called "Extending the line" for an example of how this is done.

The COMMENT directive

The **COMMENT** directive lets you comment blocks of code. **COMMENT** ignores all text from the first delimiter character and the line containing the next occurrence of the delimiter. The following example uses * as a delimiter character:

COMMENT only works in MASM mode.

```
COMMENT *
   stuff here
*
```

Extending the line

For lines of code that are longer than 80 characters, Turbo Assembler provides the \ line continuation character. Use this character at the end of your line, because Turbo Assembler ignores any characters that follow it on the same line.

The maximum line length is 1024 when you use \; however, tables, records, and enums might have definitions that are longer than 1024 characters. An alternative that does not have the 1024 character limitation is the multiline definition syntax. Here's an example of the syntax (for an enum definition):

```
foo enum {  ;Multiline version
   f1
   f2
   f3
   f4
   f5
```

```
    f6
    f7
    f8
    }
```

A more compact version of the same definition:

```
foo enum f1,f2,{   ; Compact multiline version
    f3,f4
    f5,f6
    f7,f8}
```

When using multiline definitions, remember these rules:

- The left brace that starts the definition must be the last token on the starting line. It does not, however, have to precede the first element in the list.
- You cannot include any directives such as **IF** or **INCLUDE** inside the multiline definition.

MASM-mode line continuation is available if you select **VERSION M510, M520**. Strings and other tokens can be extended across multiple lines if the "\" character is the last character on the line. For example,

```
VERSION M510
DB 'Hello out there      \
you guys'
```

You can place standard Turbo Assembler mode line continuation anywhere in a line, and it is always available. It functions as a comment as well. For example,

```
ARG  a1:word,          \first argument
     a2:word,          \second argument
     a3:word           ;final argument
```

Using INCLUDE files

Include files let you use the same block of code in several places in your program, insert the block in several source modules, or reduce the size of your source program without having to create several linkable modules. Using the **INCLUDE** directive tells Turbo Assembler to find the specified files on disk and assemble them as if they were a part of the source program.

You can nest **INCLUDE** directives as deep as you want.

The Ideal mode syntax:

```
INCLUDE "filename"
```

The MASM mode syntax:

```
INCLUDE filename
```

filename can specify any drive, directory, or extension. If *filename* does not include a directory or drive name, Turbo Assembler first searches for the file in any directories you specify with the **/I** command-line option, and then in the current directory.

Predefined symbols

Turbo Assembler provides a number of predefined symbols that you can use in your programs. These symbols can have different values at different places in your source file, and are similar to equated symbols you define using the **EQU** directive. When Turbo Assembler encounters one of these symbols in your source file, it replaces it with the current value of that predefined symbol.

Some of these symbols are text (string) equates, some are numeric equates, and others are aliases. The string values can be used anywhere that you would use a character string, for example, to initialize a series of data bytes using the **DB** directive:

```
NOW   DB   ??time
```

Numeric predefined values can be used anywhere that you would use a number:

```
IF ??version GT 100h
```

Alias values make the predefined symbol into a synonym for the value it represents, allowing you to use the predefined symbol name anywhere you would use an ordinary symbol name:

```
ASSUME  cs:@code
```

All the predefined symbols can be used in both MASM and Ideal mode.

If you use the **/ml** command-line option when assembling, you must use the predefined symbol names exactly as they are described on the following pages.

The following rule applies to predefined symbols starting with an at-sign (@): *The first letter of each word that makes up part of the symbol name is an uppercase letter (except for segment names); the rest of the word is lowercase.* As an example,

```
@FileName
```

Notice that @FileName performs an alias equate for the current assembly line.

The exception is redefined symbols, which refer to segments. Segment names begin with an at-sign (@) and are all lowercase. For example,

```
@curseg
@fardata
```

For symbols that start with two question marks (??), the letters are all lowercase. For example,

```
??date
??version
```

Note that the **??date** symbol defines a text equate that represents today's date. The exact format of the date string is determined by the country code. The **??version** symbol lets you write source files that can take advantage of features in particular versions of Turbo Assembler. This equate also lets your source files know whether they are being assembled by MASM or Turbo Assembler, since **??version** is not defined by MASM. Similarly, **??filename** defines an eight-character string that represents the file name being assembled. The file name is padded with spaces if it it contains fewer than eight characters. The **??time** symbol defines a text equate that represents the current time. The exact format of the time string is determined by the country code.

Assigning values to symbols

Turbo Assembler provides two directives that let you assign values to symbols: **EQU** and **=**. The **EQU** directive defines a string, alias, or numeric equate. To use it, specify the following syntax,

```
name EQU expression
```

where *name* is assigned the result of evaluating *expression*. *name* must be a new symbol name that you haven't previously defined in a different manner. In MASM mode, you can only redefine a symbol that you defined using the **EQU** directive if you first define it as a string equate. In MASM mode, **EQU** can generate any one of three kinds of equates: alias, expression, or string.

The **=** directive defines *only* a numeric equate. To use it, specify

```
name = expression
```

where *name* is assigned the result of evaluating *expression*, which must evaluate to either a constant or an address within a segment. *name* can either be a new symbol name, or a symbol that you previously defined with =. Since the = directive has far more predictable behavior than the **EQU** directive in MASM mode, use = instead of **EQU** wherever you can.

General module structure

Turbo Assembler provides several directives to help you work with modules of code. The remainder of this chapter describes these directives.

The VERSION directive

Using the **VERSION** directive lets you specify which version of Turbo Assembler or MASM you've written particular modules for. This is helpful for upward and downward compatibility of various versions of TASM and MASM. The **VERSION** directive also puts you into the operating mode for the specified version.

You can specify the **VERSION** directive as either a command-line switch or within program source code.

Within code, the syntax is

```
VERSION <version_ID>
```

You can specify the following legal version IDs:

M400	MASM 4.0
M500	MASM 5.0
M510	MASM 5.1
M520	MASM 5.2 (Quick ASM)
T100	Turbo Assembler 1.0
T101	Turbo Assembler 1.01
T200	Turbo Assembler 2.0
T250	Turbo Assembler 2.5
T300	Turbo Assembler 3.0
T310	Turbo Assembler 3.1
T320	Turbo Assembler 3.2
T400	Turbo Assembler 4.0
T410	Turbo Assembler 4.1

The command-line syntax is:

/**U**<*version_ID*>

As an example, if you wanted to assemble a program written for MASM 5.0, you could leave the source for the program intact and use the switch **/uM510**.

Here are the general rules:

1. The **VERSION** directive always selects MASM mode by default, because that is the starting mode of operation for both MASM and Turbo Assembler.
2. The **VERSION** directive limits the high-priority keywords available to those in the specified compiler and version. As a result, some features that were added to later versions are unavailable to you.
3. From Ideal mode, the **VERSION** directive is unavailable if you select a version prior to T300. To use the **VERSION** directive in this case, you must switch to MASM mode first.
4. No attempt is made to limit access to low priority keywords, since these will not affect compatibility.

Previous versions of Turbo Assembler controlled MASM compatibility with directives such as **MASM51**, **NOMASM51**, **QUIRKS**, **SMART**, and **NOSMART**. The **VERSION** directive supersedes these older directives. See Appendix B for a complete list of keywords available with each prior version of Turbo Assembler.

The NAME directive

Use the **NAME** directive to set the object file's module name. Here is the syntax for it:

```
NAME modulename
```

This directive only works in Ideal mode.

Turbo Assembler usually uses the source file name with any drive, directory, or extension as the module name. Use **NAME** if you wish to change this default name; *modulename* will be the new name of the module. For example,

```
NAME loader
```

The END directive

Use the **END** directive to mark the end of your source file. The syntax looks like this:

```
END [ startaddress ]
```

startaddress is an optional symbol or expression that specifies the address in your program where you want execution to begin. If your program is linked from multiple source files, only one file can specify a *startaddress*.

startaddress can be an address within the module; it can also be an external symbol defined in another module, declared with the **EXTRN** directive.

Turbo Assembler ignores any text after the **END** directive in the source file.

Example

```
.MODEL small
.CODE
START:
;Body of program goes here
END START                    ;program entry point is "START"
THIS LINE IS IGNORED
SO IS THIS ONE
```

Displaying a message during assembly

Turbo Assembler provides two directives that let you display a string on the console during assembly: **DISPLAY** and **%OUT**. You can use these directives to report on the progress of an assembly, either to let you know how far the assembly has progressed, or to let you know that a certain part of the code has been reached.

The two directives are essentially the same except that **DISPLAY** displays a quoted string onscreen, and **%OUT** displays a nonquoted string onscreen.

In both Ideal and MASM modes, the syntax for **DISPLAY** is

```
DISPLAY "text"
```

where *text* is any message you want to display.

The syntax for **%OUT** in both Ideal and MASM modes is

```
%OUT text
```

where, again, *text* is the message that you want displayed.

Displaying warning messages

WARN without warnclass enables all warnings. **WARN** with an identifier only enables that warning.

Turbo Assembler lets you choose what (if any) warning messages you'll receive for certain parts of your code. Each warning message contains a three-letter identifier, which you can specify ahead of time to let the assembler know whether or not you want to see warnings of that kind. You can use the **WARN** directive to enable warning messages, and the **NOWARN** directive to disable them.

The syntax of the **WARN** directive is

```
WARN [warnclass]
```

where *warnclass* is the three-letter identifier that represents a particular type of warning message. The available *warnclass*es are:

ALN	Segment alignment
BRK	Brackets needed
GTP	Global type doesn't match symbol type
ICG	Inefficient code generation
INT	INT 3 generation
LCO	Location counter overflow
MCP	MASM compatibility pass
OPI	Open IF conditional
OPP	Open procedure
OPS	Open segment
OVF	Arithmetic overflow
PDC	Pass-dependent construction
PRO	Write-to-memory in protected mode using CS
PQK	Assuming constant for [const] warning
RES	Reserved word warning
TPI	Borland Pascal illegal warning

Note that these are the same identifiers used by the **/W** command-line option.

Here's an example using **WARN**:

```
WARN  OVF                    ;enables arithmetic overflow warning
DW 1000h * 1234h             ;overflow warning will occur
```

NOWARN without warnclass disables all warnings. **NOWARN** with an identifier disables only that warning.

Use the **NOWARN** directive to disable specific (or all) warning messages. **NOWARN** uses the same identifiers described earlier under **WARN**. Here's an example that uses **NOWARN**:

```
NOWARN OVF                   ;disable arithmetic overflow warnings
DW 1000h * 1234h             ;doesn't warn now
```

Multiple error-message reporting

By default, Turbo Assembler only allows one error message to be reported for each line of source code. If a source line contains multiple errors, Turbo Assembler reports the most-significant error first. You can control the number of error messages you get for each source line by using the **MULTERRS** and **NOMULTERRS** directives.

The **MULTERRS** directive allows the assembler to report more than one error message for each source line. This is sometimes helpful in locating the cause of a subtle error or when the source line contains more than one error.

Note that sometimes additional error messages can be a "chain reaction" caused by the first error condition; these "chain" error messages may disappear once you correct the first error.

Here's an example of the **MULTERRS** directive:

```
MULTERRS
mov ax,[bp+abc                ;produces two errors:
                              ;1) Undefined symbol: abc
                              ;2) Need right square bracket
```

➡ The **NOMULTERRS** directive only lets one error or warning message (the most significant message) appear for each source line. When you correct this error, the other error messages may disappear as well. To avoid this problem, use the **MULTERRS** directive to see all of the error messages.

Here is an example of using the **NOMULTERRS** directive:

```
NOMULTERRS
mov ax,[bp+abc                ;one error:
                              ;1) Undefined symbol: abc
```

C H A P T E R 4

Creating object-oriented programs

Object-oriented programming is an approach to software design that is based on objects rather than procedures. This approach maximizes modularity and information hiding. The underlying premise behind object-oriented programming is the binding or encapsulation of a data structure with procedures for manipulating the data in the structure into a unit.

Object-oriented design provides many advantages. For example, every object encapsulates its data structure with the procedures used to manipulate instances of the data structure. This removes interdependencies in code that can quickly make maintenance difficult. Objects can also inherit a data structure and other characteristics from a parent object, which saves work and lets you transparently use a single chunk of code for many purposes.

If you're not an experienced Turbo Assembler user, you might want to skim through this chapter now, but come back to it later after reading the other chapters of this manual. We've put it here to make you aware of these features, but object-oriented programming in Turbo Assembler is really an advanced topic. It will make more sense after going through the rest of the manual.

Terminology

These terms are described in detail later in this chapter.

C++ and Pascal use different terms for various entities in object-oriented programming. Turbo Assembler more closely resembles Pascal in this way, although not all terms are the same. The following table outlines these differences among these languages.

Table 4.1 Object-oriented programming terminology

Turbo Assembler	Borland C++	Borland Pascal
method	member function	method
method procedure		
object	class	object
base object	base class	base object

Table 4.1: Object-oriented programming terminology (continued)

parent object	parent class	parent object
derived object	derived class	derived object
field	data member	field

Why use objects in Turbo Assembler?

Most people think of assembly language as a low-level language. Turbo Assembler, however, provides many of the features of a high-level language (such as abstract data types, and easy interfacing to other languages). The addition of object-oriented data structures gives Turbo Assembler the power to create object-oriented programs as easily as high-level languages while retaining the speed and flexibility of assembly language.

What is an object?

An *object* consists of a data structure and associated procedures (called *methods*) that manage data stored in instances of the data structure.

We strongly recommend that you use Ideal mode for object-oriented programming in Turbo Assembler, since symbol scoping is global in MASM, and you can't distinguish different positions of shown methods.

An object can inherit characteristics from a parent object. This means that the new object's data structure includes the parent object's data structure, as well as any new data. Also, the new object can call all the method procedures of the parent object, as well as any new method procedures it declares.

An object having no inheritance is called a *base* object; an object that inherits another is a *derived* object.

Turbo Assembler defines several symbols you can use when declaring objects. The following table lists these symbols.

Table 4.2 Symbols defined for objects

Symbol	Meaning
@Object	A text macro containing the name of the current object (the object last declared).
<objectname>	A **STRUC** data type that describes the object's data structure.
@Table_*<objectname>*	A **TABLE** data type containing the object's method table, which is not the same as an instance of the virtual method table.
@TableAddr_*<objectname>*	A label describing the address of the instance of the object's virtual method table, if there is one.

A sample object

As an example of where you can use objects, consider any program that uses linked lists. Think of a linked list as an object consisting of the linked list data and the operations (methods) that you can perform on it.

The linked list data consists of pointers to the head and tail of the linked list (this example contains a doubly linked list because of its flexibility). Each element of the linked list is a separate object instance.

The following operations provide the power needed to use a linked list:

- Creating the linked list (allocating memory for it).
- Destroying the linked list (deallocating memory for it).
- Initializing the linked list.
- Deinitializing the linked list.
- Inserting an item into the middle of the linked list before an existing item.
- Appending an item to the end of the linked list.
- Deleting an item from the linked list.
- Returning the first item in the linked list.
- Returning the last item in the linked list.

Keep in mind that *create* and *initialize*, as well as *destroy* and *deinitialize* methods are not synonymous. *create* and *destroy* methods allocate and deallocate memory for the linked list object, while the *initialize* and *deinitialize* methods only initialize and deinitialize previously allocated instances of the object. If you don't combine initialization with creation, it's possible to statically allocate linked list objects.

You can see how the linked list object can be inherited by a queue or stack object, since a queue or a stack can be implemented as a linked list with limited operations. For example, you can implement a queue as a linked list where items can be added to the start and taken off the end. If you implement a queue in this way, you must disable the inherited linked list methods that are illegal on a queue (such as inserting into the middle of the list).

Declaring objects

Declaring an object consists of declaring the data structure for the object, and declaring the method procedures that you can call for the object. Declaring an object does not involve creating an instance of the object. You'll learn how to do this later.

Declaring a base object

When you declare an object, Turbo Assembler creates a **STRUC** that declares the data for the object, and a **TABLE** that declares the methods for the object. The object's data declaration is a structure with the same name as the object. The object's method declarations are stored in a **TABLE** data type, named @Table_<*objectname*>.

For more on **STRUC** as it applies to declaring objects, see Chapter 8.

For example, for the *list* object, two data types are declared:

list — A **STRUC** declaring the following members:

list_head	dword pointer to head of *list*
list_tail	dword pointer to tail of *list*

@Table_list — A **TABLE** declaring the following methods:

construct	dword pointer to the procedure *list_construct*
destroy	dword pointer to the procedure *list_destroy*

and so on...

STRUC declares the data for the object that is created whenever you create an instance of the object. **TABLE** declares the table of default method procedures for the declaration. Turbo Assembler maintains this data type; it does not create an instance of the table anywhere in your program memory. However, you'll see later that you must include an instance of the table for any object that uses virtual methods. Here's an example of an object declaration for a linked list:

The **METHOD** keyword shows that you're using an extended form of **STRUC**, and are defining an object called *list*.

Each entry consists of a method name, a colon, and the size of a pointer to the method procedure (**WORD** for near procedures, **DWORD** for far procedures). This is followed by an equal sign, and the name of the procedure to call for that method.

```
list STRUC GLOBAL METHOD {
  construct:dword = list_construct   ;list constructor procedure
  destroy:dword = list_destroy       ;list destructor procedure
  init:dword = list_init             ;list initializer procedure
  deinit:dword = list_deinit         ;list deinitializer procedure
  virtual insert:word = list_insert  ;list node insert procedure
  virtual append:word = list_append  ;list node append procedure
  virtual remove:word = list_delete   ;list node remove procedure
  virtual first:word = list_first    ;list first node procedure
  virtual last:word = list_last      ;list last node procedure
  }
  list_head   dd ?                   ;list head pointer
  list_tail   dd ?                   ;list tail pointer
ENDS
```

Let's look at this example to see what's happening.

METHOD indicates an object method call and is followed by a list of the method procedure declarations for the object. These declarations are enclosed in braces ({ }) because the list of methods requires more than one line.

Each method declaration tells Turbo Assembler which procedure it should use to manipulate the object when invoking that method name. For example, the first method procedure declaration

```
construct:dword = list_construct
```

declares a method named *construct* that is a far procedure (because a DWORD stores the pointer to it). The actual procedure name of the method is *list_construct*, which should be defined elsewhere in the source code.

Turbo Assembler considers a method to be virtual if it's preceded by the keyword **VIRTUAL**. When you call such a method, Turbo Assembler will locate the method's procedure address by looking it up from a table present in memory at run time. Otherwise, the method is a static method, meaning that Turbo Assembler can determine its address at compile time. For example, the method *construct* is a static method, while the method *insert* is declared as a virtual method. Later in this chapter, we'll explain why you might want to choose virtual or static methods.

The data structure for the method immediately follows the method procedure declaration section. This definition uses the syntax for the standard **STRUC** directive. This example contains declarations for the linked list's head and tail pointers.

The method declaration portion of the object declaration doesn't place any data in the object's data structure unless you've used virtual methods. Instead, these declarations cause Turbo Assembler to build a separate table data structure that contains the specified method procedure addresses as default values. You should have an instance of this table for every object, and you must explicitly place the table. We'll explain how to do this later in this chapter.

Since the object declaration must exist in the module containing the method procedures for the object (as well as included in any source code that uses the object), you should declare the object itself in a separate file that can be **INCLUDE**d into the source code. We recommend using a file name in the form *objectname*.ASO (ASsembly Object). This file should consist of only the object declaration. The object methods should be in another source file so that you can include the object declaration wherever you need it. For example, the linked list object declaration in the previous example would be placed in the file LIST.ASO. The file LIST.ASM could be used to define the object's method procedures. Any program making use of the objects would include LIST.ASO, but not LIST.ASM.

The keyword **GLOBAL** in the object declaration causes Turbo Assembler to publish information that lets you use the object in a module other than the one it's defined in. The object declaration must also be included in all modules that use the object.

Declaring a derived object

An object that inherits another object's methods and data is called a *derived* object. You can't override the members of the parent data structure, but you can override the individual methods by respecifying them in the new object method list.

An object can inherit any other single object, whether that other object is a base or derived object itself. The inherited object is called the *parent* object. The derived object inherits the data and methods of the parent object, so you should only use inheritance when these methods and data are useful to the new object.

For example, you can define a queue object that inherits the linked list object because you can implement a queue as a linked list. Here's an example of such a derived object:

```
queue STRUC GLOBAL list METHOD {
  init:DWORD=queue_init
  virtual insert:word = queue_insert    ;(queue node insert
                                        ; procedure)
  virtual remove:word = queue_delete    ;(queue node delete
                                        ; procedure)
  virtual first:word = queue_first      ;(queue first node procedure)
  virtual last:word = queue_last       ;(queue end node procedure)
  virtual enqueue:word = list_append    ;queue enqueue procedure
  virtual dequeue:word = queue_dequeue  ;queue dequeue procedure
  }
ENDS
```

Placing the object name *list* before the **METHOD** keywords tells Turbo Assembler that the new object *queue* inherits the methods and data of the object, *list*. Any object name placed in this location will be inherited by the object being declared. You can use only one name (only single inheritance is supported).

The new *queue* object inherits all the data and methods from the *list* object, unless you override it. Note that *queue* needs its own *init* to install the pointer to the virtual method table for queues.

The inherited *insert*, *remove*, *first*, and *last* method declarations for the queue are respecified in the declaration, so these methods are replaced with the indicated procedures.

Two new methods have been declared for the queue: *enqueue* and *dequeue*. Notice that the method procedure for *enqueue* is the same as for appending to a linked list. However, we need a new procedure to dequeue from the queue, and this we call *queue_dequeue*.

The *queue* object has no additional data declared other than what it inherits from *list*. It inherits the linked list's head and tail pointers, which are still needed for the queue because of the linked list methods used to manage the queue.

Declaring a method procedure

Method procedures manipulate instances of the object. They are much like library routines in that they should have a well-defined call and a return value interface, but knowledge of how the method procedures work internally is not necessary.

The method procedures for an object should provide comprehensive management of the objects; that is, they should be the only procedures allowed direct access to the objects. Furthermore, you should use the concepts of data abstraction when you design the methods: You should be able to call the method procedures without having any knowledge of the inner workings of the method procedures.

In all other respects, you can write method procedures for any language or interface you want, although usually C++ or Pascal calling conventions are used. Any arguments to the procedures are up to you as well. One argument that is usually required is a pointer to an object instance. Some method procedures might require additional parameters. For example, the initialization method for the list object requires just the pointer to the list object, while the list insert method requires a pointer to the list, a pointer to the new node to insert, and a pointer to the node it's inserted after.

➡ There are advantages and disadvantages to using both static and virtual methods. Static methods are resolved at compile time, and result in direct calls to the method procedure. This makes the call faster, and does not require you to use intermediate registers (as in virtual method calls). However, since these calls are resolved at compile time, static method calls don't have the flexibility of virtual method calls.

Virtual method calls are made indirectly through an instance of the virtual method table for the object. The fact that the call is indirect gives virtual methods the disadvantage of requiring you to use intermediate registers when you make the call (which could complicate your code). A big advantage, however, is that virtual method calls are resolved at run time.

Thus, you can make virtual method calls for a derived object by calling a common ancestor object's method without having to know exactly what sort of descendant object you're dealing with.

Declare static and virtual method procedures exactly the same way as any other procedure, with the following exception: if you omit the procedure name for virtual methods, you'll cause an empty uninitialized location in the virtual method table and Turbo Assembler won't warn you if you do this. Omitting the procedure name is an error if the method is not virtual, since virtual methods don't go into the table.

Here's an example of a method procedure:

```
;Construct a Linked-List object.
;This is the method "construct".
;This must be a static method.
;Returns DX:AX pointing to linked-list object, null if none.
;Object is allocated but not yet initialized.
list_construct PROC PASCAL FAR
USES ds
     ;-- Allocate the Linked-List object --
     ;;<<do the allocation here>>
     ret
ENDP
```

The virtual method table

The virtual method table (VMT) is a table of addresses of the procedures that perform virtual methods. Usually this table is placed in the program's data segment. Any object having virtual methods requires an instance of the VMT somewhere in the program.

Use the **TBLINST** directive to create the instance of the VMT for an object. Since this directive creates a table for the most recently declared object, you should place this directive immediately after the object declaration, as in the following:

```
INCLUDE list.aso
DATASEG
TBLINST
```

Initializing the virtual method table

Simply creating the instance of the VMT is not enough to let you make calls to virtual methods. Every object with virtual methods includes a pointer to the VMT in its data structure. You must initialize this pointer whenever you create an instance of an object, and can use **TBLINIT** to do so.

Initialize the VMT pointer in the *init* method for the object as follows:

Notice that the *init* method must be static because you can't call a virtual method for an object instance until after you initialize the virtual table pointer.

```
;Initialize a Linked List object.
;This is the method "init".
;This must be a static method!
list_init PROC PASCAL FAR
ARG @@list:dword
USES ds,bx
     lds bx,@@list
     ;-- Initialize any virtual method table for the object at ds:bx
     TBLINIT ds:bx
     ;-- Initialize the object's data --
     ;;<<initialize any data for the object here...>>
     ret
ENDP
```

Calling an object method

The **CALL** syntax is similar for calling static or virtual methods.

Use the **CALL** instruction to invoke object methods.Turbo Assembler provides an extension to the standard **CALL** instruction, **CALL..METHOD,** for calling method procedures.

Calling a static method

When making a call to a method procedure, you should write the **CALL..METHOD** instruction as if you were making a call to a virtual method, even if you know that you're calling a static method. Doing so will have no ill effects on static method calls, and gives you the flexibility of changing methods from static to virtual or back again without having to change all the calls to the method. For the same reasons, you should specify a reasonable selection for the intermediate calling registers, even if you know that the method you're calling is static.

Calls to static methods are resolved at compile time to direct calls to the desired method procedure for the object. However, when making the call, you should not make a direct call to the method procedure; instead, use the extended **CALL..METHOD** instruction.

The following example shows a sample call to the static *init* method for the linked list object.

```
CALL foolist METHOD list:init pascal,ds offset foolist
CALL es:di METHOD list:init pascal,es di
```

The call address itself is the address of an instance of the object. This address is used for syntactic reasons only; the actual call generated is a direct call to the method procedure.

In this example, the first call is to the *init* method for the object *list*. Since this is a static method, you make a direct call to the method procedure *list_init*. Turbo Assembler ignores the object instance, *foolist* (except that it's passed as an argument to the method procedure).

The method name is followed by the usual extended call language and parameter list. The language and parameters depend on the method you're calling, and one of the parameters is generally a pointer to the instance of the object. In this example, the method accepts a single parameter, which is a pointer to the instance of the object.

Calling a virtual method

Any call to a virtual method requires an indirect call to the method procedure. You can use the extended **CALL..METHOD** instruction to let this happen. Turbo Assembler generates the following instructions to perform the call:

1. Load intermediate registers from the object instance with a pointer to the VMT.
2. Make an indirect call to the appropriate table member.

Therefore, when you specify

```
CALL <instance> METHOD <object>:<method> USES <seg>:<reg> <calling_stuff>
```

the generated instructions are as follows:

```
MOV <reg>,[<instance>.<virtual_method_table_pointer>]
CALL [(<seg>:<reg>).<method>] <calling_stuff>
```

The first instruction loads the selected register *<reg>* with the address of the table from the VMT pointer field of the object structure. The second instruction makes an indirect call to the appropriate method in the table.

For example, a call of the form

```
CALL es:di method list:insert uses ds:bx pascal,es di,es dx,es cx
```

generates a sequence like

```
mov bx,[es:di.@Mptr_list]
CALL [ds:bx.insert] pascal,\
   es di,es dx,es cx
```

Note that for objects declared with NEAR tables, only the offset register will be loaded by the **CALL..METHOD** instruction. The segment register should already contain the correct value. The following example shows how to make sure that the segment register is properly set up.

```
;Append a node at the end of a Linked-List object.
;This is the virtual method "list|append".
list_append PROC PASCAL NEAR
```

```
ARG     @@list:dword,\
        @@new:dword
USES ds,bx,es,di
        mov ax,@Data
        mov ds,ax
        les di,@@list
        sub ax,ax
        CALL es:di method list:insert uses ds:bx pascal,\
            es di,@@new,ax ax
        ret
ENDP
```

Note You can't call any virtual methods until after you initialize the VMT pointer in the object's data. This is because the pointer loads the address of the VMT (from which the address of the desired virtual method procedure is retrieved). Thus, if you haven't initialized the pointer to the VMT, any virtual method call will result in a call to some random address.

As another example, consider the base object *node,* which you can include in any object placed in a linked list or a queue.

```
node STRUC GLOBAL METHOD {
  construct:dword = node_construct      ;node constructor routine
  destroy:dword = node_destroy          ;node destructor routine
  init:dword = node_init                ;node initialization routine
  deinit:dword = node_deinit            ;node deinitialization
                                         routine
  virtual next:word = node_adv          ;next node routine
  virtual prev:word = node_back         ;previous node routine
  virtual print:word = node_print       ;print contents of node
  }
  node_next    dd ?                     ;next node pointer
  node_prev    dd ?                     ;prev node pointer
ends
```

You can define any number of other objects inheriting the *node* object, to let it use a linked list or queue. Here are two examples:

```
mlabel STRUC GLOBAL node METHOD {
  virtual print:word = label_print
  }
  label_name   db 80 dup (?)
  label_addr   db 80*2 dup (?)
  label_city   db 80 dup (?)
  label_state  db 2 dup (?)
  label_zip    db 10 dup (?)
ENDS
```

```
book STRUC GLOBAL node METHOD {
  virtual print:word = book_print
  }
  book_title   db 80 dup (?)
  book_author  db 80 dup (?)
ENDS
```

In the next example, we're making calls to methods by calling *printit* for both label and book objects. It doesn't matter what object gets passed to *printit*, as long as *node* is an ancestor. Because the print method is a virtual method, the call is made indirectly through the VMT for the object. For the first call to *printit*, the method procedure *label_print* is called, because we're passing an instance of a label object. For the second call to *printit*, the method procedure *book_print* is called, because we're passing an instance of a book object. Note that if the method *print* were static, then the call in *printit* would always call the *node_print* procedure (which is not desirable).

```
  call printit pascal,<<instance address of label object>>
  call printit pascal,<<instance address of book object>>
  :
printit proc pascal near
arg @@obj:dword
uses ds,si,es,bx
  mov ax,@data
  mov es,ax
  lds si,@@obj
  call ds:si method node:print uses es:bx pascal,ds si
  ret
endp
```

Calling ancestor virtual methods

Using ancestor virtual methods can help you write methods for derived classes since you can reuse some of the code. For example, queues can use the same listing method as a list, as long as you specify whether the item is a queue or a list. Within the list class, you can have

```
virtual  show:word  =  list_show
```

and within the queue class,

```
virtual  show:word  =  queue_show
```

The list_show routine might print `LIST SHOW:`, followed by a listing of the individual items in the list. However, if the derived class queue_show uses the listing routine, it should print its own title, `QUEUE SHOW:` and use list_show only for the mechanics of sequentially going through the list and printing individual elements. list_show can determine the kind of structure passed to it, and whether it should print the list title. If the routine for list_show looks at the pointer to the virtual method table (VMT) of the structure passed to it, it can determine whether the pointer matches the one installed for lists in the list_init routine (or if it differs). If the VMT pointer in the structure does not point to the VMT for lists, the structure is probably a derived type. list_show can do this checking with the following statements:

```
cmp     [([es:di]).@mptr_list],offset @TableAddr_LIST
jne     @@not_a_list   ; Skip over printing the list title

        ; If we come here, it is a list, and the list title
        ; should be printed.
  . . .

@@not_a_list:
        ; Now show the individual list elements.
```

So how do we call the list class show method from within a queue_show routine? If you were to directly call list_show, you could have a problem if the name of the routine used for the show method of the list class ever changes. (You might not remember to change what queue_show calls.) If you put the following statement in queue_show,

```
call (es:di) method list:show
```

you'd have an infinite loop because even though list is specified as the class for which show should be called, the VMT will be used because show is a virtual method. Since the VMT for the structure would have been pointing to queue_show, you'd end up back in the same routine.

The best way to call the list class show method would be

```
call +@table_list | show
```

Virtual routines are usually called through an indirect lookup to a VMT.

Turbo Assembler automatically translates this statement to a direct call to list_show, since list_show was specified as the value for the show element of the @table_list when the list class was declared. Note that even though list declares show to be virtual, specifying the call causes Turbo Assembler to make a direct call without the VMT lookup.

In the event that you need to use the VMT for the list class (for example, some initialization routine might change the show element of the table to point to different routines depending on what output device to use for the show command of all list class elements), the following statements use the list class VMT:

```
mov     bx,offset @TABLEADDR_LIST
call    [(@table_list ptr es:bx).SHOW]
```

This is very similar to the sequence of instructions that Turbo Assembler uses to make the indirect call using the VMT.

More on calling methods

Often, you might find it necessary to call a parent object's method from inside a derived method procedure. You can also use the **CALL..METHOD** statement to do this.

You can use the **JMP** instruction with the **METHOD** extension in the same way you use the **CALL..METHOD** instruction. This instruction provides optimal tail recursion. See Chapter 12 for more information about the **CALL..METHOD** and **JMP..METHOD** instructions.

Creating an instance of an object

To create an instance of an object, you can call an object's constructor method (which allocates memory for an object instance) or allocate an instance of the object in a predefined (static) data segment.

You can create an instance of the object exactly the same way you create an instance of a structure. For example, examine the following instances of objects:

```
foolist  list {}                        ;instance of a list
fooqueue label queue
         queue {}                       ;instance of a queue
         queue {list_head=mynode,list_tail=mynode}
                                        ;instance of a queue
```

When you create an instance of an object, you can override any of the object's default data values as defined in the object declaration by specifying the overriding values inside the braces. You can't, however, override the methods for an object when you create an instance of an object.

Programming form for objects

It's a good idea to keep method procedures in a separate file from the method declaration, and from the code that uses the object. We recommend placing method procedures in a file with the name of the object and an extension of .ASM. For example, the method procedures for the linked-list object would go into the file LIST.ASM. The method procedure file must **INCLUDE** the method declaration from the .ASO file.

An example of the method procedures for the list object is described at the end of this chapter. This excerpt from the LIST.ASM file (on the example disks) shows the general structure of this file.

```
;--------------------------------
;-- Define Linked-List objects --
;--------------------------------

MODEL SMALL
LOCALS

;** Define Linked-List object **

INCLUDE node.aso

;** Create instance of Linked-List virtual method table **

DATASEG

TBLINST

;** Linked-List methods **

CODESEG

;;<<include all method procedures here>>
```

In general, you should use the following form for object-oriented programming in Turbo Assembler:

File	Contents
<object>.ASO	**INCLUDE**s *<parent_object>*.ASO, if any; contains **GLOBAL** object declaration and a **GLOBAL** directive for each method procedure.
<object>.ASM	**INCLUDE**s *<object>*.ASO; contains **TBLINST** directive and method procedure declarations; has an *init* method with a **TBLINIT** somewhere inside.

Note that you can use the **TBLINST** and **TBLINIT** directives even when there are currently no virtual methods in the object; in that case, no action is taken.

We therefore recommend using the **TBLINST** and **TBLINIT** directives regardless of whether virtual methods are currently present in an object: Place the **TBLINST** directive in an appropriate data segment and the **TBLINIT** directive in the object's initialization method (which must be a static method). You must call this method before using any other methods for the object.

CHAPTER 5

Using expressions and symbol values

Expressions and symbols are fundamental components of an assembly language program. Use expressions to calculate values and memory addresses. Symbols represent different kinds of values. This chapter describes the different types of these language components, and how you can use them.

Constants

Constants are numbers or strings that Turbo Assembler interprets as a fixed numeric value. You can use a variety of different numeric formats, including decimal, hexadecimal, binary, and octal.

Numeric constants

A numeric constant in Turbo Assembler always starts with a digit (0-9), and consists of an arbitrary number of alphanumeric characters. The actual value of the constant depends on the radix you select to interpret it. Radixes available in Turbo Assembler are binary, octal, decimal, and hexadecimal, as shown in Table 5.1:

Table 5.1 Radixes

Radix	Legal digits
Binary	0 1
Octal	0 1 2 3 4 5 6 7
Decimal	0 1 2 3 4 5 6 7 8 9
Hexadecimal	0 1 2 3 4 5 6 7 8 9 A B C D E F

Note that for hexadecimal constants, you can use both upper- and lowercase letters.

Turbo Assembler determines the radix of a numeric constant by first checking the LAST character of the constant. The characters in the following table determine the radix used to interpret the numeric constant.

Table 5.2
Characters determining radixes

Character	Radix
B	Binary
O	Octal
Q	Octal
D	Decimal
H	Hexadecimal

You can use both uppercase and lowercase characters to specify the radix of a number. If the last character of the numeric constant is not one of these values, Turbo Assembler will use the current default radix to interpret the constant. The following table lists the available numeric constants and their values.

Table 5.3
Numeric constants

Numeric constant	Value
77d	77 decimal
77h	77 hexadecimal
ffffh	Illegal; doesn't start with a digit
0ffffh	FFFF hexadecimal
88	Interpretation depends on current default radix

Changing the default radix

You can use the **RADIX** or **.RADIX** directives to change the current default radix. Use the following syntax for Ideal mode:

```
RADIX expression
```

Here's the MASM mode syntax:

```
.RADIX expression
```

expression must have a value of either 2 (binary), 8 (octal), 10 (decimal), or 16 (hexadecimal). Turbo Assembler assumes that the current default radix is decimal while it processes the **RADIX** directive.

String constants

String constants always begin with a single or double quote, and end with a matching single or double quote. Turbo Assembler converts the characters between the quotes to ASCII values.

Sometimes, you might want to include a quote within a string constant. To do this, use a pair of matching quotes as a single matching quote character within the string. For example,

`'It''s'` represents `It's`

Symbols

A symbol represents a value, which can be a variable, address label, or an operand to an assembly instruction and directive.

Symbol names

Symbol names are combinations of letters (both uppercase and lowercase), digits, and special characters. Symbol names can't start with a digit. Turbo Assembler treats symbols as either case sensitive or case insensitive. The command line switches **/ML**, **/MU**, and **/MX** control the case sensitivity of symbols.

See Chapter 2 for information about using these command-line switches.

Symbols names can be up to 255 characters in length. By default, symbol names are significant up to 32 characters. You can use the **/MV** command-line switch to change the number of characters of significance in symbols.

The underscore (_), question mark (?), dollar sign ($), and at-sign (@) can all be used as part of a symbol name. In MASM mode only, you can use a dot (.) as the first character of a symbol name. However, since it's easy to confuse a dot at the start of a symbol with the dot operator (which performs a structure member operation), it's better not to use it in symbol names.

Symbol types

Each symbol has a type that describes the characteristics and information associated with it. The way you define a symbol determines its type. For example, you can declare a symbol to represent a numeric expression, a text string, a procedure name, or a data variable. Table 5.4 lists the types of symbols that Turbo Assembler supports.

Table 5.4
Symbol types

Symbol type	Description
address	An address. Data subtypes are UNKNOWN,BYTE,WORD,DWORD, PWORD or FWORD, QWORD, TBYTE, and an address of a named structure or table. Code subtypes are SHORT, NEAR, and FAR
text_macro	A text string
alias	An equivalent symbol
numerical_expr	The value of a numerical expression
multiline_macro	Multiple text lines with dummy arguments
struc/union	A structure or union data type
table	A table data type
struc/*table_member*	A structure or table member
record	A record data type
record_field	A record field
enum	An enumerated data type

Table 5.4: Symbol types (continued)

segment	A segment
group	A group
type	A named type
proctype	A procedure description type

Simple address subtypes

Symbols subtypes describe whether the symbol represents the address of a byte, a word, and so forth. Table 5.5 shows the simple address subtypes that Turbo Assembler provides.

Table 5.5: Address subtypes

Type expression	Meaning
UNKNOWN	Unknown or undetermined address subtype.
BYTE	Address describes a byte.
WORD	Address describes a word.
DWORD	Address describes a 4-byte quantity.
PWORD or FWORD	Address describes a 6-byte quantity.
QWORD	Address describes an 8-byte quantity.
TBYTE	Address describes a 10-byte quantity.
SHORT	Address describes a short label/procedure address.
NEAR	Address describes a near label/procedure address.
FAR	Address describes a far label/procedure address.
PROC	Address describes either a near or far label/procedure address, depending on the currently selected programming model.
DATAPTR	Address describes either a word, dword, or pword quantity, depending on the currently selected programming model.
CODEPTR	Address describes either a word, dword, or pword quantity, depending on the currently selected programming model.
struc/*union_name*	Address describes an instance of the named structure or union.
table_name	Address describes an instance of the named table.
record_name	Address describes an instance of the named record; either a byte, word, or dword quantity.
enum_name	Address describes an instance of the named enumerated data type; either a byte, word, or dword quantity.
type_name	Address describes an instance of the named type.
TYPE *expression*	Address describes an item whose subtype is the address subtype of the expression; Ideal mode only.
proctype_name	Address describes procedure of proctype.

Describing a complex address subtype

Several directives let you declare and use complex address subtypes. These type expressions are similar to C in that they can represent multiple levels of pointer indirection, for example, the complex type expression

```
PTR WORD
```

represents a pointer to a word. (The size of the pointer depends on the segmentation model you selected with **MODEL**.)

Table 5.6 shows a syntax summary of complex address subtypes:

Table 5.6
Complex address subtypes

Syntax	Meaning
simple_address_subtype	the specified address subtype
[*dist*]PTR[*complex_address_subtype*]	a pointer to the specified complex address subtype, the size of which is determined by the current **MODEL** or by the specified distance, if present

You can describe the optional distance parameter in the following ways:

Table 5.7
Distance syntax

Syntax	Meaning
NEAR	use a near pointer; can be either 16 or 32 bits, depending on the current model
FAR	use a far pointer; can be either 32 or 48 bits, depending on current model
SMALL NEAR	use a 16-bit pointer; 80386 and 80486 only
LARGE NEAR	use a 32-bit near pointer; 80386 and 80486 only
SMALL FAR	use a 32-bit far pointer; 80386 and 80486 only
LARGE FAR	use a 48-bit far pointer; 80386 and 80486 only

The type of the object being pointed to is not strictly required in complex pointer types; Turbo Assembler only needs to know the size of the type. Therefore, forward references are permitted in complex pointer types (but not in simple types).

Expressions

Using expressions lets you produce modular code, because you can represent program values symbolically. Turbo Assembler performs any recalculations required because of changes (rather than requiring you to do them).

Turbo Assembler uses standard infix notation for equations. Expressions can contain operands and unary or binary operators. Unary operators are placed before a single operand; binary operators are placed between two operands. Table 5.8 shows examples of simple expressions.

Table 5.8
Simple expressions

Expression	Evaluates to
5	constant 5
–5	constant –5
4+3	constant 7
4*3	constant 12
4*3+2*1	constant 14
4*(3+2)*1	constant 21

Appendix B contains the full Backus-Naur form (BNF) grammar that Turbo Assembler uses for expression parsing in both MASM and Ideal modes. This grammar inherently describes the valid syntax of Turbo Assembler expressions, as well as operator precedence.

Expression precision

Turbo Assembler always uses 32-bit arithmetic in Ideal mode. In MASM mode, Turbo Assembler uses either 16- or 32-bit arithmetic, depending on whether you select the 80386 processor. Therefore, some expressions might produce different results depending on which processor you've selected. For example,

(1000h * 1000h) / 1000h

evaluates to 1000h if you select the 80386 processor, or to 0 if you select the 8086, 80186, or 80286 processors.

Constants in expressions

You can use constants as operands in any expression. For example,

```
mov ax,5        ;"5" is a constant operand
```

Symbols in expressions

When you use a symbol in an expression, the returned value depends on the type of symbol. You can use a symbol by itself or in conjunction with certain unary operators that are designed to extract other information from the entity represented by the symbol.

Registers

Register names represent 8086-family processor registers, and are set aside as part of the expression value. For example,

5+ax+7

This expression has a final value of ax+12, because AX is a register symbol that Turbo Assembler sets aside. The following list contains register symbols:

8086	AX,BX,CX,DX,SI,DI,BP,CS,DS,ES,SS
80186,80286	Same as 8086
80386	8086 registers, plus EAX, EBX, ECX, EDX, ESI, EDI, EBP, FS, GS, CR0, CR2, CR3, DR0, DR1, DR2, DR3, DR6, DR7
80486	80386 registers, plus: TR3, TR4, TR5

Standard symbol values

Some symbols always represent specific values and don't have to be defined for you to use them. The following table lists these symbols and their values.

Table 5.9
Standard symbols

Symbol	Value
$	Current program counter
NOTHING	0
?	0
UNKNOWN	0
BYTE	1
WORD	2
DWORD	4
PWORD	6
FWORD	6
QWORD	8
TBYTE	10
NEAR	0ffffh
FAR	0fffeh
PROC	Either 0ffffh or 0fffeh, depending on current model
CODEPTR	Either 2 or 4, depending on current model
DATAPTR	Either 2 or 4, depending on current model

Simple symbol values

Turbo Assembler returns the following values for symbols used by themselves:

Table 5.10: Values of symbols used by themselves

Expression	Value
address_name	Returns the address.
numerical_expr_name	Returns the value of the numerical expression.
table_name | *table_member_name*	Returns the default value for the table member specified in the definition of the table.
struc/*table_member_name*	Returns the offset of the member within the table or structure (MASM mode only).
record_name	Returns a mask where the bits reserved to represent bit fields in the record definition are 1, the rest are 0.
record_name <...>	Returns the initial value a record instance would have if it were declared with the same text enclosed in angle brackets (see Chapter 12 for details).
record_name {...}	Similar to *record_name* <...>.

Table 5.10: Values of symbols used by themselves (continued)

record_field_name	Returns the number of bits the field is displaced from the low order bit of the record (also known as the *shift value*).
enum_name	Returns a mask where the bits required to represent the maximum value present in the enum definition are 1, the rest are 0.
segment_name	Returns the segment value.
group_name	Returns the group value.
struc/union_name	Returns the size in bytes of the structure or union, but only if it is 1, 2, or 4; all other sizes return a value of 0.
type_name	If the type is defined as a synonym for a structure or union, the value returned is the same as for a structure or union. Otherwise, the size of the type is returned (with 0ffffh for short and near labels, and 0fffeh for far labels).
proctype_name	Returns 0FFFFh if the proctype describes a near procedure, or 0FFFEh for a far procedure.

All other symbol types return the value 0.

Note that when you use a text macro name in an expression, Turbo Assembler substitutes the string value of the text macro for the text macro symbol. Similarly, when you use an alias name, Turbo Assembler substitutes the symbol value that the alias represents for the alias symbol.

The LENGTH unary operator

The **LENGTH** operator returns information about the count or number of entities represented by a symbol. The actual value returned depends on the type of the symbol, as shown in the following table.

Table 5.11: LENGTH operator return values

Expression	Value
LENGTH *address_name*	Returns the count of items allocated when the address name was defined.
LENGTH struc/*table_member_name*	Returns the count of items allocated when the member was defined (MASM mode only)

The length operator (when applied to all other symbol types) returns the value 1. Here are some examples using the **LENGTH** operator:

```
MSG    DB "Hello"
array  DW 10 DUP (4 DUP (1),0)
numbrs DD 1,2,3,4
lmsg = LENGTH msg                    ;=1, no DUP
larray = LENGTH array                ;=10, DUP repeat count
lnumbrs = LENGTH numbrs              ;=1, no DUP
```

The SIZE unary operator

The **SIZE** operator returns size information about the allocated data item. The value returned depends on the type of the symbol you've specified. The following table lists the available values for **SIZE**.

Table 5.12: SIZE values

Expression	Value
SIZE *address_name*	In Ideal mode, returns the actual number of bytes allocated to the data variable. In MASM mode, returns the size of the subtype of address_name (UNKNOWN=0, BYTE=1, WORD=2, DWORD=4, PWORD=FWORD=6, QWORD=8, TBYTE=10, SHORT=NEAR=0ffffh, FAR=0fffeh, structure address=size of structure) multiplied by the value of **LENGTH** *address_name*.
SIZE *struc/union_name*	Returns the number of bytes required to represent the structure or union.
SIZE *table_name*	Returns the number of bytes required to represent the table.
SIZE struc/*table_member_name*	Returns the quantity TYPE struc/*table_member_name* * LENGTH struc/*table_member_name* (MASM mode only).
SIZE *record_name*	Returns the number of bytes required to represent the total number of bits reserved in the record definition; either 1, 2, or 4.
SIZE *enum_name*	Returns the number of bytes required to represent the maximum value present in the enum definition; either 1, 2, or 4
SIZE *segment_name*	Returns the size of the segment in bytes.
SIZE *type_name*	Returns the number of bytes required to represent the named type, with short and near labels returning 0ffffh, and far labels returning 0fffeh.

The SIZE operator returns the value 0 when used on all other symbol types.

The WIDTH unary operator

The WIDTH operator returns the width in bits of a field in a record. The value depends on the type of symbol. The following table shows these types of symbols. You can't use WIDTH for any other symbol types.

Table 5.13 WIDTH values

Expression	Value
WIDTH *record_name*	Returns the total number of bits reserved in the record definition.
WIDTH *record_field_name*	Returns the number of bits reserved for the field in the record definition.
WIDTH *enum_name*	Returns the number of bits required to represent the maximum value in the enum definition.

The MASK unary operator

The MASK operator creates a mask from a bit field, where bits are set to 1 in the returned value and correspond to bits in a field that a symbol represents. The value returned depends on the type of symbol, as shown in the following table. Note that you can't use MASK on any other symbols.

Table 5.14
MASK return values

Expression	Value
MASK *record_name*	Returns a mask where the bits reserved to represent bit fields in the record definition are 1, the rest 0.
MASK *record_field_name*	Returns a mask where the bits reserved for the field in the record definition are 1, the rest 0.
MASK *enum_name*	Returns a mask where the bits required to represent up to the maximum value present in the enum definition are 1, the rest 0.

General arithmetic operators

General arithmetic operators manipulate constants, symbol values, and the values of other general arithmetic operations. Common operators are addition, subtraction, multiplication, and division. Others operators are more specifically tailored for assembly language programming. We'll discuss a little about all of these in the next few sections.

Simple arithmetic operators

Turbo Assembler supports the simple arithmetic operators shown in the following table.

Table 5.15
Simple arithmetic operators

Expression	Value
+ *expression*	Expression.
– *expression*	Negative of expression.
expr1 + *expr2*	*expr1* plus *expr2*.
expr1 – *expr2*	*expr1* minus *expr2*.
expr1 * *expr2*	*expr1* multiplied by *expr2*.
expr1 / *expr2*	*expr1* divided by *expr2* using signed integer division; note that *expr2* cannot be 0 or greater than 16 bits in extent.
expr1 MOD *expr2*	Remainder of *expr1* divided by *expr2*; same rules apply as for division.

Logical arithmetic operators

Logical operators let you perform Boolean algebra. Each of these operators performs in a bitwise manner; that is, the logical operation is performed one bit at a time. The following table shows the logical operators.

Table 5.16
Logical arithmetic operators

Expression	Value
NOT *expression*	*expression* bitwise complemented
expr1 **AND** *expr2*	*expr1* bitwise **AND**ed with *expr2*
expr1 **OR** *expr2*	*expr1* bitwise **OR**ed with *expr2*
expr1 **XOR** *expr2*	*expr1* bitwise **XOR**ed with *expr2*

Bit shift operators

Shift operators move values left or right by a fixed number of bits. You can use them to do quick multiplication or division, or to access the value of a bitfield within a value. The following table lists the bit shift operators.

Table 5.17 Bit shift operators

Expression	Value
expr1 SHL *expr2*	*expr1* shifted left by *expr2* bits (shifted right if *expr2* is negative).
expr1 SHR *expr2*	*expr1* shifted right by *expr2* bits (shifted left if *expr2* is negative).

Note that the SHL and SHR operators shift in 0s from the right or left to fill the vacated bits.

Comparison operators

Comparison operators compare two expressions to see if they're equal or unequal, or if one is greater than or less than the other. The operators return a value of –1 if the condition is true, or a value of 0 if the condition is not true. The following table shows how you can use these operators.

Table 5.18 Comparison operators

Expression	Value
expr1 EQ *expr2*	–1 if *expr1* is equal to *expr2*; otherwise, 0.
expr1 NE *expr2*	–1 if *expr1* is not equal to *expr2*; otherwise, 0.
expr1 GT *expr2*	–1 if *expr1* is greater than *expr2*; otherwise, 0.
expr1 GE *expr2*	–1 if *expr1* is greater than or equal *expr2*; otherwise, 0.
expr1 LT *expr2*	–1 if *expr1* is less than *expr2*; otherwise, 0.
expr1 LE *expr2*	–1 if *expr1* is less than or equal *expr2*; otherwise, 0.

EQ and NE treat expressions as unsigned numbers. For example, –1 EQ 0ffffh has a value of –1 (unless you've selected the 80386 processor or used Ideal mode; then, –1 EQ 0ffffffffh has a value of –1).

GT, GE, LT, and LE treat expressions as signed numbers. For example, 1 GE –1 has a value of –1, but 1 GE 0ffffh has a value of 0.

Setting the address subtype of an expression

Turbo Assembler provides operators that let you override or change the type of an expression. The following table lists these operators.

Table 5.19: Type override operators

Expression	Value
expr1 PTR *expr2*	Converts *expr2* to the type determined by *expr1*, where 0=UNKNOWN, 1=BYTE, 2=WORD, 4=DWORD, 6=PWORD, 8=QWORD, 10=TBYTE, 0ffffh=NEAR, 0fffeh=FAR, all others=UNKNOWN; MASM mode only.

Table 5.19: Type override operators (continued)

type PTR *expression* or type *expression*	Converts expression to the specified address subtype; Ideal mode only.
type LOW *expression*	Converts *expression* to the specified address subtype. Type described must be smaller in size than the type of the expression; Ideal mode only.
type HIGH *expression*	Converts *expression* to the specified address subtype. Type described must be smaller in size than the type of the expression; the resulting address is adjusted to point to the high part of the object described by the address expression; Ideal mode only.

Here are some examples:

```
IDEAL
big DD 12345678h
MOV ax,[WORD big]                  ;ax=5678h
MOV al,[BYTE PTR big]              ;al=78h
MOV ax,[WORD HIGH big]             ;ax=1234h
MOV ax,[WORD LOW big]              ;ax=5678h
MOV al,[BYTE LOW WORD HIGH big]    ;al = 3rd byte of big = 34h
MASM
MOV ax,2 PTR big                   ;ax=5678h
MOV ax,WORD PTR big                ;ax=5678h (WORD has value 2)
```

Obtaining the type of an expression

In MASM mode, you can obtain the numeric value of the type of an expression by using the TYPE operator. (You can't do this in Ideal mode, because types can never be described numerically). The syntax of the TYPE operator is

```
TYPE expression
```

The TYPE operator returns the size of the object described by the address expression, as follows:

Table 5.20 TYPE values

Type	Description
byte	1
word	2
dword	4
pword	6
qword	8
tbyte	10
short	0ffffh
near	0ffffh
far	0fffeh

Table 5.20: TYPE values (continued)

struct/union	Size of a structure or union instance
table	Size of a table instance
proctype	Returns 0FFFFh if the proctype describes a near procedure, or 0FFFEh for a far procedure

Here's an example:

```
avar = 5
bvar   db 1
darray dd 10 dup (1)

x   struc
    dw ?
    dt ?
    ends
fp label far
tavar = TYPE avar          ;=0
tbvar = TYPE bvar          ;= 1
tdarray = TYPE darray      ;= 4
tx = TYPE x                ;= 12
tfp = TYPE fp              ;= 0FFFEh
```

Overriding the segment part of an address expression

Address expressions have values consisting of a segment and an offset. You can specify the segment explicitly as a segment register, or as a segment or group value. (If you specify it as a group value, Turbo Assembler determines which segment register to use based on the values that the segment registers are ASSUMEd to be.) Use the following syntax to change the segment part of an address expression:

```
expr1 : expr2
```

This operation returns an address expression using the offset of *expr2*, and *expr1* as a segment or group value. For example,

```
VarPtr  dd  dgroup:memvar      ;dgroup is a group
        mov cl,es:[si+4]       ;segment override ES
```

Obtaining the segment and offset of an address expression

You can use the SEG and OFFSET operators to get the segment and offset of an expression. The SEG operator returns the segment value of the address expression. Here's its syntax:

```
SEG expression
```

Here is a code example:

```
DATASEG
temp  DW 0
CODESEG
```

```
mov ax,SEG temp
mov ds,ax
ASSUME  ds:SEG temp
```

The OFFSET operator returns the offset of the address expression. Its syntax follows:

```
OFFSET expression
```

Note that when you use the offset operator, be sure that the expression refers to the correct segment. For example, if you are using MASM mode and not using the simplified segmentation directives, the expression

```
OFFSET BUFFER          ;buffer is a memory address
```

is not the same as

```
OFFSET DGROUP:BUFFER   ;Dgroup is the group containing the segment that contains
                        BUFFER
```

unless the segment that contains BUFFER happens to the first segment in DGROUP.

In Ideal mode, addresses are automatically calculated relative to any group that a segment belongs to unless you override them with the **:** operator. In MASM mode, the same is true if you use the simplified segment directives. Otherwise, addresses are calculated relative to the segment an object is in, rather than any group.

Creating an address expression using the location counter

You can use the THIS operator to create an address expression that points to the current segment and location counter, and has a specific address subtype. You can use the following syntax in Ideal mode:

```
THIS type
```

The Ideal mode syntax lets you build an address expression from the current segment and location counter of the specified type.

You can use the next syntax in MASM mode:

```
THIS expression
```

The MASM mode syntax functions like the syntax in Ideal mode, but uses the numerical value of the expression to determine the type. These values are: 0=UNKNOWN, 1=BYTE, 2=WORD, 4=DWORD, 6=PWORD, 8=QWORD, 10=TBYTE, 0ffffh=NEAR, 0fffeh=FAR. For example,

```
ptr1 LABEL WORD
ptr2 EQU THIS WORD        ;similar to ptr1
```

Determining the characteristics of an expression

Sometimes, it's useful to determine (within a macro) whether an expression has specific characteristics. The SYMTYPE and .TYPE operators let this happen.

The Ideal mode syntax:

```
SYMTYPE expression
```

The SYMTYPE and .TYPE operators are exactly equivalent; however, .TYPE is available only in MASM mode, and you can use SYMTYPE only in Ideal mode.

The MASM mode syntax:

```
.TYPE expression
```

SYMTYPE and .TYPE both return a constant value that describes the expression. This value is broken down into the bit fields shown in the following table.

Table 5.21 Bit fields from SYMTYPE and .TYPE

Bit	Meaning
0	Expression is a program relative memory pointer.
1	Expression is a data relative memory pointer.
2	Expression is a constant value.
3	Expression uses direct addressing mode.
4	Expression contains a register.
5	Symbol is defined.
7	Expression contains an externally defined symbol.

The expression uses register indirection ([BX]) if bits 2 and 3 are both zero.

If Turbo Assembler can't evaluate the expression, SYMTYPE returns appropriate errors. .TYPE, however, will return a value in these situations (usually 0).

Referencing structure, union, and table member offsets

Structure, union, and table members are global variables whose values are the offset of the member within the structure, union, or table in MASM mode. In Ideal mode, however, members of these data types are considered local to the data type. The dot (.) operator lets you obtain the offsets of members. Here's the Ideal mode syntax:

```
expression . symbol
```

expression must represent an address of a structure, union, or table instance. *symbol* must be a member of the structure, union, or table. The dot operator returns the offset of the member within the structure.

MASM mode also contains a version of the dot operator. However, its function is similar to the + operator, and has the following syntax:

```
expr1 . expr2
```

Describing the contents of an address

Many instructions require you to distinguish between an address and the contents of an address. You can do this by using square brackets ([]). For example,

```
MOV AX,BX         ;move BX into AX.
MOV AX,[BX]       ;move contents of address BX into AX
```

Here's the general syntax for using square brackets:

```
[ expression ]
```

In MASM mode, the brackets are optional for expressions that are addresses. Complete addresses can't be used as an operand for any 80x86 instruction; rather, only the segment (obtained with the SEG operator) or the offset (obtained with the OFFSET operator) is used.

In Ideal mode, a warning is given when an expression is clearly an address, but no brackets are present. You can disable this warning (see Chapter 12 for further information). However, it's good programming practice to include these brackets.

Implied addition

In MASM mode, you can add expressions in several ways: using the addition operator (+), using the dot operator (.), or by implied addition (when expressions are separated by brackets or parentheses). For example,

```
MOV AX,5[BX]       ;contents of address BX+5
MOV AX,5(XYZ)      ;contents of address XYZ+5
```

Here's the general syntax for implicit addition:

```
expr1 [ expr2 ]
```

or

```
expr1 ( expr2 )
```

Obtaining the high or low byte values of an expression

You can use the HIGH and LOW operators on an expression to return its high and low byte values. This information can be useful in circumstances where, for example, only the high 8 bits of an address offset is required.

Here's the syntax of the HIGH and LOW operators:

```
HIGH expression
LOW expression
```

For example,

```
magic equ 1234h
mov cl,HIGH magic      ;cl=12h
mov cl,LOW magic       ;cl=34h
```

Specifying a 16- or 32-bit expression

When the currently selected processor is the 80386 or higher, Turbo Assembler provides two operators that let you control whether an expression is interpreted as a 16-bit value or as a 32-bit value: the SMALL and LARGE operators. Here are their syntaxes:

SMALL *expression*
LARGE *expression*

The SMALL operator flags the expression as representing a 16-bit value. LARGE flags it as representing a 32-bit value. These operators are particularly important when you program for an environment in which some segments are 32-bit and others are 16-bit. For example, the instruction

```
JMP [DWORD PTR ABC]
```

represents an indirect jump to the contents of the memory variable ABC. If you have enabled the 80386 processor, this instruction could be interpreted as either a far jump with a segment and 16-bit offset, or a near jump to a 32-bit offset. You can use SMALL or LARGE to remove the ambiguity, as follows:

```
JMP SMALL [DWORD PTR ABC]
```

This instruction causes Turbo Assembler to assemble the jump instruction so that the value read from ABC is interpreted as a 16-bit segment and 16-bit offset. Turbo Assembler then performs an indirect FAR jump.

When you use SMALL or LARGE within the address portion of an expression, the operators indicate that the address is a 32-bit address. For example,

```
JMP SMALL [LARGE DWORD PTR ABC]
```

indicates that a large 32-bit address describes the memory variable ABC, but its contents are interpreted as a 16-bit segment and 16-bit offset.

C H A P T E R 6

Choosing processor directives and symbols

The 8086 processor is actually only one of a family of processors known as the iAPx86 family. Members of this family include

- The 8088 (which contains an 8-bit data bus), the 8086 (containing a 16-bit data bus)
- The 80186 and 80188 (like the 8086 and 8088 but contain additional instructions and run faster than their predecessors)
- The 80286 (which contains instructions for protected mode)
- The 80386 (which can process 16- and 32-bit data)
- The 80486 (an enhanced version of 80386 that runs even faster).
- The Pentium (an even faster version of the 80486).

Math coprocessors such as the 8087, 80287, and 80387 work with the iAPx86 family so that you can perform floating-point operations.

Turbo Assembler provides directives and predefined symbols that let you use the instructions included for particular processors. This chapter describes these directives and symbols.

iAPx86 processor directives

To find out which instructions go with certain processors, refer to the books listed in Chapter 1.

The iAPx86 family provides a variety of directives for you to use. In the following directives, note that those beginning with . are only available in MASM mode.

Table 6.1 Processor directives

Directive	Meaning
P8086	Enables assembly of 8086 instructions only.
.8086	Enables assembly of the 8086 instructions and disables all instructions available only on the 80186, 80286, and 386 processors. It also enables the 8087 coprocessor instructions exactly as if the **.8087** or **8087** had been issued.
P186	Enables assembly of 80186 instructions.

Table 6.1: Processor directives (continued)

.186	Enables assembly of 80186 instructions.
P286	Enables assembly of all 80286 instructions.
P286N	Enables assembly of nonprivileged 80286 instructions.
P286P	Enables assembly of privileged 80286 instructions.
.286	Enables assembly of nonprivileged 80286 instructions. It also enables the 80287 numeric processor instructions exactly as if the **.286** or **P287** directive had been issued.
.286C	Enables assembly of nonprivileged 80286 instructions.
.286P	Enables assembly of all the additional instructions supported by the 80286 processor, including the privileged mode instructions. It also enables the 80287 numeric processor instructions exactly as if the **.287** or **P287** directive had been issued.
P386	Enables assembly of all 386 instructions.
P386N	Enables assembly of all nonprivileged 386 instructions.
P386P	Enables assembly of privileged 386 instructions.
.386	Enables assembly of the additional instructions supported by the 386 processor in nonprivileged mode. It also enables the 80387 numeric processor instructions exactly as if the **.387** or **P387** directive had been issued.
.386C	Enables assembly of 386 instructions.
.386P	Enables assembly of all the additional instructions supported by the 386 processor, including the privileged mode instructions. It also enables the 80387 numeric processor instructions exactly as if the **.387** or **P387** directive had been issued.
P486	Enables assembly of all i486 instructions.
P486N	Enables assembly of nonprivileged i486 instructions.
.486	Enables assembly of the additional instructions supported by the i486 processor in nonprivileged mode. It also enables the 387 numeric processor instructions exactly as if the **.387** or **P387** directive had been issued.
.486C	Enables assembly of all i486 instructions.
.486P	Enables assembly of all the additional instructions supported by the i486 processor, including the privileged mode instructions. It also enables the 80387 numeric processor instructions exactly as if the **.387** or **P387** directive had been issued.
.487	Enables assembly of 487 numeric processor instructions. This instruction works only in MASM mode.
P487	Enables assembly of 487 numeric processor instructions. This instruction works in both MASM and Ideal modes.
P586	Enables assembly of all Pentium instructions.
P586N	Enables assembly of nonprivileged Pentium instructions.
.586	Enables assembly of the additional instructions supported by the Pentium processor in nonprivileged mode.
.586C	Enables assembly of all Pentium instructions.
.586P	Enables assembly of all the additional instructions supported by the Pentium processor, including the privileged mode instructions.

Table 6.1: Processor directives (continued)

.587	Enables assembly of Pentium numeric processor instructions. This instruction works only in MASM mode.
P587	Enables assembly of Pentium numeric processor instructions. This instruction works in both MASM and Ideal modes.

Predefined symbols

Two predefined symbols, @Cpu and @WordSize, can give you information about the type of processor you're using, or the size of the current segment. Here are descriptions of these symbols:

@Cpu

Function Numeric equate that returns information about current processor

Remarks The value returned by **@Cpu** encodes the processor type in a number of single-bit fields:

Bit	Description
0	8086 instructions enabled
1	80186 instructions enabled
2	80286 instructions enabled
3	386 instructions enabled
4	486 instructions enabled
5	586 instructions enabled
7	Privileged instructions enabled (80286, 386, 486)
8	8087 numeric processor instructions
10	80287 numeric processor instructions
11	387 numeric processor instructions

The bits not defined here are reserved for future use. Mask them off when using **@Cpu** so that your programs will remain compatible with future versions of Turbo Assembler.

Since the 8086 processor family is upward compatible, when you enable a processor type with a directive like **.286**, the lower processor types (8086, 80186) are automatically enabled as well.

This equate *only* provides information about the processor you've selected at assembly time using the **.286** and related directives. The processor type and the CPU your program is executing on at run time are not indicated.

Example

```
IPUSH = @Cpu AND 2  ;allow immediate push on 186 and above
IF IPUSH
PUSH 1234
ELSE
   mov  ax,1234
   push ax
ENDIF
```

@WordSize

Function Numeric equate that indicates 16- or 32-bit segments

Remarks **@WordSize** returns 2 if the current segment is a 16-bit segment, or 4 if the segment is a 32-bit segment.

Example

```
IF @WordSize EQ 4
   mov esp,0100h
ELSE
   mov sp,0100h
ENDIF
```

8087 coprocessor directives

The following table contains the available math coprocessor directives. Again, directives beginning with a dot (.) work only in MASM mode.

Table 6.2
8087 coprocessor directives

Directive	Meaning
.287	Enables assembly of all the 80287 numeric coprocessor instructions. Use this directive if you know you'll never run programs using an 8087 coprocessor. This directive causes floating-point instructions to be optimized in a manner incompatible with the 8087, so *don't* use it if you want your programs to run using an 8087.
.387	Enables assembly of all the 80387 numeric coprocessor instructions. Use this directive if you know you'll never run programs using an 8087 coprocessor. This directive causes floating-point instructions to be optimized in a manner incompatible with the 8087, so *don't* use it if you want your programs to run using an 8087.
.487	Enables assembly of all 80486 numeric instructions.
.587	Enables assembly of all Pentium numeric instructions.
.8087	Enables all the 8087 coprocessor instructions, and disables all those coprocessor instructions available only on the 80287 and 80387. (The default.)
P287	Enables assembly of 80287 coprocessor instructions.
P387	Enables assembly of 80387 coprocessor instructions.

Table 6.2: 8087 coprocessor directives (continued)

P487	Enables assembly of all 80486 numeric instructions.
P587	Enables assembly of all Pentium numeric instructions.
P8087	Enables assembly of 8087 coprocessor instructions.

Coprocessor emulation directives

If you need to use real floating-point instructions, you must use an 80x87 coprocessor. If your program has installed a software floating-point emulation package, you can use the **EMUL** directive to use it. (**EMUL** functions like **/e**.)

Both **EMUL** and **NOEMUL** work in MASM and Ideal modes.

For example,

```
Finit         ;real 80x87 coprocessor instruction
EMUL
Fsave BUF     ;emulated instruction]
```

If you're using an 80x87 coprocessor, you can either emulate floating-point instructions using **EMUL**, or force the generation of real floating-point instructions with the **NOEMUL** directive. Note that you can use **EMUL** and **NOEMUL** when you want to generate real floating-point instructions in one portion of a file, and emulated instructions in another.

Here's an example using **NOEMUL**:

```
NOEMUL        ;assemble real FP instructions
finit
EMUL          ;back to emulation
```

CHAPTER 7

Using program models and segmentation

Each processor in the 80x86 family has at least four segment registers: CS, DS, ES, and SS. These registers contain a segment value that describes a physical block of memory up to 64K in length (or up to 4 gigabytes on the 80386 and above). All addresses are calculated using one of these segment registers as a base value.

The meaning of the value stored in a segment register differs depending on whether the processor is using *real mode* (the ONLY mode available for the 8086 and 80186), where the segment value is actually a paragraph number, or *protected mode*, where a segment register contains a *selector* (which has no numerical significance).

The operating system or platform for a program determines whether the program operates in real mode or protected mode. If you use protected mode on the 80386 or 80486, the operating system also determines whether large (4 gigabyte) segments are permitted. Turbo Assembler supports all of these environments equally well.

In the general 80x86 model, programs are composed of one or more segments, where each segment is a physically distinct piece of code or data (or both) designed to be accessed by using a segment register. From this general scheme, many arbitrary organizations are possible. To apply some order to the chaos, some standard memory models have been devised. Since many high-level languages adhere to these conventions, your assembly language programs should also.

One obvious way to break up a program is to separate the program instructions from program data. You can classify each piece of program data as initialized (containing an initial value, such as text messages), or uninitialized (having no starting value). Turbo Assembler usually assigns uninitialized data to a separate segment so that it can be placed at the end of the program, reducing the size of the executable program file.

The stack is usually a fairly large portion of the uninitialized data. It's also special because the SS and SP registers are usually initialized automatically to the stack area when you execute a program. Thus, the standard memory models treat the stack as a separate segment.

You can also combine segments into groups. The advantage of using groups is that you can use the same segment value for all the segments in the group. For example, initialized data, uninitialized data, and stack segments are often combined into a group so that the same segment value can be used for all of the program data.

This chapter describes how to use models and segments in your code and the directives that make this possible.

The MODEL directive

The **MODEL** directive lets you specify one of several standard segmentation models for your program. You can also use it to specify a language for the procedures in your program.

Here's the syntax for the **MODEL** directive:

```
MODEL [model_modifier] memory_model [code_segment_name]
  [, [language_modifier] language ]
  [, model_modifier]
```

In MASM mode, you can use the same syntax, but with the **.MODEL** directive.

memory_model and *model_modifier* specify the segmentation memory model to use for the program.

The standard memory models available in Turbo Assembler have specific segments available for:

- code
- initialized data
- uninitialized data
- far initialized data
- far uninitialized data
- constants
- stack

The code segment usually contains a module's code (but it can also contain data if necessary). Initialized data and constants are treated separately for compatibility with some high level languages. They contain data such as messages where the initial value is important. Uninitialized data and stack contain data whose initial value is unimportant. Far initialized data is initialized data that is not part of the standard data segment, and can be reached only by changing the value of a segment register. A module can have more than one far initialized data segment. Far uninitialized data is similar, except that it contains uninitialized data instead of initialized data.

The specific memory model determines how these segments are referenced with segment registers, and how they are combined into groups (if at all). When writing a program, you should keep these segments separate, regardless of the program's size. Then, you can select the proper model to group the segments together. If you keep these segments separate and your program grows, you can choose a larger model.

The memory model is the only required parameter of the **MODEL** directive. Table 7.1 describes each of the standard memory models.

The *model_modifier* field lets you change certain aspects of the model. You can specify more than one model modifier, if you wish. Table 7.2 shows the available model modifiers.

Note that you can specify the model modifier in two places, for compatibility with MASM 5.2. If you don't use a model specifier, Turbo Assembler assumes the NEARSTACK modifier, and USE32 (if the 80386 or 80486 processor is selected). Unless otherwise specified, DOS is the platform.

Use the optional *code_segment_name* field in the large code models to override the default name of the code segment. Normally, this is the module name with _TEXT appended to it.

Table 7.1: Standard memory models

Model	Code	Data	Register assumptions	Description
TINY	near	near	cs=dgroup ds=ss=dgroup	All code and data combined into a single group called DGROUP. This model is used for .COM assembly programs. Some languages don't support this model.
SMALL	near	near	cs=_text ds=ss=dgroup	Code is in a single segment. All data is combined into a group called DGROUP. This is the most common model for stand-alone assembly programs.
MEDIUM	far	near	cs=<*module*>_text ds=ss=dgroup	Code uses multiple segments, one per module. Data is in a group called DGROUP.
COMPACT	near	far	cs=_text ds=ss=dgroup	Code is in a single segment. All near data is in a group called DGROUP. Far pointers are used to reference data.

Table 7.1: Standard memory models (continued)

LARGE	far	far	cs=<*module*>_text ds=ss=dgroup	Code uses multiple segments, one per module. All near data is in a group called DGROUP. Far pointers are used to reference data.
HUGE	far	far	cs=<*module*>_text ds=ss=dgroup	Same as LARGE model, as far as Turbo Assembler is concerned.
TCHUGE	far	far	cs=<*module*>_text ds=nothing ss=nothing	This is the same as the LARGE model, but with different segment register assumptions.
TPASCAL	near	far	cs=code ds=data ss=nothing	This is a model to support early versions of Borland Pascal. It's not required for later versions.
FLAT	near	near	cs=_text ds=ss=flat	This is the same as the SMALL model, but tailored for use under OS/2.

Table 7.2
Model modifiers

Model modifier	**Function**
NEARSTACK	Indicates that the stack segment should be included in DGROUP (if DGROUP is present), and SS should point to DGROUP.
FARSTACK	Specifies that the stack segment should never be included in DGROUP, and SS should point to nothing.
USE16	Specifies (when the 80386 or 80486 processor is selected) that 16-bit segments should be used for all segments in the selected model.
USE32	Indicates (when the 80386 or 80486 processor is selected) that 32-bit segments should be used for all segments in the selected model.
DOS, OS_DOS	Specifies that DOS is the platform for the application.
NT, OS_NT	Specifies that Windows NT is the platform for the application.
OS2, OS_OS2	Specifies that OS2 is the platform for the application.

language and *language_modifier* together specify the default procedure calling conventions, and the default style of the prolog and epilog code present in each procedure. They also control how to publish symbols externally for the linker to use. Turbo Assembler will automatically generate the procedure entry and exit code that is proper for procedures using any of the following interfacing conventions: PASCAL, C, CPP (C++), SYSCALL, STDCALL, BASIC, FORTRAN, PROLOG, and NOLANGUAGE. If you don't specify a language, Turbo Assembler assumes the default language to be NOLANGUAGE.

Use *language_modifier* to specify additional prolog and epilog code when you write procedures for Windows, or for the Borland Overlay loader. These options are: NORMAL, WINDOWS, ODDNEAR and ODDFAR. If you don't specify an option, Turbo Assembler assumes the default to be NORMAL.

Also note that you can override the default language and language modifier when you define a procedure. See Chapter 10 for further details.

You can additionally override the default language when you publish a symbol.

Symbols created by the MODEL directive

When you use the **MODEL** directive, Turbo Assembler creates and initializes certain variables to reflect the details of the selected model. These variables can help you write code that's model independent, through the use of conditional assembly statements. See Chapter 15 for information about how you can use variables to alter the assembly process.

The @Model symbol

The **@Model** symbol contains a representation of the model currently in effect. It is defined as a text macro with any of the following values:

```
1 = tiny model is in effect
2 = small or flat
3 = compact
4 = medium
5 = large
6 = huge
7 = tchuge
0 = tpascal
```

The @32Bit symbol

The **@32Bit** symbol contains an indication of whether segments in the currently specified model are declared as 16 bit or 32 bit. The symbol has a value of 0 if you specified 16-bit segments in the **MODEL** directive, or 1 if you indicated 32-bit segments.

The @CodeSize symbol

The **@CodeSize** text macro symbol indicates the default size of a code pointer in the current memory model. It's set to 0 for the memory models that use **NEAR** code pointers (**TINY, SMALL, FLAT, COMPACT, TPASCAL**), and 1 for memory models that use **FAR** code pointers (all others).

The @DataSize symbol

The **@DataSize** text macro symbol indicates the default size of a data pointer in the current memory model. It's set to 0 for the memory models using **NEAR** data pointers (**TINY, SMALL, FLAT, MEDIUM**), 1 for memory models that use **FAR** data pointers (**COMPACT, LARGE, TPASCAL**), and 2 for models using huge data pointers (**HUGE** and **TCHUGE**).

The @Interface symbol

The **@Interface** symbol provides information about the language and operating system selected by the **MODEL** statement. This text macro contains a number whose bits represent the following values:

Table 7.3
Model modifiers

Value in bits 0-6	Meaning
0	NOLANGUAGE
1	C
2	SYSCALL
3	STDCALL
4	PASCAL
5	FORTRAN
6	BASIC
7	PROLOG
8	CPP

Bit 7 can have a value of 0 for DOS/Windows, or 1 for OS/2.

For example, the value 81h for **@Interface** shows that you selected the OS/2 operating system and the C language.

Simplified segment directives

Once you select a memory model, you can use simplified segment directives to begin the individual segments. You can only use these segmentation directives after a **MODEL** directive specifies the memory model for the module. Place as many segmentation directives as you want in a module; Turbo Assembler combines all the pieces with the same name to produce one segment (exactly as if you had entered all the pieces at once after a single segmentation directive). Table 7.4 contains a list of these directives.

Table 7.4
Simplified segment directives

Directive	Description
CODESEG [name]	Begins or continues the module's code segment. For models whose code is **FAR**, you can specify a name that is the actual name of the segment. Note that you can generate more than one code segment per module in this way.
.CODE [*name*]	Same as **CODESEG**. MASM mode only.
DATASEG	Begins or continues the module's **NEAR** or default initialized data segment.
.DATA	Same as **DATASEG**. MASM mode only.
CONST	Begins or continues a module's constant data segment. Constant data is always **NEAR** and is equivalent to initialized data.
.CONST	Same as **CONST**. MASM mode only.
UDATASEG	Begins or continues a module's **NEAR** or default uninitialized data segment. Be careful to include only uninitialized data in this segment or the resulting executable program will be larger than necessary. See Chapter 12 for a description of how to allocate uninitialized data.
.DATA?	Same as **UDATASEG**. MASM mode only.

Table 7.4: Simplified segment directives (continued)

STACK [size]	Begins or continues a module's stack segment. The optional *size* parameter specifies the amount of stack to reserve, in words. If you don't specify a size, Turbo Assembler assumes 200h words (1Kbytes) In MASM mode, any labels, code, or data following the **STACK** statement will not be considered part of the stack segment. Ideal mode, however, reserves the specified space, and leaves the stack segment open so that you can add labels or other uninitialized data. You usually only need to use the stack directive if you are writing a stand-alone assembly language program; most high-level languages will create a stack for you.
.STACK [*size*]	Same as **STACK**. MASM mode only.
FARDATA [*name*]	Begins or continues a **FAR** initialized data segment of the specified name. If you don't specify a name, Turbo Assembler uses the segment name **FAR_DATA**. You can have more than one **FAR** initialized data segment per module.
.FARDATA [*name*]	Same as **FARDATA**. MASM mode only.
UFARDATA [*name*]	Begins or continues a **FAR** uninitialized data segment of the specified name. If you don't specify a name, Turbo Assembler uses segment name **FAR_BSS.** You can have more than one **FAR** uninitialized data segment per module.
.FARDATA? [*name*]	Same as **UFARDATA**. MASM mode only.

See Appendix A if you need to know the actual names, class names, and alignments of the segments created with the simplified segment directives.

Symbols created by the simplified segment directives

When you use the simplified segment directives, they create variables that reflect the details of the selected segment, just as the **MODEL** directive does. See Chapter 15 for further information. The following table lists these symbols.

Table 7.5 Symbols from simplified segment directives

Symbol name	Meaning
@code	the segment or group that CS is assumed to be
@data	the segment or group that DS is assumed to be
@fardata	the current **FARDATA** segment name
@fardata?	the current **UFARDATA** segment name
@curseg	the current segment name
@stack	the segment or group that SS is assumed to be

The STARTUPCODE directive

The **STARTUPCODE** directive provides initialization code appropriate for the current model and operating system. It also marks the beginning of the program. Here's its syntax:

```
STARTUPCODE
```

or

```
.STARTUP              ;(MASM mode only)
```

STARTUPCODE initializes the DS, SS, and SP registers. For the **SMALL, MEDIUM, COMPACT, LARGE, HUGE,** and **TPASCAL** models, Turbo Assembler sets DS and SS to **@data**, and SP to the end of the stack. For **TINY** and **TCHUGE** models, the **STARTUPCODE** directive doesn't change the segment registers.

The @Startup symbol

The **@Startup** symbol is placed at the beginning of the startup code that the **STARTUPCODE** directive generates. It is a near label marking the start of the program.

The EXITCODE directive

You can use the **EXITCODE** directive to produce termination code appropriate for the current operating system. You can use it more than once in a module, for each desired exit point. Here's its syntax:

```
EXITCODE [return_value_expr]
```

You can use the following syntax only in MASM mode:

```
.EXIT [return_value_expr]
```

The optional *return_value_expr* describes the number to be returned to the operating system. If you don't specify a return value, Turbo Assembler assumes the value in the AX register.

Defining generic segments and groups

Most applications can use segments created using the standard models. These standard models, however, are limited in their flexibility. Some applications require full control over all aspects of segment generation; generic segment directives provide this flexibility.

The SEGMENT directive

The **SEGMENT** directive *opens* a segment. All code or data following it will be included in the segment, until a corresponding **ENDS** directive *closes* the segment.

The Ideal mode syntax for the **SEGMENT** directive is:

```
SEGMENT name [attributes]
```

You can use the following syntax for MASM mode:

```
name SEGMENT [attributes]
```

name is the name of the segment. You should name segments according to their usages. See Appendix A for examples of segment names.

You can open and close a segment of the same name many times in a single module. In this case, Turbo Assembler concatenates together the sections of the segment in the order it finds them. You only need to specify the *attributes* for the segment the first time you open the segment.

attributes includes any and all desired segment attribute values, for each of the following:

Note that Turbo Assembler processes attribute values from left to right.

- segment combination attribute
- segment class attribute
- segment alignment attribute
- segment size attribute
- segment access attribute

Segment combination attribute

The segment combination attribute tells the linker how to combine segments from different modules that have the same name. The following table lists the legal values of the segment combination attribute. Note that if you don't specify the combine type, Turbo Assembler assumes PRIVATE.

Table 7.6
Segment combination attribute

Attribute value	Meaning
PRIVATE	Segment will not be combined with any other segments of the same name outside of this module.
PUBLIC	Segment will be concatenated with other segments of the same name outside of this module to form a single contiguous segment.
MEMORY	Same as PUBLIC. Segment will be concatenated with other segments of the same name outside this module to form a single contiguous segment, used as the default stack. The linker initializes values for the initial SS and SP registers so that they point to the end of these segments.
COMMON	Locates this segment and all other segments with the same name at the same address. All segments of this name overlap shared memory. The length of the resulting common segment is the length of the longest segment from a single module.
VIRTUAL	Defines a special kind of segment that must be declared inside an enclosing segment. The linker treats it as a common area and attaches it to the enclosing segment. The virtual segment inherits its attributes from the enclosing segment. The assume directive considers a virtual segment to be a part of its parent segment; in all other ways, a virtual segment is a common area that is combined across modules. This permits the sharing of static data that comes into many modules from included files.
AT *xxx*	Locates the segment at the absolute paragraph address that the expression *xxx* specifies. The linker doesn't emit any data or code for AT segments. Use AT to allow symbolic access to fixed memory locations, such as the display screen or ROM areas.

Table 7.6: Segment combination attribute (continued)

UNINIT	Produces a warning message to let you know that you have inadvertently written initialized data to uninitialized data segments. For example, you can specify the following to produce a warning message: `BSS SEGMENT PUBLIC WORD UNINIT 'BSS'`. To disable this warning message, use the **NOWARN UNI** directive. You can reenable the message by using the **WARN UNI** directive.

Segment class attribute

The segment class attribute is a quoted string that helps the linker determine the proper ordering of segments when it puts together a program from modules. The linker groups together all segments with the same class name in memory. A typical use of the class name is to group all the code segments of a program together (usually the class CODE is used for this). Data and uninitialized data are also grouped using the class mechanism.

Segment alignment attribute

The segment alignment attribute tells the linker to ensure that a segment begins on a specified boundary. This is important because data can be loaded faster on the 80x86 processors if it's properly aligned. The following table lists legal values for this attribute.

Table 7.7
Segment alignment attribute

Attribute value	Meaning
BYTE	No special alignment; start segment on the next available byte.
WORD	Start segment on the next word-aligned address.
DWORD	Start segment on the next doubleword-aligned address.
PARA	Start segment on the next paragraph (16-byte aligned) address.
PAGE	Start segment on the next page (256-byte aligned) address.
MEMPAGE	Start segment on the next memory page (4Kb aligned) address.

Turbo Assembler assumes the **PARA** alignment if you don't specify the alignment type.

Segment size attribute

If the currently selected processor is the 80386, segments can be either 16 bit or 32 bit. The segment size attribute tells the linker which of these you want for a specific segment. The following table contains the legal attribute values.

Table 7.8
Segment size attribute values

Attribute value	Meaning
USE16	Segment is 16 bit. A 16-bit segment can contain up to 64K of code and/or data.
USE32	Segment is 32 bit. A 32-bit segment can contain up to 4 gigabytes of code and/or data.

Turbo Assembler assumes the USE32 value if you selected the 80386 processor in MASM mode. In Ideal mode, Turbo Assembler assumes USE16 by default.

Segment access attribute

For any segment in protected mode, you can control access so that certain kinds of memory operations are not permitted. (Note that this feature is currently supported only by the Phar Lap linker. You must generate object code compatible with it using the **/op** switch if you want to be able to use the segment access attribute.) The segment access attribute tells the linker to apply specific access restrictions to a segment.

The following table lists the legal values for this attribute.

Table 7.9
Segment access attribute

Attribute value	Meaning
EXECONLY	the segment is executable only
EXECREAD	the segment is readable and executable
READONLY	the segment is readable only
READWRITE	the segment is readable and writable

The Phar Lap linker assumes that the segment is meant to run in protected mode if you select any of these attributes, or if you select the USE32 attribute. Turbo Assembler assumes the READONLY attribute if you selected the USE32 attribute but did not specify any of these four attributes.

The ENDS directive

You can use the **ENDS** directive to *close* a segment so that no further data is emitted into it. You should use the **ENDS** directive to close any segments opened with the **SEGMENT** directive. Segments opened using the simplified segment directives don't require the **ENDS** directive.

Here's the syntax of the **ENDS** directive:

```
ENDS [name]
```

For MASM mode only, you can use the following syntax:

```
name ENDS
```

name specifies the name of the segment to be closed. Turbo Assembler will report an error message if *name* doesn't agree with the segment currently open. If you don't specify a name, Turbo Assembler assumes the currently-open segment.

The GROUP directive

You can use the **GROUP** directive to assign segments to groups. A group lets you specify a single segment value to access data in all segments in the group.

Here's the Ideal mode syntax for the **GROUP** directive:

```
GROUP name segment_name [, segment_name...]
```

You can use the following syntax for MASM mode:

```
name GROUP segment_name [, segment_name...]
```

name is the name of the group. *segment_name* is the name of a segment you want to assign to that group.

The ASSUME directive

A segment register must be loaded with the correct segment value for you to access data in a segment. Often, you must do this yourself. For example, you could use the following code to load the address of the current far data segment into DS:

```
MOV AX,@fardata
MOV DS,AX
```

When a program loads a segment value into a segment register, you use that segment register to access data in the segment. It rapidly becomes tiring (and is also poor programming practice) to specify a specific segment register every time you process data in memory.

Use the **ASSUME** directive to tell Turbo Assembler to associate a segment register with a segment or group name. This allows Turbo Assembler to be "smart enough" to use the correct segment registers when data is accessed.

In fact, Turbo Assembler uses the information about the association between the segment registers and group or segment names for another purpose as well: in MASM mode, the value that the CS register is **ASSUME**d to be is used to determine the segment or group a label belongs to. Thus, the CS register must be correctly specified in an **ASSUME** directive, or Turbo Assembler will report errors every time you define a label or procedure.

Here's the syntax of the **ASSUME** directive:

```
ASSUME segreg : expression [,segreg : expression ]
```

or

```
ASSUME NOTHING
```

segreg is one of CS, DS, ES or SS registers. If you specify the 80386 or 80486 processor, you can also use the FS and GS segment registers. *expression* can be any expression that evaluates to a group or segment name. Alternatively, it can be the keyword **NOTHING**. The **NOTHING** keyword cancels the association between the designated segment register and any segment or group name.

ASSUME NOTHING removes associations between all segment registers and segment or group names.

You can use the **ASSUME** directive whenever you modify a segment register, or at the start of a procedure to specify the assumptions at that point. In practice, **ASSUME**s are usually used at the beginning of a module and occasionally within it. If you use the **MODEL** statement, Turbo Assembler automatically sets up the initial **ASSUME**s for you.

If you don't specify a segment in an **ASSUME** directive, its **ASSUME**d value is not changed.

For example, the following code shows how you can load the current initialized far data segment into the DS register, access memory in that segment, and restore the DS register to the data segment value.

```
MOV AX,@fardata
MOV DS,AX
ASSUME DS:@fardata
MOV BX,<far_data_variable>
MOV AX,@data
MOV DS,AX
ASSUME DS:@data
```

Segment ordering

The linker arranges and locates all segments defined in a program's modules. Generally, the linker starts with the order in which it encounters the segments in a program's modules. You can alter this order using mechanisms such as segment combination and segment classing.

There are other ways to affect the way the linker arranges segments in the final program. For example, the order in which segments appear in a module's source can be changed. There are also directives that affect segment ordering. Descriptions of these follow.

Changing a module's segment ordering

The order of segments in each module determines the starting point for the linker's location of segments in the program. In MASM 1.0, 2.0, and 3.0, segments were passed to the linker in alphabetical order. Turbo Assembler provides directives (in MASM mode only) that let you reproduce this behavior.

Note that these directives have the same function as the **/A** and **/S** command line switches. See Chapter 2 for further details.

The .ALPHA directive

The **.ALPHA** directive specifies alphabetic segment ordering. This directive tells Turbo Assembler to place segments in the object file in alphabetical order (according to the segment name). Its syntax is

```
.ALPHA
```

The .SEQ directive

The **.SEQ** directive specifies sequential segment ordering, and tells Turbo Assembler to place segments in the object file in the order in which they were encountered in the source file. Since this is the default behavior of the assembler, you should usually use the **.SEQ** directive only to override a previous **.ALPHA** or a command line switch. Here's the syntax of **.SEQ**:

```
.SEQ
```

DOS ordering of segments: the DOSSEG directive

Normally, the linker arranges segments in the sequential order it encounters them during the generation of the program. When you include a **DOSSEG** directive in any module in a program, it instructs the linker to use *standard* DOS segment ordering instead. The linker defines this convention to mean the following arrangement of segments in the final program:

- segments having the class name CODE (typically code segments)
- segments that do not have class name CODE and are not part of DGROUP
- segments that are part of DGROUP in the following order:

    1. segments not of class BSS or STACK (typically initialized data)
    2. segments of class BSS (typically uninitialized data)
    3. segments of class STACK (stack space)

The segments within DGROUP are located in the order in which they were defined in the source modules.

DOSSEG is included in TASM for backward compatibility only. It is recommended that you do not use the **DOSSEG** directive in new assembly programs. In addition, do not use the **DOSSEG** directive if you're interfacing assembly programs with C programs.

Changing the size of the stack

A procedure's prolog and epilog code manipulates registers that point into the stack. On the 80386 or 80486 processor, the stack segment can be either 16 bits or 32 bits. Turbo Assembler therefore must know the correct size of the stack before it can generate correct prolog and epilog code for a procedure.

The stack size is automatically selected if you selected a standard model using the **MODEL** statement.

Turbo Assembler provides directives that can set or override the default stack size for procedure prolog and epilog generation. The following table lists these directives.

Table 7.10 Stack size modification directives

Directive	Meaning
SMALLSTACK	Indicates that the stack is 16 bit
LARGESTACK	Indicates that the stack is 32 bit

CHAPTER 8

Defining data types

Defining data types symbolically helps you write modular code. You can easily change or extend data structures without having to rewrite code by separating the definition of a data type from the code that uses it, and allowing symbolic access to the data type and its components.

Turbo Assembler supports as many or more symbolic data types than most high-level languages. This chapter describes how to define various kinds of data types.

Defining enumerated data types

An enumerated data type represents a collection of values that can be stored in a certain number of bits. The maximum value stored determines the actual number of bits required.

Here is the Ideal mode syntax for declaring an enumerated data type:

```
ENUM name [enum_var [,enum_var...]]
```

You can use the following syntax in MASM mode:

```
name ENUM [enum_var [,enum_var...]]
```

The syntax of each *enum_var* is:

```
var_name [=value]
```

Beware: If you use the same variable name in two enumerated data types, the first value of the variable will be lost, and errors could result.

Turbo Assembler will assign a value equal to that of the last variable in the list plus one if *value* isn't present when you assign the specified value to the variable *var_name*. Values can't be relative or forward referenced. Variables that **ENUM** created are redefinable numeric variables of global scope.

name is the name of the **ENUM** data type. You can use it later in the module to obtain a variety of information about the values assigned to the variables detailed. See Chapter 5 for information about using enumeration data type names in Turbo Assembler expressions.

➡ You can also use enumerated data type names to create variables and allocate memory. See Chapter 12 for details.

Enumerated data types are redefinable. You can define the same name as an enumerated data type more than once in a module.

Turbo Assembler provides a multiline syntax for enumerated data type definitions requiring a large number of variables. The symbol { starts the multiline definition, and the symbol } stops it.

The Ideal mode syntax follows:

```
ENUM name [enum_var [,enum_var...]] {
[enum_var [,enum_var]...]
:
[enum_var [,enum_var]...] }
```

You can use the following syntax in MASM mode:

```
name ENUM [enum_var [,enum_var...]] {
[enum_var [,enum_var]...]
:
[enum_var [,enum_var]...] }
```

For example, all of the following enumerated data type definitions are equivalent:

Turbo Assembler doesn't recognize any pseudo ops inside the multiline enumerated data type definition.

```
foo ENUM f1,f2,f3,f4        ;Original version

foo ENUM {                  ;Multiline version
     f1
     f2
     f3
     f4
     }

foo ENUM f1,f2,{            ;More compact multiline version
    f3,f4}
```

Defining bit-field records

A record data type represents a collection of bit fields. Each bit field has a specific width (in bits) and an initial value. The record data type width is the sum of the widths of all the fields.

You can use record data types to compress data into a form that's as compact as possible. For example, you can represent a group of 16 flags (which can be either ON or OFF) as 16 individual bytes, 16 individual words, or as a record containing 16 1-bit fields (the efficient method).

Here's the Ideal mode syntax for declaring a record data type:

```
RECORD name [rec_field [,rec_field...]]
```

The MASM mode syntax is:

```
name RECORD [rec_field [,rec_field...]]
```

Each *rec_field* has the following syntax:

```
field_name : width_expression [=value]
```

field_name is the name of a record field. Turbo Assembler will allocate a bit field of the width *width_expression* for it. *value* describes the initial value of the field (the default value used when an instance of the record is created). Values and width expressions can't be relative or forward referenced. Record field names are global in scope and can't be redefined.

name is the name of the record data type. You can use it later in the module to obtain a variety of information about the record data type. You can also use the names of individual record fields to obtain information. See Chapter 5 for details about how to obtain information from record data type names and record field names using Turbo Assembler expressions.

You can redefine record data types, and define the same name as a record data type more than once in a module.

You can also use record data type names to create variables and allocate memory. See Chapter 12 for details.

Turbo Assembler provides special support for record fields that represent flags and enumerated data types. Additional and extended instructions provide efficient access to record fields. Chapter 13 describes this concept further.

For record data type definitions requiring a large number of fields, Turbo Assembler provides a multiline syntax similar to that for enumerated data types.

For example, all of the following record data type definitions are equivalent:

Turbo Assembler does not recognize any pseudo ops inside the multiline record data type definition.

```
foo RECORD f1:1,f2:2,f3:3,f4:4 ;Original version

foo RECORD {                   ;Multiline version
    f1:1
    f2:2
    f3:3
    f4:4
    }

foo RECORD f1:1,f2:2,{         ;More compact multiline version
    f3:3,f4:4}
```

Defining structures and unions

Structures and unions let you mix and match various types. A structure in Turbo Assembler is a data type that contains one or more data elements called *members*. Structures differ from records because structure members are always an integral number of bytes, while records describe the breakdown of bit fields within bytes. The size of a structure is the combined size of all data elements within it.

Unions are similar to structures, except that all of the members in a union occupy the same memory. The size of a union is the size of its largest member. Unions are useful when a block of memory must represent one of several distinct possibilities, each with different data storage requirements.

Turbo Assembler lets you fully nest structures and unions within one another, but this can become complicated. For example, you could have a structure member that is really a union. A union could also have a full structure as each member.

Opening a structure or union definition

Use the following Ideal mode syntaxes to open a structure or union data type definition:

```
STRUC name    or    UNION name
```

You can use the following MASM mode syntaxes to do the same thing:

```
name STRUC or name UNION
```

name is the name of the structure or union data type.

Turbo Assembler considers all data or code emitted between the time a structure or union data type definition is opened and the time a corresponding **ENDS** directive is encountered to be part of that structure or union data type.

Turbo Assembler treats structure and union data type names as global but redefinable. You can define the same name as a structure or union data type more than once in a module.

Specifying structure and union members

Turbo Assembler includes data one line at a time in structures or unions. To allocate data and create members in a structure or union definition, use the same directives as those for allocating data and creating labels in an open segment. For example,

```
member1    DW 1
```

is equally valid in a segment or in a structure definition. In a segment, this statement means "reserve a word of value 1, whose name is *member1*." In a structure or union definition, this statement means "reserve a word of initial value 1, whose member name is *member1*."

You can use the initial value for a structure member if an instance of the structure or union is allocated in a segment or a structure. If you don't intend to allocate structure instances this way, the initial value of the structure member is not important. You can use the data value **?** (the uninitialized data symbol) to indicate this.

Turbo Assembler allows all methods of allocating data with a structure definition, including instances of other structures, unions, records, enumerated data types, tables, and objects. For more information on how to allocate data, see Chapter 12.

MASM and Ideal modes treat structure member names differently. In MASM mode, structure member names are global and can't be redefined. In Ideal mode, structure member names are considered local to a structure or union data type.

Defining structure member labels with LABEL

The **LABEL** directive lets you create structure members without allocating data. Normally, the **LABEL** directive establishes a named label or marker at the point it's encountered in a segment. **LABEL** directives found inside structure definitions define members of the structure. Here's the syntax of the **LABEL** directive:

```
LABEL name complex_type
```

In MASM mode only, you can use the following syntax:

```
name LABEL complex_type
```

name is the name of the structure member. *type* is the desired type for the structure member. It can be any legal type name. See Chapter 5 for a description of the available type specifiers.

Aligning structure members

You can use the **ALIGN** directive within structure definitions to align structures members on appropriate boundaries. For example,

```
        ALIGN 4         ;DWORD alignment
member  dd ?            ;member will be DWORD aligned
```

Closing a structure or union definition

You must close the structure or union definition after you define all structure or union members. Use the **ENDS** directive to do this.

ENDS has the following syntax in Ideal mode:

```
ENDS [name]
```

In MASM mode, you can use the following syntax:

```
name ENDS
```

name, if present, is the name of the currently open structure or union data type definition. If *name* is not present, the currently open structure or union will be closed.

You can also use the **ENDS** directive to close segments. This is not a conflict, because you can't open a segment inside a structure or union definition.

Nesting structures and unions

Turbo Assembler lets you nest the **STRUC**, **UNION**, and **ENDS** directives inside open structure and union data type definitions to control the offsets assigned to structure members.

In a structure, each data element begins where the previous one ended. In a union, each data element begins at the same offset as the previous data element. Allowing a single data element to consist of an entire union or structure provides enormous flexibility and power. The following table contains descriptions of **STRUC**, **UNION**, and **ENDS**.

Table 8.1
STRUC, UNION, and ENDS directives

Directive	Meaning
STRUC	Used inside an open structure or union, this directive begins a block of elements that the enclosing structure or union considers a single member. The members in the block are assigned offsets in ascending order. The size of the block is the sum of the sizes of all of the members in it.
UNION	Used inside an open structure or union, this begins a block of members that the enclosing structure or union considers a single unit. The members in the block are all assigned the same offset. The size of the block is the size of the largest member in it.
ENDS	Terminates a block of members started with a previous **STRUC** or **UNION** directive.

For example, the composite has five members in the following structure/union data definition:

```
CUNION    STRUC
CTYPE     DB ?
            UNION         ;Start of union

            ;If CTYPE=0, use this...
            STRUC
CT0PAR1       DW 1
CT0PAR2       DB 2
            ENDS

            ;If CTYPE=1, use this...
              STRUC
```

```
CT1PAR1              DB 3
CT1PAR2              DD 4
               ENDS

             ENDS         ;End of union
           ENDS           ;End of structure data type
```

The following table lists these members.

Table 8.2 Block members

Member	Type	Offset	Default value
CTYPE	Byte	0	? (uninitialized)
CT0PAR1	Word	1	1
CT0PAR2	Byte	3	2
CT1PAR1	Byte	1	3
CT1PAR2	Dword	2	4

The length of this structure/union is 6 bytes.

Including one named structure within another

Turbo Assembler provides a way of incorporating an entire existing structure or union data type, including member names, into an open structure definition to assist in the inheritance of objects. It treats the incorporated structure or union as if it were nested inside the open structure or union definition at that point. In this way, incorporating a structure or union into another is intrinsically different from including an instance of a structure or union in another; an *instance* includes only initialized or uninitialized data, while *incorporation* includes data, structure, and member names.

Here's the Ideal mode syntax:

```
STRUC struc_name fill_parameters
```

You can use the following syntax in MASM mode:

```
struc_name STRUC fill_parameters
```

Use a statement of this form only inside a structure or union definition. *struc_name* is the name of the previously defined structure or union that is to be included. *fill_parameters* represents the changes you want to make to the initial (default) values of the included structure's members. A **?** keyword indicates that all of the incorporated structure's members should be considered uninitialized. Otherwise, the syntax for the *fill_parameters* field is:

```
{ [member_name [=expression] [,member_name [=expression] ...]] }
```

member_name is the name of any member of the included structure whose initial value should be changed when it's included. *expression* is the value you want to change it to. If you have *expression*, then the initial value for that member of the structure will be unchanged when it is included. If you specify a **?** keyword for the expression field, that member's initial value will be recorded as uninitialized when it's included.

Since structure member names are global in MASM mode, they are not redefined when you copy a structure. Thus, including a structure within another is most useful in MASM mode when you do it at the beginning of the structure or union you're defining.

Usually, when you create an instance of a union, you would have to make sure that only one of the union's members contains initialized data. (See Chapter 12 for details.) Since incorporating a structure in another does not involve creating an instance, this restriction does not apply. More than one member of an included union can contain initialized data. For example,

```
FOO     STRUC
ABC        DW 1
DEF        DW 2
           UNION
A1             DB '123'
A2             DW ?
           ENDS
        ENDS

FOO2    STRUC
FOO     STRUC {A1=2}     ;Incorporates struc FOO into struc FOO2, with ;override
               ;Note that both A1 and A2 are initialized by
               ;default in FOO2!
GHI     DB 3
        ENDS
```

The definition of structure FOO2 in the previous example is equivalent to the following nested structure/union:

```
FOO2   STRUC
          STRUC ;Beginning of nested structure...
ABC          DW 1
DEF          DW 2
             UNION    ;Beginning of doubly nested union...
A1              DB '123'
A2              DW 2
             ENDS     ;End of doubly nested union...
          ENDS        ;End of nested structure...
GHI       DB 3
       ENDS
```

Note that when an instance of the structure FOO2 is made, be sure that only one value in the union is initialized.

Using structure names in expressions

Once you define a structure or union, information about the structure or union is available in many ways. You can use both the structure or union data type name and a structure member name to obtain information using Turbo Assembler expressions. See Chapter 5 for further information.

Defining tables

A table data type represents a collection of table members. Each member has a specific size (in bytes) and an initial value. A table member can be either *virtual* or *static*. A *virtual* member of a table is assigned an offset within the table data type; space is reserved for it in any instance of the table. A *static* member does not have an offset; space isn't reserved for it in an instance of the table.

The size of the table data type as a whole is the sum of the sizes of all of the virtual members.

Table data types represent method tables, used in object-oriented programming. An object usually has a number of methods associated with it, which are pointers to procedures that manipulate instances of the object. Method procedures can either be called directly (*static* methods) or indirectly, using a table of method procedure pointers (*virtual* methods).

You can use the following Ideal mode syntax for declaring a table data type:

```
TABLE name [table_member [,table_member...]]
```

The following syntax works only in MASM mode:

```
name TABLE [table_member [,table_member...]]
```

Here's the syntax of each *table_member* field:

```
table_name
```

or

```
[VIRTUAL] member_name [[count1_expression]]
    [: complex_type [:count2_expression]] [= expression ]
```

table_name is the name of an existing table data type whose members are incorporated entirely in the table you define. Use this syntax wherever you want inheritance to occur.

member_name is the name of the table member. The optional **VIRTUAL** keyword indicates that the member is virtual and should be assigned to a table offset.

complex_type can be any legal complex type expression. See Chapter 5 for a detailed description of the valid type expressions.

If you don't specify a *complex_type* field, Turbo Assembler assumes it to be WORD (DWORD is assumed if the currently selected model is a 32-bit model).

count2_expression specifies how many items of this type the table member defines. A table member definition of

```
foo TABLE VIRTUAL tmp:DWORD:4
```

defines a table member called *tmp*, consisting of four doublewords.

The default value for *count2_expression* is 1 if you don't specify it. *count1_expression* is an array element size multiplier. The total space reserved for the member is *count2_expression* times the length specified by the *memtype* field, times *count1_expression*. The default value for *count1_expression* is also 1 if you don't specify one. *count1_expression* multiplied by *count2_expression* specifies the total count of the table member.

➡ Table member names are local to a table in Ideal mode, but are global in scope in MASM mode.

name is the name of the table data type. You can use it later in the module to get a variety of information about the table data type. You can also use the names of individual table members to obtain information. See Chapter 5 for further information.

Table data types are redefinable. You can define the same name as a table data type more than once in a module.

You can also use table data type names to create variables and allocate memory. See Chapter 12 for details.

Alternatively, Turbo Assembler provides a multiline syntax for table data type definitions requiring a large number of members. This syntax is similar to that of enumerated data type definitions. Here's an example:

```
foo TABLE t1:WORD,t2:WORD,t3:WORD,t4:WORD ;Original version

foo TABLE {       ;Multiline version
   t1:WORD
   t2:WORD
```

```
   t3:WORD
   t4:WORD
   }
foo TABLE t1:WORD,t2:WORD,{      ;More compact multiline version
   t3:WORD,t4:WORD}
```

Overriding table members

If you declare two or more members of the same name as part of the same table data type, Turbo Assembler will check to be sure that their types and sizes agree. If they don't, it will generate an error. Turbo Assembler will use the last initial value occurring in the table definition for the member. In this way, you can override the initial value of a table after it is incorporated into another. For example,

```
FOO   TABLE VIRTUAL MEM1:WORD=MEM1PROC, VIRTUAL MEM2:WORD=MEM2PROC
FOO2  TABLE FOO, VIRTUAL MEM1:WORD=MEM3PROC  ;Overrides inherited ;MEM1
```

Defining a named type

Named types represent simple or complex types. You can use the **TYPEDEF** directive to define named types. Here's the Ideal mode syntax:

```
TYPEDEF type_name complex_type
```

The MASM mode syntax is:

```
type_name TYPEDEF complex_type
```

complex_type describes any type or multiple levels of pointer indirection. See Chapter 5 for further information about complex types. *type_name* is the name of the specified type.

When you use a named type in an expression, it functions as if it were a simple type of the appropriate size. For example,

```
   MOV ax,word ptr [bx]    ;Simple statement;
foo TYPEDEF near ptr byte     ;FOO is basically a word
   MOV ax,foo ptr [bx]     ;so this works too
```

Defining a procedure type

For Turbo Assembler version 3.2 or higher, you can use a user-defined data type (called a procedure type) to describe the arguments and calling conventions of a procedure. Turbo Assembler treats procedure types like any other types; you can use it wherever types are allowed. Note that since procedure types don't allocate data, you can't create an instance of a procedure type.

Use the **PROCTYPE** directive to create a procedure type. Here is the Ideal mode syntax:

```
PROCTYPE name [procedure_description]
```

The MASM mode syntax is:

```
name PROCTYPE [procedure_description]
```

See Chapter 10 for further information about the **PROC** directive.

procedure_description is similar to the language and argument specification for the **PROC** directive. Its syntax is:

```
[[language_modifier] language] [distance] [argument_list]
```

specify *language_modifier*, *language*, and *distance* exactly the same way you would for the corresponding fields in the **PROC** directive.

Use the following form for *argument_list*:

```
argument [,argument] ...
```

An individual argument has the following syntax:

```
[argname] [[count1_expression]]:complex_type [:count2_expression]
```

See Chapter 5 for a discussion of the syntax of complex types.

complex_type is the data type of the argument. It can be either a simple type or a pointer expression.

count2_expression specifies how many items of this type the argument defines. Its default value is 1, except for BYTE arguments. Those arguments have a default count of 2, since you can't **PUSH** a byte value onto the 80x86 stack.

In procedure types whose calling convention permits variable-length arguments (like C), *count2_expression* (for the last argument) can be the special keyword ?, which indicates that the procedure caller will determine the size of the array. The type UNKNOWN also indicates a variable-length parameter.

The name of each argument is optional, but *complex_type* is required because procedure types are used mainly for type checking purposes. The names of the arguments don't have to agree, but the types must.

Defining an object

OOP

An object consists of both a data structure and a list of methods that correspond to the object. Turbo Assembler uses a structure data type to represent the data structure associated with an object, and a table data type to represent the list of methods associated with an object.

An extension to the **STRUC** directive lets you define objects. The Ideal mode syntax follows:

```
STRUC name [modifiers] [parent_name] [METHOD
   [table_member    [,table_member...]]]
structure_members
ENDS [name]
```

You can use the following syntax in MASM mode:

```
name STRUC [modifiers] [parent_name] [METHOD
   [table_member  1  [,table_member...]]]
structure_members
[name] ENDS
```

name is the name of the object. *parent_name* is the optional name of the parent object. (Turbo Assembler explicitly supports only single inheritance.) The parent object's structure data will automatically be included in the new object's structure data, and the parent object's table of methods will be included in the new object's table of methods as well.

Each *table_member* field describes a method name and method procedure associated with the object. The syntax of a *table_member* field is exactly the same as in a table definition.

structure_members describe any additional structure members you want within the object's data structure. These are formatted exactly the same as in an open structure definition.

The optional *modifiers* field can be one or more of the following keywords:

Table 8.3
Available modifiers

Keyword	Meaning
GLOBAL	Causes the address of the virtual method table (if any) to be published globally.
NEAR	Forces the virtual table pointer (if any) to be an offset quantity, either 16 or 32 bits, depending on whether the current model is USE16 or USE32.
FAR	Forces the virtual table pointer (if any) to be a segment and offset quantity, either 32 or 48 bits, depending on whether the current model is USE16 or USE32.

The size of the virtual table pointer (if any) depends on whether data in the current model is addressed as NEAR or FAR if you don't specify a modifier.

The TBLPTR directive

Inherent in the idea of objects is the concept of the virtual method table. An instance of this table exists once for any object having virtual methods. The data structure for any object having virtual methods also must contain a pointer to the virtual method table for that object. Turbo Assembler automatically provides a virtual method table pointer in an object's data structure (if required) and if you don't specify it explicitly using the **TBLPTR** directive.

You should use the **TBLPTR** directive within an object data structure definition. **TBLPTR** lets you explicitly locate the virtual table pointer wherever you want. Here's its syntax:

```
TBLPTR
```

The size of the pointer that **TBLPTR** reserves is determined by whether the current model is USE16 or USE32, and what modifiers you used in the object definition.

Symbols defined by the extended STRUC directive

The extended **STRUC** directive defines or uses several symbols, which reflect the object being defined. The following table shows these symbols.

Table 8.4
Symbols used or defined by STRUC

Symbol	Meaning
@Object	A text macro containing the name of the current object
@Table_<*object_name*>	A table data type containing the object's method table
@Tableaddr_<*object_name*>	A label describing the address of the object's virtual method table

C H A P T E R 9

Setting and using the location counter

The location counter keeps track of the current address as your source files assemble. This lets you know where you are at any time during assembly of your program. Turbo Assembler supplies directives that let you manipulate the location counter to move it to a desired address.

Labels are the names used for referring to addresses within a program. Labels are assigned the value of the location counter at the time they are defined. Labels let you give names to memory variables and the locations of particular instructions.

This chapter discusses the available directives for manipulating the location counter, and declaring labels at the current location counter.

The $ location counter symbol

The predefined symbol **$** represents the current location counter. The location counter consists of two parts: a segment, and an offset. The location counter is the current offset within the current segment during assembly.

The location counter is an address that is incremented to reflect the current address as each statement in the source file is assembled. As an example,

```
helpMessage     DB      'This is help for the program.'
helpLength      = $  -  helpMessage
```

Once these two lines are assembled, the symbol *helpLength* will equal the length of the help message.

Location counter directives

Turbo Assembler provides several directives for setting the location counter. The next few sections describe these directives. Note that all of these directives work in both MASM and Ideal modes.

The ORG directive

You can use the **ORG** directive to set the location counter in the current segment. **ORG** has the following syntax:

```
ORG expression
```

expression can't contain any forward-referenced symbol names. It can either be a constant or an offset from a symbol in the current segment or from the current location counter.

You can back up the location counter before data or code that has already been emitted into a segment. You can use this to go back and fill in table entries whose values weren't known at the time the table was defined. Be careful when using this technique; you might accidentally overwrite something you didn't intend to.

You can use the **ORG** directive to connect a label with a specific absolute address. The **ORG** directive can also set the starting location for .COM files. Here's an example of how to use **ORG**:

```
; This program shows how to create a structure and macro for
; declaring instances of the structure, that allows additional
; elements to be added to the linked list without regard to
; other structures already declared in the list. If the macro
; is invoked in a section of code that is between two other
; instances of the structure, the new structure will automatically
; be inserted in the linked list at that point without your
; needing to know the names of the previous or next
; structure variables. Similarly, using the macro
; at the end of the program easily adds new structures to the
; linked list without regard for the name of the previous
; element.
; The macro also maintains variables that point to the first
; and last elements of the linked list.

ideal
p386

model OS_NT flat

codeseg

struc  a
  prev  dd 0
```

```
  next  dd 0
  info  db   100 dup (0)
ends   a

__last_a_name  equ <>

; Maintain the offsets of the head and tail of the list.
__list_a_head  dd   0
__list_a_tail  dd   0

macro makea  name:req,args
ifidni  __last_a_name,<>

        ; There is no previous item of this type.
name    a       <0,0,args>

        ; Setup the head and tail pointers
        org     __list_a_head
        dd      name
        org     __list_a_tail
        dd      name

        ; Return to the offset after the structure element
        org     name+size a

__last_a_name   equ name

else
        ; Declare it, with previous pointing to previous
        ; item of structure a.
name    a       <__last_a_name,0,args>

        ; Make the next pointer of the previous structure
        ; point to this structure.
        org     __last_a_name.next
        dd      name

        ; Setup the tail pointer for the new member
        org     __list_a_tail
        dd      name

        ; Go back to location after the current structure
        org     name+size a

        ; Set up an equate to remember the name of the
        ; structure just declared
__last_a_name    equ name
endif
endm

makea  first

       ; Miscellaneous other data
       db       5 dup (0)

makea  second
```

```
        ; More miscellaneous data
        db      56 dup (0)

; Give a string to put in the info element of this structure
makea  third,<'Hello'>

end
```

The EVEN and EVENDATA directives

You can use the **EVEN** directive to round up the location counter to the next even address. **EVEN** lets you align code for efficient access by processors that use a 16-bit data bus. It does not improve performance for processors that have an 8-bit data bus.

EVENDATA aligns evenly by advancing the location counter without emitting data, which is useful for uninitialized segments. Both **EVEN** and **EVENDATA** will cause Turbo Assembler to generate a warning message if the current segment's alignment isn't strict enough.

Note: In code segments, NOPs are emitted. In data segments, zeros are emitted.

If the location counter is odd when an **EVEN** directive appears, Turbo Assembler places a single byte of a **NOP** instruction in the segment to make the location counter even. By padding with a **NOP**, **EVEN** can be used in code segments without causing erroneous instructions to be executed at run time. This directive has no effect if the location is already even.

Similarly, if the location counter is odd when an **EVENDATA** directive appears, Turbo Assembler emits an uninitialized byte.

An example of using the **EVEN** directive follows:

```
EVEN
@@A:    lodsb
         xor    bl,al          ;align for efficient access
         loop   @@A
```

Here's an example of using the **EVENDATA** directive:

```
EVENDATA
VAR1     DW     0             ;align for efficient 8086 access
```

The ALIGN directive

You'll use the **ALIGN** directive to round up the location counter to a power-of-two address. **ALIGN** has the following syntax:

```
ALIGN boundary
```

boundary must be a power of two.

Turbo Assembler inserts **NOP** instructions into the segment to bring the location counter up to the desired address if the location counter is not already at an offset that is a multiple of *boundary*. This directive has no effect if the location counter is already at a multiple of *boundary*.

You can't reliably align to a boundary that's more strict than the segment alignment in which **ALIGN** appears. The segment's alignment is specified when the segment is first started with the **SEGMENT** directive.

For example, if you've defined a segment with

```
CODE SEGMENT PARA PUBLIC
```

you can then say **ALIGN 16** (same as **PARA**) but not **ALIGN 32**, since that's more strict than the alignment that **PARA** indicated in the **SEGMENT** directive. **ALIGN** generates a warning if the segment alignment isn't strict enough.

The following example shows how you can use the **ALIGN** directive:

```
ALIGN  4                ;align to DWORD boundary for 386
BigNum   DD   12345678
```

Defining labels

Labels let you assign values to symbols. There are three ways of defining labels:

- using the : operator
- using the **LABEL** directive
- using the :: operator (from MASM 5.1)

The : operator

The **:** operator defines a near code label, and has the syntax

```
name:
```

where *name* is a symbol that you haven't previously defined in the source file. You can place a near code label on a line by itself or at the start of a line before an instruction. You usually would use a near code label as the destination of a **JMP** or **CALL** instruction from within the same segment.

The code label will only be accessible from within the current source file unless you use the **PUBLIC** directive to make it accessible from other source files.

This directive functions the same as using the **LABEL** directive to define a **NEAR** label; for example, *A*: is the same as *A* **LABEL NEAR**. For example,

```
A:
```

is the same as

```
A LABEL NEAR
```

Here's an example of using the **:** operator.

```
jne A        ;skip following function
inc si
A:           ;jne goes here
```

The LABEL directive

You'll use the **LABEL** directive to define a symbol with a specified type. Note that the syntax is different for Ideal and MASM modes. In Ideal mode, specify

```
LABEL name complex_type
```

In MASM mode, use the following:

```
name LABEL complex_type
```

name is a symbol that you haven't previously defined in the source file. *complex_type* describes the size of the symbol and whether it refers to code or data. See Chapter 5 for further information about complex types.

The label is only accessible from within the current source file, unless you use **PUBLIC** to make it accessible from other source files.

You can use **LABEL** to access different-sized items than those in the data structure; the following example illustrates this concept.

```
WORDS  LABEL  WORD        ;access "BYTES" as WORDS
BYTES  DB  64  DUP  (0)
   mov WORDS[2],1         ;write WORD of 1
```

The :: directive

The :: directive only works when you're using MASM51.

The **::** directive lets you define labels with a scope beyond the procedure you're in. This differs from the **:** directive in that labels defined with **:** have a scope of only within the current procedure. Note that **::** is different from **:** only when you specify a language in the **.MODEL** statement.

C H A P T E R 10

Declaring procedures

Turbo Assembler lets you declare procedures in many ways. This chapter discusses NEAR and FAR procedures, declaring procedure languages using arguments and variables in procedures, preserving registers, nesting procedures, declaring method procedures for objects, and declaring procedure prototypes. You can find more information about how to call language procedures in Chapter 13.

Procedure definition syntax

You can use the PROC directive to declare procedures. Here's its Ideal mode syntax:

```
PROC name [[language modifier] language] [distance]
[ARG argument_list] [RETURNS item_list]
[LOCAL argument_list]
[USES item_list]
   :
ENDP [name]
```

Use the following syntax in MASM mode:

```
name PROC [[language modifier] language] [distance]
[ARG argument_list] [RETURNS item_list]
[LOCAL argument_list]
[USES item_list]
   :
[name] ENDP
```

Unless you specify version T310 or earlier, the Ideal mode syntax is no longer allowed in MASM mode.

Turbo Assembler also accepts MASM syntax for defining procedures. For more information on MASM syntax, see Chapter 3.

If you're using Turbo Assembler version T310 or earlier, use the following Ideal mode syntax:

```
PROC [[language modifier] language] name [distance]
  [ARG argument_list] [RETURNS item_list]
  [LOCAL argument_list]
  [USES item_list]
  :
  ENDP
```

Note that the only difference between the older versions of Turbo Assembler and the later versions is that *language* and *language_modifier* have been moved to follow the procedure name to facilitate consistent function prototyping.

Declaring NEAR or FAR procedures

NEAR procedures are called with a near call, and contain a near return; you must call them only from within the same segment in which they're defined. A near call pushes the return address onto the stack, and sets the instruction pointer (IP) to the offset of the procedure. Since the code segment (CS) is not changed, the procedure must be in the same segment as the caller. When the processor encounters a near return, it pops the return address from the stack and sets IP to it; again, the code segment is not changed.

FAR procedures are called with a far call and contain far returns. You can call **FAR** procedures from outside the segment in which they're defined. A far call pushes the return address onto the stack as a segment and offset, and then sets CS:IP to the address of the procedure. When the processor encounters a far return, it pops the segment and offset of the return address from the stack and sets CS:IP to it.

You can specify this as an argument to the **MODEL** statement. See Chapter 7 for more information.

The currently selected model determines the default distance of a procedure. For tiny, small, and compact models, the default procedure distance is **NEAR**. For all other models, **FAR** is the default. If you don't use the simplified segmentation directives, the default procedure distance is always **NEAR**.

You can override the default distance of a procedure by specifying the desired distance in the procedure definition. To do this, use the **NEAR** or **FAR** keywords. These keywords override the default procedure distance, but only for the current procedure. For example,

```
:
MODEL TINY        ;default distance near
:
;test1 is a far procedure
test1 PROC FAR
   ;body of procedure
   RET            ;this will be a far return
ENDP
```

```
;test2 is by default a near procedure
test2 PROC
   ;body of procedure
   RET           ;this will be a near return
ENDP
:
```

The same RET mnemonic is used in both **NEAR** and **FAR** procedures; Turbo Assembler uses the distance of the procedure to determine whether a near or far return is required. Similarly, Turbo Assembler uses the procedure distance to determine whether a near or far call is required to reference the procedure:

```
:
CALL test1  ;this is a far call
CALL test2  ;this is a near call
:
```

When you make a call to a forward referenced procedure, Turbo Assembler might have to make multiple passes to determine the distance of the procedure. For example,

```
:
test1 PROC NEAR
   MOV ax,10
   CALL test2
   RET
test1 ENDP
test2 PROC FAR
   ADD ax,ax
   RET
test2 ENDP
:
```

When Turbo Assembler reaches the "call test2" instruction during the first pass, it has not yet encountered test2, and therefore doesn't know the distance. It assumes a distance of **NEAR**, and presumes it can use a near call.

When it discovers that test2 is in fact a **FAR** procedure, Turbo Assembler determines that it needs a second pass to correctly generate the call. If you enable multiple passes (with the **/m** command-line switch), a second pass will be made. If you don't enable multiple passes, Turbo Assembler will report a "forward reference needs override" error.

You can specify the distance of forward referenced procedures as **NEAR PTR** or **FAR PTR** in the call to avoid this situation (and to reduce the number of passes).

```
:
test1 PROC NEAR
   MOV ax,10
   CALL FAR PTR test2
   RET
test1 ENDP
:
```

The previous example tells Turbo Assembler to use a far call, so that multiple assembler passes aren't necessary.

Declaring a procedure language

You can easily define procedures that use high-level language interfacing conventions in Turbo Assembler. Interfacing conventions are supported for the NOLANGUAGE (Assembler), BASIC, FORTRAN, PROLOG, C, CPP (C++), SYSCALL, STDCALL, and PASCAL languages.

Turbo Assembler does all the work of generating the correct prolog (procedure entry) and epilog (procedure exit) code necessary to adhere to the specified language convention.

You can specify a default language as a parameter of the **MODEL** directive. See Chapter 7 for further details. If a default language is present, all procedures that don't otherwise specify a language use the conventions of the default language.

To override the default language for an individual procedure, include the language name in the procedure definition. You can specify a procedure language by including a language identifier keyword in its declaration. For example, a definition in MASM mode for a PASCAL procedure would be

```
:
pascalproc PROC PASCAL FAR
   ;procedure body
pascalproc ENDP
:
```

Turbo Assembler uses the language of the procedure to determine what prolog and epilog code is automatically included in the procedure's body. The prolog code sets up the stack frame for passed arguments and local variables in the procedure; the epilog code restores the stack frame before returning from the procedure.

Turbo Assembler automatically inserts prolog code into the procedure before the first instruction of the procedure, or before the first "label:" tag.

Prolog code does the following:

- Saves the current BP on the stack.

- Sets BP to the current stack pointer.
- Adjusts the stack pointer to allocate local variables.
- Saves the registers specified by **USES** on the stack.

Turbo Assembler automatically inserts epilog code into the procedure at each RET instruction in the procedure (if there are multiple RETs, the epilog code will be inserted multiple times). Turbo Assembler also inserts epilog code before any object-oriented method jump (see Chapter 4).

Epilog code reverses the effects of prolog code in the following ways:

- Pops the registers specified by **USES** off the stack.
- Adjusts the stack pointer to discard local arguments.
- Pops the stored BP off the stack.
- Adjusts the stack to discard passed arguments (if the language requires it) and returns.

The last part of the epilog code, discarding passed arguments, is performed only for those languages requiring the procedure to discard arguments (for example, BASIC, FORTRAN, PASCAL). The convention for other languages (C, C++, PROLOG) is to leave the arguments on the stack and let the caller discard them. SYSCALL behaves like C++. For the STDCALL language specification, C++ calling conventions are used if the procedure has variable arguments. Otherwise, PASCAL calling conventions are used.

Turbo Assembler always implements the prolog and epilog code using the most efficient instructions for the language and the current processor selected.

Turbo Assembler doesn't generate prolog or epilog code for NOLANGUAGE procedures. If such procedures expect arguments on the stack, you must specifically include the prolog and epilog code yourself.

In general, the language of a procedure affects the procedure in the manner shown in the following figure.

Figure 10.1
How language affects procedures

Language:	None	Basic	Fortran	Pascal	C	CPP	Prolog
Argument ordering (left-to-right, right-to-left)	L-R	L-R	L-R	L-R	R-L	R-L	R-L
Who cleans up stack (caller, procedure)	PROC	PROC	PROC	PROC	CALLER	CALLER	CALLER

See Chapter 13 for further information.

You can use the **/la** command-line switch to include procedure prolog and epilog code in your listing file. This lets you see the differences between the languages.

Specifying a language modifier

Language modifiers tell Turbo Assembler to include special prolog and epilog code in procedures that interface with Windows or the VROOM overlay manager. To use them, specify one before the procedure language in the model directive, or in the procedure header. Valid modifiers are NORMAL, WINDOWS, ODDNEAR, and ODDFAR.

See Chapter 7 for more information.

Additionally, you can specify a default language modifier as a parameter of the **MODEL** directive. If a default language modifier exists, all procedures that don't otherwise specify a language modifier will use the conventions of the default.

Include the modifier in the procedure definition to specify the language modifier for an individual procedure. For example,

```
:
sample PROC WINDOWS PASCAL FAR
   ;procedure body
ENDP
:
```

If you don't specify a language modifier, Turbo Assembler uses the language modifier specified in the **MODEL** statement. Turbo Assembler will generate the standard prolog or epilog code for the procedure if there isn't a **MODEL** statement, or if **NORMAL** is specified.

If you've selected the WINDOWS language modifier, Turbo Assembler generates prolog and epilog code that lets you call the procedure from Windows. Turbo Assembler generates special prolog and epilog code only for **FAR** Windows procedures. You can't call **NEAR** procedures from Windows, so they don't need special prolog or epilog code. Procedures called by Windows typically use PASCAL calling conventions. For example,

Refer to your Windows documentation for more information on Windows procedures.

```
:
winproc PROC WINDOWS PASCAL FAR
ARG @@hwnd:WORD,@@mess:WORD,@@wparam:WORD,@@lparam:DWORD
   ;body of procedure
ENDP
:
```

The ODDNEAR and ODDFAR language modifiers are used in connection with the VROOM overlay manager. VROOM has two modes of operation: oddnear and oddfar. You can use the **/la** switch option on the Turbo Assembler command line to see the prolog and epilog code that these language modifiers produce.

Defining arguments and local variables

Turbo Assembler passes arguments to higher-level language procedures in stack frames by pushing the arguments onto the stack before the procedure is called. A language procedure reads the arguments off the stack when it needs them. When the procedure returns, it either removes the arguments from the stack at that point (the Pascal calling convention), or relies on the caller to remove the arguments (the C calling convention).

The **ARG** directive specifies, in the procedure declaration, the stack frame arguments passed to procedures. Arguments are defined internally as positive offsets from the BP or EBP registers.

The procedure's language convention determines whether or not the arguments will be assigned in reverse order on the stack. You should always list arguments in the **ARG** statement in the same order they would appear in a high-level declaration of the procedure.

The **LOCAL** directive specifies, in the procedure declaration, the stack frame variables local to procedures. Arguments are defined internally as negative offsets from the BP or EBP register.

Allocate space for local stack frame variables on the stack frame by including procedure prolog code, which adjusts the stack pointer downward by the amount of space required. The procedure's epilog code must discard this extra space by restoring the stack pointer. (Turbo Assembler automatically generates this prolog code when the procedure obeys any language convention other than NOLANGUAGE.)

Remember that Turbo Assembler assumes that any procedure using stack frame arguments will include proper prolog code in it to set up the BP or EBP register. (Turbo Assembler automatically generates prolog code when the procedure obeys any language convention other than NOLANGUAGE). Define arguments and local variables with the **ARG** and **LOCAL** directives even if the language interfacing convention for the procedure is NOLANGUAGE. No prolog or epilog code will automatically be generated, however, in this case.

ARG and LOCAL syntax

Here's the syntax for defining the arguments passed to the procedure:

```
ARG argument [,argument] ... [=symbol]
  [RETURNS argument [,argument]]
```

To define the local variables for the procedure, use the following:

```
LOCAL argument [,argument] ... [=symbol]
```

An individual argument has the following syntax:

```
argname [[count1_expression]] [: complex_type [:count2_expression]]
```

complex_type is the data type of the argument. It can be either a simple type, or a complex pointer expression. See Chapter 5 for more information about the syntax of complex types.

If you don't specify a complex_type field, Turbo Assembler assumes **WORD**. It assumes **DWORD** if the selected model is a 32-bit model.

count2_expression specifies how many items of this type the argument defines. An argument definition of

```
ARG tmp:DWORD:4
```

defines an argument called *tmp*, consisting of 4 double words.

The default value for *count2_expression* is 1, except for arguments of type **BYTE**. Since you can't push a byte value, **BYTE** arguments have a default count of 2 to make them word-sized on the stack. This corresponds with the way high-level languages treat character variables passed as parameters. If you really want to specify an argument as a single byte on the stack, you must explicitly supply a *count2_expression* field of 1, such as

```
ARG realbyte:BYTE:1
```

count1_expression is an array element size multiplier. The total space reserved for the argument on the stack is *count2_expression* times the length specified by the *argtype* field, times *count1_expression*. The default value for *count1_expression* is 1 if it is not specified. *count1_expression* times *count2_expression* specifies the total count of the argument.

For Turbo Assembler versions 3.2 or later, you can specify *count2_expression* using the **?** keyword to indicate that a variable number of arguments are passed to the procedure. For example, an argument definition of

```
ARG tmp:WORD:?
```

defines an argument called *tmp*, consisting of a variable number of words.

? must be used as the last item in the argument list. Also, you can use **?** only in procedures that support variable-length arguments (such as procedures that use the C calling conventions).

If you end the argument list with an equal sign (=) and a symbol, Turbo Assembler will equate that symbol to the total size of the argument block in bytes. If you are not using Turbo Assembler's automatic handling of high level language interfacing conventions, you can use this value at the end of the procedure as an argument to the RET instruction. Notice that this causes a stack cleanup of any pushed arguments before returning (this is the Pascal calling convention).

The arguments and variables are defined within the procedure as BP-relative memory operands. Passed arguments defined with **ARG** are positive offset from BP; local variables defined with **LOCAL** are negative offset from BP. For example,

```
:
func1 PROC NEAR
ARG a:WORD,b:DWORD:4,c:BYTE=d
LOCAL x:DWORD,y:WORD:2=z
:
```

defines *a* as [bp+4], *b* as [bp+6], *c* as [bp+14], and *d* as 20; *x* is [bp-2], *y* is [bp-6], and *z* is 8.

The scope of ARG and LOCAL variable names

All argument names specified in the procedure header, whether **ARG**s (passed arguments), **RETURN**s (return arguments), or **LOCAL**s (local variables), are global in scope unless you give them names prepended with the local symbol prefix.

See Chapter 11 for more information about controlling the scope of symbols.

The **LOCALS** directive enables locally scoped symbols. For example,

```
:
LOCALS
test1 PROC PASCAL FAR
ARG @@a:WORD,@@b:DWORD,@@c:BYTE
LOCAL @@x:WORD,@@y:DWORD
  MOV ax,@@a
  MOV @@x,ax
  LES di,@@b
  MOV WORD ptr @@y,di
  MOV WORD ptr @@y+2,es
  MOV @@c,'a'
  RET
ENDP

test2 PROC PASCAL FAR
ARG @@a:DWORD,@@b:BYTE
LOCAL @@x:WORD
  LES di,@@a
  MOV ax,es:[di]
  MOV @@x,ax
  CMP al,@@b
  jz @@dn
  MOV @@x,0
@@dn: MOV ax,@@x
  RET
```

```
ENDP
:
```

Since this example uses locally scoped variables, the names exist only within the body of the procedure. Thus, test2 can reuse the argument names @@a, @@b, and @@x.

Preserving registers

Most higher-level languages require that called procedures preserve certain registers. You can do this by pushing them at the start of the procedure, and popping them again at the end of it.

Turbo Assembler can automatically generate code to save and restore these registers as part of a procedure's prolog and epilog code. You can specify these registers with the **USES** statement. Here's its syntax:

```
USES item [,item] ...
```

item can be any register or single-token data item that can legally be pushed or popped. There is a limit of eight items per procedure. For example,

```
:
myproc PROC PASCAL NEAR
ARG @@source:DWORD,@@dest:DWORD,@@count:WORD
USES cx,si,di,foo
  MOV cx,@@count
  MOV foo,@@count
  LES di,@@dest
  LDS si,@@source
  REP MOVSB
  RE
ENDP
:
```

See Chapter 18 for information about what registers are normally preserved.

USES is only available when used with procedures that have a language interfacing convention other than NOLANGUAGE.

Defining procedures using procedure types

You can use a procedure type (defined with **PROCTYPE**) as a template for the procedure declaration itself. For example,

```
footype PROCTYPE pascal near :word, :dword,:word
   :
foo PROC footype                    ;pascal near procedure
arg a1:word,a2:dword,a3:word        ;an error would occur if
                                    ;arguments did not match
                                    ;those of footype
```

When you declare a procedure using a named procedure description, the number and types of the arguments declared for **PROC** are checked against those declared by **PROCTYPE**. The procedure description supplies the language and distance of the procedure declaration.

Nested procedures and scope rules

All procedures have global scope, even if you nest them within another procedure. For example,

```
:
test1 PROC FAR
   ;some code here
   CALL test2
   ;some more code here
   RET
test2 PROC NEAR
   ;some code here
   RET      ;near return
test2 ENDP
test1 ENDP
:
```

In this example, it's legal to call test1 or test2 from outside the outer procedure.

If you want to have localized subprocedures, use a locally scoped name. For example,

The **LOCALS** directive enables locally scoped symbols. See Chapter 11 for further information.

```
:
LOCALS
test1 PROC FAR         ;some code here
  RET
@@test2 PROC NEAR      ;some code here
  RET
@@test2 ENDP
test1 ENDP
:
```

In this case, you can only access the procedure @@test2 from within the procedure test1. In fact, there can be multiple procedures named @@test2 as long as no two are within the same procedure. For example, the following is legal:

```
:
LOCALS
test1 PROC FAR
  MOV si,OFFSET Buffer
  CALL @@test2
  RET
@@test2 PROC NEAR     ;some code here
  RET
@@test2 ENDP
test1 ENDP

test2 PROC FAR
  MOV si,OFFSET Buffer2
  CALL @@test2
  RET
@@test2 PROC NEAR     ;some code here
  RET
@@test2 ENDP
test2 ENDP
:
```

The following code is not legal:

```
:
LOCALS
test1 PROC FAR
   MOV si,OFFSET Buffer
   CALL @@test2
   RET
test1 ENDP

@@test2 PROC NEAR
   ;some code here
   RET
@@test2 ENDP
:
```

since the CALL to @@test2 specifies a symbol local to the procedure test1, and no such symbol exists.

Declaring method procedures for objects

OOP

Some special considerations apply when you create method procedures for objects. Object method procedures must be able to access the object that they are operating on, and thus require a pointer to that object as a parameter to the procedure.

Turbo Assembler's treatment of objects is flexible enough to allow a wide range of conventions for passing arguments to method procedures. The conventions are constrained only by the need to interface with objects created by a high-level language.

If you are writing a native assembly-language object method procedure, you might want to use register argument passing conventions. In this case, you should write a method procedure to expect a pointer to the object in a register or register pair (such as ES:DI).

If you are writing a method procedure that uses high-level language interfacing conventions, your procedure should expect the object pointer to be one of the arguments passed to the procedure. The object pointer passed from high-level OOP languages like C++ is an implicit argument usually placed at the start of the list of arguments. A method procedure written in assembly language must include the object pointer explicitly in its list of arguments, or unexpected results will occur. Remember that the object pointer can be either a **WORD** or **DWORD** quantity, depending on whether the object is **NEAR** or **FAR**.

You can find information about the calling conventions of Borland C++ in Chapter 18.

Other complexities arise when you write constructor or destructor procedures in assembly language. C++ uses other implicit arguments (under some circumstances) to indicate to the constructor or destructor that certain actions must be taken.

Constructors written for an application using native assembly language do not necessarily need a pointer to the object passed to them. If an object is never statically allocated, the object's constructor will always allocate the object from the heap.

Using procedure prototypes

For versions 3.2 and later, Turbo Assembler lets you declare procedure prototypes much like procedure prototypes in C. To do so, use the **PROCDESC** directive.

The Ideal mode syntax of **PROCDESC** is:

```
PROCDESC name [procedure_description]
```

Use the following syntax in MASM mode:

```
name PROCDESC [procedure_description]
```

See the beginning of this chapter for further information about **PROC**.

procedure_description is similar to the language and argument specification used in the **PROC** directive. Its syntax is:

```
[[language_modifier] language] [distance] [argument_list]
```

language_modifier, *language*, and *distance* have the same syntax as in the **PROC** directive. *argument_list* has the form:

```
argument [,argument] ...
```

An individual argument has the following syntax:

```
[argname] [[count1_expression]]:complex_type [:count2_expression]
```

See Chapter 5 for further information about the syntax of complex types.

complex_type is the data type of the argument, and can be either a simple type or a pointer expression. *count2_expression* specifies how many items of this type the argument defines. The default value of *count2_expression* is 1, except for arguments of **BYTE**, which have a default count of 2 (since you can't **PUSH** a byte value onto the 80x86 stack).

For the last argument, in procedure types whose calling convention allows variable-length arguments (like C), *count2_expression* can be ?, to indicate that the procedure caller will determine the size of the array.

Note that the name of each argument (*argname*) is optional, but *complex_type* is required for each argument because procedure types are used mainly for type checking purposes. The names of the arguments do not have to agree, but the types must.

Here's an example:

```
test PROCDESC pascal near a:word,b:dword,c:word
```

This example defines a prototype for the procedure *test* as a PASCAL procedure taking three arguments (WORD, DWORD, WORD). Argument names are ignored, and you can omit them in the **PROCDESC** directive, as follows:

```
test PROCDESC pascal near :word,:dword,:word
```

The procedure prototype is used to check calls to the procedure, and to check the **PROC** declaration against the language, number of arguments, and argument types in the prototype. For example,

```
test PROC pascal near
  ARG a1:word,a2:dword,a3:word       ;matches PROCDESC for test
```

PROCDESC also globally publishes the name of the procedure. Procedures that are not defined in a module are published as externals, while procedures that are defined are published as public. Be sure that **PROCDESC** precedes the **PROC** declaration, and any use of the procedure name.

Procedure prototypes can also use procedure types (defined with **PROCTYPE**). For example,

```
footype PROCTYPE pascal near :word,:dword,:word
foo PROCDESC footype
```

CHAPTER 11

Controlling the scope of symbols

In Turbo Assembler and most other programming languages, a symbol can have more than one meaning depending on where it's located in a module. For example, some symbols have the same meaning across a whole module, while others are defined only within a specific procedure.

Symbol scope refers to the range of lines over which a symbol has a specific meaning. Proper scoping of symbols is very important for modular program development. By controlling the scope of a symbol, you can control its use. Also, properly selecting the scope of a symbol can eliminate problems that occur when you try to define more than one symbol of the same name.

Redefinable symbols

Some symbol types that Turbo Assembler supports are considered *redefinable*. This means that you can redefine a symbol of this type to be another symbol of the same type at any point in the module. For example, numeric symbols have this property:

```
foo   = 1
   mov ax,foo  ;Moves 1 into AX.
foo   = 2
   mov ax,foo  ;Moves 2 into AX.
```

Generally, the scope of a given redefinable symbol starts at the point of its definition, and proceeds to the point where it's redefined. The scope of the last definition of the symbol is extended to include the beginning of the module up through the first definition of the symbol. For example,

```
   mov ax,foo  ;Moves 2 into AX!
foo   = 1
   mov ax,foo  ;Moves 1 into AX.
foo   = 2      ;This definition is carried around to the start
               ;of the module...
   mov ax,foo  ;Moves 2 into AX.
```

The following list contains the redefinable symbol types.

See Chapter 5 for further information about these.

- text_macro
- numerical_expr
- multiline_macro
- struc/union
- table
- record
- enum

Block scoping

By default, block-scoped symbols are disabled in Turbo Assembler.

Block scoping makes a symbol have a scope that corresponds to a procedure in a module. Turbo Assembler supports two varieties of block scoping: MASM-style, and native Turbo Assembler style.

The LOCALS and NOLOCALS directives

Turbo Assembler uses a two-character code prepended to symbols, which determines whether a symbol in a procedure has block scope. This local-symbol prefix is denoted with "@@." You can use the **LOCALS** directive to both enable block-scoped symbols, and to set the local symbol prefix. Its syntax looks like this:

```
LOCALS [prefix_symbol]
```

The optional *prefix_symbol* field contains the symbol (of two character length) that Turbo Assembler will use as the local-symbol prefix. For example,

```
LOCALS              ;@@ is assumed to be the prefix by default.

foo proc
@@a:     jmp @@a    ;This @@a symbol belongs to procedure FOO.
foo endp

bar proc
@@a:     jmp @@a    ;This @@a symbol belongs to procedure BAR.
bar endp
```

If you want to disable block-scoped symbols, you can use the **NOLOCALS** directive. Its syntax follows:

```
NOLOCALS
```

Note that you can also use block-scoped symbols outside procedures. In this case, the scope of a symbol is determined by the labels defined with the colon directive (:), which are not block-scoped symbols. For example,

```
foo:          ;Start of scope.
@@a:          ;Belongs to scope starting at FOO:
@@b = 1       ;Belongs to scope starting at FOO:
bar:          ;Start of scope.
@@a = 2       ;Belongs to scope starting at BAR:
```

MASM block scoping

In MASM versions 5.1 and 5.2, NEAR labels defined with the colon directive (:) are considered block-scoped if they are located inside a procedure, and you've selected a language interfacing convention with the **MODEL** statement. However, these symbols are not truly block-scoped; they can't be defined as anything other than a near label elsewhere in the program. For example,

```
version m510
model small,c

codeseg

foo proc
a: jmp a ;Belongs to procedure FOO
foo endp

bar proc
a: jmp a ;Belongs to procedure BAR
bar endp

a  = 1   ;Illegal!
```

MASM-style local labels

MASM 5.1 and 5.2 provide special symbols that you can use to control the scope of near labels within a small range of lines. These symbols are: **@@**, **@F**, and **@B**.

When you declare **@@** as a NEAR label using the colon (:) directive, you're defining a unique symbol of the form **@@xxxx** (where *xxxx* is a hexadecimal number). **@B** refers to the last symbol defined in this way. **@F** refers to the next symbol with this kind of definition. For example,

```
version m510
@@:
   jmp @B   ;Goes to the previous @@.
   jmp @F   ;Goes to the next @@.
@@:
   jmp @B   ;Goes to the previous @@.
   jmp @F   ;Error: no next @@.
```

C H A P T E R 12

Allocating data

Data allocation directives are used for allocating bytes in a segment. You can also use them for filling those bytes with initial data, and for defining data variables.

All data allocation directives have some features in common. First, they can generate initialized data and set aside room for uninitialized data. Initialized data is defined with some initial value; uninitialized data is defined without specifying an initial value (its initial value is said to be *indeterminate*). Data allocation directives indicate an uninitialized data value with a **?**. Anything else should represent an initialized data value. Chapter 7 explains why you should distinguish between initialized and uninitialized data.

Another feature common to all data allocation directives is the use of the **DUP** keyword to indicate a repeated block of data. Here's the general syntax of all data allocation directives:

```
[name] directive dup_expr [,dup_expr...]
```

Turbo Assembler initializes *name* to point to the space that the directive reserves. The variable will have a type depending on the actual directive used.

The syntax of each *dup_expr* can be one of the following:

- `?`
- `value`
- `count_expression DUP ( dup_expr[,dup_expr...] )`

count_expression represents the number of times the data block will be repeated. *count_expression* cannot be relative or forward referenced.

Use the **?** symbol if you want uninitialized data. The amount of space reserved for the uninitialized data depends on the actual directive used.

value stands for the particular description of an individual data element that is appropriate for each directive. For some directives, the value field can be very complex and contain many components; others may only require a simple expression.

The following example uses the **DW** directive, which allocates WORDS:

```
DW 2 DUP (3 DUP (1,3),5)    ;Same as DW 1,3,1,3,1,3,5,1,3,1,3,1,3,5
```

Simple data directives

You can define data with the **DB, DW, DD, DQ, DF, DP,** or **DT** directives. These directives define different sizes of simple data, as shown in the following table.

Table 12.1
Data size directives

Directive	Meaning
DB	Define byte-size data.
DW	Define word-size data.
DD	Define doubleword-size data.
DQ	Define quadword-size data.
DF	Define 48-bit 80386 far-pointer-size (6 byte) data.
DP	Define 48-bit 80386 far-pointer-size (6 byte) data.
DT	Define tenbyte (10-byte) size data.

The syntax of the *value* field for each of these directives differs, based on the capability of each data size to represent certain quantities. (For example, it's never appropriate to interpret byte data as a floating-point number.)

DB (byte) values can be

- A constant expression that has a value between –128 and 255 (signed bytes range from –128 to +127; unsigned byte values are from 0 to 255).
- An 8-bit relative expression using the HIGH or LOW operators.
- A character string of one or more characters, using standard quoted string format. In this case, multiple bytes are defined, one for each character in the string.

DW (word) values can be

- A constant expression that has a value between –32,768 and 65,535 (signed words range from –32,768 to 32,767; unsigned word values are from 0 to 65,535).
- A relative expression that requires 16 bits or fewer, (including an offset in a 16-bit segment, or a segment or group value).
- A one or two-byte string in standard quoted string format.

DD (doubleword) values can be

- A constant expression that has a value between –2,147,483,648 and 4,294,967,295 (when the 80386 is selected), or –32,768 and 65,535 otherwise.
- A relative expression or address that requires 32 bits or fewer (when the 80386 is selected), 16 bits or fewer for any other processor.
- A relative address expression consisting of a 16-bit segment and a 16-bit offset.
- A string of up to four bytes in length, using standard quoted string format.
- A short (32-bit) floating-point number.

DQ (quadword) values can be

- A constant expression that has a value between –2,147,483,648 and 4,294,967,295 (when the 80386 is selected), or –32,768 and 65,535 otherwise.
- A relative expression or address that requires 32 bits or fewer (when the 80386 is selected), or 16 bits or fewer for any other processor.
- A positive or negative constant that has a value between -2^{63} and 2^{64}-1 (signed quadwords range in value from -2^{63} to 2^{63}-1; unsigned quadwords have values from 0 to 2^{64}-1).
- A string of up to 8 bytes in length, using standard quoted string format.
- A long (64-bit) floating-point number.

DF, **DP** (80386 48-bit far pointer) values can be

- A constant expression that has a value between –2,147,483,648 and 4,294,967,295 (when the 80386 is selected), or –32,768 and 65,535 otherwise.
- A relative expression or address that requires 32 bits or fewer (when the 80386 is selected), or 16 bits or fewer for any other processor.
- A relative address expression consisting of a 16-bit segment and a 32-bit offset.
- A positive or negative constant that has a value between -2^{47} and 2^{48}-1 (signed 6-byte values range in value from -2^{47} to 2^{47}-1; unsigned 6-byte values have values from 0 to 2^{48}-1).
- A string of up to 6 bytes in length, in standard quoted string format.

DT values can be

- A constant expression that has a value between –2,147,483,648 and 4,294,967,295 (when the 80386 is selected), or –32,768 and 65,535 otherwise.

- A relative expression or address that requires 32 bits or fewer (when the 80386 is selected), or 16 bits or fewer for any other processor.
- A positive or negative constant that has a value between -2^{79} and 2^{80}-1 (signed tenbytes range in value from -2^{79} to 2^{79}-1; unsigned tenbytes have values from 0 to 2^{80}-1).
- A 10-byte temporary real formatted floating-point number.
- A string of up to 10 bytes in length, in standard quoted string format.
- A packed decimal constant that has a value between 0 and 99,999,999,999,999,999,999.

Data is always stored in memory low value before high value.

Numerical and string constants for the simple data allocation directives differ in some cases from those found in standard Turbo Assembler expressions. For example, the **DB**, **DP**, **DQ**, and **DT** directives accept quoted strings that are longer than those accepted within an expression.

Quoted strings are delimited either by single quotes(') or double quotes ("). Inside of a string, two delimiters together indicate that the delimiter character should be part of the string. For example,

```
'what''s up doc?'
```

represents the following characters:

```
what's up doc?
```

We recommend always using the form with the leading digit and the decimal point, for clarity.

You can have floating-point numbers as the value field for the **DD**, **DQ**, and **DT** directives. Here are some examples of floating-point numbers:

```
1.0E30          ;Stands for 1.0 x 10^30
2.56E-21        ;Stands for 2.56 x 10^-21
1.28E+5         ;Stands for 1.28 x 10^5
0.025           ;Stands for .025
```

Turbo Assembler recognizes these floating-point numbers because they contain a '.' after a leading digit. These rules are relaxed somewhat in MASM mode. For example,

```
DD 1E30         ;Legal MASM mode floating point value!
DD .123         ;Legal in MASM mode only.
```

Turbo Assembler also allows encoded real numbers for the **DD**, **DQ**, and **DT** directives. An encoded real number is a hexadecimal number of exactly a certain length. A suffix of *R* indicates that the number will be interpreted as an encoded real number. The length of the number must fill the required field (plus one digit if the leading digit is a zero); for example,

```
DD 12345678r    ;Legal number
DD 012345678r   ;Legal number
DD 1234567r     ;Illegal number (too short)
```

The other suffix values (*D, H, O, Q,* and *B*) function similarly to those found on numbers in normal expressions.

Some of the simple data allocation directives treat other numerical constant values specially. For example, if you don't specify radix for a value in the **DT** directive, Turbo Assembler uses binary coded decimal (BCD) encoding. The other directives assume a decimal value, as follows:

```
DD 1234                          ;Decimal
DT 1234                          ;BCD
```

Chapter 5 details numerical constants and the **RADIX** directive.

The default radix (that the **RADIX** directive specifies) is not applied for the **DD, DQ,** and **DT** directives if a value is a simple positive or negative constant. For example,

```
RADIX 16
DW 1234                          ;1234 hexidecimal
DD 1234                          ;1234 decimal
```

Creating an instance of a structure or union

To create an instance of a structure or a union data type, use the structure or union name as a data allocation directive. For example, assume you've defined the following:

```
ASTRUC STRUC
B  DB "xyz"
C  DW 1
D  DD 2
ASTRUC ENDS

BUNION UNION
X  DW ?
Y  DD ?
Z  DB ?
BUNION ends
```

Then the statements

```
ATEST          ASTRUC          ?
BTEST          BUNION          ?
```

would create instances of the structure *astruc* (defining the variable *atest*) and the union *bunion* (defining the variable *btest*). Since the example contained the **?** uninitialized data value, no initial data will be emitted to the current segment.

Initializing union or structure instances

Initialized structure instances are more complex than uninitialized instances. When you define a structure, you have to specify an initial default value for each of the structure members. (You can use the **?** keyword as the initial value, which indicates that no specific initial value should be saved.) When you create an instance of the structure, you can create it using the default values or overriding values. The simplest initialized instance of a structure contains just the initial data specified in the definition. For example,

```
ASTRUC {}
```

is equivalent to

```
DB "xyz"
DW 1
DD 2
```

The braces ({ }) represent a null initializer value for the structure. The initializer value determines what members (if any) have initial values that should be overridden, and by what new values, as you allocate data for the structure instance. The syntax of the brace initializer follows:

```
{ [member_name = value [,member_name = value...]] }
```

member_name is the name of a member of the structure or union. *value* is the value that you want the member to have in this instance. Specify a null value to tell Turbo Assembler to use the initial value of the member from the structure or union definition. A **?** value indicates that the member should be uninitialized. Turbo Assembler sets any member that doesn't appear in the initializer to the initial value of the member from the structure or union definition. For example,

```
ASTRUC {C=2,D=?}
```

is equivalent to

```
DB "xyz"
DW 2
DD ?
```

You can use the brace initializer to specify the value of any structure or union member, even in a nested structure or union.

Unions differ from structures because elements in a union overlap one another. Be careful when you initialize a union instance since if several union members overlap, Turbo Assembler only lets one of those members have an initialized value in an instance. For example,

```
BUNION {}
```

is valid because all three members of the union are uninitialized in the union definition. This statement is equivalent to

```
DB 4 DUP (?)
```

In this example, four bytes are reserved because the size of the union is the size of its largest member (in this case a DWORD). If the initialized member of the union is not the largest member of the union, Turbo Assembler makes up the difference by reserving space but not emitting data. For example,

```
BUNION {Z=1}
```

is equivalent to

```
DB 1
DB 3 DUP (?)
```

Finally, multiple initialized members in a union produce an error. For example, this is illegal:

```
BUNION {X=1,Z=2}
```

Note that if two or more fields of the union have initial values in the union definition, then using the simple brace initializer ({ }) will also produce an error. The initializer must set all but one value to **?** for a legal instance to be generated.

An alternative method of initializing structure and union instances is to use the bracket (< >) initializer. The values in the initializer are unnamed but are laid out in the same order as the corresponding members in the structure or union definition. Use this syntax for the bracket initializer:

```
< [value [,value...]] >
```

value represents the desired value of the corresponding member in the structure or union definition. A blank value indicates that you'll use the initial value of the member from the structure or union definition. A **?** keyword indicates that the member should be uninitialized. For example,

```
ASTRUC <"abc",,?>
```

is equivalent to

```
DB "abc"
DW 1
DD ?
```

If you specify fewer values than there are members, Turbo Assembler finishes the instance by using the initial values from the structure or union definition for the remaining members.

```
ASTRUC <"abc">                          ;Same as ASTRUC <"abc",,>
```

When you use the bracket initializer, give special consideration to nested structures and unions. The bracket initializer expects an additional matching pair of angle brackets for every level of nesting, so that Turbo Assembler will treat the nested structure or union initializer as a single entity (to match the value in the instance). Alternatively, you can skip an entire level of nesting by leaving the corresponding entry blank (for the default value of the nested structure or union), or by specifying the **?** keyword (for an uninitialized nested structure or union). For example, examine the following nested structure and union:

```
CUNION      STRUC
   CTYPE    DB ?
   UNION    ;Start of union

      ;If CTYPE=0, use this...
      STRUC
            CT0PAR1  DW 1
            CT0PAR2  DB 2
      ENDS

      ;If CTYPE=1, use this...
      STRUC
            CT1PAR1  DB 3
            CT1PAR2  DD 4
      ENDS

   ENDS     ;End of union
ENDS        ;End of structure data type
```

The bracket initializer for this complex structure/union has two levels of nesting. This nesting must appear as matched angle brackets within the initializer, like

```
CUNION <0,<<2,>,?>>
```

This directive is equivalent to

```
DB 0
DW 2
DB 2
DB 2 DUP (?)
```

Creating an instance of a record

To create an instance of a record data type, use the name of the record data type as a data allocation directive. For example, assume you've defined the following:

```
MYREC RECORD VAL:3=4,MODE:2,SZE:4=15
```

Then, the statement

```
MTEST MYREC ?
```

would create an instance of the record *myrec* (defining the variable *mtest*). No initial data is emitted to the current segment in this example because the **?** uninitialized data value was specified.

Record instances are always either a byte, a word, or a doubleword, depending on the number of bits allocated in the record definition.

Initializing record instances

You must specify an initial value for some or all of the record fields when you define a record. (Turbo Assembler assumes that any unspecified initial values are 0.) The simplest initialized instance of a record contains just the initial field data specified in the definition. For example,

```
MYREC {}
```

is equivalent to

```
DW (4 SHL 6) + (0 SHL 4) + (15 SHL 0)
   ;SHL is the shift left operator for expressions
```

The braces ({ }) represent a null initializer value for the record. The initializer value determines what initial values should be overridden, and by what new values (as you allocate data for the record instance).

Use this syntax of the brace initializer for records:

```
{ [field_name = expression [,field_name = expression...]] }
```

field_name is the name of a field in the record. *expression* is the value that you want the field to have in this instance. A blank value indicates that you'll use the initial value of the field from the record definition. A **?** value is equivalent to zero. Turbo Assembler sets any field that doesn't appear in the initializer to the initial value of the field from the record definition. For example,

```
MYREC {VAL=2,SZE=?}
```

is equivalent to

```
DW (2 SHL 6) + (0 SHL 4) + (0 SHL 0)
```

An alternative method of initializing record instances is to use the bracket (< >) initializer. In this case, brackets delineate the initializer. The values in the initializer are unnamed but are laid out in the same order as the corresponding fields in the record definition. The syntax of the bracket initializer follows:

```
< [expression [,expression...]] >
```

expression represents the desired value of the corresponding field in the record definition. A blank value indicates that you'll use the initial value of the field from the record definition. A **?** keyword indicates that the field should be zero. For example,

```
MYREC <,2,?>
```

is equivalent to

```
DW (4 SHL 6) + (2 SHL 4) + (0 SHL 0)
```

If you specify fewer values than there are fields, Turbo Assembler finishes the instance by using the initial values from the record definition for the remaining fields.

```
MYREC <1>   ;same as MYREC <1,,>
```

Creating an instance of an enumerated data type

You can create an instance of an enumerated data type by using the name of the enumerated data type as a data allocation directive. For example, assume you have defined the following:

```
ETYPE ENUM FEE,FIE,FOO,FUM
```

Then the statement

```
ETEST ETYPE ?
```

would create an instance of the enumerated data type *etype* (defining the variable *etest*). In this example, no initial data is emitted to the current segment because the **?** uninitialized data value is specified.

Enumerated data type instances are always either a byte, a word, or a doubleword, depending on the maximum value present in the enumerated data type definition.

Initializing enumerated data type instances

You can use any expression that evaluates to a number that will fit within the enumerated data type instance; for example,

```
ETYPE ?      ;uninitialized instance
ETYPE FOO    ;initialized instance, value FOO
ETYPE 255    ;a number outside the ENUM that also fits
```

Creating an instance of a table

To create an instance of a table data type, use the table name as a data allocation directive. For example, assume you have defined the following table:

```
TTYPE TABLE  VIRTUAL MoveProc:WORD=MoveRtn,      \
     VIRTUAL MsgProc:DWORD=MsgRtn,              \
     VIRTUAL DoneProc:WORD=DoneRtn
```

Then, the statement

```
TTEST TTYPE ?
```

would create an instance of the table *ttype* (defining the variable *ttest*). No initial data will be emitted to the current segment in this example because the **?** uninitialized data value was specified.

Initializing table instances

When you define a table, you must specify an initial value for all table members. The simplest initialized instance of a table contains just the initial data specified in the definition. For example,

```
TTYPE {}
```

is equivalent to

```
DW MoveRtn
DD MsgRtn
DW DoneRtn
```

The braces ({ }) represent a null initializer value for the structure. The initializer value determines what members (if any) have initial values that should be overridden, and by what new values, as you allocate data for the table instance.

Here's the syntax of the brace initializer:

```
{ [member_name = value [,member_name = value...]] }
```

member_name is the name of a member of the table. *value* is the value that you want the member to have in this instance. A blank value indicates that you'll use the initial value of the member from the table definition. A **?** value indicates that the member should be uninitialized. Turbo Assembler sets any member that doesn't appear in the initializer to the initial value of the member from the table definition. For example,

```
TTYPE {MoveProc=MoveRtn2,DoneProc=?}
```

is equivalent to

```
DW MoveRtn2
DD MsgRtn
DW ?
```

Creating and initializing a named-type instance

You can create an instance of a named type by using the type name as a data allocation directive. For example, if you define the following type:

```
NTTYPE TYPEDEF PTR BYTE
```

the statement

```
NTTEST NTTYPE ?
```

creates an instance of the named type *nttype* (defining the variable *nttest*). No initial data is emitted to the current segment in this example because you specified the **?** uninitialized data value.

The way that you initialize a named-type instance depends on the type that the named type represents. For example, NTTYPE in the previous example is a word, so it will be initialized as if you had used the **DW** directive, as follows:

```
NTTYPE 1,2,3          ;Represents pointer values 1,2,3.
DW     1,2,3          ;Same as NTTYPE 1,2,3.
```

However, if the named type represents a structure or table, it must be initialized the same way as structures and tables are. For example,

```
foo STRUC
f1  DB ?
ENDS
bar TYPEDEF foo
       bar {f1=1}        ;Must use structure initializer.
```

Creating an instance of an object

OOP

Creating an instance of an object in an initialized or uninitialized data segment is exactly the same as creating an instance of a structure. In fact, objects in Turbo Assembler *are* structures, with some extensions. One of these extensions is the @Mptr_<*object_name*> structure member.

An object data type with virtual methods is a structure having one member that points to a table of virtual method pointers. The name of this member is @Mptr_<*object_name*>. Usually, you would initialize an instance of an object using a constructor method. However, you could have objects designed to be static and have no constructor, but are instead initialized with an initializer in a data segment.

If you use the @Mptr_<*object_name*> member's default value, Turbo Assembler will correctly initialize the object instance.

Another difference between structures and objects is that objects can inherit members from previous object definitions. When this inheritance occurs, Turbo Assembler handles it as a nested structure. Because of this, we do not recommend using bracket (< >) initializers for object data.

Creating an instance of an object's virtual method table

Every object that has virtual methods requires an instance of a table of virtual methods to be available somewhere. A number of factors determine the proper placement of this table, including what program model you're using, whether you want near or far tables, and so forth. Turbo Assembler requires you to place this table. You can create an instance for the most recently defined object by using the **TBLINST** pseudo-op, with this syntax:

```
TBLINST
```

TBLINST defines @TableAddr_<*object_name*> as the address of the virtual table for the object. It is equivalent to

```
@TableAddr_<object_name> @Table_<object_name> {}
```

C H A P T E R 13

Advanced coding instructions

Turbo Assembler recognizes all standard Intel instruction mnemonics applicable to the currently selected processor(s). You can find a detailed summary of these instructions in the quick reference guide. This chapter describes Turbo Assembler's extensions to the instruction set, such as the extended **CALL** instruction for calling language procedures.

Intelligent code generation: SMART and NOSMART

Intelligent code generation means that Turbo Assembler can determine when you could have used different instructions more efficiently than those you supplied. For example, there are times when you could have replaced an **LEA** instruction by a shorter and faster **MOV** instruction, as follows:

```
LEA AX,lval
```

can be replaced with

```
MOV AX,OFFSET lval
```

Turbo Assembler supplies directives that let you use intelligent code generation. The following table lists these directives.

Table 13.1 Intelligent code generation directives

Directive	Meaning
SMART	Enables smart code generation.
NOSMART	Disables smart code generation.

See Chapter 3 for details on **VERSION**.

By default, smart code generation is enabled. However, smart code generation is affected not only by the **SMART** and **NOSMART** directives, but also by the **VERSION** directive.

Smart code generation affects the following code generation situations:

- Replacement of **LEA** instructions with **MOV** instructions if the operand of the **LEA** instruction is a simple address.

- Generation of signed Boolean instructions, where possible. For example, **AND** AX,+02 vs. **AND** AX,0002.
- Replacement of **CALL FAR** *xxxx* with a combination of **PUSH CS, CALL NEAR** *xxxx*, when the target *xxxx* shares the same CS register.

Using *smart* instructions make it easier to write efficient code. Some standard Intel instructions have also been extended to increase their power and ease of use. These are discussed in the next few sections.

Extended jumps

Conditional jumps such as **JC** or **JE** on the 8086, 80186, and 80286 processors are only allowed to be *near* (within a single segment) and have a maximum extent of –128 bytes to 127 bytes, relative to the current location counter. The same is true of the loop conditional instructions such as **JCXZ** or **LOOP** on all the Intel processors.

Turbo Assembler can generate complementary jump sequences where necessary and remove this restriction. For example, Turbo Assembler might convert

```
JC xxx
```

to

```
JNC temptag
JMP xxx
```

➡ You can enable this complementary jump sequences with the **JUMPS** directive, and disable it with the **NOJUMPS** directive. By default, Turbo Assembler doesn't generate this feature.

When you enable **JUMPS**, Turbo Assembler reserves enough space for all forward-referenced conditional jumps for a complementary jump sequence. When the actual distance of the forward jump is determined, you might not need a complementary sequence. When this happens, Turbo Assembler generates **NOP** instructions to fill the extra space.

To avoid generating extra **NOP**s, you can

- You can use an override for conditional jumps that you know are in range; for example,

```
   JC SHORT abc
   ADD ax,ax
abc:
```

- Specify the **/m** command-line switch. See Chapter 2 for more about this switch.

Additional 80386 LOOP instructions

The loop instructions for the 80386 processor can either use CX or ECX as the counting register. The standard **LOOP, LOOPE, LOOPZ, LOOPNE,** and **LOOPNZ** mnemonics from Intel select the counting register based on whether the current code segment is a 32-bit segment (when using ECX) or a 16-bit segment (when using CX).

Turbo Assembler has special instructions that increase the flexibility of the **LOOP** feature. The **LOOPW, LOOPWE, LOOPWZ, LOOPWNE,** and **LOOPWNZ** instructions use CX as the counting register, regardless of the size of the current segment. Similarly, the **LOOPD, LOOPDE, LOOPDZ, LOOPDNE,** and **LOOPDNZ** instructions use ECX as the counting register.

Additional 80386 ENTER and LEAVE instructions

Use the **ENTER** and **LEAVE** instructions for setting up and removing a procedure's frame on the stack. Depending on whether the current code segment is a 32-bit segment or a 16-bit segment, the standard **ENTER** and **LEAVE** instructions will modify either the EBP and ESP 32-bit registers, or the BP and SP 16-bit registers. These instructions might be inappropriate if the stack segment is a 32-bit segment and the code segment is a 16-bit segment, or the reverse.

Turbo Assembler provides four additional instructions that always select a particular stack frame size regardless of the code segment size. The **ENTERW** and **LEAVEW** instructions always use BP and SP as the stack frame registers, while the **ENTERD** and the **LEAVED** instructions always use EBP and ESP.

Additional return instructions

The standard **RET** instruction generates code that terminates the current procedure appropriately. This includes generating epilog code for a procedure that uses a high-level language interfacing convention. Even for a procedure with NOLANGUAGE as its calling convention, the **RET** instruction will generate different code if you declare the procedure **NEAR** or **FAR**. For a **NEAR** procedure, Turbo Assembler generates a near return instruction. For a **FAR** procedure, Turbo Assembler generates a far return instruction. (Outside of a procedure, a near return is always generated.)

Turbo Assembler contains additional instructions to allow specific return instructions to be generated (without epilog sequences). The following table lists them.

Table 13.2
Return instructions

Instruction	Function
RETN	Always generates a near return.
RETF	Always generates a far return.
RETCODE	Generates a return appropriate for the currently selected model. Generates a near return for models TINY, SMALL, COMPACT, and TPASCAL. Generates a far return for models MEDIUM, LARGE, HUGE, and TCHUGE.

Additional IRET instructions

For Turbo Assembler version 3.2 or later, you can use an expanded form of the **IRET** instruction. **IRET** will pop flags from the stack DWORD-style if the current code segment is 32-bit. Otherwise, a WORD-style **POP** is used. The **IRETW** instruction always pops WORD-style. Note that you can use these enhancements only if you select version T320. Otherwise, **IRET** will pop flags WORD-style, and **IRETW** is unavailable.

Extended PUSH and POP instructions

Turbo Assembler supports several extensions to the **PUSH** and **POP** instructions. These extensions greatly reduce the quantity of typing required to specify an extensive series of **PUSH** or **POP**s.

Multiple PUSH and POPs

You can specify more than one basic **PUSH** or **POP** instruction per line. For example,

```
PUSH ax
PUSH bx
PUSH cx
POP cx
POP bx
POP ax
```

can be written as

```
PUSH ax bx cx
POP cx bx ax
```

For Turbo Assembler to recognize there are multiple operands present, make sure that any operand cannot conceivably be considered part of an adjacent operand. For example,

```
PUSH foo [bx]
```

might produce unintended results because `foo`, `[bx]`, and `foo[bx]` are all legal expressions. You can use brackets or parentheses to clarify the instruction, as follows:

```
PUSH [foo] [bx]
```

Pointer PUSH and POPs

The standard **PUSH** and **POP** instructions can't push far pointers, which require 4 bytes on the 8086, 80186, and 80286 processors, and up to 6 bytes on the 80386 processor.

Turbo Assembler permits **PUSH** and **POP** instructions to accept DWORD-sized pointer operands for the 8086, 80186, and 80286 processors, and PWORD and QWORD-sized pointer operands for the 80386 processor. When such a **PUSH** or **POP** is encountered, Turbo Assembler will generate code to **PUSH** or **POP** the operand into two pieces.

PUSHing constants on the 8086 processor

While the 80186, 80286, and 80386 processors have basic instructions available for directly **PUSH**ing a constant value, the 8086 does not.

Turbo Assembler permits constants to be **PUSH**ed on the 8086, and generates a sequence of instructions that has the exact same result as the **PUSH** of a constant on the 80186 and higher processors.

Note: you can only do this if you've turned smart code generation on.

The sequence of instructions Turbo Assembler uses to perform the **PUSH** of a constant is about ten bytes long. There are shorter and faster ways of performing the same function, but they all involve the destruction of the contents of a register; for example,

```
MOV ax,constant
PUSH ax
```

This sequence is only four bytes long, but the contents of the AX register is destroyed in the process.

Additional PUSHA, POPA, PUSHF and POPF instructions

For Turbo Assembler versions 3.2 or later, you can use an expanded form of the **PUSHA, POPA, PUSHF** and **POPF** instructions. If the current code segment is 32-bit, the **PUSHA** instruction will push registers in DWORD-style, and **POPA** will pop registers in DWORD-style. Otherwise, Turbo

Assembler uses WORD-style **PUSH** and **POP**. Similarly, **PUSHF** and **POPF** will push and pop flags DWORD-style for a 32-bit code segment, or WORD-style otherwise.

The **PUSHAW, POPAW, PUSHFW,** and **POPFW** instructions always push and pop WORD-style. Remember that you can use these enhancements only if you're using version T320 or later; otherwise, the pushes and pops will be done WORD-style.

The PUSHSTATE and POPSTATE instructions

The PUSHSTATE directive saves the current operating state on an internal stack that is 16 levels deep. PUSHSTATE is particularly useful if you have code inside a macro that functions independently of the current operating state, but does not affect the current operating mode.

The state information that Turbo Assembler saves consists of:

- Current emulation version (for example T310)
- Mode selection (for example IDEAL, MASM, QUIRKS, MASM51)
- EMUL or NOEMUL switches
- Current processor or coprocessor selection
- MULTERRS or NOMULTERRS switches
- SMART or NOSMART switches
- The current radix
- JUMPS or NOJUMPS switches
- LOCALS or NOLOCALS switches
- The current local symbol prefix

Use the POPSTATE directive to return to the last saved state from the stack.

```
; PUSHSTATE and POPSTATE example

.386
ideal
model small
dataseg

pass_string db 'passed',13,10,36
fail_string db 'failed',13,10,36

codeseg
```

```
jumps

        ; Show changing processor selection, number radix, and JUMPS mode

        xor     eax,eax     ; Zero out eax. Can use EAX in 386 mode
        pushstate           ; Preserve state of processor, radix and JUMPS

        nojumps
        radix   2           ; Set to binary radix
        p286

        mov     ax,1        ; Only AX available now. EAX would give errors.
        cmp     ax,1
        jne     next1       ; No extra NOPS after this
                            ; Assemble with /la and check in .lst file.
        mov     ax,100      ; Now 100 means binary 100 or 4 decimal.
next1:
        popstate            ; Restores JUMPS and 386 mode and default radix.

        cmp     eax,4       ; EAX available again. Back in decimal mode.
        je      pass1       ; Extra NOPS to handle JUMPS. Check in .lst file
        mov     dx,OFFSET fail_string    ; Load the fail string
        jmp     fini
pass1:
        mov     dx,OFFSET pass_string    ; Load the pass string.
fini:
        mov     ax,@data    ; Print the string out
        mov     ds,ax
        mov     ah,9h
        int     21h

        mov     ah, 4ch     ; Return to DOS
        int     21h
end
```

Extended shifts

On the 8086 processor, the shift instructions **RCL, RCR, ROL, ROR, SHL, SHR, SAL,** and **SAR** cannot accept a constant rotation count other than 1. The 80186, 80286, and 80386 processors accept constant rotation counts up to 255.

When Turbo Assembler encounters a shift instruction with a constant rotation count greater than 1 (with the 8086 processor selected), it generates an appropriate number of shift instructions with a rotation count of 1. For example,

```
.8086
SHL ax,4
```

generates

```
SHL ax,1
SHL ax,1
SHL ax,1
SHL ax,1
```

Forced segment overrides: SEGxx instructions

Turbo Assembler provides six instructions that cause the generation of segment overrides. The following table lists these instructions.

Table 13.3 Segment override instructions

Instruction	Meaning
SEGCS	Generates a CS override prefix byte.
SEGSS	Generates an SS override prefix byte.
SEGDS	Generates a DS override prefix byte.
SEGES	Generates an ES override prefix byte.
SEGFS	Generates an FS override prefix byte.
SEGGS	Generates a GS override prefix byte.

You can use these instructions in conjunction with instructions such as **XLATB**, which do not require arguments, but can use a segment override. For example,

```
SEGCS XLATB
```

Note that most such instructions have an alternative form, where you can provide a dummy argument to indicate that an override is required. For example,

```
XLAT BYTE PTR cs:[bx]
```

These two examples generate exactly the same code.

Additional smart flag instructions

Often, you can simplify an instruction that manipulates bits in a flag to improve both code size and efficiency. For example,

```
OR ax,1000h
```

might be simplified to

```
OR ah,10h
```

if the only result desired was to set a specific bit in AX, and the processor flags that the instruction affects are unimportant. Turbo Assembler provides four additional instructions that have this functionality, as shown in the following table:

Table 13.4 Smart flag instructions

Instruction	Function	Corresponds to
SETFLAG	Set flag bit(s)	**OR**
MASKFLAG	Mask off flag bit(s)	**AND**
TESTFLAG	Test flag bit(s)	**TEST**
FLIPFLAG	Complement flag bit(s)	**XOR**

Use these instructions to enhance the modularity of records; for example,

```
FOO    RECORD R0:1,R1:4,R2:3,R3:1
    :
    TESTFLAG AX,R0
```

In this example, **TESTFLAG** will generate the most efficient instruction regardless of where R0 exists in the record.

Additional field value manipulation instructions

Turbo Assembler can generate specific code sequences for setting and retrieving values from bit fields specified with the **RECORD** statement. This lets you write code that is independent of the actual location of a field within a record. Used in conjunction with the **ENUM** statement, records can thus achieve an unprecedented level of modularity in assembly language. The following table lists these instructions:

Table 13.5 Instructions for setting and retrieving values

Instruction	Function
SETFIELD	Sets a value in a record field.
GETFIELD	Retrieves a value from a record field.

The SETFIELD instruction

SETFIELD generates code that sets a value in a record field. Its syntax follows:

```
SETFIELD field_name destination_r/m , source_reg
```

field_name is the name of a record member field. *destination_r/m* for **SETFIELD** is a register or memory address of type BYTE or WORD (or DWORD for the 80386). *source_reg* must be a register of the same size or smaller. If the source is smaller than the destination, the source register must be the least significant part of another register that is the same size as the destination. This full-size register is called the *operating register*. Use this register to shift the value in the source register so that it's aligned with the destination. For example,

```
FOO   RECORD R0:1,R1:4,R2:3,R3:1
   :
   SETFIELD R1 AX,BL              ;operating register is BX
   SETFIELD R1 AX,BH              ;illegal!
```

The entire contents of the operating register are destroyed by the **SETFIELD** operation.

SETFIELD shifts the source register efficiently to align it with the field in the destination, and **OR**s the result into the destination register. Otherwise, **SETFIELD** modifies only the operating register and the processor flags.

To perform its function, **SETFIELD** generates an efficient but extended series of the following instructions: **XOR**, **XCHG**, **ROL**, **ROR**, **OR**, and **MOVZX**.

If you're using **SETFIELD** when your source and target registers are the same, the instruction does not **OR** the source value to itself. Instead, **SETFIELD** ensures that the fields of the target register not being set will be zero.

SETFIELD does not attempt to clear the target field before **OR**ing the new value. If this is necessary, you must explicitly clear the field using the **MASKFLAG** instruction.

The GETFIELD instruction

GETFIELD retrieves data from a record field. It functions as the logical reverse of the **SETFIELD** instruction. Its syntax follows:

```
GETFIELD field_name destination_reg , source_r/m
```

field_name and *destination_reg* function as they do for **SETFIELD**. You can use *source_r/m* as you would for *source_reg* (for **SETFIELD**). For example,

```
FOO   RECORD R0:1,R1:4,R2:3,R3:1
   :
   GETFIELD R1 BL,AX              ;operating register is BX
   GETFIELD R1 BH,AX              ;illegal!
```

Note that **GETFIELD** destroys the entire contents of the operating register.

GETFIELD retrieves the value of a field found in the source register or memory address, and sets the pertinent portion of the destination register to that value. This instruction affects no other registers than the operating register and the processor flags.

To accomplish its function, **GETFIELD** generates an efficient but extended series of the following instructions: **MOV**, **XCHG**, **ROL**, and **ROR**.

If you're using the **GETFIELD** instruction when your source and target registers are the same, the instruction will not generate the nonfunctional `MOV target, source` instruction.

Additional fast immediate multiply instruction

Turbo Assembler provides a special immediate multiply operation for efficient array indexing. **FASTIMUL** addresses a typical problem that occurs when you create an array of structures. There is no immediate multiply operation available for the 8086 processor. Even for the more advanced processors, multiplication using shifts and adds is significantly faster in some circumstances than using the standard immediate **IMUL** instruction. Based on the currently specified processor, Turbo Assembler's **FASTIMUL** instruction chooses between the most efficient sequence of shifts and adds available, and the current processor's immediate **IMUL** operation (if any). **FASTIMUL** has the following syntax:

```
FASTIMUL dest_reg, source_r/m, value
```

This instruction is much like the trinary **IMUL** operation available on the 80186, 80286, and 80386 processors. The *dest_reg* destination register is a WORD register (or it can be DWORD on the 80386). *source_r/m* is a register or memory address that must match the size of the destination. *value* is a fixed, signed constant multiplicand.

FASTIMUL uses a combination of **IMUL, MOV, NEG, SHL, ADD**, and **SUB** instructions to perform its function. This function destroys the source register or memory address, and leaves the processor flags in an indeterminate state.

Extensions to necessary instructions for the 80386 processor

The 80386 processor has the ability to operate in both 16-bit and 32-bit mode. Many of the standard instructions have different meanings in these two modes. In Turbo Assembler, you can control the operating size of the instruction using the **SMALL** and **LARGE** overrides in expressions.

See Chapter 5 for further information about overriding address sizes with the **SMALL** and **LARGE** operators.

In general, when you use **SMALL** or **LARGE** as part of an address expression, the operator controls the generation of the address portion of the instruction, determining whether it should be 16- or 32-bit.

Note: Turbo Assembler selects the size of the instruction using **SMALL** and **LARGE** only when no other information is available.

When **SMALL** or **LARGE** appears outside of the address portion of an expression, it can control whether a 16-bit instruction or a 32-bit instruction is performed. In cases where you can determine the size of the instruction from the type of the operand, Turbo Assembler selects the size of the instruction. The following table shows the instructions that **SMALL** and **LARGE** affect.

Table 13.6: Instructions affected by SMALL and LARGE

Instruction	Effect
PUSH [**SMALL/LARGE**] *segreg*	Selects whether 16-bit or 32-bit form of segment register is **PUSH**ed.
POP [**SMALL/LARGE**] *segreg*	Selects whether 16-bit or 32-bit form of segment register is **POP**ped.
FSAVE [**SMALL/LARGE**] *memptr*	Selects whether small or large version of floating-point state is saved.
FRSTOR [**SMALL/LARGE**] *memptr*	Selects whether small or large version of floating-point state is restored.
FSTENV [**SMALL/LARGE**] *memptr*	Selects whether small or large version of floating-point state is stored.
FLDENV [**SMALL/LARGE**] *memptr*	Selects whether small or large version of floating-point state is loaded.
LGDT [**SMALL/LARGE**] *memptr*	Selects whether small or large version of global descriptor table is loaded.
SGDT [**SMALL/LARGE**] *memptr*	Selects whether small or large version of global descriptor table is saved.
LIDT [**SMALL/LARGE**] *memptr*	Selects whether small or large version of interrupt descriptor table is loaded.
SIDT [**SMALL/LARGE**] *memptr*	Selects whether small or large version of interrupt descriptor table is saved.
JMP [**SMALL/LARGE**] *memptr*	For DWORD-sized memory addresses, selects between FAR 16-bit **JMP** and NEAR 32-bit **JMP**.
CALL [**SMALL/LARGE**] *memptr*	For DWORD-sized memory addresses, selects between FAR 16-bit **CALL** and NEAR 32-bit **CALL**.

Calling procedures with stack frames

Turbo Assembler supports an extended form of the **CALL** instruction that lets you directly call procedures that use high-level language interfacing conventions.

Arguments to procedures that use high-level language interfacing conventions are passed on the stack in a *stack frame*. The caller must push these arguments onto the stack before calling the procedure.

The interfacing convention of the procedure determines the order arguments should be pushed into the stack frame. For BASIC, FORTRAN, and PASCAL procedures, arguments are pushed onto the stack in the order they are encountered; for C and CPP (C++), the arguments are pushed in the reverse order.

The interfacing convention of a procedure also determines whether the procedure or the caller of the procedure must remove the arguments from the stack once the procedure is called. C and C++ require the caller to clean up the stack. In all other languages, the procedure itself must remove the arguments from the stack before returning.

Turbo Assembler handles both the proper argument ordering and stack cleanup for you with the extended **CALL** instruction. The syntax for calling a procedure with parameters follows:

```
CALL expression [language] [,argument_list]
```

See Chapter 7 for further information about using **MODEL**.

expression is the target of the **CALL** instruction. *language* specifies the interfacing convention to use for the call. If you don't specify a language, Turbo Assembler uses the default language set by **MODEL**.

Arguments, if any, follow the language identifier. The syntax of each argument in the argument list is the same as for the extended **PUSH** and **POP** instructions. You can separate these arguments with commas; for example,

```
CALL test PASCAL,ax,es OFFSET buffer,blen
```

PASCAL, the language in the example, causes Turbo Assembler to push the arguments in the same order that it encounters them. This example call is equivalent to

```
PUSH ax
PUSH es OFFSET buffer
PUSH word PTR blen
CALL test
```

A call to a C procedure requires that the arguments be pushed onto the stack in the reverse order. Turbo Assembler automatically does this so that a call of the form

```
CALL test C,ax,es OFFSET buffer,word PTR blen
```

results in the following code:

```
PUSH word PTR blen
PUSH es OFFSET buffer
PUSH ax
CALL test
SUB  sp,8
```

When calling a procedure with arguments, you should always list the arguments in the same order they were listed in the procedure header. Turbo Assembler reverses them if necessary.

Remember to separate arguments with commas and components of arguments with spaces. Turbo Assembler, depending on the interfacing convention, can push arguments in reverse order on the stack, but it won't alter the ordering of argument components.

If the interfacing convention for the call is NOLANGUAGE, Turbo Assembler reports an error if any arguments are present. Although you can define arguments to a NOLANGUAGE procedure with the **ARG** directive, you must explicitly push the arguments when you make a call to a NOLANGUAGE procedure.

Calling procedures that contain RETURNS

Procedures that define some of their arguments with the **RETURNS** keyword must be considered specially. These arguments are used to return values to the caller; therefore, the caller always pops them. There is no special extension to the **CALL** instruction in Turbo Assembler to help pass those arguments specified in a procedure declaration after the **RETURNS** directive. You must explicitly **PUSH** these arguments before the **CALL**, and **POP** them afterward.

Calling procedures that have been prototyped

If you've defined the procedure prior to the call or used **PROCDESC** to prototype the procedure (see Chapter 10), Turbo Assembler will type check any language and arguments specified in the call and generate a warning if the language, number of parameters, or types of parameters don't match.

For example,

```
test PROCDESC pascal far :word,:dword,:word
   :
call test pascal ax,ds bx,cx           ;works fine
call test c, ax,dx, bx,cx              ;wrong language!
call test pascal, eax, ebx, ecx        ;wrong parameter types!
call test pascal, ax,ds bx             ;too few parameters!
```

Since the language of the procedure has been specified, you don't have to include it in the call. If you omit it, however, make sure to include the comma that would normally follow it:

```
call test,ax,ds bx,cx                  ;works fine
```

You can also use procedure types (declared with **PROCTYPE**) to supply a distance and language, and force type-checking to occur. For example,

```
footype proctype pascal near :word,:dword,:word
   :
call footype ptr[bx],ax,ds bx,cs          ;no error!
```

Calling method procedures for objects: CALL..METHOD

The **CALL** instruction is extended to support the calling of object methods. A call to an object method can generate either a direct call (for static methods) or an indirect call (for virtual methods).

Because you can use an indirect call, the instructions that perform the call can destroy the contents of some registers. Therefore, Turbo Assembler lets you select the proper registers if you're using a virtual method call.

OOP

Here's the syntax of the **CALL..METHOD** extension:

```
CALL instance_ptr METHOD [object_name:]method_name [USES [segreg:]offsreg]
[language_and_args]
```

instance_ptr must describe an instance of an object. In MASM mode, it's often impossible to determine the name of the object associated with an instance. Therefore, Turbo Assembler allows the *object_name* field, so that you can specify the instance's object name.

See Chapter 8 for further information about how to specify a method as virtual or static.

method_name contains the name of the method to be called for the specified object instance.

If the method is virtual and an indirect call is required, the **CALL..METHOD** instruction normally calls indirectly through ES:BX (or ES:EBX for USE32 models on the 80386 processor). If you want to use other registers, you can override them with the **USES** clause. *segreg* is the optional segment register to use, and *offsreg* is the offset register to use for the call.

Note: It's good programming practice to specify an appropriate selection for indirect calling registers, even if you know that the method you're calling is static. As objects are modified, methods can change from being static to virtual.

For objects declared with near tables, **CALL..METHOD** only loads the offset register. Turbo Assembler assumes that the segment register is already set up to the correct value.

The *language_and_args* field of the **CALL..METHOD** instruction contains the optional language and argument specifications, which are identical in form to that listed previously under "Calling procedures with stack frames."

Calling method procedures for C++ or Pascal usually requires that the instance of the object be passed as an argument on the stack. See Chapter 18 for further information.

Tail recursion for object methods: JMP..METHOD

Turbo Assembler provides a **JMP..METHOD** instruction that corresponds to the **CALL..METHOD** instruction. Here's its syntax:

```
JMP instance_ptr METHOD [object_name:]method_name [USES [segreg:]offsreg]
```

OOP

JMP..METHOD functions exactly like **CALL..METHOD** except that

- It generates a **JMP** instead of a **CALL** instruction.
- It generates procedure epilog code to clean up the stack before the **JMP** instruction is generated.

The **JMP..METHOD** instruction makes it possible to write efficient tail recursion code. It's intended to replace the common situation where a

CALL..METHOD instruction is issued to the current method, followed by a **RET** instruction.

Additional instruction for object-oriented programming

OOP

When an object instance is constructed, you must initialize the instance's virtual table pointer (if any) to point to the correct virtual method table. The **TBLINIT** instruction lets you do this automatically. The syntax of the **TBLINIT** instruction is

```
TBLINIT object_instance_pointer
```

The *object_instance_pointer* field is the address of the object whose virtual table pointer is to be initialized. The **TBLINIT** instruction assumes that the object instance should be of the current object type (in other words, the immediately preceding object definition determines the object type that **TBLINIT** initializes). For example,

```
TBLINIT DS:SI
```

would initialize the virtual table pointer of the object at DS:SI, if it has one.

C H A P T E R 14

Using macros

Macros let you give a symbolic name to a text string or a block of code that will be used frequently throughout your program. Macros go beyond this simple substitution, however. Turbo Assembler has macro operators that provide great flexibility in designing macros. Combined with the ability to use multiline macros with arguments, this makes Turbo Assembler's macro facility a very powerful tool. This chapter discusses how to use text and multiline macros in your program.

Text macros

A text macro is a symbol that represents a string of text characters. When Turbo Assembler encounters the symbol in expressions (and other situations), it substitutes the text characters for the symbol. For example, if *DoneMsg* is a text macro whose value is "Returning to the OS", the following statement

```
GoodBye  DB DoneMsg
```

results in

```
GoodBye DB 'Returning to the OS'
```

Defining text macros with the EQU directive

You can use the **EQU** directive to define simple text macros. Here's the syntax for defining a text macro:

```
name EQU text_string
```

text_string associates with the text macro name *name*. You should enclose *text_string* in brackets (< >) to delineate the text; for example,

```
DoneMsg  EQU <'Returning to the OS'>
```

Note: If you omit the brackets in MASM mode, Turbo Assembler will try to evaluate *text_string* to an expression, and an error may result. Only if it can't evaluate *text_string* will Turbo Assembler treat it as a text macro (to remain compatible with MASM).

In Ideal mode, **EQU** always defines a text macro. If you don't enclose *text_string* in brackets and it's the name of another text macro, Turbo Assembler will use that macro's contents. Otherwise, the macro will be defined to the text.

You should always enclose text macro strings in angle brackets to make sure they're properly defined. Consider the following mistake that can occur when you don't:

```
IDEAL
Earth  EQU dirt          ;Earth = "dirt"
Planet EQU Earth         ;Planet = "dirt"  (wrong!)
Planet EQU <Earth>       ;Planet = "Earth" (correct!)
```

In Ideal mode, the **EQU** statement always defines a text macro.

Text macros are redefinable; you can redefine a text macro name in the same module to another text string.

String macro manipulation directives

Turbo Assembler provides directives that can manipulate string macros. These directives are available in Ideal mode, and for versions M510, M520, and T300 or later (as specified by the **VERSION** directive).

A *string* argument for any of these directives can be any of the following:

- a text string enclosed in brackets; for instance, *<abc>*
- the name of a previously defined text macro
- an expression preceded by a % character, whose value is converted to the equivalent numerical string representation appropriate for the current radix

The CATSTR directive

The **CATSTR** directive defines a new text macro by concatenating strings together. **CATSTR** has the following syntax:

```
name CATSTR string[,string]...
```

CATSTR concatenates from left to right. Turbo Assembler creates a new text macro of the name *name*.

The SUBSTR directive

The **SUBSTR** directive defines a new text macro to be a substring of a string. Here's its syntax:

```
name SUBSTR string,position_expression[,size_expression]
```

The new text macro, *name* consists of the portion of *string* that starts at the *position_expression* character, and is *size_expression* characters in length. If you don't supply *size_expression*, the new text macro consists of the rest of *string* from the character at *position_expression*. Turbo Assembler considers the first character of *string* to be at position 1.

The INSTR directive

The **INSTR** directive returns the position of one string inside another string. **INSTR** has the following syntax:

```
name INSTR [start_expression,]string1,string2
```

Turbo Assembler assigns *name* a numeric value that is the position of the first instance of *string2* in *string1*. The first character in *string1* has a position of 1. If *string2* does not appear anywhere within *string1,* Turbo Assembler returns a value of 0. If you include *start_expression,* the search begins at the *start_expression* character. The first character of a string is 1.

The SIZESTR directive

The **SIZESTR** directive returns the length of a text macro (the number of characters in the string). Here's its syntax:

```
name SIZESTR string
```

name is set to the numeric value of the length of the *string*. A null string < > has a length of zero.

Text macro manipulation examples

The following examples show how these operators work:

```
VERSION T300
IDEAL
ABC     EQU     <abc>                   ;ABC = "abc"
ABC2    EQU     ABC                     ;ABC2 = "abc"
ABC     EQU     <def>                   ;ABC = "def" (redefined)
ABC3    CATSTR  ABC2,<,>,ABC,<,>,ABC2   ;ABC3 = "abc,def,abc"
ABCLEN  SIZESTR ABC                     ;ABCLEN = 3
ABC3LEN SIZESTR ABC3                    ;ABC3LEN = 11
COMMA1  INSTR   ABC3,<,>                ;COMMA1 = 4
COMMA2  INSTR   COMMA1+1,ABC3,<,>       ;COMMA2 = 8
ABC4    SUBSTR  ABC3,5                  ;ABC4 = "def,abc"
ABC5    SUBSTR  ABC3,5,3                ;ABC5 = "def"
ABC6    EQU     3+2+1                   ;ABC6 = 6 (numeric equate)
ABC7    EQU     %3+2+1                  ;ABC7 = "6" (text macro)
ABC8    EQU     %COMMA1                 ;ABC8 = "4"
```

Multiline macros

The multiline macro facility lets you define a body of instructions, directives, or other macros that you'll include in your source code whenever the macro is invoked. You can supply arguments to the macro that Turbo Assembler will substitute into the macro body when you include the macro in the module.

There are several types of multiline macros. One version substitutes each element of a string, one after the other, as an argument to the macro. Another version repeats the macro body a certain number of times. Finally, you can define still another version in one place, and invoke it many times. All versions have the definition of a macro body in common.

The multiline macro body

Regardless of its actual content, Turbo Assembler's macro processing facility treats a multiline macro body as merely a number of lines of text. Turbo Assembler lets you replace symbols within the macro body with text specified at the time a macro is invoked. This feature is called *argument substitution*. The symbols in the macro body that will be replaced are called *dummy arguments*. For example, suppose the symbol *foo* is a dummy argument in the following macro body:

```
PUSH foo
MOV  foo,1
```

If you assign *foo* with the text string AX when you invoke this macro, the actual text included in the module will be

```
PUSH AX
MOV  AX,1
```

The rules Turbo Assembler uses for recognizing a dummy argument are fairly complex. Examine the following macro body lines where the dummy argument *foo* would not be recognized:

```
symfoo:
   DB 'It is foo time'
```

In general, Turbo Assembler will not recognize a dummy argument without special help in the following situations:

- when it is part of another symbol
- when it is inside of quotation marks (' or ")
- in Ideal mode, when it appears after a semicolon not inside of quotes

Using & in macros

The **&** character has a special meaning when used with the macro parameters. In general, **&** separates a dummy argument name from surrounding text, so Turbo Assembler can recognize it for substitution. For example, given the following Ideal mode macro:

```
macro mac1 foo
sym&foo:
   DB 'It is &foo time'
endm
```

if you assign *foo* the text string *party* when this macro is invoked, the actual text included in the module will be

```
symparty:
   DB 'It is party time'
```

Another example might be

```
foo&sym:
   DB 'We are in O&foo&o'
```

If you assign *foo* the text string *hi* when this macro is invoked, the text included in the module will be

```
hisym:
   DB 'We are in Ohio'
```

Here are the rules for the **&** character:

- Outside quoted strings, the **&** serves only as a general separator.
- Inside quoted strings and after a semicolon that's not in a quoted string in Ideal mode, **&** must precede a dummy argument for it to be recognized.
- Turbo Assembler removes one **&** from any group of **&**s during a macro expansion.

The last point makes it possible to place macro definitions requiring **&** characters inside other macro definitions. Turbo Assembler will remove only one **&** from any group.

Including comments in macro bodies

For particularly complicated macros, you might want to include (in the macro body text) comments that won't be included when the macro is invoked. This also reduces the memory required for Turbo Assembler to process macros. To do this, use the double semicolon comment at the beginning of a line. For example, the following macro body

Note: comments preceded by single semicolons are always included in a macro expansion.

```
;;Wow, this is a nasty macro!
DB 'Nasty macro'
```

will only include the following text when it is invoked:

```
DB 'Nasty macro'
```

Local dummy arguments

At the beginning of any macro body, you can include one or more **LOCAL** directives. **LOCAL** declares special dummy arguments that, each time the macro expands, will be assigned a unique symbol name.

The syntax for the **LOCAL** directive in macro bodies looks like this:

```
LOCAL dummy_argument1 [,dummy_argument2]...
```

See Chapter 11 for details on how to enable local symbols and set the local symbol prefix.

If the *dummy_argument* name used in the **LOCAL** directive does not have a local symbol prefix the unique symbol name assigned to it will be in the form *??xxxx*, where *xxxx* represents a hexadecimal number. Otherwise, the unique symbol name will be `<local prefix>xxxx`.

The **LOCAL** directives must come before any other statements in a macro body.

You can use **LOCAL** dummy arguments to define labels within the macro body. For example,

```
LOCAL  @@agn,@@zero
 XOR dx,dx
 MOV cx,exp
 MOV ax,1
 JCXZ @@zero
 MOV bx,factor
@@agn: MUL bx
 LOOP @@agn
@@zero:
```

Note: In macros, you don't have to use @@ since local labels in macros are turned into consecutive numbers, like ??0001. Their names are not easily accessible outside macros.

The EXITM directive

You can use the **EXITM** directive within a macro body to prematurely terminate the assembly of an included macro body. Its syntax follows:

```
EXITM
```

When Turbo Assembler encounters **EXITM** in a macro body that has been included in the module source code, assembly of the expanded macro body stops immediately. Instead, Turbo Assembler will continue assembling the module at the end of the macro.

You can use the **EXITM** statement with a conditional assembly directive to terminate a macro expansion when certain conditions are met.

Tags and the GOTO directive

Using macro tags and the **GOTO** directive lets you control the sequence in which lines within the macro body expand. You can place a macro tag at any place within the macro body. The tag occupies an entire line in the macro, with the following syntax:

```
:tag_symbol
```

When the macro expands, all macro tags are discarded.

The **GOTO** directive tells the assembler to go to a specified point in your code, namely the tag_symbol. **GOTO** has the following syntax:

```
GOTO tag_symbol
```

Be careful not to create infinite macro loops when you use the **GOTO** directive. Infinite loops can cause Turbo Assembler to run out of memory, or even appear to stop functioning.

GOTO also terminates any conditional block that contains another **GOTO**. This lets you place **GOTO** inside conditional assembly blocks. For example,

```
IF foo
   GOTO tag1
ENDIF
   DISPLAY "foo was false!"
:tag1
          ;resume macro here...
          ;works the same whether foo was false or true
```

See Chapter 15 for further information about conditional assembly directives.

General multiline macros

Turbo Assembler associates a general multiline macro's body of directives, instructions, and other macros with a symbolic macro name. Turbo Assembler inserts the body of statements into your program wherever you use the macro name as a directive. In this way, you can use a general multiline macro more than once.

You can invoke a macro before you define it only when you use the **/m** command-line switch; see Chapter 2 for details. However, this is poor programming practice.

Here's the Ideal mode syntax for defining a general multiline macro:

```
MACRO name parameter_list
macro_body
ENDM
```

Here's the MASM mode syntax for defining a general multiline macro:

```
name MACRO parameter_list
macro_body
ENDM
```

name is the name of the multiline macro you're defining. *macro_body* contains the statements that make up the body of the macro expansion. You can place any valid (and any number of) Turbo Assembler statements within a macro. The **ENDM** keyword terminates the macro body.

This example defines a macro named PUSHALL that, when invoked, includes the macro body consisting of three **PUSH** instructions into your program.

```
PUSHALL  MACRO
   PUSH AX BX CX DX
   PUSH DS SI
   PUSH ES DI
ENDM
```

parameter_list is a list of dummy argument symbols for the macro. Here's its syntax:

```
[dummy_argument [,dummy_argument ...]]
```

You can use any number of dummy arguments with a macro, as long as they fit on one line, or you use the line continuation character (\) to continue them to the next line. For example,

```
ADDUP  MACRO dest,\       ;dest is 1st dummy argument
  s1,s2                   ;s1,s2 are 2nd and 3rd dummy arguments
  MOV dest,s1
  ADD dest,s2
ENDM
```

Each dummy argument has the following syntax:

```
dummy_name[:dummy_type]
```

dummy_name is a symbolic name used as a place holder for the actual argument passed to the macro when it's invoked. The optional *dummy_type* specifies something about the form the actual argument must take when you invoke the macro. The following types are supported:

Table 14.1
Dummy argument types

Type	Meaning
REQ	Argument cannot be null or spaces.
=<*text_string*>	Bracketed text string is the default value for the dummy argument when the actual argument is null or contains spaces.
VARARG	Actual argument consists of the rest of the macro invocation, interpreted as a list of arguments. Commas and angle brackets are added to ensure this interpretation.
REST	Actual argument consists of the rest of the macro invocation, interpreted as raw text.

Invoking a general multiline macro

To invoke a general multiline macro, use the name of the macro as a directive in your program. Turbo Assembler inserts the macro body (after all the dummy arguments are substituted) at that point in the module. The syntax for invoking a general multiline macro is as follows:

```
macro_name [argument [[,]argument]...]
```

macro_name is the symbolic name of a macro. If you invoke a macro with arguments, the arguments are listed following the macro name. You can specify any number of arguments, but they must all fit on one line. Separate multiple arguments with commas or spaces. When the macro expands, Turbo Assembler replaces the first dummy argument in the macro definition with the first argument passed, the second dummy argument with the second argument, and so forth.

Each *argument* represents a text string. You can specify this text string in the following ways:

- as a contiguous group of characters, not containing any whitespace, commas, or semicolons
- as a group of characters delineated by angle brackets (< >), which can contain spaces, commas, and semicolons
- as a single character preceded by a ! character, which is equivalent to enclosing the character in angle brackets
- as an expression preceded by a % character, which represents the text value of the expression appropriate for the currently selected radix

The < > literal string brackets

Use angle brackets to delineate a literal string that contains the characters between them. You should use them like this:

```
<text>
```

text is treated as a single string parameter, even it if contains commas, spaces, or tabs that usually separate each parameter. Use this operator when you want to pass an argument that contains any of these separator characters.

You can also use this operator to force Turbo Assembler to treat a character literally, without giving it any special meaning. For example, if you want to pass a semicolon (;) as a parameter to a macro invocation, you have to enclose it in angle brackets (<;>) to prevent it from being treated as the beginning of a comment. Turbo Assembler removes only one level of angle brackets when it converts a bracketed string to a text argument. This makes it possible to invoke a macro requiring angle brackets from inside another macro body.

The ! character

The ! character lets you invoke macros with arguments that contain special characters. Using this character prior to another is similar to enclosing the second character in angle brackets. For example, !; functions the same as <;>. Some common uses are shown in the following table.

Table 14.2 Uses for the ! character

String	Resulting character
!>	>
!<	<
!!	!

The % expression evaluation character

The **%** character causes Turbo Assembler to evaluate an expression. The assembler converts the result of the expression to an ASCII number in the current radix, which is the text that the **%** character produces. Use this character when you want to pass the string representing a calculated result, rather than the expression itself, as a macro argument. The syntax follows:

```
%expr
```

See Chapter 5 for more information about Turbo Assembler expressions.

expr can be either an expression (using any legal operands and operators), or it can be the name of a text macro. If it is an expression, the text that is produced is the result of the expression, represented as a numerical string in the current radix. If *expr* is a text macro name, the text that's produced is the string that the text macro represents.

For example, this code

```
DEFSYM MACRO NUM
TMP_&NUM:
ENDM

TNAME EQU <JUNK>            ;defining a text macro
DEFSYM %5+4
DEFSYM %TNAME
```

results in the following code macro expansions:

```
TMP_9:
TMP_JUNK:
```

Redefining a general multiline macro

You can redefine general multiline macros. The new definition automatically replaces the old definition. All preceding places where the macro had already been invoked will not change. All invocations of the macro following the redefinition use the new definition.

Deleting a general multiline macro: The PURGE directive

You can use the **PURGE** directive to delete a macro. **PURGE** has the following syntax:

```
PURGE macroname [,macroname]...
```

PURGE deletes the general multiline macro definition associated with *macroname*. After you **PURGE** a macro, Turbo Assembler no longer treats the symbol *macroname* as if it were a macro; for example,

```
ADD MACRO a1,a2
   SUB a1,a2
ENDM
   ADD ax,bx    ;This invocation will produce SUB ax,bx
PURGE ADD
   ADD ax,bx    ;This is no longer a macro, so ADD ax,bx is produced
```

You can purge several macros at a time by separating their names with commas. Note, however, that you can't redefine a purged macro symbol as anything other than another macro.

Defining nested and recursive macros

The statements in a macro body can include statements that invoke or define other macros. If you take this example,

```
MCREATE MACRO opname,op1,op2,op3,op4,op5,op6,op7
 IFNB opname
  DO&opname MACRO op,count
   IF count LE 4
    REPT count
     opname op,1
    ENDM
   ELSE
    MOV CL,count
    opname op,CL
   ENDIF
  ENDM                                   ;end of DOopname macro
 MCREATE op1,op2,op3,op4,op5,op6,op7     ;recurse!
ENDIF                                    ;end of if
ENDM                                     ;end of MCREATE macro
```

and invoke it with

```
MCREATE         ror,rol,rcl,rcr,shl,shr,sal,sar
```

it will create the additional macros DOror, DOrol, and so forth, which you can then use like this:

```
DOshr           ax,5
DOrcr           bx,3
```

You can call recursive macros with a list of parameters, and set them up so that the macro will work with anywhere from zero to a maximum number of parameters. To do this, have the macro body use the first parameter to do its expansion, then call itself with the remaining parameters. Every time it recurses, there will be one fewer parameter. Eventually, it will recurse with no parameters.

See Chapter 15 for more information about the **IFNB** directive.

When you call the macro recursively, it always needs some way to test for the end of the recursion. Usually, an **IFNB** conditional statement will do this for only the macro body if the passed parameter is present. Here is a simpler example of a recursive macro:

```
PUSHM MACRO r1,r2,r3,r4,r5,r6,r7,r8
IFNB r1
   push r1
   PUSHM r2,r3,r4,r5,r6,r7,r8
ENDIF
ENDM
```

The count repeat macro

You can use the **REPT** repeating macro directive to repeat a macro body a specific number of times, using this syntax:

```
REPT expression
macro_body
ENDM
```

expression tells Turbo Assembler how many times to repeat the macro body specified between the **REPT** and **END** directives. *expression* must evaluate to a constant and can't contain any forward-referenced symbol names. Use **ENDM** to mark the end of the repeat block. For example, this code

```
REPT 4
   SHL ax,1
ENDM
```

produces the following:

```
SHL ax,1
SHL ax,1
SHL ax,1
SHL ax,1
```

Another example shows how to use **REPT** in a macro to generate numbers that are the various powers of two:

```
count = 0

defname macro num
  Bit&num dd (1 SHL (&num))
endm
```

```
rept 32
  defname %count
  count = count + 1
endm
```

The WHILE directive

You can use the **WHILE** macro directive to repeat a macro body until a certain expression evaluates to 0 (false). **WHILE** has the following syntax:

```
WHILE while_expression
macro_body
ENDM
```

Turbo Assembler evaluates *while_expression* before each iteration of the macro body. Be careful to avoid infinite loops, which can cause Turbo Assembler to run out of memory or appear to stop functioning. Here's an example using **WHILE**:

```
WHILE 1
    ;; Do nothing
ENDM
    ; We never make it this far
```

String repeat macros

You can use the **IRP** and **IRPC** string repeat macro directives to repeat a macro body once for each element in a list or each character in a string. Each of these directives requires you to specify a single dummy argument. Here's the **IRP** syntax:

```
IRP dummy_argument, argument_list
macro_body
ENDM
```

IRPC has the following syntax:

```
IRPC dummy_argument, string
macro_body
ENDM
```

In both cases, *dummy_argument* is the dummy argument used in the macro body. **ENDM** marks the end of the macro body.

For **IRP**, *argument_list* consists of a list of arguments separated by commas. The arguments can be any text, such as symbols, strings, numbers, and so on. The form of each argument in the list is similar to that described for general multiline macro invocations, described earlier in this chapter. You must always surround the argument list with angle brackets (< >).

For **IRPC**, the argument consists of a single *string*. The string can contain as many characters as you want.

For each argument or character in a string, Turbo Assembler will include the macro body in the module, substituting the argument or character for the dummy argument wherever it finds it. For example,

```
IRP reg,<ax,bx,cx,dx>
   PUSH reg
ENDM
```

produces the following:

```
PUSH ax
PUSH bx
PUSH cx
PUSH dx
```

and the directive **IRPC**

```
IRPC LUCKY,1379
   DB LUCKY
ENDM
```

produces this:

```
DB 1
DB 3
DB 7
DB 9
```

Be careful when using **IRPC** because Turbo Assembler places each character in the string "as is" in the expanded macro, so that a string repeat macro such as

```
IRPC CHAR,HELLO
   DB CHAR
ENDM
```

might not produce DB 'H', 'E', 'L', 'L', 'O', but instead would produce DB H, E, L, L, O (where each letter is treated as a symbol name).

The % immediate macro directive

The **%** immediate macro directive treats a line of text as if it's a macro body. The dummy argument names used for the macro body include all of the text macros defined at that time. Here's its syntax:

```
% macro_body_line
```

macro_body_line represents the macro body to use for the immediate macro expansion; for example:

```
SEGSIZE EQU <TINY>
LANGUAGE EQU <WINDOWS PASCAL>

% MODEL SEGSIZE,LANGUAGE     ;Produces MODEL TINY,WINDOWS PASCAL
```

Including multiline macro expansions in the list file

Chapter 17 has more details.

Multiline macro expansions are not normally included in the listing file. However, Turbo Assembler provides the following directives that let you list macro expansions:

- **.LALL**
- **.SALL**
- **.XALL**
- **%MACS**
- **%NOMACS**

Saving the current operating state

The **PUSHSTATE** directive saves the current operating state on an internal stack that is 16 levels deep. **PUSHSTATE** is particularly useful if you have code inside a macro that functions independently of the current operating state, but does not affect the current operating mode.

Note that you can use **PUSHSTATE** outside of macros. This can be useful for include files.

The state information that Turbo Assembler saves consists of:

- current emulation version (for example, T310)
- mode selection (for example, IDEAL, MASM, QUIRKS, MASM51)
- EMUL or NOEMUL switches
- current processor or coprocessor selection
- MULTERRS or NOMULTERRS switches
- SMART or NOSMART switches
- the current radix
- JUMPS or NOJUMPS switches
- LOCALS or NOLOCALS switches
- the current local symbol prefix

Use the **POPSTATE** directive to return to the last saved state from the stack.

Here's an example of how to use **PUSHSTATE** and **POPSTATE**.

```
; PUSHSTATE and POPSTATE examples

ideal
model small
codeseg

jumps
locals  @@

    ; Show changing processor selection, number radix, and JUMPS mode
        pushstate
        nojumps
        radix   2           ; Set to binary radix
        p386
        jl      next1       ; No extra NOPS after this
        mov     eax,100     ; Now 100 means binary 100 or 4 decimal.
next1:
        popstate            ; Restores JUMPS and non 386 mode.

    ; Back to jumps directive, no 386, and decimal radix
        jl      next2       ; Three extra NOPS to handle JUMPS
        xor     eax,eax     ; Not in 386 mode anymore!

        mov     cx,100      ; Now 100 means decimal 100

        pushstate
        MULTERRS
        mov     ax,[bp+abc
        popstate

        mov     ax,[bp+abc

        ; Show disabling local scoping of symbols
        locals
next2:
@@a:    loop @@a
next3:
@@a:    loop @@a  ; Allowed because of scoping of NEXT2: and
                    ; NEXT3:

        pushstate
        nolocals
next4:
@@b:    loop @@b
next5:
@@b:    loop @@b    ; This will conflict because of nolocals
        popstate

        ; Show changing local symbol prefix and MASM/IDEAL mode
        pushstate
        masm
        locals @$
```

```
testproc proc               ; MASM mode for procedure declaration
         jmp    @$end
@$end:   nop
@@end:   ret
testproc endp

testproc2 proc
        jmp    @$end
@$end:  nop                ; This doesn't conflict with label in
                           ; TESTPROC
@@end:  ret                ; This label does conflict
testproc2 endp
        popstate

        ; Now back to @@ as a local label prefix, and IDEAL mode
testproc2b  proc            ; This won't work since we are back in
                            ; IDEAL mode!
        ret
testproc2b  endp            ; And this will give an error also.

proc    testproc3
        jmp    @$end2
@$end2: nop
@@end2: ret
endp    testproc3

proc    testproc4
        jmp    @$end2
@$end2: nop                 ; This label does conflict
@@end2: ret                 ; This label doesn't conflict with
                            ;    label in TESTPROC3
endp    testproc4

end
```

CHAPTER 15

Using conditional directives

There are two classes of conditional directives: conditional assembly directives and conditional error-generation directives. With conditional assembly directives, you can control which code gets assembled in your program under certain conditions.

Conditional error-generation directives let you generate an assembly-time error message if certain conditions occur. Turbo Assembler displays the error message on the screen and in the listing file, and it acts like any other error message in that it prevents the emission of an object file. This chapter describes how you can use the available conditional directives.

General conditional directives syntax

The three types of conditional assembly directives are **IF*xxx*** directives, **ELSEIF*xxx*** directives, and **ERR*xxx*** directives. Use these directives as you would conditional statements in high-level languages.

IFxxx conditional assembly directives

You can use **IF*xxx*** conditional assembly directives to define blocks of code that are included in the object file if certain conditions are met (such as whether a symbol is defined or set to a particular value). Here's the syntax of a conditional assembly statement:

```
IFxxx
true_conditional_body
ENDIF
```

or

```
IFxxx
true_conditional_body
ELSE
false_conditional_body
ENDIF
```

Here, **IF*xxx*** represents any of the following conditional assembly directives:

IF	**IFNB**
IF1	**IFIDN**
IF2	**IFIDNI**
IFDEF	**IFDIF**
IFNDEF	**IFDIFI**
IFB	

Each **IF*xxx*** conditional assembly directive specifies a specific condition that evaluates to either true or false. If the condition is true, the block of assembly code in *true_conditional_body* is assembled into the output object file. If the condition evaluates to false, Turbo Assembler skips over *true_conditional_body* and does not include it in the object file. If there is an **ELSE** directive, the *false_conditional_body* is assembled into the object file if the condition is false; it's ignored if the condition is true. All conditionals are terminated with an **ENDIF** directive.

Note: Except for the special cases of **IF1** and **IF2** (which are discussed later), the two bodies of code are mutually exclusive: Either *true_conditional_body* will be included in the object file or *false_conditional_body*, but never both. Also, if you use the **IF*xxx*...ELSE...ENDIF** form, one of the two bodies will be included in the generated object file. If only the **IF*xxx*...ENDIF** form is used, *true_conditional_body* may or may not be included, depending on the condition.

When you nest **IF**s and **ELSE**s, **ELSE** always pairs with the nearest preceding **IF** directive.

In this example, *test* is a symbol that flags the inclusion of test code (if the symbol is defined, then test code is generated). *color* is a symbol set to nonzero if the display is color, or 0 for a monochrome display.

The actual code generated depends on these values:

```
 :
IFDEF test                  ;T if test defined
    ;test code 1            ; if test defined
 IF color                   ;T if color <> 0
    ;color code             ; if color <> 0
 ELSE                       ;
    ;mono code              ; if color = 0
 ENDIF                      ;
    ;test code 2            ; if test defined
```

```
ELSE                          ;
    ;non-test code            ; if test not defined
ENDIF
⋮
```

Test: Color:	Defined 0	Defined Nonzero	Undefined 0	Undefined Nonzero
code:	test code 1 mono code test code 2	test code 1 color code test code 2	non-test code	non-test code

Note: If *test* is undefined, neither the color nor monochrome debug code based on the value of color is assembled, as this lies entirely within the conditional assembly for a defined test.

ELSEIFxxx conditional assembly directives

You can use the **ELSEIF*xxx*** as a shortcut where multiple **IF**s are required. **ELSEIF*xxx*** is equivalent to an **ELSE** followed by a nested **IF*xxx***, but provides more compact code. For example,

```
⋮
IF mode EQ 0
 ;mode 0 code
ELSEIF mode LT 5
 ;mode 1-4 code
ELSE
 ;mode 5+ code
ENDIF
⋮
```

compares to

```
⋮
IF mode EQ 0
 ;mode 0 code
ELSE
 IF mode LT 5
  ;mode 1-4 code
 ELSE
  ;mode 5+ code
 ENDIF
ENDIF
⋮
```

You can't use the **ELSEIF*xxx*** directives outside of an **IF*xxx*** statement.

ERRxxx error-generation directives

ERR*xxx* directives generate user errors when certain conditions are met. These conditions are the same as for the **IF*xxx*** conditional assembly directives. Here's the general syntax:

```
ERRxxx [arguments] [message]
```

In this case, **ERR*xxx*** represents any of the conditional error-generating directives (such as **ERRIFB**, **.ERRB**, and so on).

arguments represents arguments that the directive might require to evaluate its condition. Some directives require an expression, some require a symbol expression, and some require one or two text expressions. Other directives require no arguments at all.

If *message* is included, it represents an optional message that's displayed along with the error. The message must be enclosed in single (') or double (") quotation marks.

The error-generating directives generate a *user error* that is displayed onscreen and included in the listing file (if there is one) at the location of the directive in your code. If the directive specifies a message, it displays on the same line immediately following the error. For example, the directive

```
ERRIFNDEF foo "foo not defined!"
```

generates the error

```
User error: "foo not defined!"
```

if the symbol *foo* is not defined when the directive is encountered. No error would be generated in this case if *foo* were already defined.

Specific directive descriptions

Unconditional error-generation directives

The unconditional error-generation directives are **ERR** and **.ERR**. These directives always generate an error and require no arguments, although they can have an optional message. You can only use **.ERR** in MASM mode.

Expression-conditional directives

These directives provide conditional assembly or error generation based on the results of evaluating a Turbo Assembler expression. For all of these directives, the expression must evaluate to a constant and can't contain any forward references. If it evaluates to 0, Turbo Assembler considers the expression to be false; otherwise, it considers the expression to be true.

The following table shows conditional assembly directives that use expressions.

Table 15.1
Conditional assembly directives using expressions

IF*xxx* directive	**Assembles true_conditional_body if**
IF *expression*	Expression evaluates to true.
IFE *expression*	Expression evaluates to false.
ELSEIF *expression*	Expression evaluates to true.
ELSEIFE *expression*	Expression evaluates to false.

The following table shows the error-generation directives that use expressions.

Table 15.2
Error-generation directives using expressions

ERR*xxx* directive	**Generates user error if**
ERRIF *expression*	Expression evaluates to true.
.ERRNZ *expression*	Expression evaluates to true (MASM mode only).
ERRIFE *expression*	Expression evaluates to false.
.ERRE *expression*	Expression evaluates to false (MASM mode only).

Symbol-definition conditional directives

These directives provide conditional assembly or error generation based on whether one or more symbols are defined. These symbols are organized into a *symbol_expression*.

A *symbol_expression* is an expression made up of symbol names, the Boolean operators **AND**, **OR**, and **NOT**, and parentheses. In a *symbol_expression*, each symbol name is treated as a Boolean value that evaluates to true if the symbol currently exists, or false if the symbol does not exist (even if it's defined later in the module). Turbo Assembler combines these values using the Boolean operators to produce a final true or false result. In its simplest form, a symbol expression consists of a single symbol name and evaluates to true if the symbol is defined. The parsing and syntax rules for *symbol_expression* are similar to those for other Turbo Assembler expressions.

For example, if the symbol *foo* is defined but the symbol *bar* is not, the following symbol-expression evaluations are returned:

Table 15.3
Evaluation of defined and undefined symbol

Symbol expression	**Result**
foo	True
bar	False
not foo	False
not bar	True

Table 15.3: Evaluation of defined and undefined symbol (continued)

foo **OR** *bar*	True
foo **AND** *bar*	False
NOT (*foo* **AND** *bar*)	True
NOT *foo* **OR NOT** *bar*	True (same as "(**NOT** *foo*) **OR** (**NOT** *bar*)")

The directives that control assembly and use *symbol_expressions* are shown in the following table.

Table 15.4 Symbol-expression directives using symbol_expr

IF*xxx* directive	**Assembles true_conditional_body**
IFDEF *symbol_expr*	*symbol_expr* evaluates to true.
IFNDEF *symbol_expr*	*symbol_expr* evaluates to false.
ELSEIFDEF *symbol_expr*	*symbol_expr* evaluates to true.
ELSEIFNDEF *symbol_expr*	*symbol_expr* evaluates to false.

The error-generation directives that use symbol_expressions are shown in the following table.

Table 15.5 Error-generation directives

ERR*xxx* directive	**Generates user error if**
ERRIFDEF *symbol_expr*	*symbol_expr* evaluates to true.
.ERRDEF *symbol_expr*	*symbol_expr* evaluates to true (MASM mode only).
ERRIFNDEF *symbol_expr*	*symbol_expr* evaluates to false.
.ERRNDEF *symbol_expr*	*symbol_expr* evaluates to false (MASM mode only).

For example, the following error-generating conditionals are equivalent, and would generate an error only if both *foo* and *bar* are currently defined:

```
ERRIFDEF foo AND bar
ERRIFNDEF NOT ( foo AND bar )
ERRIFNDEF NOT foo OR NOT bar
```

Text-string conditional directives

These directives provide conditional assembly or error generation based on the contents of *text_string*. A *text_string* can be either a string constant delineated by brackets (< >) or a text macro name preceded by a percent sign (%). For example,

See Chapter 14 for information about how to define and manipulate text macros.

```
<ABC>                    ; text string ABC
%foo                     ; the contents of text macro foo
```

The conditional assembly directives that use *text_string* are shown in the following table:

Table 15.6: Conditional assembly directives using text_strings

IF*xxx* directive	Assembles true_conditional_body if
IFNB *txt_str*	*txt_str* is not blank.
IFB *txt_str*	*txt_str* is blank (empty).
IFIDN *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are identical text strings.
IFIDNI *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are identical text strings, ignoring case distinctions.
IFDIF *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are different text strings.
IFDIFI *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are different text strings, ignoring case distinctions.
ELSEIFNB *txt_str*	*txt_str* is not blank.
ELSEIFB *txt_str*	*txt_str* is blank (empty).
ELSEIFIDN *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are identical text strings.
ELSEIFIDNI *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are identical text strings, ignoring case distinctions.
ELSEIFDIF *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are different text strings.
ELSEIFDIFI *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are different text strings, ignoring case distinctions.

The error-generation directives that use *text_string* are shown in Table 15.7:

Table 15.7: Error-generation directives using text_strings

ERR*xxx* directive	Generates user error if
ERRIFNB *txt_str*	*txt_str* is not blank.
.ERRNB *txt_str*	*txt_str* is not blank (MASM mode only).
ERRIFB *txt_str*	*txt_str* is blank (null).
.ERRB *txt_str*	*txt_str* is blank (MASM mode only).
ERRIFIDN *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are identical text strings.
.ERRIDN *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are identical text strings (MASM mode only).
ERRIFIDNI *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are identical text strings, ignoring case distinctions.
.ERRIDNI *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are identical text strings, ignoring case distinctions (MASM mode only).
ERRIFDIF *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are different text strings.
.ERRDIF *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are different text strings (MASM mode only).
ERRIFDIFI *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are different text strings, ignoring case distinctions.
.ERRDIFI *txt_str1, txt_str2*	*txt_str1* and *txt_str2* are different text strings, ignoring case distinctions (MASM mode only).

Use these directives to check the arguments passed to macros. (Note that they are not restricted to use within macros.)

When used within a macro definition, **IFB** and **IFNB** can determine whether you've supplied the proper number of arguments to the macro. When invoking a macro, Turbo Assembler does not generate an error message if you've supplied too few arguments; instead, the unspecified arguments are blank. In this way, you can define a macro that may take arguments. For example,

```
:
load MACRO addr,reg
 IFNB <reg>
  MOV reg,addr
 ELSE
  MOV ax,addr
 ENDIF
ENDM
:
```

You could invoke this example with `load test,cx`, which would generate a `mov cx,test` instruction (or invoke simply `load test`, which will generate a `mov ax,test` instruction because the second parameter is blank). Alternately, you could use **ERRIFB** to generate an error for a macro invocation with a missing critical argument. Thus,

```
:
load MACRO addr
 ERRIFB <addr>
 MOV ax,addr
ENDM
:
```

generates an error when invoked with `load`, but would not when invoked with `load test`.

Assembler-pass conditionals

These directives provide conditional assembly or error generation based on the current assembly pass:

IF*xxx* directive	Assembles true_conditional_body if
IF1	Assembler pass 1
IF2	Assembler pass 2

ERR*xxx* directive	Generates user error if
ERRIF1	Assembling pass 1
.ERR1	Assembling pass 1 (MASM mode only)
ERRIF2	Assembling pass 2
.ERR2	Assembling pass 2 (MASM mode only)

Normally, Turbo Assembler acts as a single-pass assembler. If you use Turbo Assembler's multi-pass capability (invoked with the **/m** command-line switch), multiple passes are used if necessary.

Since there is always at least one pass through the assembler, the **IF1** conditional assembly directive will always assemble the code in its conditional block, and the **.ERR1** and **ERRIF1** directives will always generate an error (but only during the first assembly pass).

If you use any of these directives and have not enabled multiple passes, Turbo Assembler will generate `Pass dependent construction` warnings for all of these directives to alert you to a potentially hazardous code omission. If you enable multiple passes, Turbo Assembler will perform exactly two passes, and will generate the warning `Maximum compatibility pass was done.`

Including conditionals in the list file

See Chapters 2 and 17 for further information.

Normally, false conditional assembly code is not included in a listing file. You can override this through the use of assembler directives and command-line switches.

C H A P T E R 16

Interfacing with the linker

Modular programs are typically constructed from several independent sections of code, called modules. The compiler processes each of these modules independently, and the linker (TLINK) puts the resulting pieces together to create an executable file. The README file explains where you can find information about how to use TLINK, but it's also important to know how to define and include all the files and libraries you might want prior to linking. This chapter describes how to do these things.

Publishing symbols externally

You may find that you'll need to use some variables and procedures in all of your program modules. Turbo Assembler provides several directives that let you define symbols and libraries so that you can use them globally, as well as use communal variables (which the linker allocates space for). You'll also have to be careful about how you name your symbols, since different languages have particular requirements. The next few sections discuss these directives and naming requirements.

Conventions for a particular language

When you name symbols that you plan to use externally, remember to use the language specifier for your particular language. These requirements for variable names are:

- Pascal uppercase characters
- C/C++ name must start with _. Rest of name should be in lowercase characters (_name).

When you specify a language in the **MODEL** directive or in the **PROC** declaration, or declare the language in a symbol's **PUBLIC** declaration, Turbo Assembler will automatically use the proper naming conventions for that language, as follows:

- C, CPP, and PROLOG use the C/C++ naming conventions.
- BASIC, PASCAL, FORTRAN, and NOLANGUAGE languages use the Pascal naming conventions.

- SYSCALL specifies C calling conventions, but without prepending underscores to symbol names (like Pascal naming conventions).
- STDCALL uses C calling conventions for procedures with variable arguments, and Pascal calling conventions for procedures with fixed arguments. It always uses the C naming convention.

The **/ml** switch (described in Chapter 2) tells Turbo Assembler to treat all symbol names as case sensitive. The **/mx** switch (also described in Chapter 2) tells the assembler to treat only external and public symbols as case sensitive, and that all other symbols within the source file are uppercase. When you use these two switches together, they have a special meaning for symbols declared as Pascal: these switches cause the symbols in question to be published as all uppercase to the linker.

Declaring public symbols

When you declare a public symbol, you intend it to be accessible from other modules. The following types of symbols can be public:

- data variable names
- program labels
- numeric constants defined with **EQU**

You can use the **PUBLIC** directive to define public symbols. Its syntax follows:

```
PUBLIC [language] symbol [,[language] symbol] ...
```

Note that to use public symbols outside the module where they're defined, you need to use the **EXTRN** directive.

language is either C, CPP, PASCAL, BASIC, FORTRAN, PROLOG, or NOLANGUAGE, and defines any language-specific conventions to be applied to the symbol name. Using a language in the **PUBLIC** directive temporarily overrides the current language setting (the default, **NOLANGUAGE**, or one that you've established with **.MODEL**).

Turbo Assembler publishes *symbol* in the object file so that other modules can access it. If you don't make a symbol public, you can access it only from the current source file; for example:

```
PUBLIC XYPROC           ;make procedure public
XYPROC PROC NEAR
```

Declaring library symbols

You can also use symbols as dynamic link entry points for a dynamic link library. Use the **PUBLICDLL** directive to declare symbols to be accessible this way. Here's its syntax:

```
PUBLICDLL [language] symbol [,[language] symbol] ...
```

Turbo Assembler publishes *symbol* in the object file as a dynamic link entry point (using **EXPDEF** and **IMPDEF** records) so that it can be accessed by other programs. *language* causes any language-specific conventions to be applied to the symbol name. Valid language specifiers are C, PASCAL, BASIC, FORTRAN, PROLOG, and NOLANGUAGE.

Here's an example of code using **PUBLICDLL**:

```
PUBLICDLL XYPROC              ;make procedure XYPROC
XYPROC PROC NEAR              ;accessible as dynamic link entry point
```

Defining external symbols

External symbols are symbols that are defined outside a module, that you can use within the module. These symbols must have been declared using the **PUBLIC** directive. **EXTRN** has the following syntax:

```
EXTRN definition [,definition] ...
```

definition describes a symbol and has the following format:

```
[language] name [[count1]] :complex_type [:count2]
```

Defining global symbols

Global symbols function like public symbols, without your having to specify a **PUBLIC** or an **EXTRN**. If the variable is defined in the module, it functions like **PUBLIC**. If not, it functions like **EXTRN**. You can use the **GLOBAL** directive to define global symbols. **GLOBAL** has the same syntax as **PUBLIC** and **EXTRN** (see the previous few sections for syntax descriptions.)

GLOBAL lets you have an **INCLUDE** file included by all source files; the **INCLUDE** file contains all shared data defined as global symbols. When you reference these data items in each module, the **GLOBAL** definition acts as an **EXTRN** directive, describing how the data is defined in another module.

You must define a symbol as **GLOBAL** before you first use it elsewhere in your source file. Also note that each argument of **GLOBAL** accepts the same syntax as an argument of **EXTRN**.

Here's an example:

```
GLOBAL X:WORD, Y:BYTE
X DW 0                        ;made public for other module
mov al, Y                     ;Y is defined as external
```

Publishing a procedure prototype

If you're using version T320 or later and you use **PROCDESC** to describe a procedure prototype, Turbo Assembler treats the procedure name as if it were a GLOBAL symbol. If you've defined the procedure within the module, it is treated as **PUBLIC**. Otherwise, Turbo Assembler assumes it to be **EXTRN**.

You can place **PROCDESC** directives in an include file. When you reference the procedure name in the module, **PROCDESC** acts as an **EXTRN** directive, describing how the procedure is defined in another module. If the procedure is defined in the module, **PROCDESC** acts as a **PUBLIC** directive to publish the procedure.

Defining communal variables

Communal variables function like external variables, with a major difference: communal variables are allocated by the linker. Communal variables are actually like global variables, but you can't assign them initial values. These uninitialized variables can be referenced from multiple modules.

A drawback to using communal variables is that there's no guarantee that they'll appear in consecutive memory locations. If this is an issue for you, use global variables instead.

You can use the **COMM** directive to define a communal variable. Here's its syntax:

```
COMM definition [,definition]...
```

Each *definition* describes a symbol and has the following format:

```
[distance] [language] symbolname[[count1]]:complex_type [:count2]
```

distance is optional and can be either **NEAR** or **FAR**. If you don't specify a *distance*, it will default to the size of the default data memory model. If you're not using the simplified segmentation directives, the default size is **NEAR**. With the tiny, small, and medium models, the default size is also **NEAR**; all other models are **FAR**.

language is either **C, PASCAL, BASIC, FORTRAN, PROLOG**, or **NOLANGUAGE**. Using a language in the **COMM** directive temporarily overrides the current language setting (default or one established with **.MODEL**). Note that you don't need to have a **.MODEL** directive in effect to use this feature.

symbolname is the symbol that is to be communal and have storage allocated at link time. *symbolname* can also specify an array element size multiplier *count1* to be included in the total space computation. If *distance* is **NEAR**, the linker uses *count1* to calculate the total size of the array. If *distance* is **FAR**, the linker uses count2 to indicate how many elements there are of size *count1* times the basic element size (determined by *type*). *count1* defaults to a value of 1.

complex_type is the data type of the argument. It can be either a simple type, or a complex pointer expression. See Chapter 5 for more information about the syntax of complex types.

The optional *count2* specifies how many items this communal symbol defines. If you do not specify a count2, a value of 1 is assumed. The total space allocated for the communal variable is *count2* times the length specified by the *type* field times *count1*.

In MASM mode, communal symbols declared outside of any segment are presumed to be reachable using the DS register, which may not always be a valid assumption. Make sure that you either place the correct segment value in DS or use an explicit segment override when referring to these variables. In Ideal mode, Turbo Assembler correctly checks for whether the communal variable is addressable, using any of the current segment registers as described with the **ASSUME** directive.

Here's an example using the **COMM** directive.

```
COMM buffer:BYTE:512          ;512 bytes allocated at link time
COMM abc[41]:WORD:10          ;820 bytes (10 items of 41 words
                              ;each) allocated at link time
COMM FAR abc[41]:WORD:10      ;10 elements of 82 bytes (2 bytes
                              ;times 41 elements) allocated at
                              ;link time
```

Including a library

Using **INCLUDELIB** prevents you from having to remember to specify the library name in the linker commands.

For the times when you know that your source file will always need to use routines in a specified library, you can use the **INCLUDELIB** directive. **INCLUDELIB** tells the linker to include a particular library. The appropriate syntaxes for this directive are:

Ideal mode:

```
INCLUDELIB "filename"       ;note the quotes!
```

MASM mode:

```
INCLUDELIB filename
```

filename is the name of the library you want the linker to include at link time. If you don't supply an extension with *filename*, the linker assumes .LIB.

Here's an example:

```
INCLUDELIB "diskio"        ;includes DISKIO.LIB
```

The ALIAS directive

Turbo Assembler supports **ALIAS** to allow the association of an alias name with a *substitute* name. When the linker encounters an alias name, it resolves the alias by referring to the substitute name.

C H A P T E R 17

Generating a listing

See the /l and /la switches in Chapter 2.

A listing file is useful if you want to see exactly what Turbo Assembler generates when each instruction or directive is assembled. The file is basically the source file annotated with a variety of information about the results of the assembly. Turbo Assembler lists the actual machine code for each instruction, along with the offset in the current segment of the machine code for each line. What's more, Turbo Assembler provides tables of information about the labels and segments used in the program, including the value and type of each label, and the attributes of each segment.

See the **/c** command-line option in Chapter 2.

Turbo Assembler can also, on demand, generate a cross-reference table for all labels used in a source file, showing you where each label was defined and where it was referenced.

Listing format

The top of each page of the listing file displays a header consisting of the version of Turbo Assembler that assembled the file, the date and time of assembly, and the page number within the listing.

There are two parts to the listing file: the annotated source code listing and the symbol tables. The original assembly code is displayed first, with a header containing the name of the file where the source code resides. The assembler source code is annotated with information about the machine code Turbo Assembler assembled from it. Any errors or warnings encountered during assembly are inserted immediately following the line they occurred on.

The code lines in the listing file follow this format:

```
<depth> <line number> <offset> <machine code> <source>
```

<depth> indicates the level of nesting of Include files and macros within your listing file.

<line number> is the number of the line in the listing file (not including header and title lines). Line numbers are particularly useful when the cross-reference feature of Turbo Assembler, which refers to lines by line number, is used. Be aware that the line numbers in *<line number>* are not the source module line numbers. For example, if a macro is expanded or a file is included, the line-number field will continue to advance, even though the current line in the source module stays the same. To translate a line number (for example, one that the cross-referencer produced) back to the source file, you must look up the line number in the listing file, and then find that same line (by eye, not by number) in the source file.

<offset> is the offset in the current segment of the start of the machine code generated by the associated assembler source line.

<machine code> is the actual sequence of hexadecimal byte and word values that is assembled from the associated assembler source line.

<source> is simply the original assembler line, comments and all. Some assembler lines, such as those that contain only comments, don't generate any machine code; these lines have no *<offset>* or *<machine code>* fields, but do have a line number.

General list directives

There are a variety of list directives that let you control what you want in your listing file. The general list directives follow:

- **.LIST** ;MASM mode only
- **.XLIST** ;MASM mode only
- **%LIST**
- **%NOLIST**
- **%CTLS**
- **%NOCTLS**
- **%SYMS**
- **%NOSYMS**

The **%LIST** directive shows all of the source lines in your listing. This is the default condition when you create a listing file. To turn off the display of all the source lines, use the **%NOLIST** directive. Here's an example:

```
%NOLIST                         ;turn off listing
INCLUDE MORE .INC
%LIST                           ;turn on listing
```

The **.LIST** and **.XLIST** directives function the same way as **%LIST** and **%NOLIST**. Here's an example:

```
.LIST
jmp xyz             ;this line always listed
.XLIST
add dx,ByteVar      ;not in listing
```

You can use the **%CTLS** and **%NOCTLS** directives to control the listing directives. **%CTLS** causes listing control directives (such as **%LIST**, **%INCL**, and so on) to be placed in the listing file; normally, they are not listed. It takes effect on all subsequent lines, so the **%CTLS** directive itself will not appear in the listing file. **%NOCTLS** reverses the effect of a previous **%CTLS** directive. After issuing **%NOCTLS**, all subsequent listing-control directives will not appear in the listing file. (**%NOCTLS** is the default listing-control mode that Turbo Assembler uses when it starts assembling a source file.); for example,

```
%CTLS
%NOLIST                       ;this will be in listing file
%NOCTLS
%LIST                         ;this will not appear in listing
```

You can use the **%SYMS** and **%NOSYMS** directives to cause the symbol table to either appear or not to appear in your listing file (the default is for it to appear). The symbol table will appear at the end of the listing file.

Here's the syntax for **%SYMS**:

```
%SYMS
```

Here's the syntax for **%NOSYMS**:

```
%NOSYMS
```

Include file list directives

In the event that you might want to list the include files in your listing file, you can turn this capability on and off using the **%INCL** and **%NOINCL** directives. By default, INCLUDE files are normally contained in the listing file. **%NOINCL** stops all subsequent INCLUDE files source lines from appearing in the listing until a **%INCL** is enabled. This is useful if you have a large INCLUDE file that contains things such as a lot of **EQU** definitions that never change.

Here's an example:

```
%INCL
INCLUDE DEFS.INC            ;contents appear in listing
%NOINCL
INCLUDE DEF1.INC            ;contents don't appear
```

Conditional list directives

When you have conditional blocks of code in your source files, you might not want all of that information to appear in the listing file. Showing conditional blocks can be very helpful in some instances when you want to see exactly how your code is behaving. Turbo Assembler provides the following conditional list directives:

- **.LFCOND** ;MASM mode only
- **.SFCOND** ;MASM mode only
- **.TFCOND** ;MASM mode only
- **%CONDS**
- **%NOCONDS**

Turbo Assembler does not usually list conditional blocks.

The **%CONDS** directive displays all statements in conditional blocks in the listing file. This includes the listing of false conditional blocks in assembly listings. The **.LFCOND** directive functions the same as **%COND.** **%NOCONDS** prevents statements in false conditional blocks from appearing in the listing file. The **.SFCONDS** directive functions exactly the same as **%NOCOND**. If you want to toggle conditional block-listing mode, use the **.TFCOND** directive.

The first **.TFCOND** that Turbo Assembler encounters enables a listing of conditional blocks. If you use the **/X** command-line option, conditional blocks start off being listed, and the first **.TFCOND** encountered disables listing them. Each time **.TFCOND** appears in the source file, the state of false conditional listings is reversed.

To invoke any of these directives, place it by itself on a line in your code. They will affect the conditional blocks that immediately follow them.

Macro list directives

Macro expansions are not normally included in listing files. Having this information in listing files can be very helpful when you want to see what your code is doing. Turbo Assembler provides several directives that let turn this feature on and off. They are:

- **.LALL** ;MASM mode only
- **.SALL** ;MASM mode only
- **.XALL** ;MASM mode only
- **%MACS**
- **%NOMACS**

The **%MACS** directive enables the listing of macro expansions. The **.LALL** directive does the same thing, but only works in MASM mode. You can use these macros to toggle macro expansion in listings on.

%MACS has the following syntax:

```
%MACS
```

You can specify **.LALL** as follows:

```
.LALL
```

If you want to suppress the listing of all statements in macro expansions, use either the **%NOMACS** or **.SALL** directives. Note that you can use these directives to toggle macro expansion in listings off.

%NOMACS has the following syntax:

```
%NOMACS
```

You can specify **.SALL** as follows:

```
.SALL
```

The **.XALL** directive, which is only available in MASM mode, lets you list only the macro expansions that generate code or data. .XALL has the following syntax:

```
.XALL
```

Cross-reference list directives

The symbol table portion of the listing file normally tells you a great deal about labels, groups, and segments, but there are two things it doesn't tell you: where labels, groups, and segments are defined, and where they're used. Cross-referenced symbol information makes it easier to find labels and follow program execution when debugging a program.

There are several ways of enabling cross-referencing information in your listing file. You can use **/c** to produce cross-referencing information for an entire file (see Chapter 2 for details), or you can include directives in your code that let you enable and disable cross-referencing in selected portions of your listings.These directives are:

- **.CREF** ;MASM mode only
- **.XCREF** ;MASM mode only
- **%CREF**
- **%NOCREF**
- **%CREFALL**
- **%CREFREF**
- **%CREFUREF**

Turbo Assembler includes cross referencing information in the listing file. You can specify a .XRF in your Turbo Assembler command to get a separate .XRF file as well.

The **%CREF** and **.CREF** directives let you accumulate cross-reference information for all symbols encountered from that point forward in the source file. **%CREF** and **.CREF** reverse the effects of any **%NOCREF** or **.XCREF** directives, which inhibit the collection of cross-reference information.

%CREF and **.CREF** have the following syntaxes:

```
%CREF
```

or

```
.CREF
```

%NOCREF and **.XCREF** have the following syntaxes:

```
%NOCREF [symbol, ...]
```

or

```
.XCREF [symbol, ...]
```

If you use **%NOCREF** or **.XCREF** alone without specifying any *symbols*, cross-referencing is disabled completely. If you supply one or more symbol names, cross-referencing is disabled only for those symbols.

The **%CREFALL** directive lists all symbols in the cross reference. **%CREFALL** reverses the effect of any previous **%CREFREF** (which disables listing of unreferenced symbols in the cross reference), or **%CREFUREF** (which lists only the unreferenced symbols in the cross reference). After issuing **%CREFALL**, all subsequent symbols in the source file will appear in the cross-reference listing. This is the default mode that Turbo Assembler uses when assembling your source file.

The syntax for **%CREFALL**, **%CREFREF**, and **%CREFUREF** follows:

```
%CREFALL
```

```
%CREFREF
```

```
%CREFUREF
```

Changing list format parameters

The listing format control directives alter the format of the listing file. You can use these directives to tailor the appearance of the listing file to your tastes and needs.

The **PAGE** directive sets the listing page height and width, and starts new pages. **PAGE** only works in MASM mode. **PAGE** has the following syntax:

```
PAGE [rows] [,cols]
PAGE +
```

rows specifies the number of lines that will appear on each listing page. The minimum is 10 and the maximum is 255. *cols* specifies the number of columns wide the page will be. The minimum width is 59; the maximum is 255. If you omit either *rows* or *cols*, the current setting for that parameter will remain unchanged. To change only the number of columns, precede the column width with a comma; otherwise, you'll end up changing the number of rows instead.

If you follow the **PAGE** directive with a plus sign (+), a new page starts, the section number is incremented, and the page number restarts at 1. If you use **PAGE** with no arguments, the listing resumes on a new page, with no change in section number.

The **%PAGESIZE** directive functions exactly like the **PAGE** directive, except that it doesn't start a new page and that it works in both MASM and Ideal modes. **%PAGESIZE** has the following syntax:

```
%PAGESIZE [rows] [,cols]
```

%NEWPAGE functions like **PAGE**, with no arguments. Source lines appearing after **%NEWPAGE** will begin at the start of a new page in the listing file. **%NEWPAGE** has the following syntax:

```
%NEWPAGE
```

The **%BIN** directive sets the width of the object code field in the listing file. **%BIN** has the following syntax:

```
%BIN size
```

size is a constant. If you don't use this directive, the instruction opcode field takes up 20 columns in the listing file. For example,

```
%BIN 12         ;set listing width to 12 columns
```

%DEPTH sets the size of the depth field in the listing file. **%DEPTH** has the following syntax:

```
%DEPTH width
```

width specifies how many columns to reserve for the nesting depth field in the listing file. The depth field indicates the nesting level for INCLUDE files and macro expansions. If you specify a width of 0, this field does not appear in the listing file. Usually, you won't need to specify a width of more than 2, since that would display a depth of up to 99 without truncation. The default width for this field is 1 column.

%LINUM sets the width of the line-number field in the listing file. **%LINUM** has the following syntax:

```
%LINUM size
```

%LINUM lets you set how many columns the line numbers take up in the listing file. *size* must be a constant. If you want to make your listing as narrow as possible, you can reduce the width of this field. Also, if your source file contains more than 9,999 lines, you can increase the width of this field so that the line numbers are not truncated. The default width for this field is 4 columns.

%TRUNC truncates listing fields that are too long. **%TRUNC** has the following syntax:

```
%TRUNC
```

The object code field of the listing file has enough room to show the code emitted for most instructions and data allocations. You can adjust the width of this field with **%BIN**. If a single source line emits more code than can be displayed on a single line, the rest is normally truncated and therefore not visible. When you want to see all the code generated, use **%NOTRUNC** (which wordwraps too-long fields in the listing file). Otherwise, use **%TRUNC**. You can use these directives to toggle truncation on and off.

%NOTRUNC has the following syntax:

```
%NOTRUNC
```

%PCNT sets the segment:offset field width in the listing file. **%PCNT** has the following syntax:

```
%PCNT width
```

where *width* is the number of columns you want to reserve for the offset within the current segment being assembled. Turbo Assembler sets the width to 4 for ordinary 16-bit segments and sets it to 8 for 32-bit segments used by the 386 processor. **%PCNT** overrides these defaults.

The **TITLE** directive, which you can use only in MASM mode, sets the title in the listing file. **TITLE** has the following syntax:

```
TITLE text
```

The title *text* appears at the top of each page, after the name of the source file and before any subtitle set with the **SUBTTL** directive. You can use **TITLE** as many times as you want.

%TITLE functions like **TITLE**, but you can use it for either MASM or Ideal mode. **%TITLE** has the following syntax:

```
%TITLE "text"
```

SUBTTL, which only works in MASM mode, sets the subtitle in the listing file. **SUBTTL** has the following syntax:

```
SUBTTL text
```

The subtitle appears at the top of each page, after the name of the source file, and after any title set with **TITLE**.

You can place as many **SUBTTL** directives in your program as you wish. Each directive changes the subtitle that will appear at the top of the next listing page.

%SUBTTL functions like **SUBTTL**, but it works in both MASM and Ideal modes. **%SUBTTL** has the following syntax:

```
%SUBTTL "text"
```

%TABSIZE sets the tab column width in the listing file. **%TABSIZE** has the following syntax:

```
%TABSIZE width
```

width is the number of columns between tabs in the listing file. The default tab column width is 8 columns.

You can use the **%TEXT** directive to set the width of the source field in the listing file. It has the following syntax:

```
%TEXT width
```

width is the number of columns to use for source lines in the listing file. If the source line is longer than this field, it will either be truncated or wrapped to the following line, depending on whether you've used **%TRUNC** or **%NOTRUNC**.

You can use the **%PUSHLCTL** directive to save the listing controls on a 16-level stack. It only saves the listing controls that can be enabled or disabled (**%INCL**, **%NOINCL**, and so on). The listing field widths are not saved. This directive is particularly useful in macros, where you can invoke special listing modes that disappear once the macro expansion terminates.

%PUSHLCTL has the following syntax:

```
%PUSHLCTL
```

Conversely, the **%POPLCTL** directive recalls listing controls from the stack. Here's its syntax:

```
%POPLCTL
```

%POPLCTL resets the listing controls to the way they were when the last **%PUSHLCTL** directive was issued. None of the listing controls that set field width are restored (such as **%DEPTH**, **%PCNT**).

C H A P T E R 18

Interfacing Turbo Assembler with Borland C++

While many programmers can—and do—develop entire programs in assembly language, many others prefer to do the bulk of their programming in a high-level language, dipping into assembly language only when low-level control or very high-performance code is required. Still others prefer to program primarily in assembler, taking occasional advantage of high-level language libraries and constructs.

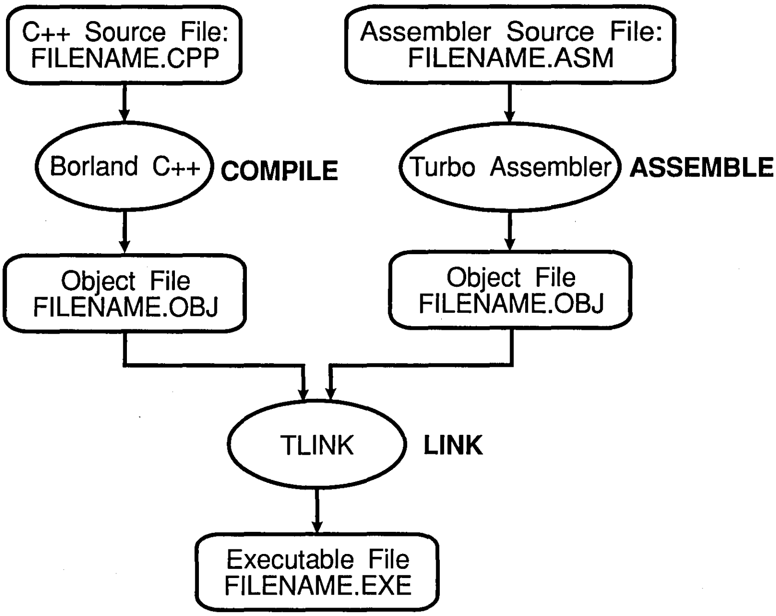
Borland C++ lends itself particularly well to supporting mixed C++ and assembler code on an as-needed basis, providing not one but three mechanisms for integrating assembler and C++ code. The inline assembly feature of Borland C++ provides a quick and simple way to put assembler code directly into a C++ function. You can assemble the inline code with Turbo Assembler or use Borland C++'s built-in assembler. For further information about using in-line assembly in Borland C++ or the built-in assembler, see the *Borland C++ Programmer's Guide*. For those who prefer to do their assembler programming in separate modules written entirely in assembly language, Turbo Assembler modules can be assembled separately and linked to Borland C++ code.

First, we'll discuss the details of linking separately assembled Turbo Assembler modules to Borland C++, and explore the process of calling Turbo Assembler functions from Borland C++ code. Then, we'll cover calling Borland C++ functions from Turbo Assembler code.

Calling Turbo Assembler functions from Borland C++

C++ and assembler have traditionally been mixed by writing separate modules entirely in C++ or assembler, compiling the C++ modules and assembling the assembler modules, and then linking the separately compiled modules together. Borland C++ modules can readily be linked with Turbo Assembler modules in this fashion. Figure 18.1 shows how to do this.

Figure 18.1
Compile, assemble, and link with Borland C++, Turbo Assembler, and TLINK



The executable file is produced from mixed C++ and assembler source files. You start this cycle with

```
bcc filenam1.cpp filenam2.asm
```

This instructs Borland C++ to first compile FILENAM1.CPP to FILENAM1.OBJ, then invoke Turbo Assembler to assemble FILENAM2.ASM to FILENAM2.OBJ, and finally invoke TLINK to link FILENAM1.OBJ and FILENAM2.OBJ into FILENAM1.EXE.

Separate compilation is very useful for programs that have sizable amounts of assembler code, since it makes the full power of Turbo Assembler available and allows you to do your assembly language programming in a pure assembler environment, without the **asm** keywords, extra compilation time, and C++-related overhead of inline assembly.

There is a price to be paid for separate compilation: The assembler programmer must attend to all the details of interfacing C++ and assembler code. Where Borland C++ handles segment specification, parameter-passing, reference to C++ variables, register variable preservation, and the like for inline assembly, separately compiled assembler functions must explicitly do all that and more.

There are two major aspects to interfacing Borland C++ and Turbo Assembler. First, the various parts of the C++ and assembler code must be linked together properly, and functions and variables in each part of the code must be made available to the rest of the code as needed. Second, the assembler code must properly handle C-style function calls. This includes accessing passed parameters, returning values, and following the register preservation rules required of C++ functions.

Let's start by examining the rules for linking together Borland C++ and Turbo Assembler code.

The framework

In order to link Borland C++ and Turbo Assembler modules together, three things must happen:

- The Turbo Assembler modules must use a Borland C++-compatible segment-naming scheme.
- The Borland C++ and Turbo Assembler modules must share appropriate function and variable names in a form acceptable to Borland C++.
- TLINK must be used to combine the modules into an executable program.

This says nothing about what the Turbo Assembler modules actually *do*; at this point, we're only concerned with creating a framework within which C++-compatible Turbo Assembler functions can be written.

Linking assembly language modules with C++

Type-safe linkage is an important concept in C++. The compiler and linker must work together to ensure function calls between modules use the correct argument types. A process called *name-mangling* provides the necessary argument type information. Name-mangling modifies the name of the function to indicate what arguments the function takes.

When you build a program entirely in C++, name-mangling occurs automatically and transparently. However, when you write a module in assembly language to be linked into a C++ program, you must be sure the assembler module contains mangled names. You can do this easily by writing a dummy function in C++ and compiling it to assembler. The .ASM file that Borland C++ generates will have the proper mangled names. You use these names when you write the real assembler module.

For example, the following code fragment defines four different versions of the function named test:

```
void test()
{
}
```

```
void test( int )
{
}

void test( int, int )
{
}

void test( float, double )
{
}
```

If the code is compiled using the **–S** option, the compiler produces an assembly language output file (.ASM). This is how the output looks (edited to remove extraneous details):

```
;       void test()
@test$qv        proc    near
        push    bp
        mov     bp,sp
        pop     bp
        ret
@test$qv        endp

;       void test( int )
@test$qi        proc    near
        push    bp
        mov     bp,sp
        pop     bp
        ret
@test$qi        endp

;       void test( int, int )
@test$qii       proc    near
        push    bp
        mov     bp,sp
        pop     bp
        ret
@test$qii       endp

;       void test( float, double )
@test$qfd       proc    near
        push    bp
        mov     bp,sp
        pop     bp
        ret
@test$qfd       endp
```

Using Extern "C" to simplify linkage

If you prefer, you can use unmangled names for your assembler functions, instead of trying to figure out what the mangled names would be. Using unmangled names will protect your assembler functions from possible future changes in the name-mangling algorithm. Borland C++ allows you to define standard C function names in your C++ programs.

Look at this example:

```
extern "C" {
     int add(int *a,int b);
}
```

Any functions declared within the braces will be given C style names. Here is the matching assembler procedure definition.

```
     public _add
_add proc
```

Declaring an assembler function with an extern "C" block can save you the trouble of determining what the mangled names will be. Your code will be more readable, also.

Memory models and segments

For a given assembler function to be callable from C++, that function must use the same memory model as the C++ program and must use a C++-compatible code segment. Likewise, in order for data defined in an assembler module to be accessed by C++ code (or for C++ data to be accessed by assembler code), the assembler code must follow C++ data segment-naming conventions.

Memory models and segment handling can be quite complex to implement in assembler. Fortunately, Turbo Assembler does virtually all the work of implementing Borland C++-compatible memory models and segments for you in the form of the simplified segment directives.

Simplified segment directives and Borland C++

The **.MODEL** directive tells Turbo Assembler that segments created with the simplified segment directives should be compatible with the selected memory model (tiny, small, compact, medium, large, huge, or tchuge), and controls the default type (near or far) of procedures created with the **PROC** directive. Memory models defined with the **.MODEL** directive are compatible with the equivalently named Borland C++ models except that you should use Turbo Assembler's **tchuge** memory model when you want to support Borland C++'s huge memory model. (The **huge** memory model is more appropriate for compatibility with other C compilers.) You should use the FARSTACK modifier with the **.MODEL** directive for large model, so the stack does not become a part of DGROUP.

Finally, the **.CODE**, **.DATA**, **.DATA?**, **.FARDATA**, and **.FARDATA?** simplified segment directives generate Borland C++-compatible segments. (Don't use **.DATA?** or **FARDATA?** in huge model as they do not exist in Borland C++.)

For example, consider the following Turbo Assembler module, named DOTOTAL.ASM:

```
; select Intel-convention segment ordering
       .MODEL  small    ;select small model (near code and data)
       .DATA            ;TC-compatible initialized data segment
       EXTRN   _Repetitions:WORD    ;externally defined
       PUBLIC  _StartingValue       ;available to other modules
_StartingValue  DW  0
       .DATA?           ;TC-compatible uninitialized data segment
RunningTotal    DW  ?
       .CODE            ;TC-compatible code segment
       PUBLIC  _DoTotal
_DoTotal        PROC    ;function (near-callable in small model)
       mov     cx,[_Repetitions]    ;# of counts to do
       mov     ax,[_StartingValue]
       mov     [RunningTotal],ax    ;set initial value
TotalLoop:
       inc     [RunningTotal]       ;RunningTotal++
       loop    TotalLoop
       mov     ax,[RunningTotal]    ;return final total
       ret
_DoTotal        ENDP
       END
```

The assembler procedure *_DoTotal* is readily callable from a small-model Borland C++ program with the statement

```
DoTotal();
```

Note that *_DoTotal* expects some other part of the program to define the external variable *Repetitions*. Similarly, the variable *StartingValue* is made public, so other portions of the program can access it. The following Borland C++ module, SHOWTOT.CPP, accesses public data in DOTOTAL.ASM and provides external data to DOTOTAL.ASM:

```
#include <stdio.h>

extern "C" int DoTotal(void);
extern int StartingValue;

int Repetitions;
```

```
int main()
{
   Repetitions = 10;
   StartingValue = 2;
   printf("%d\n", DoTotal());
   return 0;
}
```

StartingValue doesn't have to go in the Extern "C" block because variable names are not mangled.

To create the executable program SHOWTOT.EXE from SHOWTOT.CPP and DOTOTAL.ASM, enter the command line

```
bcc showtot.cpp dototal.asm
```

If you wanted to link *_DoTotal* to a compact-model C++ program, you would simply change the **.MODEL** directive to **.MODEL** *COMPACT*. If you wanted to use a far segment in DOTOTAL.ASM, you could use the **.FAR-DATA** directive.

In short, generating the correct segment ordering, memory model, and segment names for linking with Borland C++ is easy with the simplified segment directives.

Old-style segment directives and Borland C++

Simply put, it's a nuisance interfacing Turbo Assembler code to C++ code using the old-style segment directives. For example, if you replace the simplified segment directives in DOTOTAL.ASM with old-style segment directives, you get

```
DGROUP  GROUP    _DATA,_BSS
_DATA   SEGMENT  WORD PUBLIC 'DATA'
        EXTRN    _Repetitions:WORD    ;externally defined
        PUBLIC   _StartingValue       ;available to other modules
_StartingValue    DW 0
_DATA   ENDS
_BSS    SEGMENT   WORD PUBLIC 'BSS'
RunningTotal      DW  ?
_BSS    ENDS
_TEXT   SEGMENT   BYTE PUBLIC 'CODE'
        ASSUME    cs:_TEXT,ds:DGROUP,ss:DGROUP
        PUBLIC    _DoTotal
_DoTotal          PROC                ;function (near-callable
                                      ; in small model)
        mov       cx,[_Repetitions]   ;# of counts to do
        mov       ax,[_StartingValue]
        mov       [RunningTotal],ax   ;set initial value
```

```
TotalLoop:
        inc     [RunningTotal]          ;RunningTotal++
        loop    TotalLoop
        mov     ax,[RunningTotal]       ;return final total
        ret
_DoTotal ENDP
_TEXT   ENDS
        END
```

The version with old-style segment directives is not only longer, but also much harder to read and harder to change to match a different C++ memory model. When you're interfacing to Borland C++, there's generally no advantage to using the old-style segment directives. If you still want to use the old-style segment directives when interfacing to Borland C++, you'll have to identify the correct segments for the memory model your C++ code uses.

The easy way to determine the appropriate old-style segment directives for linking with a given Borland C++ program is to compile the main module of the Borland C++ program in the desired memory model with the **–S** option. This causes Borland C++ to generate an assembler version of the C++ code. In that C++ code, you'll find all the old-style segment directives used by Borland C++; just copy them into your assembler code.

You can also find out what the appropriate old-style directives are by compiling as you normally would (without the **–S** option) and then using TDUMP, a utility that comes with Turbo Assembler, to display all the segment definition records. Use this command line:

```
tdump -OIsegdef module.obj
```

Segment defaults: When is it necessary to load segments?

Under some circumstances, your C++-callable assembler functions might have to load DS and/or ES in order to access data. It's also useful to know the relationships between the settings of the segment registers on a call from Borland C++, since sometimes assembler code can take advantage of the equivalence of two segment registers. Let's take a moment to examine the settings of the segment registers when an assembler function is called from Borland C++, the relationships between the segment registers, and the cases in which an assembler function might need to load one or more segment registers.

On entry to an assembler function from Borland C++, the CS and DS registers have the following settings, depending on the memory model in use (SS is always used for the stack segment, and ES is always used as a scratch segment register):

Table 18.1
Register settings when Borland C++ enters assembler

Model	CS	DS
Tiny	_TEXT	DGROUP
Small	_TEXT	DGROUP
Compact	_TEXT	DGROUP
Medium	filename_TEXT	DGROUP
Large	filename_TEXT	DGROUP
Huge	filename_TEXT	calling_filename_DATA

filename is the name of the assembler module, and *calling_filename* is the name of the module calling the assembler module.

In the tiny model, **_TEXT** and **DGROUP** are the same, so CS equals DS on entry to functions. Also in the tiny, small, and medium models, SS equals DS on entry to functions.

So, when is it necessary to load a segment register in a C++-callable assembler function? First, you should never have to (or want to) directly load the CS or SS registers. CS is automatically set as needed on far calls, jumps, and returns, and can't be tampered with otherwise. SS always points to the stack segment, which should never change during the course of a program (unless you're writing code that switches stacks, in which case you had best know *exactly* what you're doing).

ES is always available for you to use as you wish. You can use ES to point at far data, or you can load ES with the destination segment for a string instruction.

That leaves the DS register; in all Borland C++ models other than the huge model, DS points to the static data segment (**DGROUP**) on entry to functions, and that's generally where you'll want to leave it. You can always use ES to access far data, although you may find it desirable to instead temporarily point DS to far data that you're going to access intensively, thereby saving many segment override instructions in your code. For example, you could access a far segment in either of the following ways:

```
        :
        .FARDATA
Counter DW      0
        :
        .CODE
        PUBLIC  _AsmFunction
```

```
_AsmFunction     PROC
          :
          mov     ax,@fardata
          mov     es,ax              ;point ES to far data segment
          inc     es:[Counter]       ;increment counter variable
          :
_AsmFunction     ENDP
          :
```

or

```
          :
          .FARDATA
Counter   DW      0
          :
          .CODE
          PUBLIC  _AsmFunction
_AsmFunction     PROC
          :
          ASSUME  ds:@fardata
          mov     ax,@fardata
          mov     ds,ax              ;point DS to far data segment
          inc      [Counter]         ;increment counter variable
          ASSUME   ds:@data
          mov      ax,@data
          mov      ds,ax             ;point DS back to DGROUP
          :
_AsmFunction     ENDP
          :
```

The second version has the advantage of not requiring an ES: override on each memory access to the far data segment. If you do load DS to point to a far segment, be sure to restore it like in the preceding example before attempting to access any variables in **DGROUP**. Even if you don't access **DGROUP** in a given assembler function, be sure to restore DS before exiting since Borland C++ assumes that functions leave DS unchanged.

Handling DS in C++-callable huge model functions is a bit different. In the huge model, Borland C++ doesn't use **DGROUP** at all. Instead, each module has its own data segment, which is a far segment relative to all the other modules in the program; there is no commonly shared near data segment. On entry to a function in the huge model, DS should be set to point to that module's far segment and left there for the remainder of the function, as follows:

```
        :
        .FARDATA
        :
        .CODE
        PUBLIC  _AsmFunction
_AsmFunction    PROC
        push    ds
        mov     ax,@fardata
        mov     ds,ax
        :
        pop     ds
        ret
_AsmFunction    ENDP
        :
```

Note that the original state of DS is preserved with a **PUSH** on entry to *AsmFunction* and restored with a **POP** before exiting; even in the huge model, Borland C++ requires all functions to preserve DS.

Publics and externals

Turbo Assembler code can call C++ functions and reference external C++ variables. Borland C++ code can likewise call public Turbo Assembler functions and reference public Turbo Assembler variables. Once Borland C++-compatible segments are set up in Turbo Assembler, as described in the preceding sections, only the following few simple rules are necessary to share functions and variables between Borland C++ and Turbo Assembler.

Underscores and the C language

If you are programming in C or C++, all external labels should start with an underscore character (_). The C and C++ compilers automatically prefix an underscore to all function and external variable names when they're used in C/C++ code, so you only need to attend to underscores in your assembler code. You must be sure that all assembler references to C and C++ functions and variables begin with underscores, and you must begin all assembler functions and variables that are made public and referenced by C/C++ code with underscores.

For example, the following C code (link2asm.cpp),

```
int ToggleFlag();
int Flag;
main()
{
   ToggleFlag();
}
```

links properly with the following assembler program (CASMLINK.ASM):

Labels not referenced by C code, such as *SetTheFlag*, don't need leading underscores.

```
        .MODEL  small
        .DATA
        EXTRN   _Flag:WORD
        .CODE
        PUBLIC  _ToggleFlag
_ToggleFlag     PROC
        cmp     [_Flag],0               ;is the flag reset?
        jz      SetTheFlag              ;yes, set it
        mov     [_Flag],0               ;no, reset it
        jmp     short EndToggleFlag     ;done
SetTheFlag:
        mov     [_Flag],1               ;set flag
EndToggleFlag:
        ret
_ToggleFlag     ENDP
        END
```

When you use the C language specifier in your **EXTRN** and **PUBLIC** directives, as in the following program (CSPEC.ASM),

```
     .MODEL    small
     .DATA
     EXTRN     C Flag:word
     .CODE
     PUBLIC    C ToggleFlag
ToggleFlag     PROC
     cmp       [Flag],0
     jz        SetTheFlag
     mov       [Flag],0
     jmp       short EndToggleFlag
SetTheFlag:
     mov       [Flag],1
EndToggleFlag:
     ret
ToggleFlag     ENDP
     END
```

Turbo Assembler causes the underscores to be prefixed automatically when *Flag* and *ToggleFlag* are published in the object module.

The significance of uppercase and lowercase

Turbo Assembler is normally insensitive to case when handling symbolic names, making no distinction between uppercase and lowercase letters. Since C++ is case-sensitive, it's desirable to have Turbo Assembler be case-sensitive, at least for those symbols that are shared between assembler and C++. **/ml** and **/mx** make this possible.

The **/ml** command-line switch causes Turbo Assembler to become case-sensitive for all symbols. The **/mx** command-line switch causes Turbo Assembler to become case-sensitive for public (**PUBLIC**), external (**EXTRN**), global (**GLOBAL**), and communal (**COMM**) symbols only. When Borland C++ calls Turbo Assembler, it uses the **/ml** switch. Most of the time you should use **/ml** also.

Label types

While assembler programs are free to access any variable as data of any size (8 bit, 16 bit, 32 bit, and so on), it is generally a good idea to access variables in their native size. For instance, it usually causes problems if you write a word to a byte variable:

```
             :
SmallCount  DB  0
             :
             mov  WORD PTR [SmallCount],0ffffh
             :
```

Consequently, it's important that your assembler **EXTRN** statements that declare external C++ variables specify the right size for those variables, since Turbo Assembler has only your declaration to go by when deciding what size access to generate to a C++ variable. Given the statement

```
char c
```

in a C++ program, the assembler code

```
             :
EXTRN   c:WORD
             :
inc     [c]
             :
```

could lead to problems, since every 256th time the assembler code incremented *c*, *c* would turn over. And, since *c* is erroneously declared as a word variable, the byte at **OFFSET** c + 1 is incorrectly incremented, and with unpredictable results.

Correspondence between C++ and assembler data types is as follows:

C++ data type	Assembler data type
unsigned char	byte
char	byte
enum	word
unsigned short	word
short	word

C++ data type	Assembler data type
unsigned int	word
int	word
unsigned long	dword
long	dword
float	dword
double	qword
long double	tbyte
near *	word
far *	dword

Far externals

If you're using the simplified segment directives, **EXTRN** declarations of symbols in far segments must not be placed within any segment, since Turbo Assembler considers symbols declared within a given segment to be associated with that segment. This has its drawbacks: Turbo Assembler cannot check the addressability of symbols declared **EXTRN** outside any segment, and so can neither generate segment overrides as needed nor inform you when you attempt to access that variable when the correct segment is not loaded. Turbo Assembler still assembles the correct code for references to such external symbols, but can no longer provide the normal degree of segment addressability checking.

You can use the old-style segment directives to explicitly declare the segment each external symbol is in, and then place the **EXTRN** directive for that symbol inside the segment declaration. This is a lot of work, however; if you make sure that the correct segment is loaded when you access far data, it's easiest to just put **EXTRN** declarations of far symbols outside all segments. For example, suppose that FILE1.ASM contains

```
   :
          .FARDATA
File1Variable   DB   0
   :
```

Then if FILE1.ASM is linked to FILE2.ASM, which contains

```
   :
       .DATA
       EXTRN  File1Variable:BYTE
       .CODE
Start  PROC
       mov    ax,SEG File1Variable
       mov    ds,ax
   :
```

SEG *File1Variable* will not return the correct segment. The **EXTRN** directive is placed within the scope of the **DATA** directive of FILE2.ASM, so Turbo Assembler considers *File1Variable* to be in the near **DATA** segment of FILE2.ASM rather than in the **FARDATA** segment.

The following code for FILE2.ASM allows **SEG** *File1Variable* to return the correct segment:

```
      :
         .DATA
@curseg  ENDS
         EXTRN   File1Variable:BYTE
         .CODE
Start    PROC
         mov     ax,SEG File1Variable
         mov     ds,ax
      :
```

Here, the **@curseg ENDS** directive ends the **.DATA** segment, so no segment directive is in effect when *File1Variable* is declared external.

Linker command line

The simplest way to link Borland C++ modules with Turbo Assembler modules is to enter a single Borland C++ command line and let Borland C++ do all the work. Given the proper command line, Borland C++ will compile the C++ code, invoke Turbo Assembler to do the assembling, and invoke TLINK to link the object files into an executable file. Suppose, for example, that you have a program consisting of the C++ files MAIN.CPP and STAT.CPP and the assembler files SUMM.ASM and DISPLAY.ASM. The command line

```
bcc main.cpp stat.cpp summ.asm display.asm
```

compiles MAIN.CPP and STAT.CPP, assembles SUMM.ASM and DISPLAY.ASM, and links all four object files, along with the C++ start-up code and any required library functions, into MAIN.EXE. You only need remember the .ASM extensions when typing your assembler file names.

If you use TLINK in stand-alone mode, the object files generated by Turbo Assembler are standard object modules and are treated just like C++ object modules. See Appendix C for more information about using TLINK in stand-alone mode.

Parameter passing

Borland C++ passes parameters to functions on the stack. Before calling a function, Borland C++ first pushes the parameters to that function onto the stack, starting with the rightmost parameter and ending with the leftmost parameter. The C++ function call

```
      ⋮
   Test(i, j, 1);
      ⋮
```

compiles to

```
mov   ax,1
push  ax
push  WORD PTR DGROUP:_j
push  WORD PTR DGROUP:_i
call  NEAR PTR _Test
add   sp,6
```

in which you can clearly see the rightmost parameter, 1, being pushed first, then *j*, and finally *i*.

Upon return from a function, the parameters that were pushed on the stack are still there, but are no longer useful. Consequently, immediately following each function call, Borland C++ adjusts the stack pointer back to the value it contained before the parameters were pushed, thereby discarding the parameters. In the previous example, the three parameters of 2 bytes each take up 6 bytes of stack space altogether, so Borland C++ adds 6 to the stack pointer to discard the parameters after the call to *Test*. The important point here is that under the default C/C++ calling conventions, the *calling* code is responsible for discarding the parameters from the stack.

Assembler functions can access parameters passed on the stack relative to the BP register. For example, suppose the function *Test* in the previous example is the following assembler function, called PRMSTACK.ASM:

```
        .MODEL  small
        .CODE
        PUBLIC  _Test
_Test   PROC
        push    bp
        mov     bp,sp
        mov     ax,[bp+4]           ;get parameter 1
        add     ax,[bp+6]           ;add parameter 2 to parameter 1
        sub     ax,[bp+8]           ;subtract parameter 3 from sum
        pop     bp
        ret
_Test   ENDP
        END
```

You can see that *Test* is getting the parameters passed by the C++ code from the stack, relative to BP. (Remember that BP addresses the stack segment.) But just how are you to know *where* to find the parameters relative to BP?

Figure 18.2 shows what the stack looks like just before the first instruction in *Test* is executed.

```
i = 25;
j = 4;
Test(i, j, 1);
```

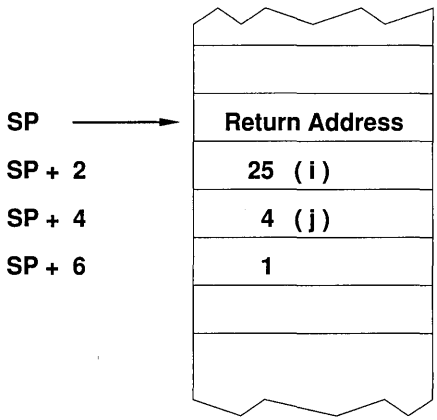
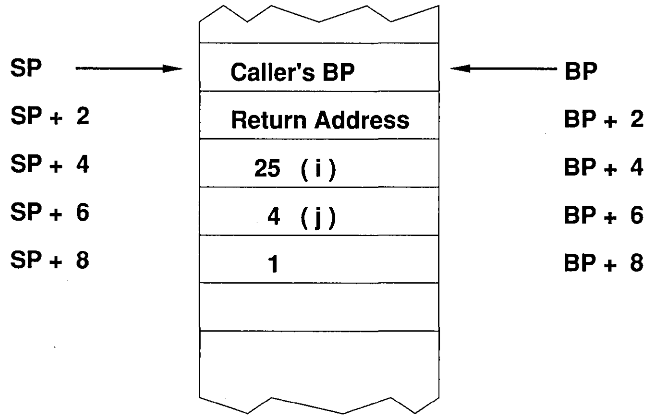


Figure 18.2
State of the stack just before executing Test's first instruction

The parameters to *Test* are at fixed locations relative to SP, starting at the stack location 2 bytes higher than the location of the return address that was pushed by the call. After loading BP with SP, you can access the parameters relative to BP. However, you must first preserve BP, since the calling C++ code expects you to return with BP unchanged. Pushing BP changes all the offsets on the stack. Figure 18.3 shows the stack after these lines of code are executed:

```
   .
   .
   .
push  bp
mov   bp,sp
   .
   .
   .
```

Figure 18.3
State of the stack
after PUSH and MOV



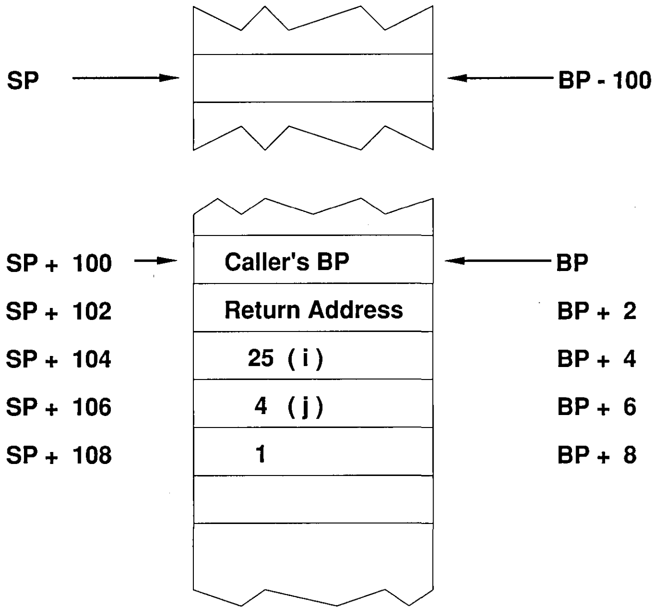
This is the standard C++ stack frame, the organization of a function's parameters and automatic variables on the stack. As you can see, no matter how many parameters a C++ program might have, the leftmost parameter is always stored at the stack address immediately above the pushed return address, the next parameter to the right is stored just above the leftmost parameter, and so on. As long as you know the order and type of the passed parameters, you always know where to find them on the stack.

Space for automatic variables can be reserved by subtracting the required number of bytes from SP. For example, room for a 100-byte automatic array could be reserved by starting *Test* with

```
:
push  bp
mov   bp,sp
sub   sp,100
:
```

as shown in Figure 18.4.

Figure 18.4
State of the stack
after PUSH, MOV,
and SUB



Since the portion of the stack holding automatic variables is at a lower address than BP, negative offsets from BP are used to address automatic variables. For example,

```
mov  BYTE PTR [bp-100],0
```

would set the first byte of the 100-byte array you reserved earlier to zero. Passed parameters, on the other hand, are always addressed at positive offsets from BP.

While you can, if you wish, allocate space for automatic variables as shown previously, Turbo Assembler provides a special version of the **LOCAL** directive that makes allocation and naming of automatic variables a snap. When **LOCAL** is encountered within a procedure, it is assumed to define automatic variables for that procedure. For example,

```
LOCAL   LocalArray:BYTE:100,LocalCount:WORD = AUTO_SIZE
```

defines the automatic variables *LocalArray* and *LocalCount*. *LocalArray* is actually a label equated to [*BP-100*], and *LocalCount* is actually a label equated to [*BP-102*], but you can use them as variable names without ever needing to know their values. *AUTO_SIZE* is the total number of bytes of automatic storage required; you must subtract this value from SP in order to allocate space for the automatic variables.

Here's how you might use **LOCAL**:

```
        .
        .
        .
_TestSub  PROC
     LOCAL     LocalArray:BYTE:100,LocalCount:WORD=AUTO_SIZE
     push bp                ;preserve caller's stack frame pointer
     mov  bp,sp             ;set up our own stack frame pointer
     sub  sp,AUTO_SIZE      ;allocate room for automatic variables
     mov  [LocalCount],10   ;set local count variable to 10
                            ; (LocalCount is actually [BP-102])
        .
        .
        .
     mov  cx,[LocalCount]   ;get count from local variable
     mov  al,'A'            ;we'll fill with character "A"
     lea  bx,[LocalArray]   ;point to local array
                            ; (LocalArray is actually [BP-100])
FillLoop:
     mov  [bx],al           ;fill next byte
     inc  bx                ;point to following byte
     loop FillLoop          ;do next byte, if any
     mov  sp,bp             ;deallocate storage for automatic
                            ; variables (add sp,AUTO_SIZE would
                            ; also have worked)
     pop  bp                ;restore caller's stack frame pointer
     ret
_TestSub  ENDP
        .
        .
        .
```

In this example, note that the first field after the definition of a given automatic variable is the data type of the variable: **BYTE**, **WORD**, **DWORD**, **NEAR**, and so on. The second field after the definition of a given automatic variable is the number of elements of that variable's type to reserve for that variable. This field is optional and defines an automatic array if used; if it is omitted, one element of the specified type is reserved. Consequently, *LocalArray* consists of 100 byte-sized elements, while *LocalCount* consists of 1 word-sized element.

Also note that the **LOCAL** line in the preceding example ends with *=AUTO_SIZE*. This field, beginning with an equal sign, is optional; if present, it sets the label following the equal sign to the number of bytes of automatic storage required. You must then use that label to allocate and deallocate storage for automatic variables, since the **LOCAL** directive only generates labels, and doesn't actually generate any code or data storage. To put this another way: **LOCAL** doesn't allocate automatic variables, but simply generates labels that you can readily use to both allocate storage for and access automatic variables.

As you can see, **LOCAL** makes it much easier to define and use automatic variables. Note that the **LOCAL** directive has a completely different meaning when used in macros.

By the way, Borland C++ handles stack frames in just the way we've described here. You might find it instructive to compile a few Borland C++ modules with the **–S** option, and then look at the assembler code Borland C++ generates to see how Borland C++ creates and uses stack frames.

This looks good so far, but there are further complications. First of all, this business of accessing parameters at constant offsets from BP is a nuisance; not only is it easy to make mistakes, but if you add another parameter, all the other stack frame offsets in the function must be changed. For example, suppose you change *Test* to accept four parameters:

```
Test(Flag, i, j, 1);
```

Suddenly *i* is at offset 6, not offset 4, *j* is at offset 8, not offset 6, and so on. You can use equates for the parameter offsets:

```
          :
Flag          EQU  4
AddParm1      EQU  6
AddParm2      EQU  8
SubParm1      EQU  10

         mov  ax,[bp+AddParm1]
         add  ax,[bp+AddParm2]
         sub  ax,[bp+SubParm1]
          :
```

but it's still a nuisance to calculate the offsets and maintain them. There's a more serious problem, too: The size of the pushed return address grows by 2 bytes in far code models, as do the sizes of passed code pointers and data pointer in far code and far data models, respectively. Writing a function that can be easily assembled to access the stack frame properly in any memory model would thus seem to be a difficult task.

Turbo Assembler, however, provides you with the **ARG** directive, which makes it easy to handle passed parameters in your assembler routines.

The **ARG** directive automatically generates the correct stack offsets for the variables you specify. For example,

```
arg  FillArray:WORD,Count:WORD,FillValue:BYTE
```

specifies three parameters: *FillArray*, a word-sized parameter; *Count*, a word-sized parameter, and *FillValue*, a byte-sized parameter. **ARG** actually sets the label *FillArray* to [*BP+4*] (assuming the example code resides in a near procedure), the label *Count* to [*BP+6*], and the label *FillValue* to [*BP+8*].

However, **ARG** is valuable precisely because you can use **ARG**-defined labels without ever knowing the values they're set to.

For example, suppose you've got a function *FillSub*, called from C++ as follows:

```
extern "C" {
  void FillSub(
     char *FillArray,
     int  Count,
     char FillValue);
}

main()
{
const int ARRAY_LENGTH=100;
   char TestArray[ARRAY_LENGTH];

   FillSub(TestArray,ARRAY_LENGTH,'*');
}
```

You could use **ARG** in *FillSub* to handle the parameters as follows:

```
_FillSub  PROC NEAR
     ARG  FillArray:WORD,Count:WORD,FillValue:BYTE
     push bp                     ;preserve caller's stack frame
     mov  bp,sp                  ;set our own stack frame
     mov  bx,[FillArray]         ;get pointer to array to fill
     mov  cx,[Count]             ;get length to fill
     mov  al,[FillValue]         ;get value to fill with
FillLoop:
     mov  [bx],al                ;fill a character
     inc  bx                     ;point to next character
     loop FillLoop               ;do next character
     pop  bp                     ;restore caller's stack frame
     ret
_FillSub  ENDP
```

That's really all it takes to handle passed parameters with **ARG**. Better yet, **ARG** automatically accounts for the different sizes of near and far returns.

Preserving registers

As far as Borland C++ is concerned, C++-callable assembler functions can do anything as long as they preserve the following registers: BP, SP, CS, DS, and SS. While these registers can be altered during the course of an assembler function, when the calling code is returned, they must be exactly as they were when the assembler function was called. AX, BX, CX, DX, ES, and the flags can be changed in any way.

SI and DI are special cases, since they're used by Borland C++ as register variables. If register variables are enabled in the C++ module calling your assembler function, you must preserve SI and DI; but if register variables are not enabled, SI and DI need not be preserved.

It's good practice to always preserve SI and DI in your C++-callable assembler functions, regardless of whether register variables are enabled. You never know when you might link a given assembler module to a different C++ module, or recompile your C++ code with register variables enabled, without remembering that your assembler code needs to be changed as well.

Returning values

A C++-callable assembler function can return a value, just like a C++ function. Function values are returned as follows:

Return value type	Return value location
unsigned char	AX
char	AX
enum	AX
unsigned short	AX
short	AX
unsigned int	AX
int	AX
unsigned long	DX:AX
long	DX:AX
float	8087 top-of-stack (TOS) register (ST(0))
double	8087 top-of-stack (TOS) register (ST(0))
long double	8087 top-of-stack (TOS) register (ST(0))
near *	AX
far *	DX:AX

In general, 8- and 16-bit values are returned in AX, and 32-bit values are returned in DX:AX, with the high 16 bits of the value in DX. Floating-point values are returned in ST(0), which is the 8087's top-of-stack (TOS) register, or in the 8087 emulator's TOS register if the floating-point emulator is being used.

Structures are a bit more complex. Structures that are 1 or 2 bytes in length are returned in AX, and structures that are 4 bytes in length are returned in DX:AX. When a function that returns a three-byte structure or a structure larger than 4 bytes is called, the caller must allocate space for the return value (usually on the stack), and pass the address of this space to the function as an additional "hidden" parameter. The function assigns the return value through this pointer argument, and returns that pointer as its result. As with all pointers, near pointers to structures are returned in AX, and far pointers to structures are returned in DX:AX.

Let's look at a small model C++-callable assembler function, *FindLastChar*, that returns a near pointer to the last character of a passed string. The C++ prototype for this function would be

```
extern char * FindLastChar(char * StringToScan);
```

where *StringToScan* is the nonempty string for which a pointer to the last character is to be returned.

Here's *FindLastChar*, from FINDCHAR.ASM:

```
        .MODEL  small
        .CODE
        PUBLIC _FindLastChar
_FindLastChar   PROC
        ARG     StringToScan:WORD
        push    bp
        mov     bp,sp
        cld             ;we need string instructions to count up
        mov     ax,ds
        mov     es,ax   ;set ES to point to the near data segment
        mov     di, [StringToScan]    ;point ES:DI to start of
                                      ;passed string
        mov     al,0    ;search for the null that ends the string
        mov     cx,0ffffh  ;search up to 64K-1 bytes
        repnz   scasb      ;look for the null
        dec     di         ;point back to the null
        dec     di         ;point back to the last character
        mov     ax,di      ;return the near pointer in AX
        pop     bp
        ret
_FindLastChar   ENDP
        END
```

The final result, the near pointer to the last character in the passed string, is returned in AX.

Calling an assembler function from C++

Now look at an example of Borland C++ code calling a Turbo Assembler function. The following Turbo Assembler module, COUNT.ASM, contains the function *LineCount*, which returns counts of the number of lines and characters in a passed string:

```
; Small model C++-callable assembler function to count the number
; of lines and characters in a zero-terminated string.
;
; Function prototype:
;       extern unsigned int LineCount(char * near StringToCount,
;               unsigned int near * CharacterCountPtr);
```

```
; Input:
;       char near * StringToCount: pointer to the string on which
;       a line count is to be performed
;
;       unsigned int near * CharacterCountPtr: pointer to the
;               int variable in which the character count is
;               to be stored
;
NEWLINE  EQU     0ah            ;the linefeed character is C's
                                ; newline character
         .MODEL  small
         .CODE
         PUBLIC  _LineCount
_LineCount       PROC
         push    bp
         mov     bp,sp
         push    si             ;preserve calling program's
                                ; register variable, if any
         mov     si,[bp+4]      ;point SI to the string
         sub     cx,cx          ;set character count to 0
         mov     dx,cx          ;set line count to 0
LineCountLoop:
         lodsb                  ;get the next character
         and     al,al          ;is it null, to end the string?
         jz      EndLineCount   ;yes, we're done
         inc     cx             ;no, count another character
         cmp     al,NEWLINE     ;is it a newline?
         jnz     LineCountLoop  ;no, check the next character
         inc     dx             ;yes, count another line
         jmp     LineCountLoop
EndLineCount:
         inc     dx             ;count the line that ends with the
                                ; null character
         mov     bx,[bp+6]      ;point to the location at which to
                                ; return the character count
         mov     [bx],cx        ;set the character count variable
         mov     ax,dx        ;return line count as function value
         pop     si           ;restore calling program's register
                              ; variable, if any
         pop     bp
         ret
_LineCount       ENDP
         END
```

The following C++ module, CALLCT.CPP, is a sample invocation of the *LineCount* function:

```
#include <stdio.h>

char * TestString="Line 1\nline 2\nline 3";
extern "C" unsigned int LineCount(char * StringToCount,
                                  unsigned int * CharacterCountPtr);

int main()
{
   unsigned int LCount;
   unsigned int CCount;
   LCount = LineCount(TestString, &CCount);
   printf("Lines: %d\nCharacters: %d\n", LCount, CCount);
   return 0;
}
```

The two modules are compiled and linked together with the command line

```
bcc -ms callct.cpp count.asm
```

As shown here, *LineCount* will work only when linked to small-model C++ programs since pointer sizes and locations on the stack frame change in other models. Here's a version of *LineCount*, COUNTLG.ASM, that will work with large-model C++ programs (but not small-model ones, unless far pointers are passed, and *LineCount* is declared far):

```
; Large model C++-callable assembler function to count the number
; of lines and characters in a zero-terminated string.
;
; Function prototype:
;       extern unsigned int LineCount(char * far StringToCount,
;              unsigned int * far CharacterCountPtr);
;       char far * StringToCount: pointer to the string on which
;                                 a line count is to be performed
;
;       unsigned int far * CharacterCountPtr: pointer to the
;               int variable in which the character count
;               is to be stored
;
NEWLINE  EQU     0ah      ;the linefeed character is C's newline
                          ; character
         .MODEL  large
         .CODE
         PUBLIC  _LineCount
```

```
_LineCount      PROC
        push    bp
        mov     bp,sp
        push    si              ;preserve calling program's
                                ; register variable, if any
        push    ds              ;preserve C's standard data seg
        lds     si,[bp+6]       ;point DS:SI to the string
        sub     cx,cx           ;set character count to 0
        mov     dx,cx           ;set line count to 0
LineCountLoop:
        lodsb                   ;get the next character
        and     al,al           ;is it null, to end the string?
        jz      EndLineCount    ;yes, we're done
        inc     cx              ;no, count another character
        cmp     al,NEWLINE      ;is it a newline?
        jnz     LineCountLoop   ;no, check the next character
        inc     dx              ;yes, count another line
        jmp     LineCountLoop
EndLineCount:
        inc     dx              ;count line ending with null
                                ; character
        les     bx,[bp+10]      ;point ES:BX to the location at
                                ; which to return char count
        mov     es:[bx],cx      ;set the char count variable
        mov     ax,dx           ;return the line count as
                                ; the function value
        pop     ds              ;restore C's standard data seg
        pop     si              ;restore calling program's
                                ; register variable, if any
        pop     bp
        ret
_LineCount      ENDP
        END
```

COUNTLG.ASM can be linked to CALLCT.CPP with the following command line:

```
bcc -ml callct.cpp countlg.asm
```

Writing C++ member functions in assembly language

While you can write a member function of a C++ class completely in assembly language, it is not easy. For example, all member functions of C++ classes are name-mangled to provide the type-safe linkage that makes things like overridden functions available, and your assembler function would have to know exactly what name C++ would be expecting for the member function. To access the member variables you must prepare a STRUC definition in your assembler code that defines all the member variables with exactly the same sizes and locations. If your class is a derived class, there may be other member variables derived from a base class. Even if your class is not a descendant of another class, the location of member variables in memory changes if the class includes any virtual functions.

If you write your function using inline assembler, Borland C++ can take care of these issues for you. But if you must write your function in assembly language, (perhaps because you are reusing some existing assembler code), there are some special techniques you can use to make things easier.

For an example of how to write assembly functions using mangled names, see the example on page 222.

Create a dummy stub C++ function definition for the assembler function. This stub will satisfy the linker because it will have a properly mangled name for the member function. The dummy stub then calls your assembler function and passes to it the member variables and other parameters. Since your assembler code has all the parameters it needs passed as arguments, you don't have to worry about changes in the class definition. Your assembler function can be declared in the C++ code as an extern "C" function, just as we have shown you in other examples.

Here's an example, called COUNTER.CPP:

```
#include <stdio.h>

class counter {
     // Private member variables:
     int count;          // The ongoing count
   public:
        counter(void){ count=0; }
        int  get_count(void){return count;}

        // Two functions that will actually be written
        // in assembler:
        void increment(void);
        void add(int what_to_add=-1);
         // Note that the default value only
         // affects calls to add, it does not
         // affect the code for add.
};

extern "C" {
  // To create some unique, meaningful names for the
  // assembler routines, prepend the name of the class
  // to the assembler routine. Unlike some assemblers,
  // Turbo Assembler has no problem with long names.
  void counter_increment(int *count);     // We will pass a
                                          // pointer to the
                                          // count variable.
                                          // Assembler will
                                          // do the incrementing.
```

```
    void counter_add(int *count,int what_to_add);
}

void counter::increment(void) {
    counter_increment(&count);
}

void counter::add(int what_to_add) {
    counter_add(&count, what_to_add);
}

int main() {
   counter Counter;

   printf( "Before count: %d\n", Counter.get_count() );
   Counter.increment();
   Counter.add( 5 );
   printf( "After count:  %d\n", Counter.get_count() );
   return 0;
}
```

Your assembler module that defines the count_add_increment and count_add_add routines could look like this example, called COUNTADD.ASM:

```
     .MODEL small   ; Select small model (near code and data)
     .CODE
     PUBLIC _counter_increment
_counter_increment  PROC
     ARG count_offset:word      ; Address of the member variable
     push bp                    ; Preserve caller's stack frame
     mov  bp,sp                 ; Set our own stack frame
     mov  bx,[count_offset]     ; Load pointer
     inc  word ptr [bx]         ; Increment member variable
     pop  bp                    ; Restore callers stack frame
     ret
_counter_increment  ENDP

     PUBLIC _counter_add
_counter_add  PROC
     ARG count_offset:word,what_to_add:word
     push bp
     mov  bp,sp
     mov  bx,[count_offset]     ; Load pointer
     mov  ax,[what_to_add]
     add  [bx],ax
     pop  bp
     ret
_counter_add  ENDP

     end
```

Using this method, you don't have to worry about changes in your class definition. Even if you add or delete member variables, make this class a derived class, or add virtual functions, you won't have to change your assembler module. You need to reassemble your module only if you change the structure of the count member variable, or if you make a large model version of this class. You need to reassemble because you have to deal with a segment and an offset when referring to the count member variable.

Pascal calling conventions

So far, you've seen how C++ normally passes parameters to functions by having the calling code push parameters right to left, call the function, and discard the parameters from the stack after the call. Borland C++ is also capable of following the conventions used by Pascal programs in which parameters are passed from left to right, and the *called* function discards the parameters from the stack. In Borland C++, Pascal conventions are enabled with the **-p** command-line option or the **pascal** keyword.

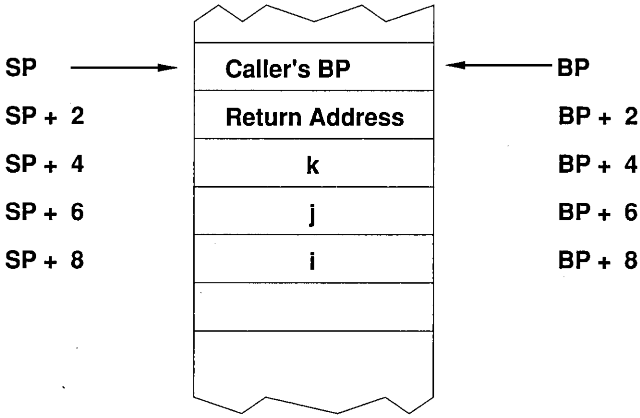
The following example, ASMPSCL.ASM, shows an assembler function that uses Pascal conventions:

```
; Called as: TEST_PROC(i, j, k);

i          equ     8          ;leftmost parameter
j          equ     6
k          equ     4          ;rightmost parameter

           .MODEL  small
           .CODE
           PUBLIC  TEST_PROC
TEST_PROC  PROC
           push    bp
           mov     bp,sp
           mov     ax,[bp+i]  ;get i
           add     ax,[bp+j]  ;add j to i
           sub     ax,[bp+k]  ;subtract k from the sum
           pop     bp
           ret     6          ;return, discarding 6 parameter bytes
TEST_PROC  ENDP
           END
```

Note that **RET 6** is used by the called function to clear the passed parameters from the stack.

Figure 18.5 shows the stack frame after **MOV BP,SP** has been executed.

Figure 18.5
State of the stack immediately after MOV BP, SP



Pascal calling conventions also require all external and public symbols to be in uppercase, with no leading underscores. Why would you want to use Pascal calling conventions in a C++ program? Code that uses Pascal conventions tends to be somewhat smaller and faster than normal C++ code since there's no need to execute an **ADD SP** *n* instruction to discard the parameters after each call.

Calling Borland C++ from Turbo Assembler

Although it's most common to call assembler functions from C++ to perform specialized tasks, you might occasionally want to call C++ functions from assembler. As it turns out, it's actually easier to call a Borland C++ function from a Turbo Assembler function than the reverse since no stack-frame handling on the part of the assembler code is required. Let's take a quick look at the requirements for calling Borland C++ functions from assembler.

Link in the C++ startup code

As a general rule, you should only call Borland C++ library functions from assembler code in programs that link in the C++ startup module as the first module linked.

Generally, you should not call Borland C++ library functions from programs that don't link in the C++ startup module since some Borland C++ library functions will not operate properly if the startup code is not linked in. If you really want to call Borland C++ library functions from such programs, we suggest you look at the startup source code (the file C0.ASM on the Borland C++ distribution disks) and purchase the C++ library source code from Borland. This way, you can be sure to provide the proper initialization for the library functions you need.

➡ Calling user-defined C++ functions that in turn call C++ library functions falls into the same category as calling library functions directly; lack of the C++ startup can potentially cause problems for *any* assembler program that calls C++ library functions, directly or indirectly.

The segment setup

As we learned earlier, you must make sure that Borland C++ and Turbo Assembler are using the same memory model and that the segments you use in Turbo Assembler match those used by Borland C++. Turbo Assembler has a **tchuge** memory model that supports Borland C++'s huge memory model. Refer to the previous section if you need a refresher on matching memory models and segments. Also, remember to put **EXTRN** directives for far symbols either outside all segments or inside the correct segment.

Performing the call

All you need to do when passing parameters to a Borland C++ function is push the rightmost parameter first, then the next rightmost parameter, and so on, until the leftmost parameter has been pushed. Then just call the function. For example, when programming in Borland C++, to call the Borland C++ library function **strcpy** to copy *SourceString* to *DestString*, you would type

```
strcpy(DestString, SourceString);
```

To perform the same call in assembler, you would use

```
lea  ax,SourceString   ;rightmost parameter
lea  bx,DestString     ;leftmost parameter
push ax                ;push rightmost first
push bx                ;push leftmost next
call _strcpy           ;copy the string
add  sp,4              ;discard the parameters
```

Don't forget to discard the parameters by adjusting SP after the call.

You can simplify your code and make it language independent at the same time by taking advantage of Turbo Assembler's **CALL** instruction extension:

```
call destination [language [,arg1] ...]
```

where *language* is C, CPP, PASCAL, BASIC, FORTRAN, PROLOG or NOLANGUAGE, and *arg* is any valid argument to the routine that can be directly pushed onto the processor stack.

Using this feature, the preceding code can be reduced to

```
lea   ax,SourceString
lea   bx,DestString
call  strcpy c,bx,ax
```

Turbo Assembler automatically inserts instructions to push the arguments in the correct order for C++ (AX first, then BX), performs the call to **_strcpy** (Turbo Assembler automatically inserts an underscore in front of the name for C++), and cleans up the stack after the call.

If you're calling a C++ function that uses Pascal calling conventions, you have to push the parameters left to right and not adjust SP afterward:

```
lea   bx,DestString     ;leftmost parameter
lea   ax,SourceString   ;rightmost parameter
push  bx                ;push leftmost first
push  ax                ;push rightmost next
call  STRCPY            ;copy the string
                        ;leave the stack alone
```

Again, you can use Turbo Assembler's **CALL** instruction extension to simplify your code:

```
lea   bx,DestString      ;leftmost parameter
lea   ax,SourceString    ;rightmost parameter
call  strcpy pascal,bx,ax
```

Turbo Assembler automatically inserts instructions to push the arguments in the correct order for Pascal (BX first, then AX) and performs the call to **STRCPY** (converting the name to all uppercase, as is the Pascal convention).

The last example assumes that you've recompiled **strcpy** with the **–p** switch, since the standard library version of **strcpy** uses C++ rather than Pascal calling conventions.

Rely on C++ functions to preserve the following registers and *only* the following registers: SI, DI, BP, DS, SS, SP, and CS. Registers AX, BX, CX, DX, ES, and the flags may be changed arbitrarily.

Calling a Borland C++ function from Turbo Assembler

One case in which you may wish to call a Borland C++ function from Turbo Assembler is when you need to perform complex calculations. This is especially true when mixed integer and floating-point calculations are involved; while it's certainly possible to perform such operations in assembler, it's simpler to let C++ handle the details of type conversion and floating-point arithmetic.

Let's look at an example of assembler code that calls a Borland C++ function in order to get a floating-point calculation performed. In fact, let's look at an example in which a Borland C++ function passes a series of integer numbers to a Turbo Assembler function, which sums the numbers and in turn calls another Borland C++ function to perform the floating-point calculation of the average value of the series.

The C++ portion of the program in CALCAVG.CPP is

```
#include <stdio.h>

extern "C" float Average(int far * ValuePtr, int NumberOfValues);

#define NUMBER_OF_TEST_VALUES 10
int TestValues[NUMBER_OF_TEST_VALUES] = {
   1, 2, 3, 4, 5, 6, 7, 8, 9, 10
};

int main()
{
   printf("The average value is: %f\n",
          Average(TestValues, NUMBER_OF_TEST_VALUES));
   return 0;
}

extern "C"
float IntDivide(int Dividend, int Divisor)
{
   return( (float) Dividend / (float) Divisor );
}
```

and the assembler portion of the program in AVERAGE.ASM is

```
;
; Borland C++-callable small-model function that returns the average
; of a set of integer values. Calls the Borland C++ function
; IntDivide() to perform the final division.
;
; Function prototype:
;   extern float Average(int far * ValuePtr, int NumberOfValues);
;
; Input:
;   int far * ValuePtr:          ;the array of values to average
;   int NumberOfValues:          ;the number of values to average

        .MODEL  small
        EXTRN   _IntDivide:PROC
        .CODE
        PUBLIC  _Average
_Average        PROC
        push    bp
        mov     bp,sp
        les     bx,[bp+4]        ;point ES:BX to array of values
        mov     cx,[bp+8]        ;# of values to average
        mov     ax,0             ;clear the running total
```

```
AverageLoop:
        add     ax,es:[bx]        ;add the current value
        add     bx,2              ;point to the next value
        loop    AverageLoop
        push    WORD PTR [bp+8]   ;get back the number of values
                                  ; passed to IntDivide as the
                                  ; rightmost parameter
        push    ax      ;pass the total as the leftmost parameter
        call    _IntDivide  ;calculate the floating-point average
        add     sp,4              ;discard the parameters
        pop     bp
        ret                  ;average is in 8087's TOS register
_Average        ENDP
        END
```

The C++ **main** function passes a pointer to the array of integers *TestValues* and the length of the array to the assembler function *Average*. *Average* sums the integers, then passes the sum and the number of values to the C++ function *IntDivide*. *IntDivide* casts the sum and number of values to floating-point numbers and calculates the average value, doing in a single line of C++ code what would have taken several assembler lines. *IntDivide* returns the average to *Average* in the 8087 TOS register, and *Average* just leaves the average in the TOS register and returns to **main**.

CALCAVG.CPP and AVERAGE.ASM could be compiled and linked into the executable program CALCAVG.EXE with the command

```
bcc calcavg.cpp average.asm
```

Note that *Average* will handle both small and large data models without the need for any code change since a far pointer is passed in all models. All that would be needed to support large code models (huge, large, and medium) would be use of the appropriate **.MODEL** directive.

Taking full advantage of Turbo Assembler's language-independent extensions, the assembly code in the previous example could be written more concisely as shown here in CONCISE.ASM:

```
    .MODEL   small,C
    EXTRN    C IntDivide:PROC
    .CODE
    PUBLIC   C Average
Average      PROC  C ValuePtr:DWORD,NumberOfValues:WORD
    les      bx,ValuePtr
    mov      cx,NumberOfValues
    mov      ax,0
```

```
AverageLoop:
    add     ax,es:[bx]
    add     bx,2            ;point to the next value
    loop    AverageLoop
    call    IntDivide C,ax,NumberOfValues
    ret
Average     ENDP
    END
```

A P P E N D I X A

Program blueprints

This appendix describes basic program construction information depending on specific memory models and executable object formats.

Simplified segmentation segment description

The following tables show the default segment attributes for each memory model.

Table A.1
Default segments and types for TINY memory model

Directive	Name	Align	Combine	Class	Group
.CODE	_TEXT	WORD	PUBLIC	'CODE'	DGROUP
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the **MODEL** directive.

Table A.2
Default segments and types for SMALL memory model

Directive	Name	Align	Combine	Class	Group
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the **MODEL** directive.

Table A.3
Default segments and types for MEDIUM memory model

Directive	Name	Align	Combine	Class	Group
.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the **MODEL** directive.

Table A.4
Default segments and types for COMPACT memory model

Directive	Name	Align	Combine	Class	Group
.CODE	_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the **MODEL** directive.

Table A.5
Default segments and types for LARGE or HUGE memory model

Directive	Name	Align	Combine	Class	Group
.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	_DATA	WORD	PUBLIC	'DATA'	DGROUP
.CONST	CONST	WORD	PUBLIC	'CONST'	DGROUP
.DATA?	_BSS	WORD	PUBLIC	'BSS'	DGROUP
.STACK*	STACK	PARA	STACK	'STACK'	DGROUP

* STACK not assumed to be in DGROUP if FARSTACK specified in the **MODEL** directive.

Table A.6
Default segments and types for Borland C++ HUGE (TCHUGE) memory model

Directive	Name	Align	Combine	Class	Group
.CODE	*name*_TEXT	WORD	PUBLIC	'CODE'	
.FARDATA	FAR_DATA	PARA	private	'FAR_DATA'	
.FARDATA?	FAR_BSS	PARA	private	'FAR_BSS'	
.DATA	*name*_DATA	PARA	private	'DATA'	
.STACK*	STACK	PARA	STACK	'STACK'	

* STACK is automatically FAR

OS/2 programs

Programs designed to be executed under the OS/2 operating system can use one of several formats, depending on the capabilities you want. OS/2 can execute programs designed for DOS program formats, as well as programs and DLLs written for Windows. However, the most powerful format available under OS/2 is the linear executable format, in which a program no longer has to manipulate segment registers, and 512 megabytes of virtual memory is available. This format is also known as flat model.

OS/2 flat-model program blueprint

Turbo Assembler assumes that all 32-bit segments in a flat-model program belong to a supergroup called FLAT, and share the same segment selector. (Segments that are 16-bit are allowed, but are of little use.)

When you execute a flat-model program, Turbo Assembler initializes the registers as follows:

Register	Value
CS,DS,ES,SS	Contains the segment selector of the 32-bit linear address for the program. These registers should never have to be changed.
CS:EIP	Contains the address of the **STARTUPCODE** directive.
SS:ESP	Contains the address of the last word of the stack segment, which the **STACK** directive specifies.
FS,GS	Contain special values that the application should not modify.

You *must* define linear-executable programs with the FLAT model. This instructs Turbo Assembler to consider all 32-bit segments and groups to be a member of the FLAT supergroup. You can also optionally specify the OS/2 operating system, which allows the **STARTUPCODE** and **EXITCODE** directives to function correctly (for example, `MODEL OS2 FLAT`).

The **STARTUPCODE** directive produces instructions that automatically initialize all necessary registers to conform to FLAT model. Similarly, the **EXITCODE** directive produces instructions that automatically return control to the operating system, while letting you specify an optional return value.

EXE2PROG.ASM on your example Turbo Assembler disks illustrates these topics.

You can use the MAKE utility to build a linear executable file. MAKEFILE should include all modules to link together to form the program, as follows:

```
EXE2PROG.EXE:EXE2PROG.ASM
  TLINK EXE2PROG,,,,EXE2PROG;
EXE2PROG.OBJ:EXE2PROG.ASM
  TASM EXE2PROG
```

You'll also need a file called EXE2PROG.DEF, containing the following:

```
NAME EXE2PROG WINDOWCOMPAT
PROTMODE
CODE PRELOAD
DATA PRELOAD
```

A P P E N D I X B

Turbo Assembler syntax summary

This appendix describes the syntax of Turbo Assembler expressions in a modified Backus-Naur form (BNF). The symbol ::= describes a syntactical production. Ellipses (...) indicate that an element is repeated as many times as it is found. This appendix also discusses keywords and their precedences.

Lexical grammar

valid_line ::=

white_space valid_line
punctuation valid_line
number_string valid_line
id_string valid_line
null

white_space ::=

space_char white_space
space_char

space_char ::=

All control characters, character > 128, ' '

id_string ::=

id_char id_strng2

id_strng2 ::=

id_chr2 id_strng2
null

id_char ::=

Any of $, %, _, ?, or any alphabetic characters

id_chr2 ::=

id_chars plus numerics

number_string ::=

num_string
str_string

num_string ::=

digits alphanums
digits '.' digits exp
digits exp ;Only MASM mode **DD**, **DQ**, or **DT**

digits ::=

digit digits
digit

digit ::=

0 through 9

alphanums ::=

digit alphanum
alpha alphanum
null

alpha ::=

alphabetic characters

exp ::=

E + digits
E – digits
E digits
null

str_string ::=

Quoted string, quote enterable by two quotes in a row

punctuation ::=

Everything that is not a space_char, id_char, ' " ', " ' ", or digits

The period (.) character is handled differently in MASM mode and Ideal mode. This character is not required in floating-point numbers in MASM mode and also can't be part of a symbol name in Ideal mode. In MASM mode, it is sometimes the start of a symbol name and sometimes a punctuation character used as the structure member selector.

Here are the rules for the period (.) character:

1. In Ideal mode, it's always treated as punctuation.
2. In MASM mode, it's treated as the first character of an ID in the following cases:

   a. When it is the first character on the line, or in other special cases like **EXTRN** and **PUBLIC** symbols, it gets attached to the following symbol if the character that follows it is an *id_chr2*, as defined in the previous rules.

   b. If it appears other than as the first character on the line, or if the resulting symbol would make a defined symbol, the period gets appended to the start of the symbol following it.

MASM mode expression grammar

Expression parsing starts at *MASM_expr*.

MASM_expr ::=

mexpr1

mexpr1 ::=
SHORT mexpr1
.TYPE mexpr1
SMALL mexpr1 ;If 386
LARGE mexpr1 ;If 386
expr2

expr2 ::=

expr3 OR expr3 ...
expr3 XOR expr3 ...
expr3

expr3 ::=

expr4 AND expr4 ...
expr4

expr4 ::=

NOT expr4
expr5

expr5 ::=

expr6 EQ expr6 ...
expr6 NE expr6 ...
expr6 LT expr6 ...
expr6 LE expr6 ...
expr6 GT expr6 ...
expr6 GE expr6 ...
expr6

expr6 ::=

expr7 + expr7 ...
expr7 – expr7 ...
expr7

expr7 ::=

*mexpr10 * mexpr10 ...*
mexpr10 / mexpr10 ...
mexpr10 MOD mexpr10 ...
mexpr10 SHR mexpr10 ...
mexpr10 SHL mexpr10 ...
mexpr10

expr8 ::=

+ expr8
– expr8
expr12

expr10 ::=

OFFSET pointer
SEG pointer
SIZE symbol
LENGTH symbol
WIDTH symbol
MASK symbol
THIS itype
symbol
(pointer)
[pointer]

mexpr10 ::=

mexpr11 PTR mexpr10
mexpr11
TYPE mexpr10
HIGH mexpr10
LOW mexpr10
OFFSET mexpr10
SEG mexpr10
THIS mexpr10

mexpr11 ::=

expr8 : expr8 ...

mexpr12 ::=

mexpr13 [mexpr13 ...	;Implied addition if bracket
mexpr13 (mexpr13 ...	;Implied addition if parenthesis
mexpr13 '.' mexpr10	

mexpr13 ::=
LENGTH symbol
SIZE symbol
WIDTH symbol
MASK symbol
(mexpr1)
[mexpr1]
expr10

Ideal mode expression grammar

Expression parsing starts at *ideal_expr*.

ideal_expr ::=

pointer

itype ::=

UNKNOWN
BYTE
WORD
DWORD
PWORD
FWORD
QWORD

TBYTE
SHORT
NEAR
FAR
PROC
DATAPTR
CODEPTR
structure_name
table_name
enum_name
record_name
TYPE pointer

pointer ::=

SMALL pointer	;If 386
LARGE pointer	;If 386

itype PTR pointer
itype LOW pointer
itype HIGH pointer
itype pointer
pointer2

pointer2 ::=

pointer3 . symbol ...
pointer3

pointer3 ::=

expr : pointer3
expr

expr ::=

SYMTYPE expr
expr2

expr2 ::=

expr3 OR expr3 ...
expr3 XOR expr3 ...
expr3

expr3 ::=

expr4 AND expr4 ...
expr4

expr4 ::=

NOT expr4
expr5

expr5 ::=

expr6 EQ expr6 ...
expr6 NE expr6 ...
expr6 LT expr6 ...
expr6 LE expr6 ...
expr6 GT expr6 ...
expr6 GE expr6 ...
expr6

expr6 ::=

expr7 + expr7 ...
expr7 – expr7 ...
expr7

expr7 ::=

*expr8 * expr8 ...*
expr8 / expr8 ...
expr8 MOD expr8 ...
expr8 SHR expr8 ...
expr8 SHL expr8 ...
expr8

expr8 ::=

+ expr8
– expr8
expr9

expr9 ::=

HIGH expr9
LOW expr9
expr10

expr10 ::=

OFFSET pointer
SEG pointer
SIZE symbol
LENGTH symbol
WIDTH symbol
MASK symbol

THIS itype
symbol
(pointer)
[pointer]

Keyword precedence

It's important to understand how Turbo Assembler parses source lines so that you can avoid writing code that produces unexpected results. For example, examine the following program fragment:

```
NAME SEGMENT
```

If you had written this line hoping to open a segment called NAME, you would be disappointed. Turbo Assembler recognizes the **NAME** directive before the **SEGMENT** directive, thus naming your code SEGMENT.

In general, Turbo Assembler determines the meaning of a line based on the first two symbols on the line. The leftmost symbol is in the first position, while the symbol to its right is in the second position.

Ideal mode precedence

The following precedence rules for parsing lines apply to Ideal mode:

1. All keywords in the first position of the line have the highest priority (priority 1) and are checked first.
2. The keywords in the second position have priority 2 and are checked second.

MASM mode precedence

The precedence rules for parsing lines in MASM mode are much more complicated than in Ideal mode. There are three levels of priority instead of two, as follows:

Note: Turbo Assembler treats priority 1 keywords like priority 3 keywords inside structure definitions. In this case, priority 2 keywords have the highest priority.

1. The highest priority (priority 1) is assigned to certain keywords found in the first position, such as **NAME** or **%OUT**.
2. The next highest priority (priority 2) belongs to all symbols found in the second position.
3. All other keywords found in first position have the lowest priority (priority 3).

For example, in the code fragment

```
NAME SEGMENT
```

NAME is a priority 1 keyword, while **SEGMENT** is a priority 2 keyword. Therefore, Turbo Assembler will interpret this line as a **NAME** directive rather than a **SEGMENT** directive. In another example,

```
MOV INSTR,1
```

MOV is a priority 3 keyword, while **INSTR** is a priority 2 keyword. Thus, Turbo Assembler interprets this line as an **INSTR** directive, not a **MOV** instruction (which you might have wanted).

Keywords and predefined symbols

This section contains a complete listing of all Turbo Assembler keywords.

The values in parentheses next to keywords indicate the priority of the keyword (1 or 2) in MASM mode. Keywords are labeled with a priority only if they have priority 1 or 2. All others are assumed to be priority 3. Turbo Assembler recognizes the keyword only if it finds them. In MASM mode, priority 1 or 3 keywords always are located in the first position, while priority 2 keywords occur in the second position.

An *M* next to a keyword indicates that you can use a keyword only in MASM mode, and an *I* indicates a keyword that is available only in Ideal mode. If there is no letter, the keyword works in either mode. A number next to the keyword indicates its priority.

Directive keywords

The following list contains all Turbo Assembler directive keywords. The keywords are grouped by the version of Turbo Assembler in which they were introduced.

These keywords were introduced in Turbo Assembler 1.0.

Table B.1: Turbo Assember V1.0 (VERSION T100) keywords

% (1)	.8087 (M)	.ALPHA (M)	BTC
.186 (M)	: (2)	AND	BTR
.286 (M)	= (2)	ARG	BTS
.286c (M)	AAA	ARPL	CALL
.286p (M)	AAD	ASSUME	CATSTR (2)
.386 (M)	AAM	%BIN	CBW
.386c (M)	AAS	BOUND	CDQ
.386p (M)	ADC	BSF	CLC
.387 (M)	ADD	BSR	CLD
.8086 (M)	ALIGN	BT	CLI

Table B.1: Turbo Assember V1.0 (VERSION T100) keywords (continued)

CLTS	ELSEIFDIF (1)	EXTRN (1)	FLDZ
CMC	ELSEIFDIFI (1)	F2XM1	FMUL
CMP	ELSEIFE (1)	FABS	FMULP
CMPBW	ELSEIFIDN (1)	FADD	FNCLEX
CMPS	ELSEIFIDNI (1)	FADDP	FNDISI
CMPSB	ELSEIFNB (1)	FARDATA	FNENI
CMPSD	ELSEIFNDEF (1)	.FARDATA (M)	FNINIT
.CODE (M)	EMUL	.FARDATA? (M)	FNOP
CODESEG	END	FBLD	FNSAVE
COMM (1)	ENDIF (1)	FBSTP	FNSTCW
COMMENT (1)	ENDM	FCHS	FNSTENV
%CONDS	ENDP (2)	FCLEX	FNSTSW
CONST	ENDS (2)	FCOM	FPATAN
.CONST (M)	ENTER	FCOMP	FPREM
%CREF	EQU (2)	FCOMPP	FPTAN
.CREF (M)	.ERR (1)(M)	FDECSTP	FRNDINT
%CREFALL	ERR	FDISI	FRSTOR
%CREFREF	.ERR1 (1)(M)	FDIV	FSAVE
%CREFUREF	.ERR2 (1)(M)	FDIVP	FSCALE
%CTLS	.ERRB (1)(M)	FDIVR	FSQRT
CWD	.ERRDEF (1)(M)	FDIVRP	FST
CWDE	.ERRDIF (1)(M)	FENI	FSTCW
DAA	.ERRDIFI (1)(M)	FFREE	FSTENV
DAS	.ERRE (1)(M)	FIADD	FSTP
.DATA (M)	.ERRIDN (1)(M)	FICOM	FSTSW
.DATA? (M)	.ERRIDNI (1)(M)	FICOMP	FSUB
DATASEG	ERRIF	FIDIV	FSUBP
DB (2)	ERRIF1	FIDIVR	FSUBR
DD (2)	ERRIF2	FILD	FSUBRP
DEC	ERRIFB	FIMUL	FTST
%DEPTH	ERRIFDEF	FINCSTP	FWAIT
DF (2)	ERRIFDIF	FINIT	FXAM
DISPLAY	ERRIFDIFI	FIST	FXCH
DIV	ERRIFE	FISTP	FXTRACT
DOSSEG	ERRIFIDN	FISUB	FYL2X
DP (2)	ERRIFIDNI	FISBR	FYL2xP1
DQ (2)	ERRIFNB	FLD	FSETPM
DT (2)	ERRIFNDEF	FLD!	FPCOS
DW (2)	.ERRNB (1)(M)	FLDCW	FPREM1
ELSE (1)	.ERRNDEF (1)(M)	FLDENV	FPSIN
ELSEIF (1)	.ERRNZ (1)(M)	FLDL2E	FPSINCOS
ELSEIF1 (1)	ESC	FLDL2T	FUCOM
ELSEIF2 (1)	EVEN	FLDLG2	FUCOMP
ELSEIFB (1)	EVENDATA	FLDLN2	FUCOMPP
ELSEIFDEF (1)	EXITM	FLDPI	GLOBAL (1)

Table B.1: Turbo Assember V1.0 (VERSION T100) keywords (continued)

GROUP (2)	JNA	LODSD	NOMASM51
HLT	JNAE	LODSW	%NOMACS
IDEAL	JNB	LOOP	NOMULTERRS
IDIV	JNBE	LOOPD	NOP
IF (1)	JNC	LOOPDE	NOSMART
IF1 (1)	JNE	LOOPDNE	%NOSYMS
IF2 (1)	JNG	LOOPDNZ	NOT
IFb (1)	JNGE	LOOPDZ	%NOTRUNC
IFDEF (1)	JNL	LOOPE	NOWARN
IFDIF (1)	JNLE	LOOPNE	OR
IFDIFI (1)	JNO	LOOPNZ	ORG
IFE (1)	JNP	LOOPW	OUT
IFIDN (1)	JNS	LOOPWE	%OUT (1)
IFIDNI (1)	JNZ	LOOPWNE	OUTS
IFNB (1)	JO	LOOPWNZ	OUTSB
IFNDEF (1)	JP	LOOPWZ	OUTSD
IJECXZ	JPE	LOOPZ	OUTSW
IMUL	JPO	LSL	P186
IN	JS	LSS	P286
INC	JUMP	LTR	P286N
%INCL	JUMPS	%MACS	P287
INCLUDE (1)	JZ	MACRO (2)	P386
INCLUDELIB (1)	LABEL (2)	MASM	P386N
INS	LAHF	MODEL	P387
INSB	.LALL (M)	.MODEL (M)	P8086
INSD	LAR	MOV	P8087
INSTR (2)	LDS	MOVMOVS	PAGE
INSW	LEA	MOVSB	%PAGESIZE
INT	LEAVE	MOVSD	%PCNT
INTO	LES	MOVSW	PN087
IRET	.LFCOND (M)	MOVSX	POP
IRETD	LFS	MOVZX	POPA
IRP (1)	LGDT	MUL	POPAD
IRPC (1)	LGS	MULTERRS	POPFD
JA	LIDT	NAME (1)	%POPLCTL
JAE	%LINUM	NEG	PPF
JB	%LIST	%NEWPAGE	PROC (2)
JBE	.LIST (M)	%NOCONDS	PUSH
JC	LLDT	%NOCREF	PUSHA
JCXZ	LMSW	%NOCTLS	PUSHAD
JE	LOCAL	NOEMUL	PUSHF
JG	LOCALS	%NOINCL	PUSHFD
JGE	LOCK	NOJUMPS	%PUSHLCTL
JL	LODS	%NOLIST	PUBLIC (1)
JLE	LODSB	NOLOCALS	PURGE

Table B.1: Turbo Assember V1.0 (VERSION T100) keywords (continued)

%PAGESIZE	SCASB	SETPE	%SUBTTL
%PCNT	SCASD	SETPO	%SYMS
PN087	SCASW	SETS	%TABSIZE
%POPLCTL	SEGMENT (2)	SETZ	TEST
PROC (2)	.SEQ (M)	.SFCOND (M)	%TEXT
%PUSHLCTL	SETA	SGDT	.TFCOND (M)
PUBLIC (1)	SETAE	SHL	TITLE (1)
PURGE	SETB	SHLD	%TITLE
QUIRKS	SETBE	SHR	%TRUNC
RADIX	SETC	SHRD	UDATASEG
.RADIX (M)	SETE	SIDT	UFARDATA
RCL	SETG	SIZESTR (2)	UNION (2)
RCR	SETGE	SLDT	USES
RECORD (2)	SETL	SMART	VERR
REPT (1)	SETLE	SMSW	VERW
REP	SETNA	SOR	WAIT
REPE	SETNAE	STACK	WARN
REPNE	SETNB	.STACK (M)	.XALL (M)
REPNZ	SETNBE	.STARTUP (M)	XCHG
REPZ	SETNC	STC	.XCREF (M)
RET	SETNE	STD	XLAT
RETF	SETNG	STI	XLATB
RETN	SETNGE	STOS	.XLIST (M)
ROL	SETNL	STOSB	USECS
ROR	SETNLE	STOSD	USEDS
SAHF	SETNO	STOSW	USEES
SAL	SETNP	STR	USEFS
.SALL (M)	SETNS	STRUC (2)	USEGS
SAR	SETNZ	SUB	USESS
SBB	SETO	SUBSTR (2)	
SCAS	SETP	SUBTTL (1)	

Turbo Assembler version 2.0 supports all version 1.0 keywords, with the following additions:

Table B.2 Turbo Assembler V2.0 (VERSION T200) new keywords

BSWAP	P486	STARTUPCODE
CMPXCHG	P486N	WBINVD
INVD	P487	PUBLICDLL (I)
XADD	INVLPG	RETCODE

Turbo Assembler version 2.5 supports all version 2.0 keywords, plus the following keyword additions:

Table B.3 Turbo Assembler V2.5 (VERSION T250) new keywords

ENTERD	LEAVED
ENTERW	LEAVEW

Turbo Assembler version 3.0 supports keywords from all previous versions, with the following additions:

Table B.4
Turbo Assembler V3.0 (VERSION T300) new keywords

CLRFLAG	GOTO (1)	TBLINIT
ENUM (2)	LARGESTACK	TBLINST
EXITCODE	SETFIELD	TBLPTR
FASTIMUL	SETFLAG	TESTFLAG
FLIPFLAG	SMALLSTACK	TYPEDEF
GETFIELD	TABLE (2)	VERSION
	WHILE (1)	

Turbo Assembler version 3.1 supports keywords from all previous versions, with the following additions:

Table B.5
Turbo Assembler V3.1 (VERSION T310) new keywords

PUSHSTATE	POPSTATE

Turbo Assembler version 3.2 supports keywords from all previous versions, with the following additions:

Table B.6
Turbo Assembler V3.2 (VERSION T320) new keywords

IRETW	POPFW	PROCTYPE(2)
POPAW	PROCDESC(2)	PUSHAW
PUSHFW		

Turbo Assembler version 4.0 supports keywords from all previous versions, with the following additions:

Table B.7
Turbo Assembler V4.0 (VERSION T400) new keywords

ALIAS	P586N	RSM
CMPXCHG8B	P587	WRMSR
CPUID	RDMSR	
P586	RDTSC	

APPENDIX C

Compatibility issues

Turbo Assembler in MASM mode is very compatible with MASM version 5.2. However, 100% compatibility is an ideal that can only be approached, since there is no formal specification for the language and different versions of MASM are not even compatible with each other.

For most programs, you will have no problem using Turbo Assembler as a direct replacement for MASM. Occasionally, Turbo Assembler will issue warnings or errors where MASM would not, which usually means that MASM has not detected an erroneous statement. For example, MASM accepts

```
abc  EQU [BP+2]
     PUBLIC  abc
```

and generates a nonsense object file. Turbo Assembler correctly detects this and many other questionable constructs.

If you are having trouble assembling a program with Turbo Assembler, you might try using the **QUIRKS** directive (which enables potentially troublesome features of MASM). For example,

```
TASM /JQUIRKS MYFILE
```

might make your program assemble properly. If it does, add **QUIRKS** to the top of your source file. Even better, review Chapter 3 and determine which statement in your source file *needs* the **QUIRKS** directive. Then you can rewrite the line(s) of code so that you don't even have to use **QUIRKS**.

For maximum compatibility with MASM, you should use the **NOSMART** directive along with **QUIRKS** mode.

One-pass versus two-pass assembly

Normally, Turbo Assembler performs only one pass when assembling code, while MASM performs two. This feature gives Turbo Assembler a speed advantage, but can introduce minor incompatibilities when forward references and pass-dependent constructions are involved. The command-line option **/m** specifies the number of passes desired. For maximum compatibility with MASM, two passes (**/m2**) should be used. (See Chapter 2 for a complete discussion of this option.) The **/m2** command-line switch will generate a MASM-style compatibility when the following constructs are present:

- **IF1** and **IF2** directives
- **ERR1** and **ERR2** directives
- **ESLEIF1** and **ELSEIF2** directives
- Forward references with **IFDEF** or **IFNDEF**
- Forward references with the **.TYPE** operator
- Recursively defined numbers, such as `NMBR=NMBR+1`
- Forward-referenced or recursively defined text macros, such as

```
LNAME CATSTR LNAME,<1>
```

- Forward-referenced macros

Environment variables

Turbo Assembler doesn't use environment variables to control default options. However, you can place default options in a configuration file and then set up different configuration files for different projects.

If you use **INCLUDE** or **MASM** environment variables to configure MASM, you'll have to make a configuration file for Turbo Assembler. Any options that you have specified using the **MASM** variable can simply be placed in the configuration file. Any directories that you have specified using the **INCLUDE** variable should be placed in the configuration file using the **/I** command-line option.

Microsoft binary floating-point format

By default, older versions of MASM generated floating-point numbers in a format incompatible with the IEEE standard floating-point format. MASM version 5.1 generates IEEE floating-point data by default and has the **.MSFLOAT** directive to specify that the older format be used.

Turbo Assembler does not support the old floating-point format, and therefore does not let you use **.MSFLOAT**.

A P P E N D I X D

Error messages

This chapter describes all the messages that Turbo Assembler generates. Messages usually appear on the screen, but you can redirect them to a file or printer using the standard OS/2 redirection mechanism of putting the device or file name on the command line, preceded by the greater than (>) symbol. For example,

```
TASM MYFILE >ERRORS
```

Turbo Assembler generates several types of messages:

- Information messages
- Warning messages
- Error messages
- Fatal error messages

Information messages

Turbo Assembler displays two information messages: one when it starts assembling your source file(s) and another when it has finished assembling each file. Here's a sample startup display:

```
Turbo Assembler Version 4.0 Copyright (C) 1988, 1994 Borland International
Assembling file: TEST.ASM
```

When Turbo Assembler finishes assembling your source file, it displays a message that summarizes the assembly process; the message looks like this:

```
Error messages: None
Warning messages: None
Passes: 1
Remaining memory: 279k
```

You can suppress all information messages by using the **/T** command-line option. This only suppresses the information messages if no errors occur during assembly. If there are any errors, the **/T** option has no effect and the normal startup and ending messages appear.

Warning and error messages

Warning messages let you know that something undesirable may have happened while assembling a source statement. This might be something such as the Turbo Assembler making an assumption that is usually valid, but might not always be correct. You should always examine the cause of warning messages to see if the generated code is what you wanted. Warning messages *won't* stop Turbo Assembler from generating an object file. These messages are displayed using the following format:

```
**Warning** filename(line) message
```

If the warning occurs while expanding a macro or repeat block, the warning message contains additional information, naming the macro and the line within it where the warning occurred:

```
**Warning** filename(line) macroname(macroline) message
```

Error messages, on the other hand, *will* prohibit Turbo Assembler from generating an object file, but assembly will continue to the end of the file. Here's a typical error message format:

```
**Error** filename(line) message
```

If the error occurs while expanding a macro or repeat block, the error message contains additional information, naming the macro and the line within it where the error occurred:

```
**Error** filename(line) macroname(macroline) message
```

Fatal error messages cause Turbo Assembler to immediately stop assembling your file. Whatever caused the error prohibited the assembler from being able to continue.

The following list arranges Turbo Assembler's messages alphabetical order:

32-bit segments not allowed without .386

Has been extended to work with the new ability to specify **USE32** in the **.MODEL** statement and the **LARGESTACK** command. Formerly was "**USE32** not allowed without .386."

Argument mismatch

Argument sizes did not agree. For example,

```
foo proctype pascal :word, :dword
fooproc proc foo a1:word, a2:dword
 :
endp

call fooproc,ax,bx            ;Argument mismatch.
```

Argument needs type override

The expression needs to have a specific size or type supplied, since its size can't be determined from the context. For example,

```
mov  [bx],1
```

You can usually correct this error by using the **PTR** operator to set the size of the operand:

```
mov  WORD PTR[bx],1
```

Argument to operation or instruction has illegal size

An operation was attempted on something that could not support the required operation. For example,

```
Q LABEL QWORD
QNOT = not Q              ;can't negate a qword
```

Arithmetic overflow

A loss of arithmetic precision occurred somewhere in the expression. For example,

```
X = 20000h * 20000h    ;overflows 32 bits
```

All calculations are performed using 32-bit arithmetic.

ASSUME must be segment register

You have used something other than a segment register in an **ASSUME** statement. For example,

```
ASSUME ax:CODE
```

You can only use segment registers with the **ASSUME** directive.

Bad keyword in SEGMENT statement

One of the align/combine/use arguments to the **SEGMENT** directive is invalid. For example,

```
DATA SEGMENT PAFA PUBLIC    ;PAFA should be PARA
```

Bad switch __

You have used an invalid command-line option. See Chapter 2 for a description of the command-line options.

Can't add relative quantities

You have specified an expression that attempts to add together two addresses, which is a meaningless operation. For example,

```
ABC   DB  ?
DEF = ABC + ABC               ;error, can't add two relatives
```

You can subtract two relative addresses, or you can add a constant to a relative address, as in:

```
XYZ    DB  5 DUP (0)
XYZEND EQU $
XYZLEN = SYZEND - XYZ         ;perfectly legal
XYZ2 = XYZ + 2                ;legal also
```

Can't address with currently ASSUMEd segment registers

An expression contains a reference to a variable for which you have not specified the segment register needed to reach it. For example,

```
DSEG SEGMENT
     ASSUME ds:DSEG
     mov  si,MPTR        ;no segment register to reach XSEG
DSEG ENDS
XSEG SEGMENT
MPTR DW   ?
XSEG ENDS
```

Can't convert to pointer

Part of the expression could not be converted to a memory pointer, for example, by using the **PTR** operator,

```
mov cl,[BYTE PTR al]        ;can't make AL into pointer
```

Can't emulate 8087 instruction

The Turbo Assembler is set to generate emulated floating-point instructions, either via the **/E** command-line option or by using the **EMUL** directive, but the current instruction can't be emulated. For example,

```
EMUL
FNSAVE [WPTR]               ;can't emulate this
```

The following instructions are not supported by floating-point emulators: **FNSAVE**, **FNSTCW**, **FNSTENV**, and **FNSTSW.**

Can't find @file __

You have specified an indirect command file name that does not exist. Make sure that you supply the complete file name. Turbo Assembler does not presume any default extension for the file name. You've probably run out of space on the disk where you asked the cross-reference file to be written.

Can't generate instance of type

You attempted to generate an instance of a named type that does not have an instance. For example,

```
foo typedef near
   foo ?                 ;NEARs have no instance.
```

Can't locate file ___

You have specified a file name with the **INCLUDE** directive that can't be found.

An INCLUDE file could not be located. Make sure that the name contains any necessary disk letter or directory path.

Can't make variable public

The variable is already declared in such a way that it can't be made public. For example,

```
EXTRN  ABC:NEAR
PUBLIC ABC                  ;error, already EXTRN
```

Can't override ES segment

The current statement specifies an override that can't be used with that instruction. For example,

```
stos  DS:BYTE PTR[di]
```

Here, the **STOS** instruction can only use the **ES** register to access the destination address.

Can't subtract dissimilar relative quantities

An expression subtracts two addresses that can't be subtracted from each other, such as when they are each in a different segment:

```
SEG1 SEGMENT
A:
SEG1 ENDS
SEG2 SEGMENT
B:
     mov  ax,B-A      ;illegal, A and B in different segments
SEG2 ENDS
```

Can't use macro name in expression

A macro name was encountered as part of an expression. For example,

```
MyMac    MACRO
     ENDM
     mov ax,MyMac     ;wrong!
```

Can't use this outside macro

You have used a directive outside a macro definition that can only be used inside a macro definition. This includes directives like **ENDM** and **EXITM**. For example,

```
DATA SEGMENT
     ENDM             ;error, not inside macro
```

Code or data emission to undeclared segment

A statement that generated code or data is outside of any segment declared with the **SEGMENT** directive. For example,

```
;First line of file
     inc  bx          ;error, no segment
     END
```

You can only emit code or data from within a segment.

Constant assumed to mean immediate constant

This warning appears if you use an expression such as [0], which under MASM is interpreted as simply 0. For example,

```
mov  ax,[0]   ;means mov ax,0 NOT mov ax,DS:[0]
```

Constant too large

You have entered a constant value that is properly formatted, but is too large. For example, you can only use numbers larger than 0ffffh when you have enabled 386 or i486 instructions with the **.386/.386P** or **.486/.486P** directives.

CS not correctly assumed

A near **CALL** or **JMP** instruction can't have as its target an address in a different segment. For example,

```
SEG1  SEGMENT
LAB1  LABEL NEAR
SEG1  ENDS
SEG2  SEGMENT
      jmp LAB1        ;error, wrong segment
SEG2  ENDS
```

This error only occurs in MASM mode. Ideal mode correctly handles this situation.

CS override in protected mode

The current instruction requires a CS override, and you are assembling instructions for the 80286, 386, or i486 in protected mode (**P286P**, **P386P**, or **P486** directives). For example,

```
      P286P
      .CODE
CVAL DW   ?
      mov  CVAL,1      ;generates CS override
```

The **/P** command-line option enables this warning. When running in protected mode, instructions with CS overrides won't work without you taking special measures.

CS unreachable from current segment

When defining a code label using colon (:), **LABEL** or **PROC**, the **CS** register is not assumed to either the current code segment or to a group that contains the current code segment. For example,

```
PROG1    SEGMENT
     ASSUME cs:PROG2
START:              ;error, bad CS assume
```

This error only occurs in MASM mode. Ideal mode correctly handles this situation.

Data or code written to uninitialized segment

You have inadvertently written initialized code or data to an uninitialized segment. For example,

```
.data?
msg db 'Hello',0      ; error, uninitialized segment
```

Declaration needs name

You have used a directive that needs a symbol name, but none has been supplied. For example,

```
PROC                ;error, PROC needs a name
     ret
ENDP
```

You must always supply a name as part of a **SEGMENT**, **PROC**, or **STRUC** declaration. In MASM mode, the name precedes the directive; in Ideal mode, the name comes after the directive.

Directive not allowed inside structure definition

You have used a directive inside a **STRUC** definition block that can't be used there. For example,

```
X STRUC
MEM1 DB  ?
     ORG $+4        ;error, can't use ORG inside STRUC
MEM2 DW  ?
ENDS
```

Also, when declaring nested structures, you cannot give a name to any that are nested. For example,

```
FOO STRUC
    FOO2 STRUC      ;can't name inside
    ENDS
ENDS
```

If you want to use a named structure inside another structure, you must first define the structure and then use that structure name inside the second structure.

Duplicate dummy argument: __

A macro defined with the **MACRO** directive has more than one dummy parameter with the same name. For example,

```
XYZ  MACRO A,A        ;error, duplicate dummy name
     DB A
     ENDM
```

Each dummy parameter in a macro definition must have a different name.

ELSE or ENDIF without IF

An **ELSE** or **ENDIF** directive has no matching **IF** directive to start a conditional assembly block. For example,

```
BUF  DB 10 DUP (?)
     ENDIF            ;error, no matching IFxxx
```

Error writing to listing file

You've probably run out of space on the disk where you asked the listing file to be written.

Error writing to object file

You've probably run out of space on the disk where you asked the object file to be written.

Expecting METHOD keyword

The extended structure statement for defining objects expects the keyword **METHOD** after the parent object.

Expecting offset quantity

An expression expected an operand that referred to an offset within a segment, but did not encounter the right sort of operand. For example,

```
CODE SEGMENT
     mov  ax,LOW CODE
CODE ENDS
```

Expecting offset or pointer quantity

An expression expected an operand that referred to an offset within a specific segment, but did not encounter the right sort of operand. For example,

```
CODE SEGMENT
     mov  ax,SEG CODE    ;error, code is a segment not
                         ; a location within a segment
CODE ENDS
```

Expecting pointer type

The current instruction expected an operand that referenced memory. For example,

```
les di,4        ;no good, 4 is a constant
```

Expecting record field name

You used a **SETFIELD** or **GETFIELD** instruction without a field name following it.

Expecting register ID

The **USES** part of the **CALL..METHOD** expects register name(s).

Expecting scalar type

An instruction operand or operator expects a constant value. For example,

```
BB   DB   4
     rol  ax,BB        ;ROL needs constant
```

Expecting segment or group quantity

A statement required a segment or group name, but did not find one. For example,

```
DATA SEGMENT
     ASSUME  ds:FOO ;error, FOO is not group or segment
                    ;name
FOO  DW  0
DATA ENDS
```

Extra characters on line

A valid expression was encountered, but there are still characters left on the line. For example,

```
ABC = 4 shl 3 3   ;missing operator between 3 and 3
```

This error often happens in conjunction with another error that caused the expression parser to lose track of what you intended to do.

File not found

The source file name you specified on the command line does not exist. Make sure you typed the name correctly, and that you included any necessary drive or path information if the file is not in the current directory.

File was changed or deleted while assembly in progress

Another program, such as a pop-up utility, has changed or deleted the file after Turbo Assembler opened it. Turbo Assembler can't reopen a file that was previously opened successfully.

Forward reference needs override

An expression containing a forward-referenced variable resulted in more code being required than Turbo Assembler anticipated. This can happen either when the variable is unexpectedly a far address for a **JMP** or **CALL** or when the variable requires a segment override in order to access it. For example,

```
       ASSUME cs:DATA
       call A            ;presume near call
A PROC FAR               ;oops, it's far
       mov ax,MEMVAR     ;doesn't know it needs override
DATA   SEGMENT
MEMVAR DW  ?             ;oops, needs override
```

Correct this by explicitly supplying the segment override or **FAR** override.

Global type doesn't match symbol type

This warning is given when a symbol is declared using the **GLOBAL** statement and is also defined in the same module, but the type specified in the **GLOBAL** and the actual type of the symbol don't agree.

ID not member of structure

In Ideal mode, you have specified a symbol that is not a structure member name after the period (.) structure member operator. For example,

```
      IDEAL
STRUC DEMO
      DB  ?
ENDS
COUNT     DW 0
      mov ax,[(DEMO bx).COUNT] ;COUNT not part of structure
```

You must follow the period with the name of a member that belongs to the structure name that precedes the period.

This error often happens in conjunction with another error that caused the expression parser to lose track of what you intended to do.

Illegal forward reference

A symbol has been referred to that has not yet been defined, and a directive or operator requires that its argument not be forward-referenced. For example,

```
IF MYSYM       ;error, MYSYM not defined yet
      ;
ENDIF
MYSYM EQU 1
```

Forward references may not be used in the argument to any of the **IFxxx** directives, nor as the count in a **DUP** expression.

Illegal immediate

An instruction has an immediate (constant) operand where one is not allowed. For example,

```
mov  4,al
```

Illegal indexing mode

An instruction has an operand that specifies an illegal combination of registers. For example,

```
mov  al,[si+ax]
```

On all processors except the 386, the only valid combinations of index registers are: BX, BP, SI, DI, BX+SI, BX+DI, BP+SI, BP+DI.

Illegal instruction

A source line starts with a symbol that is neither one of the known directives nor a valid instruction mnemonic.

```
move ax,4        ;should be "MOV"
```

Illegal instruction for currently selected processor(s)

A source line specifies an instruction that can't be assembled for the current processor. For example,

```
.8086
push  1234h      ;no immediate push on 8086
```

When Turbo Assembler first starts assembling a source file, it generates instructions for the 8086 processor, unless told to do otherwise.

If you wish to use the extended instruction mnemonics available on the 186/286/386 processors, you must use one of the directives that enables those instructions (**P186**, **P286**, **P386**).

Illegal local argument

The **LOCAL** directive inside a macro definition has an argument that is not a valid symbol name. For example,

```
X     MACRO
      LOCAL 123         ;not a symbol
      ENDM
```

Illegal local symbol prefix

The argument to the **LOCALS** directive specifies an invalid start for local symbols. For example,

```
LOCALS  XYZ         ;error, not 2 characters
```

The local symbol prefix must be exactly two characters that themselves are a valid symbol name, such as __, @@, and so on (the default is @@).

Illegal macro argument

A macro defined with the **MACRO** directive has a dummy argument that is not a valid symbol name. For example,

```
X     MACRO 123         ;invalid dummy argument
      ENDM
```

Illegal memory reference

An instruction has an operand that refers to a memory location, but a memory location is not allowed for that operand. For example,

```
mov  [bx],BYTE PTR A    ;error, can't move from MEM to MEM
```

Here, both operands refer to a memory location, which is not a legal form of the **MOV** instruction. On the 80x86 family of processors, only one of the operands to an instruction can refer to a memory location.

Illegal number

A number contains one or more characters that are not valid for that type of number. For example,

```
Z = 0ABCGh
```

Here, *G* is not a valid letter in a hexadecimal number.

Illegal origin address

You have entered an invalid address to set the current segment location ($). You can enter either a constant or an expression using the location counter ($), or a symbol in the current segment.

Illegal override in structure

You have attempted to initialize a structure member that was defined using the **DUP** operator. You can only initialize structure members that were declared without **DUP**.

Illegal override register

A register other than a segment register (CS, DS, ES, SS, and on the 386, FS and GS) was used as a segment override, preceding the colon (:) operator. For example,

```
mov dx:XYZ,1            ;DX not a segment register
```

Illegal radix

The number supplied to the **.RADIX** directive that sets the default number radix is invalid. For example,

```
.RADIX  7               ;no good
```

The radix can only be set to one of 2, 8, 10, or 16. The number is interpreted as decimal no matter what the current default radix is.

Illegal register for instruction
An illegal register was used as the source of a **SETFIELD** instruction or the destination of a **GETFIELD** instruction.

Illegal register multiplier
You have attempted to multiply a register by a value, which is not a legal operation; for example,

```
mov ax*3,1
```

The only context where you can multiply a register by a constant expression is when specifying a scaled index operand on the 386 processor.

Illegal segment address
This error appears if an address greater than 65,535 is specified as a constant segment address; for example,

```
FOO SEGMENT AT 12345h
```

Illegal use of constant
A constant appears as part of an expression where constants can't be used. For example,

```
mov bx+4,5
```

Illegal use of register
A register name appeared in an expression where it can't be used. For example,

```
X = 4 shl ax  ;can't use register with SHL operator
```

Illegal use of segment register
A segment register name appears as part of an instruction or expression where segment registers cannot be used. For example,

```
add  SS,4            ;ADD can't use segment regs
```

Illegal USES register
You have entered an invalid register to push and pop as part of entering and leaving a procedure. The valid registers follow:

AX	CX	DS	ES
BX	DI	DX	SI

If you have enabled the 386 processor with the **.386** or **.386P** directive, you can use the 32-bit equivalents for these registers.

Illegal version ID
Occurs when an illegal version ID was selected in the **VERSION** statement or **/U** switch.

Illegal warning ID
You have entered an invalid three-character warning identifier. See the options discussed in Chapter 2 for a complete list of the allowed warning identifiers.

Instruction can be compacted with override
The code generated contains **NOP** padding, due to some forward-referenced symbol. You can either remove the forward reference or explicitly provide the type information as part of the expression. For example,

```
    jmp X          ;warning here
    jmp SHORT X    ;no warning
X:
```

Insufficient memory to process command line

You have specified a command line that is either longer than 64K or can't be expanded in the available memory. Either simplify the command line or run Turbo Assembler with more memory free.

Internal error

This message should never happen during normal operation of Turbo Assembler. Save the file(s) that caused the error and report it to Borland's Technical Support department.

Invalid command line

The command line that you used to start Turbo Assembler is badly formed. For example,

```
TASM ,MYFILE
```

does not specify a source file to assemble. See Chapter 2 for a complete description of the Turbo Assembler command line.

Invalid model type

The model directive has an invalid memory model keyword. For example,

```
.MODEL GIGANTIC
```

Valid memory models are tiny, small, compact, medium, large, and huge.

Invalid number after __

You have specified a valid command-line switch (option), but have not supplied a valid numeric argument following the switch. See Chapter 2 for a discussion of the command-line options.

Invalid operand(s) to instruction

The instruction has a combination of operands that are not permitted. For example,

```
fadd  ST(2),ST(3)
```

Here, *FADD* can only refer to one stack register by name; the other must be the stack top.

Labels can't start with numeric characters

You have entered a symbol that is neither a valid number nor a valid symbol name, such as *123XYZ*.

Language differs from procedure type

You attempted to use a different language than what was contained in the procedure type declaration. For example,

```
foo proctype windows pascal :word
fooproc proc foo al:word
   :
endp

call fooproc c,ax          ;Language doesn't match.
```

Language doesn't support variable-length arguments

You specified a variable-length stack frame with a language that doesn't support it. For example,

```
foo proctype pascal :word, :unknown   ;Pascal can't have
                                      ;variable arguments.
```

Line too long—truncating

The current line in the source file is longer than 255 characters. The excess characters will be ignored.

Location counter overflow

The current segment has filled up, and subsequent code or data will overwrite the beginning of the segment. For example,

```
ORG 0FFF0h
ARRAY  DW 20 DUP (0)    ;overflow
```

Method CALL requires object name

The **CALL..METHOD** statement cannot obtain the object type from this instance pointer. You must specify the object name.

Missing argument list

An **IRP** or **IRPC** repeat block directive does not have an argument to substitute for the dummy parameter. For example,

```
IRP X           ;no argument list
      DB X
ENDM
```

IRP and **IRPC** must always have both a dummy parameter and an argument list.

Missing argument or <

You forgot the angle brackets or the entire expression in an expression that requires them. For example,

```
ifb    ;needs an argument in <>s
```

Missing argument size variable

An **ARG** or **LOCAL** directive does not have a symbol name following the optional = at the end of the statement. For example,

```
ARG   A:WORD,B:DWORD=      ;error, no name after =
LOCAL X:TBYTE=             ;same error here
```

ARG and **LOCAL** must always have a symbol name if you have used the optional equal sign (=) to indicate that you want to define a size variable.

Missing COMM ID

A **COMM** directive does not have a symbol name before the type specifier. For example,

```
COMM  NEAR  ;error, no symbol name before "NEAR"
```

COMM must always have a symbol name before the type specifier, followed by a colon (:) and then the type specifier.

Missing dummy argument

An **IRP** or **IRPC** repeat block directive does not have a dummy parameter. For example,

```
RP              ;no dummy parameter
    DB X
ENDM
```

IRP and **IRPC** must always have both a dummy parameter and an argument list.

Missing end quote

A string or character constant did not end with a quote character. For example,

```
DB   "abc       ;missing " at end of ABC
mov  al,'X      ;missing ' after X
```

You should always end a character or string constant with a quote character matching the one that started it.

Missing macro ID

A macro defined with the **MACRO** directive has not been given a name. For example,

```
MACRO              ;error, no name
      DB A
      ENDM
```

Macros must always be given a name when they are defined.

Missing module name

You have used the **NAME** directive but you haven't supplied a module name after the directive. Remember that the **NAME** directive only has an effect in Ideal mode.

Missing or illegal language ID

You have entered something other than one of the allowed language identifiers after the **.MODEL** directive. See Chapter 7 for a complete description of the **.MODEL** directive.

Missing or illegal type specifier

A statement that needed a type specifier (like **BYTE**, **WORD**, and so on) did not find one where expected. For example,

```
RED  LABEL XXX     ;error, "XXX" is not a type specifier
```

Missing table member ID

A **CALL..METHOD** statement was missing the method name after the **METHOD** keyword.

Missing term in list

In Ideal mode, a directive that can accept multiple arguments (**EXTRN**, **PUBLIC**, and so on) separated by commas does not have an argument after one of the commas in the list. For example,

```
EXTRN XXX:BYTE,,YYY:WORD
```

In Ideal mode, all argument lists must have their elements separated by precisely one comma, with no comma at the end of the list.

Missing text macro

You have not supplied a text macro argument to a directive that requires one. For example,

```
NEWSTR  SUBSTR          ;ERROR - SUBSTR NEEDS ARGUMENTS
```

Model must be specified first

You used one of the simplified segmentation directives without first specifying a memory model. For example,

```
.CODE        ;error, no .MODEL first
```

You must always specify a memory model using the **.MODEL** directive before using any of the other simplified segmentation directives.

Module is pass-dependent—compatibility pass was done

This warning occurs if a pass-dependent construction was encountered and the **/m** command-line switch was specified. A MASM-compatible pass was done.

You put a symbol name after a directive, and the symbol name should come first. For example,

```
STRUC ABC          ;error, ABC must come before STRUC
```

Since Ideal mode expects the name to come after the directive, you will encounter this error if you try to assemble Ideal mode programs in MASM mode.

Near jump or call to different CS

This error occurs if the user attempts to perform a **NEAR CALL** or **JMP** to a symbol that's defined in an area where CS is assumed to a different segment.

Need address or register

An instruction does not have a second operand supplied, even though there is a comma present to separate two operands; for example,

```
mov ax,              ;no second operand
```

Need angle brackets for structure fill

A statement that allocates storage for a structure does not specify an initializer list. For example,

```
STR1 STRUC
M1   DW   ?
M2   DD   ?
     ENDS
     STR1          ;no initializer list
```

Need colon

An **EXTRN, GLOBAL, ARG**, or **LOCAL** statement is missing the colon after the type specifier (**BYTE**, **WORD**, and so on). For example,

```
EXTRN X BYTE,Y:WORD      ;X has no colon
```

Need expression

An expression has an operator that is missing an operand. For example,

```
X = 4 + * 6
```

Need file name after INCLUDE

An **INCLUDE** directive did not have a file name after it. For example,

```
INCLUDE      ;include what?
```

In Ideal mode, the file name must be enclosed in quotes.

Need left parenthesis

A left parenthesis was omitted that is required in the expression syntax. For example,

```
DB 4 DUP 7
```

You must always enclose the expression after the **DUP** operator in parentheses.

Need method name

The **CALL..METHOD** statement requires a method name after the **METHOD** keyword.

Need pointer expression

This error only occurs in Ideal mode and indicates that the expression between brackets ([ ]) does not evaluate to a memory pointer. For example,

```
mov ax,[WORD PTR]
```

In Ideal mode, you must always supply a memory-referencing expression between the brackets.

Need quoted string

You have entered something other than a string of characters between quotes where it is required. In Ideal mode, several directives require their argument to be a quoted string. For example,

```
IDEAL
DISPLAY "ALL DONE"
```

Need register in expression

You have entered an expression that does not contain a register name where one is required.

Need right angle bracket

An expression that initializes a structure, union, or record does not end with a > to match the < that started the initializer list. For example,

```
MYSTRUC STRUCNAME <1,2,3
```

Need right curly bracket

Occurs during a named structure, table, or record fill when a '}' is expected but not found.

Need right parenthesis

An expression contains a left parenthesis, but no matching right parenthesis. For example,

```
X = 5 * (4 + 3
```

You must always use left and right parentheses in matching pairs.

Need right square bracket

An expression that references a memory location does not end with a] to match the [that started the expression. For example,

```
mov   ax,[si      ;error, no closing ] after SI
```

You must always use square brackets in matching pairs.

Need stack argument

A floating-point instruction does not have a second operand supplied, even though there is a comma present to separate two operands. For example,

```
fadd ST,
```

Need structure member name

In Ideal mode, the period (.) structure member operator was followed by something that was not a structure member name. For example,

```
      IDEAL
STRUC DEMO
      DB   ?
ENDS
COUNT      DW 0
      mov  ax,[(DEMO bx).]
```

You must always follow the period operator with the name of a member in the structure to its left.

Not expecting group or segment quantity

You have used a group or segment name where it can't be used. For example,

```
CODE SEGMENT
     rol  ax,CODE      ;error, can't use segment name here
```

One non-null field allowed per union expansion

When initializing a union defined with the **UNION** directive, more than one value was supplied. For example,

```
U      UNION
       DW   ?
       DD   ?
ENDS
UINST U <1,2>      ;error, should be <?,2> or <1,?>
```

A union can only be initialized to one value.

Only one startup sequence allowed

This error appears if you have more than one **.STARTUP** or **STARTUPCODE** statement in a module.

Open conditional

The end of the source file has been reached as defined with the **END** directive, but a conditional assembly block started with one of the **IFxxx** directives has not been ended with the **ENDIF** directive. For example,

```
IF BIGBUF
END            ;no ENDIF before END
```

This usually happens when you type **END** instead of **ENDIF** to end a conditional block.

Open procedure

The end of the source file has been reached as defined with the **END** directive, but a procedure block started with the **PROC** directive has not been ended with the **ENDP** directive. For example,

```
MYFUNC PROC
   END            ;no ENDIF before ENDP
```

This usually happens when you type **END** instead of **ENDP** to end a procedure block.

Open segment

The end of the source file has been reached as defined with the **END** directive, but a segment started with the **SEGMENT** directive has not been ended with the **ENDS** directive. For example,

```
DATA SEGMENT
     END          ;no ENDS before END
```

This usually happens when you type **END** instead of **ENDS** to end a segment.

Open structure definition

The end of the source file has been reached as defined with the **END** directive, but a structure started with the **STRUC** directive has not been ended with the **ENDS** directive. For example,

```
X     STRUC
VAL1 DW    ?
     END          ;no ENDS before it
```

This usually happens when you type **END** instead of **ENDS** to end a structure definition.

Operand types do not match
The size of an instruction operand does not match either the other operand or one valid for the instruction; for example,

```
ABC  DB   5
   :
     mov  ax,ABC
```

Operation illegal with procedure type
You used the structure member operator on an expression whose type is a procedure. For example,

```
foo proctype pascal :word
   :
   mov ax,[foo ptr [bx]).member     ;Things of type FOO
                                   ;have no members
```

Operation illegal with static table member
A '.' operator was used to obtain the address of a static table member. This is illegal.

Out of hash space
The hash space has one entry for each symbol you define in your program. It starts out allowing 16,384 symbols to be defined, as long as Turbo Assembler is running with enough free memory. If your program has more than this many symbols, use the **/KH** command-line option to set the number of symbol entries you need in the hash table.

Out of memory
You don't have enough free memory for Turbo Assembler to assemble your file.

If you have any TSR (RAM-resident) programs installed, you can try removing them from memory and try assembling your file again. You may have to reboot your system in order for memory to be properly freed.

Another solution is to split the source file into two or more source files, or rewrite portions of it so that it requires less memory to assemble. You can also use shorter symbol names, reduce the number of comments in macros, and reduce the number of forward references in your program.

Out of string space
You don't have enough free memory for symbol names, file names, forward-reference tracking information, and macro text. A maximum of 512K is allowed, and your module has exceeded this maximum.

Pass-dependent construction encountered
The statement may not behave as you expect, due to the one-pass nature of Turbo Assembler. For example,

```
IF1
              ;Happens on assembly pass
ENDIF
IF2
              ;Happens on listing pass
ENDIF
```

Most constructs that generate this error can be re-coded to avoid it, often by removing forward references.

Pointer expression needs brackets

In Ideal mode, the operand contained a memory-referencing symbol that was not surrounded by brackets to indicate that it references a memory location. For example,

```
B  DB   0
   mov  al,B       ;warning, Ideal mode needs [B]
```

Since MASM mode does not require the brackets, this is only a warning.

Positive count expected

A **DUP** expression has a repeat count less than zero. For example,

```
BUF  -1 DUP (?)      ;error, count < 0
```

The count preceding a **DUP** must always be 1 or greater.

Procedure has too many arguments

A procedure was declared with too many arguments. For example,

```
footype PROCTYPE pascal :word, :dword

foo proc footype
arg a1:word,a2:dword,a3:word
   nop                ;too many arguments were declared for
                      ;for this proc
endp
```

Procedure needs more arguments

A procedure was declared with too few arguments. For example,

```
footype PROCTYPE pascal :word , :dword

foo proc footype
arg a1:word
   nop             ;Needs a DWORD argument somewhere too.
   ret
endp
```

Record field too large

When you defined a record, the sum total of all the field widths exceeded 32 bits. For example,

```
AREC RECORD   RANGE:12,TOP:12,BOTTOM:12
```

Record member not found

A record member was specified in a named record fill that was not part of the specified record.

Recursive definition not allowed for EQU

An **EQU** definition contained the same name that you are defining within the definition itself. For example,

```
ABC EQU TWOTIMES ABC
```

Register must be AL or AX

An instruction which requires one operand to be the AL or AX register has been given an invalid operand. For example,

```
IN  CL,dx      ;error, "IN" must be to AL or AX
```

Register must be DX

An instruction which requires one operand to be the DX register has been given an invalid operand. For example,

```
IN  AL,cx      ;error, must be DX register instead of CX
```

Relative jump out of range by __ bytes

A conditional jump tried to reference an address that was greater than 128 bytes before or 127 bytes after the current location. If this is in a **USE32** segment, the conditional jump can reference between 32,768 bytes before and 32,767 bytes after the current location.

Relative quantity illegal

An instruction or directive has an operand that refers to a memory address in a way that can't be known at assembly time, and this is not allowed. For example,

```
DATA SEGMENT PUBLIC
X    DB   0
     IF   OFFSET X GT 127    ;not known at assemble time
```

Reserved word used as symbol

You have created a symbol name in your program that Turbo Assembler reserves for its own use. Your program will assemble properly, but it is good practice not to use reserved words for your own symbol names.

Rotate count must be constant or CL

A shift or rotate instruction has been given an operand that is neither a constant nor the CL register. For example,

```
rol  ax,DL      ;error, can't use DL as count
```

You can only use a constant value or the CL register as the second operand to a rotate or shift instruction.

Rotate count out of range

A shift or rotate instruction has been given a second operand that is too large. For example,

```
.8086
shl   DL,3     ;error, 8086 can only shift by 1
.286
ror   ax,40    ;error, max shift is 31
```

The 8086 processor only allows a shift count of 1, but the other processors allow a shift count up to 31.

Segment alignment not strict enough

The align boundary value supplied is invalid. Either it is not a power of 2, or it specifies an alignment stricter than that of the align type in the **SEGMENT** directive. For example,

```
DATA SEGMENT PARA
     ALIGN 32        ;error, PARA is only 16
     ALIGN 3         ;error, not power of 2
```

Segment attributes illegally redefined

A **SEGMENT** directive reopen a segment that has been previously defined, and tries to give it different attributes. For example,

```
DATA SEGMENT BYTE PUBLIC
DATA ENDS
```

```
DATA SEGMENT PARA ;error, previously had byte alignment
DATA ENDS
```

If you reopen a segment, the attributes you supply must either match exactly or be omitted entirely. If you don't supply any attributes when reopening a segment, the old attributes will be used.

Segment name is superfluous
This warning appears with a **.CODE** *xxx* statement, where the model specified doesn't allow more than code segment.

String too long
You have built a quoted string that is longer than the maximum allowed length of 255.

Style differs from procedure type
You attempted to use a different language style than the declaration of the procedure type contained. For example,

```
foo proctype windows pascal :word
fooproc proc foo a1:word
   :
endp

call fooproc normal pascal,ax        ;Style doesn't match.
```

Symbol already defined: __
The indicated symbol has previously been declared with the same type. For example,

```
BB  DB  1,2,3
BB  DB  ?          ;error, BB already defined
```

Symbol already different kind
The indicated symbol has already been declared before with a different type. For example,

```
BB  DB  1,2,3
BB  DW  ?          ;error, BB already a byte
```

Symbol has no width or mask
The operand of a **WIDTH** or **MASK** operator is not the name of a record or record field. For example,

```
B  DB  0
   mov ax,MASK B      ;B is not a record field
```

Symbol is not a segment or already part of a group
The symbol has either already been placed in a group or it is not a segment name. For example,

```
DATA     SEGMENT
DATA     ENDS
DGROUP     GROUP DATA
DGROUP2    GROUP DATA     ;error, DATA already belongs to
                          ;DGROUP
```

Text macro expansion exceeds maximum line length
This error occurs when expansion of a text macro causes the maximum allowable line length to be exceeded.

Too few arguments to procedure

You called a procedure using too few arguments. For example,

```
foo proctype pascal :word, :dword
fooproc proc foo a1:word, a2:dword
   :
endp
call fooproc,ax              ;Too few arguments.
```

Too few operands to instruction

The instruction statement requires more operands than were supplied. For example,

```
add  ax      ;missing second arg
```

Too many arguments to procedure

You called a procedure using too many arguments. For example,

```
foo proctype pascal :word, :dword
fooproc proc foo a1:word, a2:dword
   :
endp
call fooproc,ax,bx cx,dx              ;Too many arguments.
```

Too many errors found

Turbo Assembler has stopped assembling your file because it contained so many errors. You may have made a few errors that have snowballed. For example, failing to define a symbol that you use on many lines is really a single error (failing to define the symbol), but you will get an error message for each line that referred to the symbol.

Turbo Assembler will stop assembling your file if it encounters a total of 100 errors or warnings.

Too many errors or warnings

No more error messages will be displayed. The maximum number of errors that will be displayed is 100; this number has been exceeded. Turbo Assembler continues to assemble and prints warnings rather than error messages.

Too many initial values

You have supplied too many values in a structure or union initialization. For example,

```
XYZ   STRUC
A1 DB ?
A2 DD ?
XYZ   ENDS
ANXYZ XYZ   <1,2,3>      ;error, only 2 members in XYZ
```

You can supply fewer initializers than there are members in a structure or union, but never more.

Too many register multipliers in expression

An 386 scaled index operand had a scale factor on more than one register. For example,

```
mov EAX,[2*EBX+4*EDX]      ;too many scales
```

Too many registers in expression

The expression has more than one index and one base register. For example,

```
mov ax,[BP+SI+DI]       ;can't have SI and DI
```

Too many USES registers

You specified more than 8 **USES** registers for the current procedure.

Trailing null value assumed

A data statement like **DB**, **DW**, and so on, ends with a comma. TASM treats this as a null value. For example,

```
db 'hello',13,10,   ;same as ...,13,10,?
```

Undefined symbol

The statement contains a symbol that wasn't defined anywhere in the source file.

Unexpected end of file (no END directive)

The source file does not have an **END** directive as its last statement. All source files must have an **END** statement.

Unknown character

The current source line contains a character that is not part of the set of characters that make up Turbo Assembler symbol names or expressions. For example,

```
add  ax,!1      ;error, exclamation is illegal character
```

Unmatched ENDP: __

The **ENDP** directive has a name that does not match the **PROC** directive that opened the procedure block. For example,

```
ABC PROC
XYZ ENDP      ;error, XYZ should be ABC
```

Unmatched ENDS: __

The **ENDS** directive has a name that does not match either the **SEGMENT** directive that opened a segment or the **STRUC** or **UNION** directive that started a structure or union definition. For example,

```
ABC STRUC
XYZ ENDS      ;error, XYZ should be ABC
DATA SEGMENT
CODE ENDS     ;error, code should be DATA
```

User-generated error

An error has been forced by one of the directives, which then forces an error. For example,

```
.ERR     ;shouldn't get here
```

USES has no effect without *language*

This warning appears if you specify a **USES** statement when no *language* is in effect.

Value out of range

The constant is a valid number, but it is too large to be used where it appears. For example,

```
DB 400
```

Variable length parameter must be last parameter

If a variable-length parameter is present, it must be the last parameter. For example,

```
foo proctype pascal :word, :unknown, :word   ;Not allowed.
```

Index

80287 coprocessor
  .287 directive *86*
  P287 directive *86*
80387 coprocessor
  .387 directive *86*
  P387 directive *86*
.8086 directive *83*
.8087 directive *86*
80186 processor
  .186 directive *83*
  P186 directive *83*
80286 processor
  .286 directive *83*
  .286C directive *83*
  .286P directive *83*
  protected mode *25*
80386 processor *See also* 386 processor
  .386 directive *83*
  .386C directive *83*
  .386P directive *83*
  loop instructions for *161*
  P386 directive *83*
  P386N directive *83*
  P386P directive *83*
  protected mode *25*
80486 processor
  .486 directive *83*
  .486C directive *83*
  .486P directive *83*
  P486 directive *83*
  P486N directive *83*
  protected mode *25*
80487 processor
  .487 directive *83*
  P487 directive *83*
{ } (brace) initializer *150, 155*
  records and *153*
< > (bracket) initializer *151*
  nested structures/unions and *152, 157*
  records and *154*
< > (brackets)
  literal string *183*
  macros and *175*
8087 coprocessor *18, 87*
  .8087 directive *86*
  Borland C++ and *241*
  P8087 directive *86*
.186 directive *83*
.286 directive *83*
.287 directive *86*
.386 directive *83*
.387 directive *86*
.486 directive *83*
.487 directive *83*
8086 processor
  .8086 directive *83*
  P8086 directive *83*
  PUSHing constants *163*
  segments and *89*
.286C directive *83*
.386C directive *83*
.486C directive *83*
.286P directive *83*
.386P directive *83*
.486P directive *83*
386 processor
  protected mode *25*
[] (square brackets)
  describing address contents *80*
  Ideal mode *36*
  MASM mode *36*
;; comment character *179*
:: directive *124*
= (equals) directive *43*
. (period) character
  Ideal mode *36*
@@ symbol *143*
+ addition operator *80*
@32Bit symbol *93*
! character *183*
& character, in macros *178*
= directive *17*
% expression evaluation character *184*
% immediate macro directive *188*
? keyword *111, 132*
  as initial value *150*

\ line continuation character *182*
@-sign *30*
= sign, argument lists and *132*
$ symbol *119*
? symbol *145*

A

/a option *16, 26*
address expressions *See* expressions
address subtypes
  complex *68*
  setting *75*
address subtypes of symbols
  distance parameter and *69*
addresses, calculating *78*
ALIAS *208*
alias values *42*
ALIGN directive *109, 122*
.ALPHA directive *102*
ALPHA directive *16, 26*
ancestor virtual methods *60*
ARG directive *131*
  Borland C++ and *239*
arguments
  BYTE *132*
  names (scope of) *133*
  substitution (defined) *178*
arithmetic operators *74, 80*
.ASM files *1, 14*
assembling
  first program *10*
  multiple passes *127*
  number of passes *21*
ASSUME directive *100*
at-sign *30*
attribute values of segments *97*
attributes
  segment
    access *99*
    alignment *98*
    class *98*
    combination *97*
    size *98*
  values of segments *97*

B

@B symbol *143*
/b option *16*
\ comment character *40*
Backus-Naur form (BNF) grammar *70*
%BIN directive *215, 216*
binary coded decimal (BCD) encoding
  DT directive and *149*
bit-field records, defining *106*
bit shift operators *75*
BIX, JOIN BORLAND *5*
block scoping of symbols, defined *142*
books
  assembly language *11*
Boolean algebra and operators *74*
  symbol expressions and *197*
Borland
  contacting *4*
Borland, contacting *4-5*
Borland C++
  ARG directive and *239*
  assembler modules in *18*
  case sensitivity *23, 230*
  code segment *223*
  data types *231*
  external symbols *232*
  floating-point emulation *18*
  linking to *249*
  LOCAL directive and *237*
  memory models *223*
  parameter passing *233*
  Pascal calling conventions *248*
  public functions and *229*
  register preservation *240*
  returning values *241*
  segment directives and *223*
  structures *241*
BOUND instruction
  Ideal mode *37*
buffers, size of *16*
bulletin board, Borland *5*
BYTE arguments *132*
byte values *146*

C

C++ *See* Borland C++

/c option *17, 214*
calculating addresses *78*
CALL..METHOD instruction *57, 58, 61*
  near tables and *173*
CALL instruction *170*, *See also* CALL..METHOD
  extended *170*
case sensitivity
  assembler routines and *23*
  Borland C++ *230*
CATSTR directive *176*
code-checking *25*
.CODE directive *94*
@code symbol *95*
code generation, intelligent (directives for) *159*
code segments *90*
  Borland C++ *223*
CODESEG directive *94*
@CodeSize symbol *93*
: (colon) operator *78*
: operator *123*
COMM directive *206*
command files, indirect *30*
command-line options *13*
command-line syntax *14*
  help screen *18*
COMMENT directive *40*
comments
  ;; comment character *179*
  ; (semicolon) comment character *40*
  \ comment character *40*
  COMMENT directive *40*
  end of line *40*
  including in macros *179*
communal variables *206*
  MASM mode and *207*
comparison operators *75*
compatibility, MASM vs. Ideal mode *273*
compiler options *See* individual listings
complementary jumps *160*
complex types *114, 115*
compressing data, record data types and *106*
CompuServe, GO BORLAND *5*
conditional blocks (terminating) *See* GOTO directive
conditional directives
  assembly pass *200*
  defining blocks of code *193*
  expression *196*
  nesting *194*
  symbol-definition *197*
  text string *198*
  when to use *193*
conditional jumps *See* jumps, conditional
conditional list directives *212*
%CONDS directive *212*
configuration files *31*
.CONST directive *94*
CONST directive *94*
constants
  defined *65*
  in expressions *70*
  numeric *65*
  rotation counts and shift instructions *165*
  string *66*
constructor and destructor procedures
  writing *137*
coprocessor directives *86*
@Cpu symbol *85*
%CREF directive *214*
.CREF directive *214*
%CREFALL directive *214*
%CREFREF directive *214*
%CREFUREF directive *214*
cross-reference
  generating *15*
  in listing files *17*
  symbol information *213*
cross-reference utility *See* TCREF utility
CS override *25*
%CTLS directive *211*
@curseg symbol *95*
customer assistance *4-5*

D

/d option *17*
data
  allocating *145*
    constants and *148*
    WORDS *146*
  defining *146*
  initialized (defined) *145*
  repeated blocks *145*
  storage in memory *148*

structures *See* structures
uninitialized
defined *145*
specifying *145*
.DATA? directive *94*
.DATA directive *94*
@data symbol *95*
data structures *See* structures
data types
Borland C++ *231*
creating named *156*
creating record *153*
declaring record *106*
enumerated *105*
creating instances of *154*
initializing instances of *155*
multiline syntax *106*
pseudo ops and *106*
objects and *52*
record, multiline syntax for *107*
table *113*
multiline syntax *114*
with virtual methods *157*
DATASEG directive *94*
@DataSize symbol *93*
??date symbol *43*
DB directive *146*
DD directive *146, 148*
debugging information *29, 30*
%DEPTH directive *216*
derived objects *54*
development cycle, program *10*
DF directive *146*
directives *See also* individual listings
conditional *193*
assembly pass *200*
symbol-definition *197*
conditional expression *196*
coprocessor *86*
displaying assembly messages *46*
error-generation *195*
using symbol expressions *198*
include files *41*
module names *45*
processor *83*
program termination *45*
startup *19*
symbols *20*
DISPLAY directive *46*
distance parameter
complex subtypes and *69*
DOSSEG directive *102*
:: directive *124*
doubleword values *146*
DP directive *146*
DQ directive *146, 148*
DT directive *146, 148*
dummy arguments
defined *178*
in macros *182*
local *180*
recognizing *178*
types of *182*
DUP keyword *145*
DW directive *146*

E

/e option *18, 87*
ELSEIF directive *197*
ELSEIFB directive *199*
ELSEIFDEF directive *198*
ELSEIFDIF directive *199*
ELSEIFDIFI directive *199*
ELSEIFE directive *197*
ELSEIFIDN directive *199*
ELSEIFIDNI directive *199*
ELSEIFNB directive *199*
ELSEIFNDEF directive *198*
ELSEIFxxx directives *195*
EMUL directive *18, 87*
encoded real numbers *148*
END directive *45*
ENDM keyword *181*
ENDS directive *99, 109, 110*
ENTER instruction *161*
ENTERD instruction *161*
ENTERW instruction *161*
ENUM directive *105*
enumerated data types
creating instances of *154*
defined *105*
initializing instances of *155*
multiline syntax for *106*
pseudo ops and *106*

environment variables, MASM mode *274*
epilog code
  defined *128*
  how it works *129*
  languages and *128*
  NOLANGUAGE procedures and *129*
  register preservation and *134*
  specifying default style *92*
EQU directive *42, 43, 175*
  Ideal vs. MASM mode *34*
equal (=) directive *17*
equate substitutions *42*
.ERR1 directive *200*
.ERR2 directive *200*
ERR directive *196*
.ERR directive *196*
.ERRB directive *199*
.ERRDEF directive *198*
.ERRDIFI directive *199*
.ERRE directive *197*
.ERRIDN directive *199*
.ERRIDNI directive *199*
ERRIF1 directive *200*
ERRIF2 directive *200*
ERRIF directive *197*
ERRIFB directive *199*
ERRIFDEF directive *198*
ERRIFDIF directive *199*
ERRIFDIFI directive *199*
ERRIFE directive *197*
ERRIFIDN directive *199*
ERRIFIDNI directive *199*
ERRIFNB directive *199*
ERRIFNDEF directive *198*
.ERRNB directive *199*
.ERRNDEF directive *198*
.ERRNZ directive *197*
error-generation directives *195*
error messages *275-297*
  fatal *276*
  reporting *47*
  source file line display *29*
  warning *276*
ERRxxx directives *195*
EVEN directive *122*
EVENDATA directive *122*
.EXE files *1*
.EXIT directive *96*
EXITCODE directive *96*
EXITM directive *180*
expressions
  16-bit vs. 32-bit *81*
  BNF grammar and *70*
  byte values *80*
  constants in *70*
  contents of *69*
  determining characteristics *79*
  evaluation character *184*
  Ideal mode *36*
  obtaining type of *76*
  precision of *70*
  register names and *70*
  segment overrides of *77*
  setting address subtypes *75*
  structure names in *113*
  symbols in *70*
  syntax of *259*
  text macro names and *72*
  why to use *65*
extended CALL instruction *See* CALL..METHOD instruction
extern "C" *222*
external symbols *See* symbols, external
EXTRN directive *205*
  Borland C++ and *232*

F

@F symbol *143*
far data
  initialized *90*
  uninitialized *90*
far pointer values *147*
FAR procedures *126*
far returns, instructions for *162*
.FARDATA? directive *95*
@fardata? symbol *95*
FARDATA directive *95*
.FARDATA directive *95*
@fardata symbol *95*
fast immediate multiply instruction *See* FASTIMUL instruction
FASTIMUL instruction *169*
fatal error messages *276*
field value manipulation instructions *167*

file names *44*
  object-oriented programming format *63*
??filename symbol *44*
@FileName symbol *44*
files
  .ASM *14*
  assembly *44*
  configuration *31*
  indirect *30*
  listing *See* listing files
flag instructions, smart *167*
FLDENV instruction *170*
FLIPFLAG instruction *167*
floating-point
  emulation *18*
  Ideal vs. MASM mode *274*
  instructions *2*
floating-point instructions *See* coprocessor emulation directives
floating-point numbers *148*
FRSTOR instruction *170*
FSAVE instruction *170*
FSTENV instruction *170*

G

GEnie, BORLAND *5*
GETFIELD instruction *168*
GLOBAL directive *205*
  in .ASO files *63*
  objects and *53*
global symbols, include files and *205*
GOTO directive *180*
GREP utility *See* the README file
GROUP directive *99*
  Ideal vs. MASM mode *37*
groups
  assigning segments to *99*
  segment registers and *100*
  segments in Ideal mode *37*

H

H2ASH utility *See* the README file
/h option *18*
hardware and software requirements *2*
HELLO.ASM *10*
help
  displaying screen *18*
  online *8*
HIGH operator *80*

I

i486 processor
  protected mode *25*
/i option *19*
IDEAL directive *34*
Ideal mode *1*
  BOUND instruction *37*
  expressions *36*
  features *34*
  include files *42*
  operands *36*
  operators *36*
  predefined symbols *42*
  segment fixups *36*
  segment groups *37*
  speed *34*
  why to use *33, 34*
IF1 directive *194, 200*
IF2 directive *194, 200*
IF directive *194, 197*
IFB directive *194, 199*
IFDEF directive *194, 198*
IFDIF directive *194, 199*
IFDIFI directive *194, 199*
IFE directive *197*
IFIDN directive *194, 199*
IFIDNI directive *194, 199*
IFNB directive *186, 194, 199*
IFNDEF directive *194, 198*
IFxxx directives *193*
immediate macro directive (%) *188*
implied addition *80*
IMUL instruction *See* FASTIMUL instruction
%INCL directive *211*
INCLUDE directive *19, 41*
include files
  Ideal mode *42*
  setting path *19*
INCLUDELIB directive *207*
indirect command files *30*
information
  technical support *4*

inheritance
  defined *54*
  example of *60*
  objects and *117*
  previous object definitions *157*
  structure definitions and *111*
initialization code *95*
installation instructions *7*
instances
  creating object *62*
  creating structure or union *149*
  creating table *155*
  initializing instances *150*
  initializing table *155*
  initializing union or structure *150*
  named-type, creating *156*
  of objects *157*
  of records *153*
  virtual method table *60, 157*
  virtual method table (VMT) *56*
INSTR directive *177*
instruction set *See* individual listings
instruction size *See* size of instructions
intelligent code generation
  directives for *159*
@Interface symbol
  MODEL directive and *93*
IRET instruction
  expanded *162*
IRETW instruction *162*
IRP directive *187*
IRPC directive *187*

J

/j option *19*
jEMUL option *18*
JMP..METHOD instruction *62, 173*
JMP instruction *170*
jumps
  complementary *160*
  conditional *160*
JUMPS directive *160*

K

keyword precedence *266*
  Ideal mode *266*
  MASM mode *266*
keywords *22, See also* individual listings
  list of available *267*
/kh option *20*

L

/l option *17, 20, 23*
/la option *21, 129*
  language modifiers and *130*
LABEL directive *109, 124*
labels
  defining *123*
  external *229*
  local in MASM *143*
.LALL directive *189, 213*
language modifiers
  WINDOWS procedures and *130*
languages
  MODEL and *128*
  modifiers and Windows procedures *129*
  overriding default for procedures *128*
  preserving registers and *134*
  procedures and arguments *131*
  setting in CALL statement *250*
LARGE operator *81, 169*
  instructions it affects *170*
LARGESTACK directive *103*
LEAVE instruction *161*
LEAVED instruction *161*
LEAVEW instruction *161*
length of symbols *22*
LENGTH unary operator *72*
LFCOND directive *29*
.LFCOND directive *212*
LGDT instruction *170*
libraries (including) *See* TLINK
LIDT instruction *170*
line continuation character (\) *182*
line number information *29*
linker *See also* TLINK utility
  Borland C++ *233, 249*
  Phar Lap *99*
  segment ordering and *101*
%LINUM directive *216*
%LIST directive *210*
.LIST directive *210*
listing files *14*

/X command-line option and *212*
conditional listing directives *212*
cross-reference information *17*
cross-reference table and *209*
directives for *210*
false conditionals in *28*
format of *209*
format parameters *215*
generating *20*
high-level code in *21*
including files in *211*
including multiline macros *189*
macro expansions in *213*
symbol table and *211*
symbol table in *213*
symbol tables
suppressing *23*
why to use *209*
literal string brackets *183*
LOCAL directive *131*
Borland C++ and *237*
in macros *180*
local labels
in MASM *143*
LOCALS directive *135, 142*
location counter
creating address expressions *78*
defined *119*
directives for *120*
location counter symbol *119*
LOOP instruction *161*
loop instructions for 80386 processor *161*
LOOPD instruction *161*
LOOPDE instruction *161*
LOOPDNE instruction *161*
LOOPDNZ instruction *161*
LOOPDZ instruction *161*
LOOPE instruction *161*
LOOPNE instruction *161*
LOOPWE instruction *161*
LOOPWNE instruction *161*
LOOPWNZ instruction *161*
LOOPWZ instruction *161*
LOOPZ instruction *161*
LOW operator *80*
.LST files *15*

M

@Mptr member *157*
/m option *21, 127, 273*
macros
& character in *178*
body of *178*
controlling expansion *180*
defining new text *176*
defining substring *176*
deleting multiline *184*
dummy arguments within *182*
expansions in listing files *213*
including comments in *179*
invoking arguments with special characters *183*
invoking general multiline *182*
length of text *177*
manipulating string *176*
multiline *177*
defining general *181*
multiline expansions in listing file *189*
names in expressions *72*
nested and recursive *185*
redefining general multiline *184*
repeating *186, 187*
returning positions of strings *177*
string repeat *187*
terminating assembly of *180*
terminating body of *181*
text
defined *175*
examples of manipulation *177*
how to define *175*
why to use *175*
%MACS directive *189, 213*
MAKE utility *See also* the README file
MASK unary operator *73*
MASKFLAG instruction *167*
MASM compatibility *273*
environment variables *274*
expressions *36*
floating-point format *274*
NOSMART directive *273*
predefined symbols *42*
Quirks mode *273*
segment groups *37*
two-pass asssembly *273*
MASM directive *34*

MASM mode *See* MASM compatibility
math coprocessor *See* numeric coprocessor
member functions *246*
memory models
  available segments *90*
  Borland C++ *223*
  FAR code pointers *93*
  modifiers of *92*
  NEAR code pointers *93*
  segment attributes for *255*
  specifying values *93*
  standard *91*
messages
  reporting error *47*
  suppressing *26*
  warning *46*
METHOD keyword *52, 54*
method procedures
  creating *136*
  defined *55*
  example of *56*
  structure of *62*
methods
  calling ancestor virtual *61*
  calling static *57*
  calling virtual *58, 59*
  defined *50*
  static versus virtual
    advantages of *55*
  tables and *113*
  virtual *173*
Microsoft Assembler *See* MASM compatibility
/ML command-line switch *67*
/ml option *21, 42, 204*
MODEL directive *90, 93*
  language modifiers and *130*
.MODEL directive *90*
@Model symbol *93*
models, determining procedure distance *126*
modifiers, language *129*
modular programming, module names *45*
modules, defined *203*
/MU command-line switch *67*
/mu option *22*
MULTERRS directive *47*
multiline definition syntax *40*
multiline macros *177*
  defining general *181*
  deleting general *184*
  including in listing file *189*
  invoking general *182*
  redefining general *184*
multiline syntax
  enumerated data types and *106*
  record data types and *107*
  table data type definitions and *114*
multiple assembly passes *21, 127*
/MV command-line switch *67*
/mv# option *22*
/MX command-line switch *67*
/mx option *23, 204*

N

/n option *23*
NAME directive *45*
name-mangling *221*
named structures, including *111*
naming conventions of symbols *203*
NEAR procedures *126*
near returns, instructions for *162*
near tables, objects and *58*
NEARSTACK modifier *91*
nested macros *See* macros
nested procedures *135*
%NEWPAGE directive *215*
%NOCONDS directive *212*
%NOCREF directive *214*
%NOCTLS directive *211*
NOEMUL directive *18, 87*
%NOINCL directive *211*
NOJUMPS directive *160*
NOLANGUAGE interfacing convention *171*
NOLANGUAGE procedures
  prolog and epilog code and *129*
%NOLIST directive *210*
NOLOCALS directive *142*
%NOMACS directive *189, 213*
NOMULTERRS directive *47*
NOPs, avoiding generation of *160*
NOSMART directive *159*
  MASM compatibility *273*
%NOSYMS directive *211*
NOTHING keyword *101*
%NOTRUNC directive *216*

NOWARN directive *46*
NUL device *15*
null string, length of *177*
numbers
  encoded real *148*
  floating-point *148*
numeric constants *65*
numeric coprocessor *18*

O

/o option *24*
.OBJ files *1*
  suppressing *25*
object files
  debugging information in *29, 30*
  line number information in *29*
  module name *45*
  segment ordering *16, 26*
object methods
  calling *172*
  tail recursion for *173*
object modules, defined *10*
@Object symbol *118*
object-oriented programming
  advantages of using *49, 50*
  defined *49*
  filename format *63*
  table data types and *113*
objects *See also* methods
  creating instances of *62, 157*
  data types and *52*
  declaring *51, 53*
  defined *50*
  defining symbols *118*
  derived *54*
  differences between structures and *157*
  GLOBAL directive and *53*
  how to define *117*
  initializing instance's VMT pointer *174*
  linked list example *51*
  method procedures and *117, 136*
  near tables and *58*
  structures and *117*
  TLINK compatable without overlay code *24*
  virtual method table instances *157*
  what they consist of *116*
OBJXREF utility *See* the README file
OFFSET operator *36, 77*
  MASM vs. Ideal mode *37*
offsets, getting segments and *77*
/oi option *24*
online help *8*
/op option *24, 99*
operands, Ideal mode *36*
operators *See also* individual listings
  bit shift *75*
  Boolean algebra and *74*
  comments *40*
  comparison *75*
  general arithmetic *74*
  Ideal vs. MASM mode *36*
options, command line *See* command-line options
ORG directive *120*
OS/2 programs
  flat model format and *256*
/os option *24*
%OUT directive *46*
overlay code
  generating *24*
  IBM linker *24*
  Phar Lap linker *24*

P

P8086 directive *83*
P8087 directive *86*
P186 directive *83*
P287 directive *86*
P386 directive *83*
P387 directive *86*
P486 directive *83*
P487 directive *83*
P386N directive *83*
P486N directive *83*
P386P directive *83*
/p option *25*
PAGE directive *215*
%PAGESIZE directive *215*
parameter passing
  Borland C++ *233*
%PCNT directive *216*
. (period) character
  MASM vs. Ideal mode *260*
. (period) operator *79*
period, Ideal mode structures *36*

Phar Lap linker *99*
plus sign *14*
pointers
  virtual method table *56, 58, 59, 60*
POP instruction *170*
  multiple *162*
  pointers and *163*
POPA instruction
  expanded *163*
POPAW instruction *164*
POPFW instruction *164*
%POPLCTL directive *218*
POPSTATE instruction *164*
precedence
  keyword *266*
    Ideal mode *266*
    MASM mode *266*
predefined symbols *See* symbols
PROC directive *125*
PROC keyword, Ideal mode *35*
PROCDESC directive *137, 172*
procedure prototypes *137*
procedure types, defining *134*
procedures
  calling and having RETURNS *172*
  calling with arguments *171*
  declaring *125*
  defining types *115*
  determining distance of *127*
  FAR *126*
  interfacing conventions of *170*
  languages for
    arguments and *131*
    MODEL and *128*
    overriding default *128*
  method *62*
    creating *136*
  models and distance of *126*
  NEAR *126*
  nesting and scope rules *135*
  NOLANGUAGE *129*
  prototyping *172*
  publishing prototypes *205*
  specifying languages for *128*
  stack frames and *131, 161, 170*
  writing constructor and destructor *137*
processor directives *83*
processor type, determining *85*
PROCTYPE directive *115, 134*
program development cycle *10*
program termination, END directive and *45*
prolog code
  defined *128*
  languages and *128*
  NOLANGUAGE procedures and *129*
  register preservation and *134*
  specifying default style *92*
  what it does *128*
protected mode *25*
  segment registers and *89*
prototypes
  procedure *137*
  procedure types and *138*
  publishing procedure *205*
prototyping procedures *172*
PUBLIC directive *204*
public functions, Borland C++ and *229*
PUBLICDLL directive *204*
PURGE directive *184*
PUSH instruction *170*
  multiple *162*
  pointers and *163*
PUSHA instruction
  expanded *163*
PUSHAW instruction *164*
PUSHF instruction
  expanded *163*
PUSHFW instruction *164*
PUSHing constants *163*
%PUSHLCTL directive *218*
PUSHSTATE instruction *164*

Q

/q option *25*
quadword values *147*
question mark
  symbols using *43*
QUIRKS directive *273*

R

/r option *18, 25*
RADIX directive *66, 149*
.RADIX directive *66*

radixes
available *65*
changing default *66*
characters determining *65*
default *149*
real mode, segment registers and *89*
record data types, multiline syntax for *107*
RECORD directive *106*
records
< > and *154*
{ } and *153*
creating instances of *153*
defining *153*
initializing instances *153*
retrieving data from *168*
setting values in *167*
recursive macros *See* macros
reference books *11*
registers *See also* individual listings
names of and expressions *70*
preserving *134*
preserving (Borland C++) *240*
segment *89*
registration (product)
by phone *4*
REPT directive *186*
RET instruction, NEAR or FAR and *161*
RETCODE instruction *162*
RETF instruction *162*
RETN instruction *162*
return instructions *161*
RETURNS directive *172*

S

/s option *16, 17, 26*
.SALL directive *189, 213*
scope of symbols, defined *141*
scope rules for nested procedures *135*
SEG operator *77*
SEGCS instruction *166*
SEGDS instruction *166*
SEGES instruction *166*
SEGFS instruction *166*
SEGGS instruction *166*
SEGMENT directive *96*
SEGMENT keyword, Ideal mode *35*
segments
8086 processor and *89*
assigning to groups *99*
attributes
access *99*
alignment *98*
class *98*
combination *97*
size *98*
Borland C++ and *223*
closing *99, 110*
code *90*
default attributes *255*
directives (Borland C++ and) *223*
fixups (Ideal vs. MASM mode) *36*
forced overrides *166*
generic *96*
getting offsets and *77*
groups
Ideal mode and *34, 37*
MASM mode *37*
groups and *90*
how the stack is treated *89*
memory models and *91*
opening *96*
ordering *16, 101*
alphabetic *102*
changing *101*
DOS *102*
sequential *102*
overrides of expressions *77*
registers *89, See also* individual listings
registers and *100*
sequential order *16, 26*
simplified directives *94*
size *86*
symbols and *95*
writing to uninitialized *97*
SEGSS instruction *166*
semicolon *14*
within macros *40*
SEQ directive *16*
.SEQ directive *102*
SETFIELD instruction *167*
SETFLAG instruction *167*
SFCOND directive *29*
.SFCONDS directive *212*

SGDT instruction *170*
shift instructions, rotation counts and *165*
SHL operator *75*
SHR operator *75*
SIDT instruction *170*
simplified segment directives *See also* individual listings
  symbols and *95*
size of data *See* data
size of instructions, controlling *169*
SIZE unary operator *72*
SIZESTR directive *177*
SMALL operator *81, 169*
  instructions it affects *170*
SMALLSTACK directive *103*
SMART directive *159*
  MASM compatibility *273*
smart flag instructions, why they're useful *166*
software and hardware requirements *2*
source files
  include files *19*
  symbols *17*
square brackets
  Ideal mode *36*
  MASM mode *36*
stack
  changing size of *103*
  MODEL directive and *103*
  segments and *89*
STACK directive *94*
stack frame
  defined *170*
  specifying arguments *131*
.STACK directive *95*
@stack symbol *95*
.STARTUP directive *95*
@Startup symbol *96*
STARTUPCODE directive *95*
static methods
  calling *57*
  versus virtual (advantages of) *55*
statistics, displaying *16*
string constants *66*
strings, quoted *148*
STRUC directive *52, 108, 110, 117, 118*
structures
  aligning members *109*
  Borland C++ *241*
  bracket initializer and nested *152*
  closing *109*
  creating instances of *149*
  creating members *109*
  defined *108*
  differences between objects and *157*
  including named *111*
  initializing instances *150*
  member names and *109, 112*
  members and *108*
  names in expressions *113*
  nested *112*
  nesting *110*
  objects and *117*
  opening a definition *108*
SUBSTR directive *176*
SUBTTL directive *217*
%SUBTTL directive *217*
support, technical *4-5*
symbol tables
  listing files and cross-referencing *17*
  suppressing *23*
symbols
  address subtypes
    complex *68*
    simple *68*
  aliases *42*
  block-scoped *142*
  block-scoped (disabling) *142*
  case sensitivity of *21, 23, 67*
  @Cpu *85*
  ??date *43*
  defined *67*
  defining *17*
  dynamic link entry points *204*
  enabling locally scoped *135*
  external *23, 205*
    Borland C++ and *232*
  ??filename *44*
  @FileName *44*
  global *205*
  in expressions *70*
  length of *22*
  location counter *119*
  MASM block scoping *143*
  names of *67*

naming conventions for languages *203*
overriding language setting *204*
public *23, 204*
publishing external *204*
redefinable *141, 142*
restrictions *17*
scope of (defined) *141*
standard values *71*
??time *44*
types of *67*
uppercase *22*
values used by themselves *71*
why to use *65*
@WordSize *86*
%SYMS directive *211*
SYMTYPE operator *79*
syntax, command-line *See* command-line syntax

T

/t option *26*
TABLE directive *52*
@Table symbol *118*
@TableAddr member *157*
@Tableaddr symbol *118*
tables
creating instances of *155*
data types *113*
initializing instances of *155*
overriding members *115*
static members *113*
virtual members *113*
%TABSIZE directive *217*
tags, macro *180*
tail recursion code, instruction for *173*
TASM.CFG *31*
TBLINIT directive *56*
in .ASM files *63*
TBLINIT instruction *174*
TBLINST directive *56*
in .ASM files *63*
TBLINST pseudo-op *157*
TBLPTR directive *118*
TCREF utility *15, See also* the README file
Technical Support
contacting *4*
technical support *4-5*
termination, END directive and *45*
termination code *96*
TESTFLAG instruction *167*
text macro names, in expressions *72*
text strings *See* strings
%TEXT directive *217*
TFCOND directive *29*
.TFCOND directive *212*
THELP utility *See* the README file
THIS operator *78*
time *44*
??time symbol *44*
TITLE directive *217*
%TITLE directive *217*
TLIB utility *See* the README file
TLINK utility *233, 249, See also* the README file
example of *11*
%TRUNC directive *216*
Turbo Librarian *See* the README file
Turbo Link *See* TLINK utility
two-pass assembly
MASM compatibility *273*
type checking, Ideal mode *33*
TYPE operator *76*
type override operators *75*
type-safe linkage *221*
.TYPE operator *79*
TYPEDEF directive *115*
typefaces in this manual *4*
types *See also* data types
complex *114, 115*
defining named *115*
defining procedure *115*
of expressions *76*
procedure *134*
symbol *67*

U

/u option *27*
/u command-line switch *44*
UDATASEG directive *94*
UFARDATA directive *95*
underscore, and the C language *229*
UNINIT *98*
UNION directive *108, 110*
unions
bracket initializer and nested *152*
closing *109*

creating instances of *149*
defined *108*
initialized data *112*
initializing instances *150*
members and *108*
multiple initialized members *151*
nested *112*
nesting *110*
opening a definition *108*
uppercase, converting symbols to *22*
USE32 modifier *91*
USES directive *134*
utilities *See* individual listings
UTILS.TSM *8*

V

/v option *16, 27*
variables, communal *206*
VERSION directive *44, 45*
line continuation and *41*
MASM compatability and *45*
VIRTUAL keyword *53, 113*
virtual method table
initializing *56*
initializing pointer to *174*
instances of *56, 60, 157*
modifiers and *117*
objects and *118*
pointers *59*
pointers to *56, 58, 60, 118*
virtual methods
ancestor *60*
calling *58, 59*
object data types and *157*
versus static (advantages of) *55*
virtual table pointers
determining size of *118*
modifiers and *117*

W

/w option *27*
WARN directive *46*
warning messages *46, 276*
"mild" *27*
generating *27*
WHILE directive *187*
WIDTH unary operator *73*
word values *146*
@WordSize symbol *86*

X

/x option *28*
.XRF files *15*
.XALL directive *189, 213*
.XCREF directive *214*
.XLIST directive *210*

Z

/z option *29*
/zd option *29*
/zi option *29*
/zn option *30*

Borland

Corporate Headquarters: 100 Borland Way, Scotts Valley, CA 95066-3249, (408) 431-1000. Offices in: Australia, Belgium, Canada, Chile, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Latin America, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan, and United Kingdom • Part # LSM1441WW21771 • BOR 7028