

BORLAND® C++

TOOLS AND UTILITIES GUIDE

- ERROR MESSAGES
- WINSIGHT™
- MAKE
- HELP/RESOURCE COMPILERS
- TLINK

B O R L A N D

Borland[®] C++

Version 3.1

Tools and Utilities Guide

Copyright © 1991, 1992 by Borland International. All rights reserved.
All Borland products are trademarks or registered trademarks of
Borland International, Inc. Other brand and product names are
trademarks or registered trademarks of their respective holders.
Windows, as used in this manual, refers to Microsoft's
implementation of a windows system.

C O N T E N T S

Introduction	1	Examples	24
Chapter 1 Import library tools	3	Automatic dependency checking	25
IMPDEF: The module definitions manager	3	Implicit rules	25
Classes in a DLL	4	Macros	28
Functions in a DLL	5	Defining macros	29
IMPLIB: The import librarian	6	Using macros	29
Re-creating IMPORT.LIB	7	Using environment variables as macros	29
IMPLIBW: The import librarian for Windows	7	Substitution within macros	30
Select an import library	7	Special considerations	30
From a DLL	7	Predefined macros	30
From a module definition file	7	Defined Test Macro (\$d)	31
Creating the import library	8	File name macros	32
Chapter 2 Make: The program manager	9	Base file name macro (\$*)	32
How MAKE works	10	Full file name macro (\$<)	32
Starting MAKE	10	File name path macro (\$:)	33
Command-line options	11	File name and extension macro (\$.)	33
The BUILTINS.MAK file	13	File name only macro (\$&)	33
A simple use of MAKE	13	Full target name with path macro (\$@)	33
Creating makefiles	16	All dependents macro (\$**)	34
Components of a makefile	16	All out of date dependents macro (\$?)	34
Comments	17	Macro modifiers	34
Command lists for implicit and explicit rules	17	Directives	35
Prefixes	18	Dot directives	36
Command body and operators	18	.precious	36
Compatibility option	19	.path.ext	36
Batching programs	20	.suffixes	37
Executing commands	21	File-inclusion directive	38
Explicit rules	22	Conditional execution directives	38
Special considerations	23	Expressions allowed in conditional directives	40
Multiple explicit rules for a single target	24	Error directive	41
		Macro undefinition directive	42

The compatibility option -N	42	/o (overlays)	68
Chapter 3 TLIB: The Turbo librarian	45	/P (pack code segments)	69
Why use object module libraries?	46	/t (tiny model .COM file)	69
The TLIB command line	46	/Td and /Tw (target options)	70
The operation list	47	/v (debugging information)	70
File and module names	48	/ye (expanded memory)	71
TLIB operations	48	/yx (extended memory)	71
Using response files	49	The module definition file	72
Creating an extended dictionary: The /E		Module definition file defaults	72
option	49	A quick example	73
Setting the page size: The /P option	50	Module definition reference	74
Advanced operation: The /C option	50	CODE	74
Examples	51	DATA	75
Chapter 4 TLINK: The Turbo linker	53	DESCRIPTION	76
Invoking TLINK	53	EXETYPE	76
An example of linking for DOS	54	EXPORTS	76
An example of linking for Windows ..	55	HEAPSIZE	77
File names on the TLINK command		IMPORTS	78
line	55	LIBRARY	78
Using response files	56	NAME	79
The TLINK configuration file	57	SEGMENTS	79
Using TLINK with Borland C++		STACKSIZE	80
modules	58	STUB	80
Startup code	59	Chapter 5 Using WinSight	83
Libraries	59	Getting started	83
BGI graphics library	60	Getting out	84
Math libraries	60	Choosing a view	84
Run-time libraries	61	Picking a pane	84
Using TLINK with BCC	62	Arranging the panes	85
TLINK options	62	Getting more detail	85
The TLINK configuration file	62	Class detail	85
/3 (32-bit code)	63	Window detail	85
/A (align segments)	63	Using the window tree	85
/c (case sensitivity)	63	Pruning the tree	86
/C (case sensitive exports)	64	Showing child windows	86
/d (duplicate symbols)	64	Hiding child windows	86
/e (no extended dictionary)	65	Finding a window	86
/i (uninitialized trailing segments) ..	65	Leaving Find Window mode	86
/l (line numbers)	65	Spying on windows	87
/L (library search paths)	65	Working with classes	87
/m, /s, and /x (map options)	66	Using the Class List pane	87
/n (ignore default libraries)	68	Spying on classes	87
		Taking time out	88

Turning off tracing	88	Layout of the Help text	110
Suspending screen updates	88	Type fonts and sizes	111
Choosing messages to trace	88	Graphic images	112
Filtering out messages	89	Creating the Help topic files	113
Message tracing options	89	Choosing an authoring tool	113
Formatting message parameters ...	89	Structuring Help topic files	113
Logging traced messages	89	Coding Help topic files	114
WinSight windows	93	Assigning build tags	115
Class List pane	93	Assigning context strings	116
Display format	93	Assigning titles	117
Window Tree pane	94	Assigning keywords	118
Display format	94	Creating multiple keyword	
Message Trace pane	94	tables	119
Format	95	Assigning browse sequence	
Chapter 6 RC: The Windows resource		numbers	120
compiler	97	Organizing browse sequences ..	120
Creating resources	97	Coding browse sequences	122
Adding resources to an executable	98	Creating cross-references between	
Resource compiling from the IDE	98	topics	122
Resource compiling from the command		Defining terms	123
line	99	Creating definition topics	123
Resource compiling from a makefile ..	99	Coding definitions	124
Resource Compiler syntax	99	Inserting graphic images	124
Chapter 7 HC: The Windows Help		Creating and capturing bitmaps ..	124
compiler	101	Placing bitmaps using a graphical	
Creating a Help system: The development		word processor.	125
cycle	101	Placing bitmaps by reference	125
How Help appears to the user	102	Managing topic files	126
How Help appears to the help writer ..	103	Keeping track of files and topics ..	127
How Help appears to the help		Creating a help tracker	127
programmer	104	Building the Help file	129
Planning the Help system	104	Creating the Help project file	129
Developing a plan	104	Specifying topic files	130
Defining the audience	104	Specifying build tags	131
Planning the contents	105	Specifying options	131
Planning the structure	106	Specifying error reporting	132
Displaying context-sensitive Help		Specifying build topics	132
topics	107	Specifying the root directory	133
Determining the topic file structure ..	108	Specifying the index	134
Choosing a file structure for your		Assigning a title to the Help	
application	108	system	134
Designing Help topics	110	Converting fonts	134
		Changing font sizes	135
		Multiple keyword tables	136

Compressing the file	136	Canceling Help	148
Specifying alternate context strings ..	137	Help examples	149
Mapping context-sensitive topics	138	The Helpex project file	151
Including bitmaps by reference	140	Appendix A Error messages	153
Compiling Help files	140	Finding a message in this	
Using the Help Compiler	141	appendix	153
Programming the application to access		Types of messages	154
Help	141	Compile-time messages	154
Calling WinHelp from an		DPMI server messages	155
application	142	Help compiler messages	155
Getting context-sensitive Help	143	MAKE messages	157
Shift+F1 support	144	Run-time error messages	157
F1 support	145	TLIB messages	157
Getting help on items on the Help		TLINK messages	158
menu	147	Message explanations	158
Accessing additional keyword		Index	241
tables	147		

T A B L E S

1.1: IMPLIB options	6	5.9: Non-client messages	91
2.1: MAKE options	12	5.10: Control messages	91
2.2: MAKE prefixes	18	5.11: Other messages	92
2.3: MAKE predefined macros	31	6.1: Resource Compiler options	100
2.4: MAKE filename macros	31	7.1: Your application audience	105
2.5: MAKE macro modifiers	35	7.2: Help design issues	110
2.6: MAKE directives	35	7.3: Windows fonts	112
2.7: MAKE operators	41	7.4: Help control codes	114
3.1: TLIB options	47	7.5: Restrictions of Help titles	118
3.2: TLIB action symbols	48	7.6: Help keyword restrictions	119
4.1: TLINK options	53	7.7: Help project file sections	129
4.2: DOS application .OBJ and .LIB files ..	61	7.8: The Help [Options] options	131
4.3: Windows application .OBJ and .LIB files	61	7.9: WARNING levels	132
4.4: DLL object and library files	61	7.10: Build tag order of precedence	133
4.5: TLINK overlay options	68	7.11: Build expression examples	133
5.1: Mouse and keyboard actions	84	7.12: <i>wCommand</i> values	142
5.2: Mouse messages	89	7.13: <i>dwData</i> formats	143
5.3: Window messages	90	7.14: MULTIKEYHELP structure formats	148
5.4: Input messages	90	A.1: Compile-time message variables ..	155
5.5: System messages	90	A.2: Help message variables	156
5.6: Initialization messages	90	A.3: MAKE error message variables ..	157
5.7: Clipboard messages	91	A.4: TLIB message variables	157
5.8: DDE messages	91	A.5: TLINK error message variables ..	158

F I G U R E S

4.1: Detailed map of segments	67	7.6: Help topic display showing bitmaps by reference	126
7.1: Helpex help window	103	7.7: Help tracker text file example	128
7.2: Topic file	103	7.8: Help tracker worksheet example ...	128
7.3: Example of a help hierarchy	106	7.9: Word for Windows topic	149
7.4: Basic help file structure	108	7.10: Help topic display	150
7.5: Help file structure showing hypertext jumps	109	7.11: Bitmap by reference in topic	150
		7.12: Help topic display	151

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

Borland C++ comes with a host of powerful standalone utilities that you can use with your Borland C++ files or other modules to ease your DOS and Windows programming.

This book describes IMPDEF, IMPLIB, IMPLIBW, MAKE, TLIB, TLINK, and WinSight, and illustrates, with code and command-line examples, how to use them. The rest of the Borland C++ utilities are documented in a text file called UTIL.DOC that the INSTALL utility placed in the DOC subdirectory.

Name	Description
<i>Documented in this book</i>	
IMPDEF	Creates a module definition file
IMPLIB	Generates an import library
IMPLIBW	Windows application that generates an import library
MAKE	Standalone program manager
TLIB	Turbo Librarian
TLINK	Turbo Linker
WinSight	Windows message monitor
<i>Documented in the online document UTIL.DOC</i>	
BGIOBJ	Conversion utility for graphics drivers and fonts
CPP	Preprocessor
GREP	File-search utility
OBJXREF	Object module cross-referencer
PRJCFG	Updates options in a project file from a configuration file, or converts a project file to a configuration file
PRJCNVT	Converts Turbo C project files to the Borland C++ format
PRJ2MAK	Converts Borland C++ project files to MAKE files
THELP	Turbo Help utility
TOUCH	Updates file date and time
TRANCOPY	Copies transfer items from one project to another
TRIGRAPH	Character-conversion utility

Import library tools

Dynamic link libraries (DLLs) are an important part of Windows programming. You can create DLLs of your commonly used code, and can use DLLs in your applications, both of your own code and from third parties.

TLINK uses import libraries as it builds a Windows application to know when a function is defined in and imported from a DLL. Import libraries generally replace module definition (.DEF) files.

IMPDEF creates a module definition file containing an EXPORT statement for each of the exported functions in the DLL. IMPLIB creates an import library for a DLL. IMPLIBW also creates an import library for a DLL, but is a Windows application.

IMPDEF: The module definitions manager

An import library provides access to the functions in a Windows DLL. See page 6 for more details.

IMPDEF works with IMPLIB to let you customize an import library to suit the needs of a specific application. The syntax is

```
IMPDEF DestName.DEF SourceName.DLL
```



This creates a module definition file named *DestName.DEF* from the file *SourceName.DLL*. The module definition file would look something like this:

```
LIBRARY FileName  
DESCRIPTION 'Description'
```

```

EXPORTS
    ExportFuncName           @Ordinal
    ...
    ExportFuncName           @Ordinal

```

where *FileName* is the DLL's root filename, *Description* is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement, *ExportFuncName* names an exported function and *Ordinal* is that function's ordinal value (an integer).

Classes in a DLL

This utility is particularly handy for a DLL that uses C++ classes, for two reasons. First, if you use the **_export** keyword when defining a class, all of the non-inline member functions and static data members for that class are exported. It's easier to let IMPDEF make a module definition file for you because it lists all the exported functions, automatically including the member functions and static data members.

Since the names of these functions are mangled, it would be very tedious to list them all in the EXPORTS section of a module definition file simply to create an import library from the module definition file. If you use IMPDEF to create the module definition file, it will include the ordinal value for each exported function. If the exported name is mangled, IMPDEF will also include that function's unmangled, original name as a comment following the function entry. So, for instance, the module definition file created by IMPDEF for a DLL that used C++ classes would look something like this:

```

LIBRARY   FileName
DESCRIPTION 'Description'
EXPORTS
    MangledExportFuncName @Ordinal ; ExportFuncName
    ...
    MangledExportFuncName @Ordinal ; ExportFuncName

```

where *FileName* is the DLL's root filename, *Description* is the value of the DESCRIPTION statement if the DLL was previously linked with a module definition file that included a DESCRIPTION statement, *MangledExportFuncName* provides the mangled name, *Ordinal* is that function's ordinal value (an integer), and *ExportFuncName* gives the function's original name.

Functions in a DLL

IMPDEF creates an editable source file that lists all the exported functions in the DLL. You can edit this .DEF file to contain only those functions that you want to make available to a particular application, then run IMPLIB on the edited .DEF file. This results in an import library that contains import information for a specific subset of a DLL's export functions.

For instance, let's say you're distributing a DLL that provides functions to be used by several applications. Every export function in the DLL is defined with **_export**. Now, if all the applications used all the DLL's exports, then you could simply use IMPLIB to make one import library for the DLL, and deliver that import library with the DLL, which would provide import information for all of the DLL's exports. The import library could be linked to any application, thus eliminating the need for the particular application to list every DLL function it uses in the IMPORTS section of its module definition file.

Now, let's say you want to give only a handful of the DLL's exports to a particular application. Ideally, you want a customized import library to be linked to that application—an import library that only provides import information for the subset of functions that the application will use. All of the other export functions in the DLL will be hidden to that client application.

To create an import library that satisfies these conditions, run IMPDEF on the compiled and linked DLL. IMPDEF produces a module definition file that contains an EXPORT section listing all of the DLL's export functions. You can edit that module definition file, removing EXPORTS section entries for those functions that you *don't* want in the customized import library. Once you've removed the exports that you don't want, run IMPLIB on the module definition file. The result will be an import library that contains import information for only those export functions listed in the EXPORTS section of the module definition file.

IMPLIB: The import librarian



The IMPLIB utility creates an import library that can be substituted for part or all of the IMPORTS section of a module definition file for a Windows application.

If a module uses functions from DLLs, you have two ways to tell the linker about them:

Since TLINK's default settings suffice for most applications, module definition files aren't usually needed.

- You can add an IMPORTS section to the module definition file and list every function from DLLs that the module will use.
- Or you can include the import library for the DLLs when you link the module.

If you've created a Windows application, you've already used at least one import library, IMPORT.LIB. IMPORT.LIB is the import library for the standard Windows DLLs. (IMPORT.LIB is linked automatically when you build a Windows application in the IDE and when using BCC to link. You only have to explicitly link with IMPORT.LIB if you're using TLINK to link separately.)

See page 3 for information on using IMPDEF and IMPLIB to customize an import library for a specific application.

An import library lists some or all of the exported functions for one or more DLLs. IMPLIB creates an import library directly from DLLs or from module definition files for DLLs (or a combination of the two).

To create an import library for a DLL, type

```
IMPLIB Options LibName [ DefFiles... | DLLs... ]
```

Note that a DLL can also have an extension of .EXE or .DRV, not just .DLL.

where *Options* is an optional list of one or more IMPLIB options (see Table 1.1), *LibName* (required) is the name for the new import library, *DefFiles* is a list of one or more existing module definition files for one or more DLLs, and *DLLs* is a list of one or more existing DLLs. You must specify at least one DLL or module definition file.

Table 1.1
IMPLIB options

You can use either a hyphen or a slash to precede IMPLIB's options but the options must be lowercase.

Option	What it does
-i	Tells IMPLIB to ignore WEP, the Windows exit procedure required to end a DLL. Use this option if you are specifying more than one DLL on the IMPLIB command line.
Warning control:	
-t	Terse warnings.
-v	Verbose warnings.
-w	No warnings.

Re-creating IMPORT.LIB

When Microsoft releases new versions of Windows, you will probably need to replace the current version of IMPORT.LIB with a new one. The easiest way to do this is to build it yourself.

This command line builds the current version of IMPORT.LIB:

Enter this command on a single line while at the Windows SYSTEM directory.

```
IMPLIB -i IMPORT.LIB GDI.EXE KERNEL.EXE USER.EXE KEYBOARD.DRV  
SOUND.DRV WIN87EM.DLL
```

If Windows is extended so that it uses additional DLLs, any new DLLs will also have to appear on the command line.

IMPLIBW: The import librarian for Windows



See the discussion of IMPLIB on page 6 for more information on the uses of import libraries and DLLs.

The IMPLIBW Windows utility creates an import library that can be substituted for part or all of the IMPORTS section of a module definition file for a Windows application. Unlike most of the other tools and utilities discussed in this book, IMPLIBW is an application that runs under Windows.

Select an import library

To create an import library with IMPLIBW, double-click on the IMPLIBW icon. An empty window will appear. Selecting File | Create... will bring up a dialog box listing all DLLs in the current directory. You can navigate between directories as normal by double-clicking on directory names in the list box.

From a DLL

When you select a DLL and click on Create (or simply double-click on a DLL filename), IMPLIBW will create an import library with the same base name as the selected DLL, and an extension of .LIB.

From a module definition file

If you have a module definition (.DEF) file for a DLL, you can use it instead of the DLL itself to create an import library.

Instead of selecting a DLL, type the name of a .DEF file into the edit control and click on Create. IMPLIBW will create an import

library with the same base name as the selected .DEF file, and an extension of .LIB.

Creating the import library

See Appendix A, "Error messages," for possible IMPLIBW error messages.

If there were no errors as IMPLIBW examined the DLL or module definition file and created the import library, the window will report "No Warnings." That's all there is to it. You now have a new import library that you can include in a project.

Make: The program manager

Borland's command-line MAKE, derived from the UNIX program of the same name, helps you keep the executable versions of your programs current. Many programs consist of many source files, each of which may need to pass through preprocessors, assemblers, compilers, and other utilities before being combined with the rest of the program. Forgetting to recompile a module that has been changed—or that depends on something you've changed—can lead to frustrating bugs. On the other hand, recompiling *everything* just to be safe can be a tremendous waste of time.

To find out how to create a makefile for a Windows application, see Chapter 8, "Building a Windows application" in the Programmer's Guide.

MAKE solves this problem. You provide MAKE with a description of how the source and object files of your program are processed to produce the finished product. MAKE looks at that description and at the date stamps on your files, then does what's necessary to create an up-to-date version. During this process, MAKE may invoke many different compilers, assemblers, linkers, and utilities, but it never does more than is necessary to update the finished program.

MAKE's usefulness extends beyond programming applications. You can use MAKE to control any process that involves selecting files by name and processing them to produce a finished product. Some common uses include text processing, automatic backups, sorting files by extension into other directories, and cleaning temporary files out of your directory.

How MAKE works

MAKE keeps your program up-to-date by performing the following tasks:

- Reads a special file (called a makefile) that you have created. This file tells MAKE which .OBJ and library files have to be linked in order to create your executable file, and which source and header files have to be compiled to create each .OBJ file.
- Checks the time and date of each .OBJ file against the time and date of the source and header files it depends on. If any of these is later than the .OBJ file, MAKE knows that the file has been modified and that the source file must be recompiled.
- Calls the compiler to recompile the source file.
- Once all the .OBJ file dependencies have been checked, checks the date and time of each of the .OBJ files against the date and time of your executable file.
- If any of the .OBJ files is later than the .EXE file, calls the linker to recreate the .EXE file.

The original IBM PC and compatibles didn't come with a built-in clock. If your system falls into this category, be sure to set the system time and date correctly.

MAKE relies completely upon the time stamp DOS places on each file. This means that, in order for MAKE to do its job, your system's time and date *must* be set correctly. Make sure the system battery is in good repair. Weak batteries can cause your system's clock to lose track of the date and time, and MAKE will no longer work as it should.

Starting MAKE

There are two versions of MAKE: A protected mode version—MAKE.EXE—and a real mode version—MAKER.EXE. They work identically; the only difference is that the protected mode version can process larger make files. When we refer to MAKE, we mean either version.

To use MAKE, type `make` at the DOS prompt. MAKE then looks for a file specifically named MAKEFILE. If MAKE can't find MAKEFILE, it looks for MAKEFILE.MAK; if it can't find that or BUILTINS.MAK (described later), it halts with an error message.

What if you want to use a file with a name other than MAKEFILE or MAKEFILE.MAK? You give MAKE the file (**-f**) option, like this:

```
MAKE -f MYFILE.MAK
```

The general syntax for MAKE is

```
make [option...] [target...]
```

where *option* is a MAKE option (discussed later), and *target* is the name of a target file to make.

Here are the MAKE syntax rules:

MAKE stops if any command it has executed is aborted via a Ctrl-Break. Thus, a Ctrl-Break stops the currently executing command and MAKE as well.

- The word *make* is followed by a space, then a list of make options.
- Each make option must be separated from its adjacent options by a space. Options can be placed in any order, and any number of these options can be entered (as long as there is room in the command line). All options that do not specify a string (**-s** or **±a**, for example) can have an optional **-** or **+** after them. This specifies whether you wish to turn the option off (**-**) or on (**+**).
- The list of MAKE options is followed by a space, then an optional list of targets.
- Each target must also be separated from its adjacent targets by a space. MAKE evaluates the target files in the order listed, re-compiling their constituents as necessary.

If the command line does not include any target names, MAKE uses the first target file mentioned in an explicit rule. If one or more targets are mentioned on the command line, they will be built as necessary.

Command-line options

Here's a complete list of MAKE's command-line options. Note that case (upper or lower) *is* significant; for example, the option **-d** is not a valid substitution for **-D**. Also note that you can use either a **-** or a **/** to introduce the options.

Table 2.1: MAKE options

Option	What it does
-? or -h	Prints a help message. The default options are displayed with plus signs following.
-a	Causes an automatic dependency check on .OBJ files.
-B	Builds all targets regardless of file dates.
-ddirectory	When used with the -S options, tells MAKE to write its swap file in the specified directory. <i>directory</i> can include a drive letter. Has no effect with the protected-mode version of MAKE.
-Didentifier	Defines the named identifier to the string consisting of the single character 1 (one).
[-D]iden=string	Defines the named identifier <i>iden</i> to the string after the equal sign. If the string contains any spaces or tabs, it must be enclosed in quotes. The -D option is optional.
-e	Ignores any attempt to redefine a macro whose name is the same as an environment variable. (In other words, causes the environment variable to take precedence.)
-f filename	Uses <i>filename</i> as the MAKE file. If <i>filename</i> does not exist and no extension is given, tries <i>filename</i> .MAK. The space after the -f is optional.
-i	Does not check (ignores) the exit status of all programs run. Continues regardless of exit status. This is equivalent to putting '-' in front of all commands in the MAKEFILE (described below).
-ldirectory	Searches for include files in the indicated directory (as well as in the current directory).
-K	Keeps (does not erase) temporary files created by MAKE. All temporary files have the form MAKE $nnnn$.\$\$\$, where $nnnn$ ranges from 0000 to 9999. See page 18 for more on temporary files.
-m	Displays the date and time stamp of each file as MAKE processes it.
-n	Prints the commands but does not actually perform them. This is useful for debugging a makefile.
-N	Increases MAKE's compatibility by resolving conflicts between MAKE's syntax and the syntax of Microsoft's NMAKE. See the rest of this chapter and Chapter 7, "Converting from Microsoft C" in the <i>Programmer's Guide</i> for the exact differences.
-p	Displays all macro definitions, implicit rules, and macro definitions before executing the makefile.
-r	Ignores the rules (if any) defined in BUILTINS.MAK.
-s	Does not print commands before executing. Normally, MAKE prints each command as it is about to be executed.
-S	Swaps MAKE out of memory while executing commands. This significantly reduces the memory overhead of MAKE, allowing it to compile very large modules. This option has no effect on the protected-mode version of MAKE.
-Uidentifier	Undefines any previous definitions of the named identifier.
-W	Writes the current specified non-string options (like -s and -a) to MAKE.EXE. (This makes them default.)

The BUILTINS.MAK

file

You will often find that there are MAKE macros and rules that you use again and again. There are three ways of handling them.

- ▣ First, you can put them in every makefile you create.
- ▣ Second, you can put them all in one file and use the **!include** directive in each makefile you create. (See page 35 for more on directives.)
- ▣ Third, you can put them all in a BUILTINS.MAK file.

Each time you run MAKE, it looks for a BUILTINS.MAK file; however, there is no requirement that any BUILTINS.MAK file exist. If MAKE finds a BUILTINS.MAK file, it interprets that file first. If MAKE cannot find a BUILTINS.MAK file, it proceeds directly to interpreting MAKEFILE or MAKEFILE.MAK (or whatever makefile you specify with the **-f** option).

The first place MAKE searches for BUILTINS.MAK is the current directory. If it's not there, MAKE then searches the directory from which MAKE.EXE was invoked. You should place the BUILTINS.MAK file in the same directory as the MAKE.EXE file.

MAKE always searches for the makefile in the current directory only. This file contains the rules for the particular executable program file being built. Both BUILTINS.MAK and the makefile files have identical syntax rules.

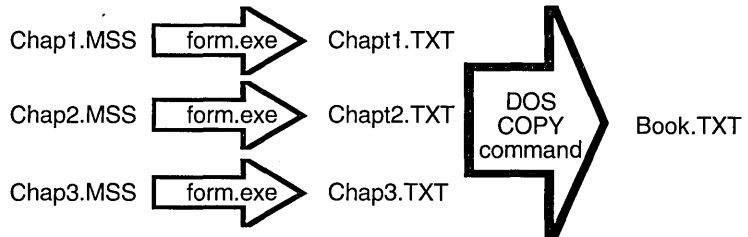
MAKE also searches for any **!include** files (see page 38 for more on this MAKE directive) in the current directory. If you use the **-I** (include) option, it will also search in the directory specified with the **-I** option.

A simple use of MAKE

For our first example, let's look at a simple use of MAKE that doesn't involve programming. Suppose you're writing a book, and decide to keep each chapter of the manuscript in a separate file. (Let's assume, for the purposes of this example, that your

MAKE can also back up files, pull files out of different subdirectories, and even automatically run your programs should the data files they use be modified.

book is quite short: It has three chapters, in the files CHAP1.MSS, CHAP2.MSS, and CHAP3.MSS.) To produce a current draft of the book, you run each chapter through a formatting program, called FORM.EXE, then use the DOS COPY command to concatenate the outputs to make a single file containing the draft, like this:



Like programming, writing a book requires a lot of concentration. As you write, you may modify one or more of the manuscript files, but you don't want to break your concentration by noting which ones you've changed. On the other hand, you don't want to forget to pass any of the files you've changed through the formatter before combining it with the others, or you won't have a fully updated draft of your book!

One inelegant and time-consuming way to solve this problem is to create a batch file that reformats every one of the manuscript files. It might contain the following commands:

```
FORM CHAP1.MSS
FORM CHAP2.MSS
FORM CHAP3.MSS
COPY /A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT BOOK.TXT
```

Running this batch file would always produce an updated version of your book. However, suppose that, over time, your book got bigger and one day contained 15 chapters. The process of reformatting the entire book might become intolerably long.

MAKE can come to the rescue in this sort of situation. All you need to do is create a file, usually named MAKEFILE, which tells MAKE what files BOOK.TXT depends on and how to process them. This file will contain rules that explain how to rebuild BOOK.TXT when some of the files it depends on have been changed.

In this example, the first rule in your makefile might be

```
BOOK.TXT: CHAP1.TXT CHAP2.TXT CHAP3.TXT
          COPY /A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT BOOK.TXT
```

What does this mean? The first line (the one that begins with `BOOK.TXT:`) says that `BOOK.TXT` depends on the formatted text of each of the three chapters. If any of the files that `BOOK.TXT` depends on are newer than `BOOK.TXT` itself, `MAKE` must rebuild `BOOK.TXT` by executing the `COPY` command on the subsequent line.

This one rule doesn't tell the whole story, though. Each of the chapter files depends on a manuscript (`.MSS`) file. If any of the `CHAP?.TXT` files is newer than the corresponding `.MSS` file, the `.MSS` file must be recreated. Thus, you need to add more rules to the makefile as follows:

```
CHAP1.TXT: CHAP1.MSS
           FORM CHAP1.MSS

CHAP2.TXT: CHAP2.MSS
           FORM CHAP2.MSS

CHAP3.TXT: CHAP3.MSS
           FORM CHAP3.MSS
```

Each of these rules shows how to format one of the chapters, if necessary, from the original manuscript file.

`MAKE` understands that it must update the files that another file depends on before it attempts to update that file. Thus, if you change `CHAP3.MSS`, `MAKE` is smart enough to reformat Chapter 3 before combining the `.TXT` files to create `BOOK.TXT`.

We can add one more refinement to this simple example. The three rules look very much the same—in fact, they're identical except for the last character of each file name. And, it's pretty easy to forget to add a new rule each time you start a new chapter. To solve these problems, `MAKE` allows you to create something called an *implicit rule*, which shows how to make one type of file from another, based on the files' extensions. In this case, you can replace the three rules for the chapters with one implicit rule:

```
.MSS.TXT:
          FORM $*.MSS
```

This rule says, in effect, "If you need to make a `.TXT` file out of an `.MSS` file to make things current, here's how to do it." (You'll still have to update the first rule—the one that makes `BOOK.TXT`, so that `MAKE` knows to concatenate the new chapters into the

output file. This rule, and others following, make use of a *macro*. See page 28 for an in-depth discussion of macros.)

Once you have the makefile in place, all you need to do to create an up-to-date draft of the book is type a single command at the DOS prompt: MAKE.

Creating makefiles

Creating a program from an assortment of program files, include files, header files, object files, and so on, is very similar to the text-processing example you just looked at. The main difference is that the commands you'll use at each step of the process will invoke preprocessors, compilers, assemblers, and linkers instead of a text formatter and the DOS COPY command. Let's explore how to create makefiles—the files that tell MAKE how to do these things—in greater depth.

A makefile contains the definitions and relationships needed to help MAKE keep your program(s) up-to-date. You can create as many makefiles as you want and name them whatever you want; MAKEFILE is just the default name that MAKE looks for if you don't specify a makefile when you run MAKE.

You create a makefile with any ASCII text editor, such as the IDE built-in editor, Sprint, or SideKick. All rules, definitions, and directives end at the end of a line. If a line is too long, you can continue it to the next line by placing a backslash (\) as the last character on the line.

Use whitespace (blanks and tabs) to separate adjacent identifiers (such as dependencies) and to indent commands within a rule.

Components of a makefile

Creating a makefile is like writing a program, with definitions, commands, and directives. These are the constructs allowed in a makefile:

- comments
- explicit rules
- implicit rules

- macro definitions
- directives:
 - file inclusion directives
 - conditional execution directives
 - error detection directives
 - macro undefinition directives

Let's look at each of these in more detail.

Comments

Comments begin with a pound sign (#) character; the rest of the line following the # is ignored by MAKE. Comments can be placed anywhere; they don't have to start in a particular column.

A backslash will *not* continue a comment onto the next line; instead, you must use a # on each line. In fact, you cannot use a backslash as a continuation character in a line that has a comment. If the backslash precedes the #, it is no longer the last character on the line; if it follows the #, then it is part of the comment itself.

Here are some examples of comments in a makefile:

```
# Makefile for my book

# This file updates the file BOOK.TXT each time I
# change one of the .MSS files

# Explicit rule to make BOOK.TXT from six chapters. Note the
# continuation lines.
BOOK.TXT: CHAP1.TXT CHAP2.TXT CHAP3.TXT\
          CHAP4.TXT CHAP5.TXT CHAP6.TXT
COPY /A CHAP1.TXT+CHAP2.TXT+CHAP3.TXT+CHAP4.TXT+\
        CHAP5.TXT+CHAP6.TXT BOOK.TXT

# Implicit rule to format individual chapters
.MSS.TXT:
        FORM $*.MSS
```

Explicit and implicit rules are discussed following the section on commands.

Command lists for implicit and explicit rules

Both explicit and implicit rules (discussed later) can have lists of commands. This section describes how these commands are processed by MAKE.

Commands in a command list take the form

[*prefix ...*] *command_body*

Each command line in a command list consists of an (optional) list of prefixes, followed by a single command body.

Prefixes The prefixes allowed in a command modify the treatment of these commands by MAKE. The prefix is either the at-sign (@) or a hyphen (-) followed immediately by a number.

Table 2.2
MAKE prefixes

Prefix	What it does
@	Prevents MAKE from displaying the command before executing it. The display is hidden even if the -s option is not given on the MAKE command line. This prefix applies only to the command on which it appears.
-num	Affects how MAKE treats exit codes. If a number (<i>num</i>) is provided, then MAKE aborts processing only if the exit status exceeds the number given. In this example, MAKE aborts only if the exit status exceeds 4: -4 MYPROG SAMPLE.X If no -num prefix is given and the status is nonzero, MAKE stops and deletes the current target file.
-	With a hyphen but no number, MAKE will not check the exit status at all. Regardless of the exit status, MAKE continues.
&	The ampersand operator causes MAKE to execute the command for each of the dependents the \$\$ or ?\$ macros in an explicit rule expands to. See page 34 for more information on these macros.

Exit codes are status codes returned by the executed commands.

Command body and operators

The command body is treated exactly as if it were entered as a line to the DOS command line, with the exception that pipes (!) are not supported.

In addition to the <, >, and >> redirection operators, MAKE adds the << and && operators. These operators create a file on the fly for input to a command. The << operator creates a temporary file and redirects the command's standard input so that it comes from the created file. If you have a program that accepts input from *stdin*, the command

```
MYPROG <<!
This is a test
!
```

would create a temporary file containing the string "This is a test \n", redirecting it to be the sole input to *myprog*. The exclamation point (!) is a delimiter in this example; you can use any character except # or \ as a delimiter for the file. The first line containing the delimiter character as its first character ends the

file. The rest of the line following the delimiter character (in this case, an exclamation point) is considered part of the preceding command.

The **&&** operator is similar to **<<**. It creates a temporary file, but instead of making the file the standard input to the command, the **&&** operator is replaced with the temporary file's name. This is useful when you want MAKE to create a file that's going to be used as input to a program. The following example creates a "response file" for TLINK:

Macros are covered starting on page 28.

```
MYPROG.EXE: $(MYOBSJ)
    TLINK /c @&&!
COS $(MYOBSJ)
$*
$*
$(MYLIBS) EMU.LIB MATHS.LIB CS.LIB
!
```

Note that macros (indicated by **\$** signs) are expanded when the file is created. The **\$*** is replaced with the name of the file being built, without the extension, and **\$(MYOBSJ)** and **\$(MYLIBS)** are replaced with the values of the macros MYOBSJ and MYLIBS. Thus, TLINK might see a file that looks like this:

```
COS A.OBJ B.OBJ C.OBJ D.OBJ
MYPROG
MYPROG
W.LIB X.LIB Y.LIB Z.LIB EMU.LIB MATHS.LIB CS.LIB
```

The KEEP option for the << operator in compatibility mode tells MAKE not to delete specific temporary files. See the next section.

All temporary files are deleted unless you use the **-K** command-line option. Use the **-K** option to "debug" your temporary files if they don't appear to be working correctly.

Compatibility option

If you specified **-N** on the MAKE command line, the **<<** operator changes its behavior to be more like that of the **&&** operator; that is, the temporary file isn't redirected to standard input, it's just created on the fly for use mainly as a response file. This behavior is consistent with Microsoft's NMAKE.

The format for this version of the **<<** operator is:

```
command <<[filename1] ... <<[filenameN]
text
:
```

Note that there must be no space after the << and before the KEEP or NOKEEP option.

```
<<[KEEP | NOKEEP]  
text  
:  
<<[KEEP | NOKEEP]
```

The KEEP option tells MAKE to not delete the file after it's been used. If you don't specify anything or specify NOKEEP, MAKE will delete the temporary file (unless you specified the **-K** option to keep temporary files).

Batching programs

MAKE allows utilities that can operate on a list of files to be batched. Suppose, for example, that MAKE needs to submit several C files to Borland C++ for processing. MAKE could run BCC once for each file, but it's much more efficient to invoke BCC with a list of all the files to be compiled on the command line. This saves the overhead of reloading Borland C++ each time.

MAKE's batching feature lets you accumulate the names of files to be processed by a command, combine them into a list, and invoke that command only once for the whole list.

To cause MAKE to batch commands, you use braces in the command line:

```
command { batch-item } ...rest-of-command
```

This command syntax delays the execution of the command until MAKE determines what command (if any) it has to invoke next. If the next command is identical except for what's in the braces, the two commands will be combined by appending the parts of the commands that appeared inside the braces.

Here's an example that shows how batching works. Suppose MAKE decides to invoke the following three commands in succession:

```
BCC {file1.c }  
BCC {file2.c }  
BCC {file3.c }
```

Rather than invoking Borland C++ three times, MAKE issues the single command

```
BCC file1.c file2.c file3.c
```

Note that the spaces at the ends of the file names in braces are essential to keep them apart, since the contents of the braces in each command are concatenated exactly as-is.

Here's an example that uses an implicit rule. Suppose your makefile had an implicit rule to compile C programs to .OBJ files:

```
.c.obj:
    BCC -c {< }
```

As MAKE uses the implicit rule on each C file, it expands the macro `<` into the actual name of the file and adds that name to the list of files to compile. (Again, note the space inside the braces to keep the names separate.) The list grows until one of three things happens:

- MAKE discovers that it has to run a program other than BCC
- there are no more commands to process
- MAKE runs out of room on the command line

If MAKE runs out of room on the command line, it puts as much as it can on one command line, then puts the rest on the next command line. When the list is done, MAKE invokes BCC (with the `-c` option) on the whole list of files at once.

Executing commands MAKE searches for any other command name using the DOS search algorithm:

1. MAKE first searches for the file in the current directory, then searches each directory in the path.
2. In each directory, MAKE first searches for a file of the specified name with the extension .COM. If it doesn't find it, it searches for the same file name with an .EXE extension. Failing that, MAKE searches for a file by the specified name with a .BAT extension.
3. If MAKE finds a .BAT file, it invokes a copy of COMMAND.COM to execute the batch file.
4. If MAKE can't find a .COM, .EXE, or .BAT file matching the command to be executed, it will invoke a copy of the DOS command processor (COMMAND.COM by default) to execute the command.

If you supply a file name extension in the command line, MAKE searches only for that extension. Here are some examples:

- This command causes COMMAND.COM to change the current directory to C:\INCLUDE:

```
cd c:\include
```

- MAKE uses the full search algorithm in searching for the appropriate files to perform this command:

```
tlink lib\c0s x y,z,z,lib\cs
```

- MAKE searches for this file using only the .COM extension:

```
myprog.com geo.xyz
```

- MAKE executes this command using the explicit file name provided:

```
c:\myprogs\fil.exe -r
```

Explicit rules

The first rule in the example on page 17 is an explicit rule—a rule that specifies complete file names explicitly. Explicit rules take the form

Note that the braces must be included if you use the paths parameter.

```
target [target ... ] : [{paths}] [dependent ... ]
    [command]
    :
```

where *target* is the file to be updated, *dependent* is a file on which *target* depends, *paths* is a list of directories, separated by semicolons and enclosed in braces, in which dependent files might reside, and *command* is any valid DOS command (including invocation of .BAT files and execution of .COM and .EXE files).

Explicit rules define one or more target names, zero or more dependent files, and an optional list of commands to be performed. Target and dependent file names listed in explicit rules can contain normal DOS drive and directory specifications; they can also contain wildcards.



Syntax here is important.

- *target* must be at the start of a line (in column 1).
- The *dependent* file(s) must be preceded by at least one space or tab, after the colon.
- *paths*, if included, must be enclosed in braces.
- Each *command* must be indented, (must be preceded by at least one blank or tab). As mentioned before, the backslash can be used as a continuation character if the list of dependent files or a given command is too long for one line.

Both the dependent files and the commands are optional; it is possible to have an explicit rule consisting only of *target [target ...]* followed by a colon.

The idea behind an explicit rule is that the command or commands listed will create or update *target*, usually using the *dependent* files. When MAKE encounters an explicit rule, it first checks to see if any of the *dependent* files are themselves target files elsewhere in the makefile. If so, MAKE evaluates that rule first.

MAKE will check for dependent files in the current directory first. If it can't find them, MAKE will then check each of the directories specified in the path list.

Once all the *dependent* files have been created or updated based on other rules, MAKE checks to see if *target* exists. If not, each *command* is invoked in the order given. If *target* does exist, its time and date of last modification are compared against the time and date for each *dependent*. If any *dependent* has been modified more recently than *target*, the list of commands is executed.

A given file name can occur on the left side of an explicit rule only once in a given execution of MAKE.

Each command line in an explicit rule begins with whitespace. MAKE considers all lines following an explicit rule to be part of the command list for that rule, up to the next line that begins in column 1 (without any preceding whitespace) or to the end of the file. Blank lines are ignored.

Special considerations An explicit rule with no command lines following it is treated a little differently than an explicit rule with command lines.

- If an explicit rule includes commands, the only files that the target depends on are the ones listed in the explicit rule.
- If an explicit rule has no commands, the targets depend on two sets of files: the files given in the explicit rule, and any file that matches an implicit rule for the target(s). This lets you specify a dependency to be handled by an implicit rule. For example in

```
.c.obj:  
    BCC -c $<  
prog.obj:
```

PROG.OBJ depends on PROG.C; If PROG.OBJ is out of date, MAKE executes the command line

```
BCC -c prog.c
```


Multiple explicit rules for a single target

A single target may have more than a single explicit rule. You might use multiple explicit rules to create a module library with TLIB, for example, since the .OBJ object module files might be built differently (some with BCC and some with TASM, for example).

The format is the same as for normal explicit rules, except there are two colons following the target. The second colon tells MAKE to expect additional explicit rules for this target.

In the following example, MYLIB.LIB consists of four object modules, two of which are C++ modules. The other two are assembly modules. The first explicit rule compiles the C++ modules and updates the library. The second explicit rule assembles the ASM files and also updates the library.

```
mylib.lib:: f1.cpp f2.cpp
  bcc -c f1.cpp f2.cpp
  tlib mylib -+f1.obj -+f2.obj

mylib.lib:: f3.asm f4.asm
  tasm /mx f3.asm f4.asm
  tlib mylib -+f3.obj -+f4.obj
```

Examples Here are some examples of explicit rules:

1. prog.exe: myprog.obj prog2.obj
BCC myprog.obj prog2.obj
2. myprog.obj: myprog.c include\stdio.h
BCC -c myprog.c
3. prog2.obj: prog2.c include\stdio.h
BCC -c -K prog2.c

The three examples are from the same makefile. Only the modules affected by a change are rebuilt. If PROG2.C is changed, it's the only one recompiled; the same holds true for MYPROG.C. But if the include file stdio.h is changed, both are recompiled. (The link step is done if any of the .OBJ files in the dependency list have changed, which will happen when a recompile results from a change to a source file.)

Automatic dependency checking

Borland C++ works with MAKE to provide automatic dependency checking for include files. BCC and BC produce .OBJ files that tell MAKE what include files were used to create those .OBJ files. MAKE's `-a` command-line option checks this information and makes sure that everything is up-to-date.

When MAKE does an automatic dependency check, it reads the include files' names, times, and dates from the .OBJ file. The autodependency check will also work for include files inside of include files. If any include files have been modified, MAKE causes the .OBJ file to be recompiled. For example, consider the following explicit rule:

```
myprog.obj: myprog.c include\stdio.h
    BCC -c myprog.c
```

Now assume that the following source file, called MYPROG.C, has been compiled with BCC:

```
#include <stdio.h>
#include "dcl.h"

void myprog() {}
```

If you then invoke MAKE with the following command line

```
make -a myprog.obj
```

it checks the time and date of MYPROG.C, and also of `stdio.h` and `dcl.h`.

Implicit rules

MAKE allows you to define *implicit* rules as well as explicit ones. Implicit rules are generalizations of explicit rules; they apply to all files that have certain identifying extensions.

Here's an example that illustrates the relationship between the two rules. Consider this explicit rule from the preceding example. The rule is typical because it follows a general principle: An .OBJ file is dependent on the .C file with the same file name and is created by executing BCC. In fact, you might have a makefile where you have several (or even several dozen) explicit rules following this same format.

By rewriting the explicit rule as an implicit rule, you can eliminate all the explicit rules of the same form. As an implicit rule, it would look like this:

```
.c.obj:  
    BCC -c $<
```

The symbol `$<` is a special macro. Macros are discussed starting on page 28. The `$<` macro will be replaced by the full name of the appropriate .C source file each time the command executes.

This rule means “Any file with the extension .C can be translated to a file of the same name with the extension .OBJ using this sequence of commands.” The .OBJ file is created with the second line of the rule, where `$<` represents the file’s name with the source (.C) extension.

Here’s the syntax for an implicit rule:

```
[[source_dir]].source_extension.{{target_dir}}target_extension:  
    [command]  
    :
```

As before, the commands are optional and must be indented.

`source_dir` (which must be enclosed by braces) tells MAKE to search for source files in the specified directory. `target_dir` tells MAKE where the target files will be placed.

`source_extension` is the extension of the source file; that is, it applies to any file having the format

fname.source_extension

Likewise, the `target_extension` refers to the file

fname.target_extension

where *fname* is the same for both files. In other words, this implicit rule replaces all explicit rules having the format

```
fname.target_extension: fname.source_extension  
    [command]  
    :
```

for any *fname*.



MAKE uses implicit rules if it can’t find any explicit rules for a given target, or if an explicit rule with no commands exists for the target.

The extension of the file name in question is used to determine which implicit rule to use. The implicit rule is applied if a file is found with the same name as the target, but with the mentioned source extension.

For example, suppose you had a makefile (named MAKEFILE) whose contents were

```
.c.obj:  
    BCC -c $<
```

If you had a C program named RATIO.C that you wanted to compile to RATIO.OBJ, you could use the command

```
make ratio.obj
```

MAKE would take RATIO.OBJ to be the target. Since there is no explicit rule for creating RATIO.OBJ, MAKE applies the implicit rule and generates the command

```
BCC -c ratio.c
```

which, of course, does the compile step necessary to create RATIO.OBJ.

MAKE also uses implicit rules if you give it an explicit rule with no commands. Suppose you had the following implicit rule at the start of your makefile:

```
.c.obj:  
    BCC -c $<
```

You could then remove the command from the rule:

```
myprog.obj: myprog.c include\stdio.h  
    BCC -c myprog.c
```

and it would execute exactly as before.

If you're using Borland C++ and you enable automatic dependency checking in MAKE, you can remove all explicit dependencies that have .OBJ files as targets. With automatic dependency checking enabled and implicit rules, the three-rule C example shown in the section on explicit rules becomes

```
.c.obj:  
    BCC -c $<  
  
prog.exe: myprog.obj prog2.obj  
    tlink lib\c0s myprog prog2, prog, , lib\cs
```

*Note that with the -N compatibility option, the searches go in the **opposite** direction: from the bottom of the makefile up.*

You can write several implicit rules with the same target extension. If more than one implicit rule exists for a given target extension, the rules are checked in the order in which they appear in the makefile, until a match is found for the source extension, or until MAKE has checked all applicable rules.

MAKE uses the first implicit rule that involves a file with the source extension. Even if the commands of that rule fail, no more implicit rules are checked.

All lines following an implicit rule, up to the next line that begins without whitespace or to the end of the file, are considered to be part of the command list for the rule.

Macros

Often, you'll find yourself using certain commands, file names, or options again and again in your makefile. For instance, if you're writing a C program that uses the medium memory model, all your BCC commands will use the option `-mm`, which means to compile to the medium memory model. But suppose you wanted to switch to the large memory model. You could go through and change all the `-mm` options to `-ml`. Or, you could define a macro.

A *macro* is a name that represents some string of characters. A macro definition gives a macro name and the expansion text; thereafter, when MAKE encounters the macro name, it replaces the name with the expansion text.

Suppose you defined the following macro at the start of your makefile:

```
MODEL = m
```

This line defines the macro `MODEL`, which is now equivalent to the string `m`. Using this macro, you could write each command to invoke the C compiler to look something like this:

```
BCC -c -m$(MODEL) myprog.c
```

When you run MAKE, each macro (in this case, `$(MODEL)`) is replaced with its expansion text (here, `m`). The command that's actually executed would be

```
BCC -c -mm myprog.c
```

Now, changing memory models is easy. If you change the first line to

```
MODEL = l
```

you've changed all the commands to use the large memory model. In fact, if you leave out the first line altogether, you can specify which memory model you want each time you run MAKE, using the `-D` (define) command-line option:

```
make -DMODEL=l
```

This tells MAKE to treat **MODEL** as a macro with the expansion text *l*.

Defining macros Macro definitions take the form

```
macro_name = expansion text
```

where *macro_name* is the name of the macro. *macro_name* should be a string of letters and digits with no whitespace in it, although you can have whitespace between *macro_name* and the equal sign (=). The *expansion text* is any arbitrary string containing letters, digits, whitespace, and punctuation; it is ended by newline.

If *macro_name* has previously been defined, either by a macro definition in the makefile or on the MAKE command line, the new definition replaces the old.

Case is significant in macros; that is, the macro names **model**, **Model**, and **MODEL** are all different.

Using macros You invoke macros in your makefile using this format

```
$(macro_name)
```

You need the parentheses for all invocations, except when the macro name is just one character long. This construct—*\$(macro_name)*—is known as a *macro invocation*.

When MAKE encounters a macro invocation, it replaces the invocation with the macro's expansion text. If the macro is not defined, MAKE replaces it with the null string.

Using environment variables as macros

If you invoke a macro where *macro_name* hasn't been defined in the makefile or on the command line, MAKE will try to find *macro_name* as a DOS environment variable. If MAKE does find it in the environment, the expansion text will be the value of the environment variable.



Macros defined in the makefile or on the command line override environment variables of the same name unless the **-e** option was specified.

Substitution within macros

You can invoke a macro while simultaneously changing some of its text by using a special format of the macro invocation format. Instead of the standard macro invocation form, use

```
$(macro_name:text1=text2)
```

In this form, every occurrence of *text1* in *macro_name* will be replaced with *text2*. *macro_name* can also be one of the predefined macros. This is useful if you'd prefer to keep only a single list of files in a macro. For example, in the following example, the macro **SOURCE** contains a list of all the C++ source files a target depends on. The TLINK command line changes all the .CPP extensions to the matching .OBJ object files and links those.

Note that no extraneous whitespace should appear between the : and =. If spaces appear after the colon, MAKE will attempt to find the string, including the preceding space.

```
SOURCE = f1.cpp f2.cpp f3.cpp
myapp.exe: $(SOURCE)
    bcc -c $(SOURCE)
    tlink c0s $(SOURCE:.cpp=.obj),myapp,,cs
```

Special considerations

Macros in macros: Macros cannot be invoked on the left side (*macro_name*) of a macro definition. They can be used on the right side (*expansion text*), but they are not expanded until the macro being defined is invoked. In other words, when a macro invocation is expanded, any macros embedded in its expansion text are also expanded.

Macros in rules: Macro invocations are expanded immediately in rule lines.

See page 35 for information on directives.

Macros in directives: Macro invocations are expanded immediately in **!if** and **!elif** directives. If the macro being invoked in an **!if** or **!elif** directive is not currently defined, it is expanded to the value 0 (FALSE).

Macros in commands: Macro invocations in commands are expanded when the command is executed.

Predefined macros

MAKE comes with several special macros built in: **\$d**, **\$***, **\$<**, **\$:**, **\$.**, **\$&**, **\$@**, **\$****, and **\$?**. The first is a test to see if a macro name is defined; it's used in the conditional directives **!if** and **!elif**. The others are file name macros, used in explicit and implicit rules. Finally, MAKE defines several other macros; see Table 2.3.

Table 2.3
MAKE predefined macros

<code>--MSDOS--</code>	"1" if running MAKE under DOS
<code>--MAKE--</code>	MAKE's version number in hexadecimal (for this version, "0x0360")
MAKE	MAKE's executable filename (usually MAKE or MAKER)
MAKEFLAGS	Any options used on the MAKE command line
MAKEDIR	The directory from which MAKE was run

Table 2.4
MAKE filename macros

Macro	What part of the file name it returns in an	
	implicit rule	explicit rule
\$*	Dependent base with path	Target base with path
\$<	Dependent full with path	Target full with path
\$:	Dependent path only	Target path only
\$.	Dependent full without path	Target full without path
\$&	Dependent base without path	Target base without path
\$@	Target full with path	Target full with path
**	Dependent full with path	All dependents
 \$?	Dependent full with path	All out of date dependents

Defined Test Macro (\$d)

The defined test macro (**\$d**) expands to 1 if the given macro name is defined, or to 0 if it is not. The content of the macro's expansion text does not matter. This special macro is allowed only in **!if** and **!elif** directives.

For example, suppose you want to modify your makefile so that if you don't specify a memory model, it'll use the medium one. You could put this at the start of your makefile:

```
!if !$d(MODEL) # if MODEL is not defined
MODEL=m      # define it to m (MEDIUM)
!endif
```

If you then invoke MAKE with the command line

```
make -DMODEL=l
```

then **MODEL** is defined as *l*. If, however, you just invoke MAKE by itself,

```
make
```

then **MODEL** is defined as *m*, your "default" memory model.

File name macros The various file name macros work in similar ways, expanding to some variation of the full path name of the file being built.

Base file name macro (\$*)

The base file name macro is allowed in the commands for an explicit or an implicit rule. This macro (\$*) expands to the file name being built, excluding any extension, like this:

```
File name is A:\P\TESTFILE.C
$* expands to A:\P\TESTFILE
```

For example, you could modify this explicit rule

```
prog.exe: myprog.obj prog2.obj
         tlink lib\c0s myprog prog2, prog, , lib\cs
```

to look like this:

```
prog.exe: myprog.obj prog2.obj
         tlink lib\c0s myprog prog2, $*, , lib\cs
```

When the command in this rule is executed, the macro \$* is replaced by the target file name without an extension and with a path. For implicit rules, this macro is very useful.

For example, an implicit rule might look like this:

```
.cpp.obj:
         BCC -c $*
```

Full file name macro (\$<)

The full file name macro (\$<) is also used in the commands for an explicit or implicit rule. In an explicit rule, \$< expands to the full target file name (including extension), like this:

```
File name is A:\P\TESTFILE.C
$< expands to A:\P\TESTFILE.C
```

For example, the rule

```
mylib.obj: mylib.c
         copy $< \oldobjs
         BCC -c $*.c
```

copies MYLIB.OBJ to the directory \OLDOBJS before compiling MYLIB.C.

In an implicit rule, `$<` takes on the file name plus the source extension. For example, the implicit rule

```
.c.obj:  
    BCC -c $*.c
```

produces exactly the same result as

```
.c.obj:  
    BCC -c $<
```

because the extension of the target file name *must* be `.C`.

File name path macro (`$:`)

This macro expands to the path name (without the file name), like this:

```
File name is A:\P\TESTFILE.C  
$: expands to A:\P\
```

File name and extension macro (`$.`)

This macro expands to the file name, with an extension but without the path name, like this:

```
File name is A:\P\TESTFILE.C  
$. expands to TESTFILE.C
```

File name only macro (`$&`)

This macro expands to the file name only, without path or extension, like this:

```
File name is A:\P\TESTFILE.C  
$& expands to TESTFILE
```

Full target name with path macro (`$@`)

This macro expands to the full target file name with path and extension, like this:

```
File name is A:\P\TESTFILE.C  
$@ expands to A:\P\TESTFILE.C
```

The `$@` macro is similar to the `$<` macro, except that `$@` expands to the full target file name in *both* implicit and explicit rules, which

expands to the target in an explicit rule and the dependent in an implicit rule.

All dependents macro (\$**)

In an explicit rule, this macro expands to all the dependents of the target, including the full filename with path and extension. For example, in the following explicit rule, the **\$**** will be replaced with `myprog.obj prog2.obj`, the two dependents of `prog.exe`:

```
prog.exe: myprog.obj prog2.obj
        tlink lib\c0s $**, $*, , lib\cs
```

All out of date dependents macro (\$?)

In an explicit rule, this macro expands to all the out of date dependents of the target, including the full filename with path and extension. Out of date dependents are those that have been modified since the target was last made. Therefore, in the following example explicit rule, the **\$?** will be replaced with `f1.cpp` and/or `f2.cpp` depending on which dependent(s) were out of date:

```
mylib.lib: f1.cpp f2.cpp
        bcc -c $?
        &tlib mylib -+$(?:.cpp=.obj)
```

Note the use of the **&** prefix so MAKE will repeat the command for each of the out of date dependents.

Macro modifiers If there isn't a predefined filename macro to give you the parts of a filename you need, you can use *macro modifiers* to extract any part of a filename macro. The format is:

`$(macro[D | F | B | R])`

where *macro* is one of the predefined filename macros and D, F, B, and R are the modifiers. Note that since the macro is now longer than a single character, parentheses are necessary. The following table describes what each modifier does. The examples assume that **\$<** returns `C:\OBJ\BOB.OBJ`.

Table 2.5
MAKE macro modifiers

Modifier	What part of the filename	Example
D	Drive and directory	\$(<D) = C:\OBJS\
F	Base and extension	\$(<F) = BOB.OBJ
B	Base only	\$(<B) = BOB
R	Drive, directory, and base	\$(<R) = C:\OBJS\BOB

Directives

Borland's MAKE allows something that other versions of MAKE don't: directives similar to those allowed in C, assembler, and Turbo Pascal. You can use these directives to perform a variety of useful and powerful actions. Some directives in a makefile begin with an exclamation point (!) as the first character of the line. Others begin with a period. Here is the complete list of MAKE directives:

Table 2.6
MAKE directives

.autodepend	Turns on autodependency checking.
!elif	Conditional execution.
!else	Conditional execution.
!endif	Conditional execution.
!error	Causes MAKE to stop and print an error message.
!if	Conditional execution.
!ifdef	Conditional execution.
!ifndef	Conditional execution.
.ignore	Tells MAKE to ignore return value of a command.
!include	Specifies a file to include in the makefile.
.noautodepend	Turns off autodependency checking.
.noignore	Turns off .ignore .
.nosilent	Tells MAKE to print commands before executing them.
.noswap	Tells the real mode version of MAKE to not swap itself in and out of memory. Has no effect in the protected-mode version of MAKE.
.path.ext	Gives MAKE a path to search for files with extension .EXT .
.precious	Tells MAKE to not delete the specified target even if the commands to build the target fail.
.silent	Tells MAKE to not print commands before executing them.
.swap	Tells the real mode version of MAKE to swap itself in and out of memory. Has no effect in the protected-mode version of MAKE.
.suffixes	Tells MAKE the implicit rule to use when a target's dependent is ambiguous.
!undef	Causes the definition for a specified macro to be forgotten.

Dot directives

Each of the following directives has a corresponding command-line option, but takes precedence over that option. For example, if you invoke MAKE like this:

```
make -a
```

but the makefile has a **.noautodepend** directive, then autodependency checking will be off.

.autodepend and **.noautodepend** turn on or off autodependency checking. They correspond to the **-a** command-line option.

.ignore and **.noignore** tell MAKE whether or not to ignore the return value of a command, much like placing the prefix **-** in front of it (described earlier). They correspond to the **-i** command-line option.

.silent and **.nosilent** tell MAKE whether or not to print commands before executing them. They correspond to the **-s** command-line option.

.swap and **.noswap** tell MAKE whether or not to swap itself out of memory. They correspond to the **-S** option.

.precious The syntax for the **.precious** directive is:

```
.precious: target [...]
```

where *target* is one or more target files. **.precious** prevents MAKE from deleting the target if the commands building the target fail. In some cases, the target is still viable. For example, if an object module can't be added to a library, the library shouldn't be deleted, or, when building a Windows application, if the Resource Compiler fails, the .EXE executable shouldn't be deleted.

.path.ext This directive, placed in a makefile, tells MAKE where to look for files of the given extension. For example, if the following is in a makefile:

```
.path.c = C:\CSOURCE
.c.obj:
    BCC -c $<
tmp.exe: tmp.obj
    BCC tmp.obj
```

MAKE will look for TMP.C, the implied source file for TMP.OBJ, in C:\CSOURCE instead of the current directory.

The **.path** is also a macro that has the value of the path. The following is an example of the use of **.path**. The source files are contained in one directory, the .OBJ files in another, and all the .EXE files in the current directory.

```
.path.c = C:\CSOURCE
.path.obj = C:\OBJS

.c.obj:
    BCC -c -o$(.path.obj)\$& $<

.obj.exe:
    BCC -e$&.exe $<

tmp.exe: tmp.obj
```

.suffixes In the following example, MYPROG.OBJ can be created from MYPROG.ASM, MYPROG.CPP, and MYPROG.C.

```
myprog.exe: myprog.obj
    tlink myprog.obj

.asm.obj:
    tasm /mx $<

.cpp.obj:
    bcc -P $<

.c.obj:
    bcc -P- $<
```

If more than one of these sources is available, the **.suffixes** directive determines which will be used. The syntax for **.suffixes** is:

```
.suffixes: source_extension ...
```

where *source_extension* is a list of the extensions for which there are implicit rules, in order of which source extension implicit rule should be used.

For example, if we add **.suffixes: .asm .c .cpp** to the top of the previous makefile example, MAKE would first look for MYPROG.ASM, then MYPROG.C, and finally MYPROG.CPP.

File-inclusion directive

A file-inclusion directive (**!include**) specifies a file to be included into the makefile for interpretation at the point of the directive. It takes the following form:

```
!include filename
```

filename can be surrounded by quotes ("*filename*") or angle brackets (<*filename*>). You can nest these directives to any depth. If an include directive attempts to include a file that has already been included in some outer level of nesting (so that a nesting loop is about to start), the inner include directive is rejected as an error.

How do you use this directive? Suppose you created the file MODEL.MAC that contained the following:

```
!ifndef MODEL
MODEL=m
!endif
```

You could use this conditional macro definition in any makefile by including the directive

```
!include "MODEL.MAC"
```

When MAKE encounters **!include**, it opens the specified file and reads the contents as if they were in the makefile itself.

Conditional execution directives

Conditional execution directives (**!if**, **!ifdef**, **!ifndef**, **!elif**, **!else**, and **!endif**) give you a measure of flexibility in constructing makefiles. Rules and macros can be made conditional, so that a command-line macro definition (using the **-D** option) can enable or disable sections of the makefile.

The format of these directives parallels those in C, assembly language, and Turbo Pascal:

```
!if expression
[lines]
:
!endif

!if expression
```

```

[lines]
:
!else
[lines]
:
!endif

!if expression
[lines]
:
!elif expression
[lines]
:
!endif

!ifdef macro
[lines]
:
!endif

!ifndef macro
[lines]
:
!endif

```

[lines] can be any of the following statement types:

- macro definition
- explicit rule
- implicit rule
- include directive
- if group
- error directive
- undef directive

The conditional directives form a group, with at least an **!if**, **!ifdef**, or **!ifndef** directive beginning the group and an **!endif** directive closing the group.

- One **!else** directive can appear in the group.
- **!elif** directives can appear between the **!if** (or **!ifdef** and **!ifndef**) and any **!else** directives.
- Rules, macros, and other directives can appear between the various conditional directives in any number. Note that complete rules, with their commands, cannot be split across conditional directives.
- Conditional directive groups can be nested to any depth.

Any rules, commands, or directives must be complete within a single source file.

All **lif**, **lifdef**, and **lifndef** directives must have matching **lendif** directives within the same source file. Thus the following include file is illegal, regardless of what's in any file that might include it, because it doesn't have a matching **lendif** directive:

```
!if $(FILE_COUNT) > 5
    some rules
!else
    other rules
<end-of-file>
```

The **lifdef** directive is another way of testing whether a macro is defined. **lifdef** MACRO is equivalent to **!if** \$d(MACRO). The same holds true for **lifndef**; **lifndef** MACRO is equivalent to **!if** ! \$d(MACRO).

Expressions allowed in conditional directives

If you program in assembly language or Turbo Pascal, be sure to look closely at the examples that follow.

Expressions are allowed in an **!if** or an **!elif** directive; they use a C-like syntax. The expression is evaluated as a simple 32-bit signed integer or strings of characters.

You can enter numbers as decimal, octal, or hexadecimal constants. If you know the C language, you already know how to write constants in MAKE; the formats are exactly the same. These are legal constants in a MAKE expression:

```
4536 # decimal constant
0677 # octal constant (distinguished by leading 0)
0x23aF # hexadecimal constant (distinguished by leading 0x)
```

Any expression that doesn't follow one of those formats is considered a string.

An expression can use any of the following operators (an asterisk indicates the operator is also available with string expressions):

Table 2.7
MAKE operators

See Chapter 2 of the
Programmer's Guide for
complete descriptions of
these operators.

Operator	Operation	Operator	Operation
<i>Unary operators</i>			
-	Negation	&	Bitwise AND
~	Bit complement		Bitwise OR
!	Logical NOT	^	Bitwise XOR
<i>Binary operators</i>			
+	Addition	&&	Logical AND
-	Subtraction		Logical OR
*	Multiplication	>	Greater than*
/	Division	<	Less than*
%	Remainder	>=	Greater than or equal*
>>	Right shift	<=	Less than or equal*
<<	Left shift	==	Equality*
		!=	Inequality*
		<i>Ternary operator</i>	
		?:	Conditional expression

The operators have the same precedences as they do in the C language. Parentheses can be used to group operands in an expression. Unlike the C language, MAKE can compare strings using the normal ==, !=, >, <, >=, and => operators. You can't compare a string expression with a numeric expression, nor can you use numeric operators (like + or *) with strings.

A string expression may contain spaces, but if it does it must be enclosed in quotes:

```
Model = "Medium model"
:
:
!if $(Model) == "Medium model"
CFLAGS = -mm
!elif $(Model) == "Large model"
CFLAGS = -ml
!endif
```

You can invoke macros within an expression; the special macro **\$d()** is recognized. After all macros have been expanded, the expression must have proper syntax.

Error directive

The error directive (**!error**) causes MAKE to stop and print a fatal diagnostic containing the text after **!error**. It takes the format

```
!error any_text
```

This directive is designed to be included in conditional directives to allow a user-defined error condition to abort MAKE. For example, you could insert the following code in front of the first explicit rule:

```
!if !$d(MODEL)
# if MODEL is not defined
!error MODEL not defined
!endif
```

If you reach this spot without having defined **MODEL**, then MAKE stops with this error message:

```
Fatal makefile 4: Error directive: MODEL not defined
```

Macro undefinition directive

The macro “undefinition” directive (**!undef**) causes any definition for the named macro to be forgotten. If the macro is currently undefined, this directive has no effect. The syntax is

```
!undef macro_name
```

The compatibility option **-N**

The **-N** command line option increases compatibility with Microsoft's NMAKE. You should use it only when you need to build a project using makefiles created for NMAKE tools. Running MAKE without the **-N** option is preferred, since **-N** introduces some subtle differences in how makefiles work:

- `$$` expands to a single `$` and a single `$` expands to nothing.
- The caret character `^` causes the character that follows, if a special character, to be treated literally. For example,

```
TEST = this is ^
a test
```

will cause **TEST** to expand to `this is \na test` where the `\n` is the C symbol for a new line. It's especially useful when you need to end a line with the line continuation character:

```
SOURCEDIR = C:\BOB\OBJ$^\
```



- If the caret is followed by a normal character (one without a special meaning), the caret will be ignored.

- ❑ The **\$d** macro won't be the special defined test macro. Use the **ifdef** and **ifndef** directives instead.
- ❑ Predefined macros that return paths only will *not* end in a trailing backslash. For example, without the **-N** switch $\$(<D)$ might return `C:\OBJS\`, but with the **-N** switch, the same $\$(<D)$ macro would return `C:\OBJS`.
- ❑ Unless there's a matching **.suffixes** directive, MAKE will search for implicit rules from the bottom of the makefile up.
- ❑ The **\$*** macro always expands to the target name. (In normal mode, **\$*** expands to the dependent in an implicit rule.)

TLIB: The Turbo librarian

TLIB is a utility that manages libraries of individual .OBJ (object module) files. A library is a convenient tool for dealing with a collection of object modules as a single unit.

When it modifies an existing library, TLIB always creates a copy of the original library with a .BAK extension. Better safe than sorry!

The libraries included with Borland C++ were built with TLIB. You can use TLIB to build your own libraries, or to modify the Borland C++ libraries, your own libraries, libraries furnished by other programmers, or commercial libraries you've purchased. You can use TLIB to

- *create* a new library from a group of object modules
- *add* object modules or other libraries to an existing library
- *remove* object modules from an existing library
- *replace* object modules from an existing library
- *extract* object modules from an existing library
- *list* the contents of a new or existing library

See the section on the /E option (page 49) for details.

TLIB can also create (and include in the library file) an Extended Dictionary, which may be used to speed up linking.

Although TLIB is not essential for creating executable programs with Borland C++, it is a useful programming productivity tool. You will find TLIB indispensable for large development projects. If you work with object module libraries developed by others, you can use TLIB to maintain those libraries when necessary.

Why use object module libraries?

When you program in C and C++, you often create a collection of useful functions and classes. Because of C and C++'s modularity, you are likely to split those functions into many separately compiled source files. You use only a subset of functions from the entire collection in any particular program. It can become quite tedious, however, to figure out exactly which files you are using. On the other hand, if you always include all the source files, your program becomes extremely large and unwieldy.

An object module library solves the problem of managing a collection of functions and classes. When you link your program with a library, the linker scans the library and automatically selects only those modules needed for the current program.

The TLIB command line

To get a summary of TLIB's usage, just type `TLIB` and press Enter.

The TLIB command line takes the following general form, where items listed in square brackets (*like this*) are optional:

```
tlib [/C] [/E] [/Psize] libname [operations] [, listfile]
```

Table 3.1: TLIB options

Option	Description
<i>libname</i>	The DOS path name of the library you want to create or manage. Every TLIB command must be given a <i>libname</i> . Wildcards are not allowed. TLIB assumes an extension of .LIB if none is given. We recommend that you do not use an extension other than .LIB, since both BCC and BC's project-make facility require the .LIB extension in order to recognize library files. Note: If the named library does not exist and there are <i>add</i> operations, TLIB creates the library.
<i>/C</i>	The case-sensitive flag. This option is not normally used; see page 50 for a detailed explanation.
<i>/E</i>	Creates Extended Dictionary; see page 49 for a detailed explanation.
<i>/Psize</i>	Sets the library page size to <i>size</i> ; see page 50 for a detailed explanation.
<i>operations</i>	The list of operations TLIB performs. Operations may appear in any order. If you only want to examine the contents of the library, don't give any operations.
<i>listfile</i>	The name of the file listing library contents. The <i>listfile</i> name (if given) must be preceded by a comma. No listing is produced if you don't give a file name. The listing is an alphabetical list of each module. The entry for each module contains an alphabetical list of each public symbol defined in that module. The default extension for the <i>listfile</i> is .LST. You can direct the listing to the screen by using the <i>listfile</i> name CON, or to the printer by using the name PRN.

This section summarizes each of these command-line components; the following sections provide details about using TLIB. For TLIB examples, refer to the "Examples" section on page 51.

The operation list

The operation list describes what actions you want TLIB to do. It consists of a sequence of operations given one after the other. Each operation consists of a one- or two-character *action symbol* followed by a file or module name. You can put whitespace around either the action symbol or the file or module name, but not in the middle of a two-character action or in a name.

You can put as many operations as you like on the command line, up to DOS's COMMAND.COM-imposed line-length limit of 127 characters. The order of the operations is not important. TLIB always applies the operations in a specific order:

1. All extract operations are done first.
2. All remove operations are done next.
3. All add operations are done last.

You can replace a module by first removing it, then adding the replacement module.

File and module names

TLIB finds the name of a module by taking the given file name and stripping any drive, path, and extension information from it. (Typically, drive, path, and extension are not given.)

Note that TLIB always assumes reasonable defaults. For example, to add a module that has an .OBJ extension from the current directory, you only need to supply the module name, not the path and .OBJ extension.

Wildcards are never allowed in file or module names.

TLIB operations

TLIB recognizes three action symbols (-, +, *), which you can use singly or combined in pairs for a total of five distinct operations. The order of the characters is not important for operations that use a pair of characters. The action symbols and what they do are listed here:

Table 3.2
TLIB action symbols

To create a library, add modules to a library that does not yet exist.

Action symbol	Name	Description
+	Add	TLIB adds the named file to the library. If the file has no extension given, TLIB assumes an extension of .OBJ. If the file is itself a library (with a .LIB extension), then the operation adds all of the modules in the named library to the target library. If a module being added already exists, TLIB displays a message and does not add the new module.
-	Remove	TLIB removes the named module from the library. If the module does not exist in the library, TLIB displays a message. A remove operation only needs a module name. TLIB allows you to enter a full path name with drive and extension included, but ignores everything except the module name.
*	Extract	TLIB creates the named file by copying the corresponding module from the library to the file. If the module does not exist, TLIB displays a message and does not create a file. If the named file already exists, it is overwritten.

Table 3.2: TLIB action symbols (continued)

You can't directly rename modules in a library. To rename a module, extract and remove it, rename the file just created, then add it back into the library.

-*	Extract &	TLIB copies the named module to the
*-	Remove	corresponding file name and then removes it from the library. This is just shorthand for an <i>extract</i> followed by a <i>remove</i> operation.
-+	Replace	TLIB replaces the named module with the corresponding file. This is just shorthand for a <i>remove</i> followed by an <i>add</i> operation.
+ -		

Using response files

When you are dealing with a large number of operations, or if you find yourself repeating certain sets of operations over and over, you will probably want to start using *response files*. A response file is simply an ASCII text file (which can be created with the Borland C++ editor) that contains all or part of a TLIB command. Using response files, you can build TLIB commands larger than would fit on one DOS command line.

See "Examples" for a sample response file and a TLIB command line incorporating it.

To use a response file *pathname*, specify *@pathname* at any position on the TLIB command line.

- ▣ More than one line of text can make up a response file; you use the "and" character (&) at the end of a line to indicate that another line follows.
- ▣ You don't need to put the entire TLIB command in the response file; the file can provide a portion of the TLIB command line, and you can type in the rest.
- ▣ You can use more than one response file in a single TLIB command line.

Creating an extended dictionary: The /E option

To speed up linking with large library files (such as the standard Cx.LIB library), you can direct TLIB to create an *extended dictionary* and append it to the library file. This dictionary contains, in a very compact form, information that is not included in the standard library dictionary. This information enables TLINK to process library files faster.

To create an extended dictionary for a library that is being modified, use the **/E** option when you invoke TLIB to add, remove, or replace modules in the library. To create an extended dictionary for an existing library that you don't want to modify, use the **/E** option and ask TLIB to remove a nonexistent module from the library. TLIB will display a warning that the specified module was not found in the library, but it will also create an extended dictionary for the specified library. For example, if you enter

```
tlib /E mylib -bogus
```

TLINK will ignore the debugging information in a library that has an extended dictionary, unless the **/e** option is used on the TLINK command line.

Setting the page size: The **/P** option

Every DOS library file contains a dictionary (which appears at the end of the .LIB file, following all of the object modules). For each module in the library, this dictionary contains a 16-bit address of that particular module within the .LIB file; this address is given in terms of the library page size (it defaults to 16 bytes).

The library page size determines the maximum combined size of all object modules in the library—it cannot exceed 65,536 pages. The default (and minimum) page size of 16 bytes allows a library of about 1 MB in size. To create a larger library, the page size must be increased using the **/P** option; the page size must be a power of 2, and it may not be smaller than 16 or larger than 32,768.

All modules in the library must start on a page boundary. For example, in a library with a page size of 32 (the lowest possible page size higher than the default 16), on the average 16 bytes will be lost per object module in padding. If you attempt to create a library that is too large for the given page size, TLIB will issue an error message and suggest that you use **/P** with the next available higher page size.

Advanced operation: The **/C** option

When you add a module to a library, TLIB maintains a dictionary of all public symbols defined in the modules of the library. All symbols in the library must be distinct. If you try to add a module

to the library that would cause a duplicate symbol, TLIB displays a message and won't add the module.

Normally, when TLIB checks for duplicate symbols in the library, uppercase and lowercase letters are not considered as distinct. For example, the symbols *lookup* and *LOOKUP* are treated as duplicates. Since C and C++ *do* treat uppercase and lowercase letters as distinct, use the **/C** option to add a module to a library that includes a symbol differing *only in case* from one already in the library. The **/C** option tells TLIB to accept a module with symbols in it that differ only in case from symbols already in the library.

*If you want to use the library with other linkers (or allow other people to use the library with other linkers), for your own protection you should **not** use the /C option.*

It may seem odd that, without the **/C** option, TLIB rejects symbols that differ only in case, especially since C and C++ are case-sensitive languages. The reason is that some linkers fail to distinguish between symbols in a library that differ only in case. Such linkers, for example, will treat *stars*, *Stars*, and *STARS* as the same identifier. TLINK, on the other hand, has no problem distinguishing uppercase and lowercase symbols, and it will properly accept a library containing symbols that differ only in case. In this example, then, Borland C++ would treat *stars*, *Stars*, and *STARS* as three separate identifiers. As long as you use the library only with TLINK, you can use the TLIB **/C** option without any problems.

Examples

Here are some simple examples demonstrating the different things you can do with TLIB.

1. To create a library named MYLIB.LIB with modules X.OBJ, Y.OBJ, and Z.OBJ, type

```
tlib mylib +x +y +z
```

2. To create a library as in #1 and get a listing in MYLIB.LST too, type

```
tlib mylib +x +y +z, mylib.lst
```

3. To get a listing in CS.LST of an existing library CS.LIB, type

```
tlib cs, cs.lst
```

4. To replace module X.OBJ with a new copy, add A.OBJ and delete Z.OBJ from MYLIB.LIB, type

```
tlib mylib -x +a -z
```

5. To extract module Y.OBJ from MYLIB.LIB and get a listing in MYLIB.LST, type

```
tlib mylib *y, mylib.lst
```

6. To create a new library named ALPHA, with modules A.OBJ, B.OBJ, ..., G.OBJ using a response file:

First create a text file, ALPHA.RSP, with

```
+a.obj +b.obj +c.obj &  
+d.obj +e.obj +f.obj &  
+g.obj
```

Then use the TLIB command, which produces a listing file named ALPHA.LST:

```
tlib alpha @alpha.rsp, alpha.lst
```

TLINK: The Turbo linker

Appendix A, "Error messages," lists linker messages generated by TLINK and by the built-in IDE linker.

The IDE has its own built-in linker. When you invoke the command-line compiler BCC, TLINK is invoked automatically unless you suppress the linking stage. If you suppress the linking stage, you must invoke TLINK manually. This chapter describes how to use TLINK as a standalone linker.

By default, the command-line compiler calls TLINK when compilation is successful; TLINK then combines object modules and library files to produce the executable file.

Invoking TLINK

Note that this version of TLINK is sensitive to the case of its options; */t* is not the same option as *T*.

You can invoke TLINK at the command line by typing `tlink` with or without parameters. When invoked without parameters, TLINK displays a summary of parameters and options. Table 4.1 briefly describes the TLINK options.

Table 4.1
TLINK options

You can use either a hyphen or a slash to precede TLINK's commands.

Option	What it does
<i>/3</i>	Enables processing of 32-bit modules.
<i>/A=nnnn</i>	Specifies segment alignment for NewExe (Windows) images.
<i>/c</i>	Treats case as significant in symbols.
<i>/C</i>	Treats EXPORTS and IMPORTS section of module definition file as case sensitive.
<i>/d</i>	Warns if duplicate symbols in libraries.
<i>/e</i>	Ignores Extended Dictionary.
<i>/i</i>	Initializes all segments.

Table 4.1: TLINK options (continued)

/l	Includes source line numbers.
/L	Specifies library search paths.
/m	Creates map file with publics.
/n	Doesn't use default libraries.
/b	Overlays following modules or libraries.
/P	Packs code segments.
/s	Creates detailed map of segments.
/t	Generates .COM file. (Also /Tdc.)
/Td	Creates target DOS executable.
/Tdc	Creates target DOS .COM file.
/Tde	Creates target DOS .EXE file.
/Tw	Creates target Windows executable (.DLL or .EXE).
/Twe	Creates target Windows application (.EXE).
/Twd	Creates target Windows DLL (.DLL).
/v	Includes full symbolic debug information.
/x	Doesn't create map file.
/ye	Uses expanded memory for swapping.
/yx	Configures TLINK's use of extended memory.

The general syntax of a TLINK command line is

TLINK *objfiles, exe file, mapfile, libfiles, deffile*

This syntax specifies that you supply file names *in the given order*, separating the file *types* with commas.

An example of linking for DOS

If you supply the TLINK command line

```
tlink /c mainline wd ln tx,fin,mfin,work\lib\comm work\lib\support
```

TLINK will interpret it to mean that

- ▣ Case is significant during linking (**/c**).
- ▣ The .OBJ files to be linked are MAINLINE.OBJ, WD.OBJ, LN.OBJ, and TX.OBJ.
- ▣ The executable program name will be FIN.EXE.
- ▣ The map file is MFIN.MAP.
- ▣ The library files to be linked in are COMM.LIB and SUPPORT.LIB, both of which are in subdirectory WORK\LIB.
- ▣ No module definition file is specified.

An example of linking for Windows



This example shows how complex linking a Windows application can get. It's much easier to use the command line compiler or IDE project manager, because they will automatically provide the correct link switches and libraries for the memory model.

To create a Windows application executable, you might use this command line:

```
tlink /Tw /c \BORLANDC\lib\c0ws winapp1 winapp2, winapp, winapp,  
\BORLANDC\lib\cws \BORLANDC\lib\import, winapp.def
```

where

- The **/Tw** option tells TLINK to generate Windows executables.
- The **/c** option tells TLINK to be case-sensitive during linking. Note that the EXPORTS and IMPORTS sections in the module definition file will be still treated as case-insensitive unless the **/C** option is used.
- **\BORLANDC\LIB\C0WS** is the standard Windows initialization file and **WINAPP1** and **WINAPP2** are the module's object files; for all three files the **.OBJ** extension is assumed.
- **WINAPP.EXE** is the name of the target Windows executable.
- **WINAPP.MAP** is the name of the map file.
- **\BORLANDC\LIB\CWS** is the small memory model runtime library for Windows and **\BORLANDC\LIB\IMPORT** is the library that provides access to the built-in Windows functions.
- **WINAPP.DEF** is the Windows module definition file used to specify additional link options.

File names on the TLINK command line

If you don't specify an executable file name, TLINK derives the name of the executable by appending **.EXE** or **.DLL** to the first object file name listed.

If you specify a complete file name for the executable file, TLINK will create the file with that name, but the actual nature of that executable depends on other options or on settings in the module definition file. For instance, if you specify **WINAPP.EXE**, but you provide the **/Twd** option, the executable will be created as a **DLL** but named **WINAPP.EXE**—probably not what you intended. Similarly, if you give **WINAPP.DLL** as the executable name, but include a **/Td** option on the command line, the file will be a **DOS** executable.

If no map file name is given, TLINK adds a .MAP extension to the .EXE file name. If no libraries are included, none will be linked. If you don't specify a module definition (.DEF) file *and* you have used the **/Tw** option, TLINK creates a Windows application based on default settings.

TLINK assumes or appends extensions to file names that have none:



- .OBJ for object files
- .EXE for DOS and Windows executable files (when you use the **/t** or the **/Tdc** option, the executable file extension defaults to .COM rather than .EXE)
- .DLL for Windows dynamic link libraries (when you use the **/Twd** option, or the **/Tw** option and the module definition file specifies a library)
- .MAP for map files
- .LIB for library files
- .DEF for module definition files.

All of the file names *except* object files are optional. So, for instance,

```
TLINK dosapp dosapp2
```

links the files DOSAPP.OBJ and DOSAPP2.OBJ, creates a DOS executable file called DOSAPP.EXE, creates a map file called DOSAPP.MAP, links no libraries, and uses no module definition file.

Using response files

TLINK lets you supply the various parameters on the command line, in a response file, or in any combination of the two.

A response file is just a text file that contains the options and file names that you would usually type in after the name TLINK on your command line.

Unlike the command line, however, a response file can be continued onto several lines of text. You can break a long list of object or library files into several lines by ending one line with a plus character (+) and continuing the list on the next line. When a plus occurs at the end of a line but it immediately follows one of the TLINK options that uses + to enable the option (such as **/ye+**), the + is not treated as a line continuation character.

You can also start each of the four components on separate lines: object files, executable file, map file, libraries. When you do this, you must leave out the comma used to separate components.

To illustrate these features, suppose that you rewrote the command line

```
tlink /c mainline wd ln tx,fin,mfin,work\lib\comm work\lib\support
```

with the following response file, FINRESP:

```
/c mainline wd+
ln tx,fin
mfin
work\lib\comm work\lib\support
```

You would then enter your TLINK command as

```
tlink @finresp
```

Note that you must precede the file name with an “at” character (@) to indicate that the next name is a response file.

Alternately, you may break your link command into multiple response files. For example, you can break the previous command line into the following two response files:

File name	Contents
LISTOBS	mainline+ wd+ ln tx
LISTLIBS	lib\comm+ lib\support

You would then enter the TLINK command as

```
tlink /c @listobjs,fin,mfin,@listlibs
```

The TLINK configuration file

The command line version of TLINK looks for a file called TLINK.CFG first in the current directory, or in the directory from which it was loaded.

TLINK.CFG is a regular text file that contains a list of valid TLINK options. Unlike a response file, TLINK.CFG can't list the groups of file names to be linked.

For instance, the following TLINK.CFG file

```
/Lc:\BORLANDC\lib;c:\winapps\lib  
/v /s  
/Tw
```

tells TLINK to search the specified directories for libraries, include debug information, create a detailed segment map, and produce a Windows program.

Using TLINK with Borland C++ modules

Borland C++ supports six different memory models: tiny, small, compact, medium, large, and huge. When you create an executable Borland C++ file using TLINK, you must include the initialization module and libraries for the memory model being used.

The general format for linking Borland C++ programs with TLINK is

```
tlink C0[W | D | F]x myobjs, exe, [map], [IMPORT] [mylibs]  
[OVERLAY] [CWx | Cx] [EMU | FP87 mathx], [deffile]
```

where

- *myobjs* is the .OBJ files you want linked, specify path if not in current directory
- *exe* is the name to be given the executable file
- (optional) ■ *map* is the name to be given the map file
- (optional) ■ *mylibs* is the library files you want included at link time. You must specify the path if not in current directory, or use **/L** option to specify search paths
- *deffile* is the module definition file for a Windows executable

Be sure to include paths for the startup code and libraries (or use the **/L** option to specify a list of search paths for startup and library files). The other file names on this general TLINK command line represent Borland C++ files, as follows:

- C0x | C0Fx | C0Wx | C0Dx is the initialization module for DOS executable, DOS executable written for another compiler, Windows application, or Windows DLL (choose one) with memory model **t** (DOS only), **s**, **c**, **m**, **l**, or **h** (DOS only).
- IMPORT is the Windows import library; the library that provides access to the built-in Windows functions.
- OVERLAY is the overlay manager library; needed only for overlaid programs (not compatible with Windows).

If you are using the tiny model and you want TLINK to produce a .COM file, you must also specify the /t or /Tdc option.

DOS only!

- **CWx** is the run-time library for executables under Windows with memory model **s**, **c**, **m**, or **l**.
- **EMU|FP87** is the floating-point libraries (choose one).
- **MATHx** is the math library for memory model **s**, **c**, **m**, **l**, or **h**.
- **Cx** is the run-time library for memory model **s**, **c**, **m**, **l**, or **h**.

Startup code



The initialization modules have the name **C0x.OBJ**, **C0Wx.OBJ**, or **C0Dx.OBJ** (for DOS, a Windows application, and a Windows DLL, respectively), where *x* is a single letter corresponding to the model: *t* for tiny (DOS only), *s* for small, *c* for compact, *m* for medium, *l* for large, and *h* for huge (DOS only).

The **C0Fx.OBJ** modules are provided for compatibility with source files intended for compilers from other vendors. The **C0Fx.OBJ** modules substitute for the **C0x.OBJ** modules; they are to be linked with DOS applications only, not Windows applications or DLLs. These initialization modules alter the memory model so that the stack segment is inside the data segment. The appropriate **C0Fx.OBJ** module will be used automatically if you use either the **-Fs** or the **-Fm** command-line compiler option.

Failure to link in the correct initialization module usually results in a long list of error messages telling you that certain identifiers are unresolved, that no stack has been created, or that fixup overflows occurred.


The initialization module must also appear as the first object file in the list. The initialization module arranges the order of the various segments of the program. If it is not first, the program segments may not be placed in memory properly, causing some frustrating program bugs.

Be sure that you give an explicit name for the executable file name on the **TLINK** command line. Otherwise, your program name will be something like **C0x.EXE**—probably not what you wanted!

Libraries



The order of objects and libraries is very important. You must always put the Borland C++ start-up module (**C0x.OBJ**, **C0Fx**, **C0Wx.OBJ**, or **C0Dx.OBJ**) first in the list of objects. Then, the library list should contain, in this order:

- your own libraries (if any)
 - if you want to overlay your program (DOS only), you must include OVERLAY.LIB; this library must precede the Cx.LIB library
- 
- CWx.LIB, the run-time library for Windows, or Cx.LIB, run-time library for DOS
 - if you are using floating point math, FP87.LIB or EMU.LIB (required for DOS only), followed by MATHx.LIB or MATHWx.LIB



If you want to create a Windows application or DLL you must link IMPORT.LIB to provide access to the built-in Windows functions. IMPORT.LIB can be included anywhere in the list.

BGI graphics library

DOS only! If you are using any Borland C++ BGI graphics functions, you must link in GRAPHICS.LIB anywhere in the list. The BGI graphics library is independent of memory models, but is for DOS only (not Windows).

Math libraries

If your program uses any floating-point, you must include a math library (MATHx.LIB or MATHWx.LIB) in the link command. For DOS applications (but not for Windows applications or DLLs), you will also need to include either the EMU.LIB or FP87.LIB floating-point libraries. Borland C++'s two floating-point libraries are independent of the program's memory model.

- Use EMU.LIB if you want to include floating-point emulation logic. With EMU.LIB the program will work on machines whether they have a math coprocessor (80x87) chip or not.
- If you know that the program will always be run on a machine with a math coprocessor chip, the FP87.LIB library will produce a smaller and faster executable program.

The math libraries have the name MATHx.LIB, where *x* is a single letter corresponding to the model: s, c, m, l, h (the tiny and small models share the library MATHS.LIB).

You can always include the emulator (DOS only) and math libraries in a link command line. If you do so, and if your program does no floating-point work, nothing from those libraries

will be added to your executable program file. However, if you know there is no floating-point work in your program, you can save some time in your links by excluding those libraries from the command line.

Run-time libraries

You must always include the C run-time library for the program's memory model. The C run-time libraries have the name Cx.LIB, where x is a single letter corresponding to the model, as before. Use the same C run-time library for both DOS and Windows executables.

If you don't use all six memory models, you may want to keep only the files for the model(s) you use.

Here's a list of the library files needed for each memory model (you'll also need FP87.LIB or EMU.LIB for DOS only, and IMPORT.LIB for Windows):

Table 4.2

DOS application .OBJ and .LIB files

Model	Regular Startup Module	Compatibility Startup Module	Math Library	Run-time Library
Tiny	C0T.OBJ	C0FT.OBJ	MATHS.LIB	CS.LIB
Small	C0S.OBJ	C0FS.OBJ	MATHS.LIB	CS.LIB
Compact	C0C.OBJ	C0FC.OBJ	MATHC.LIB	CC.LIB
Medium	C0M.OBJ	C0FM.OBJ	MATHM.LIB	CM.LIB
Large	C0L.OBJ	C0FL.OBJ	MATHL.LIB	CL.LIB
Huge	C0H.OBJ	C0FH.OBJ	MATHH.LIB	CH.LIB

Note that the tiny and small models use the same libraries, but have different startup files (C0T.OBJ vs. C0S.OBJ).

Table 4.3

Windows application .OBJ and .LIB files

Model	Startup for applications	Windows RTL	Math Library
Small	C0WS.OBJ	CWS.LIB	MATHWS.LIB
Compact	C0WC.OBJ	CWC.LIB	MATHWC.LIB
Medium	C0WM.OBJ	CWM.LIB	MATHWM.LIB
Large	C0WL.OBJ	CWL.LIB	MATHWL.LIB

Table 4.4

DLL object and library files

Model	Startup for DLLs	Windows RTL	Math Library
Small	C0DS.OBJ	CWC.LIB	MATHWC.LIB
Compact	C0DC.OBJ	CWC.LIB	MATHWC.LIB
Medium	C0DM.OBJ	CWL.LIB	MATHWL.LIB
Large	C0DL.OBJ	CWL.LIB	MATHWL.LIB

See Chapter 8, "Building a Windows application" in the Programmer's Guide for more information on DLLs.

Using TLINK with BCC

See Chapter 5, "The command-line compiler," in the Programmer's Guide for more on BCC.

You can also use BCC, the standalone Borland C++ compiler, as a "front end" to TLINK that will invoke TLINK with the correct startup file, libraries, and executable program name.

To do this, you give file names on the BCC command line with explicit .OBJ and .LIB extensions. For example, given the following BCC command line,

```
BCC -mx MAINFILE.OBJ SUB1.OBJ MYLIB.LIB
```

BCC will invoke TLINK with the files C0x.OBJ, EMU.LIB, MATHx.LIB and Cx.LIB (initialization module, default 8087 emulation library, math library and run-time library for memory model *x*). TLINK will link these along with your own modules MAINLINE.OBJ and SUB1.OBJ, and your own library MYLIB.LIB.



To compile and link a Windows program, include one of the **-W** options on the compiler command-line, as well as any other options. The compiler will take care of linking in C0Wx.OBJ, CWx.LIB, and IMPORT.LIB.

When BCC invokes TLINK, it uses the **/c** (case-sensitive link) option by default. You can override this default with **-l -c**.

TLINK options

TLINK options can occur anywhere on the command line. The options consist of a slash (/), a hyphen (-), or the DOS switch character, followed by the option.

If you have more than one option, spaces are not significant (**/m/c** is the same as **/m /c**), and you can have them appear in different places on the command line. The following sections describe each of the options.

The TLINK configuration file

The command-line version of TLINK looks for a file called TLINK.CFG first in the current directory, or in the directory from which it was loaded.

TLINK.CFG is a regular text file that contains a list of valid TLINK options. Unlike a response file, TLINK.CFG can't list the groups of file names to be linked. Whitespace is ignored.

For instance, the following TLINK.CFG file

```
/Lc:\BORLANDC\lib;c:\winapps\lib  
/v /s  
/Tw
```

tells TLINK to search the specified directories for libraries, include debug information, create a detailed segment map, and produce a Windows program.

/3 (32-bit code)

This option increases the memory requirements of TLINK and slows down linking, so it should be used only when necessary.

The **/3** option should be used when one or more of the object modules linked has been produced by TASM or a compatible assembler, and contains 32-bit code for the 80386 or the i486 processor.

/A (align segments)

The **/A** option specifies a byte value on which to align segments. Segments smaller than the specified value will be padded up to the value. The syntax is

```
/A=nnnn
```

where *nnnn* is a number which represents the alignment factor. *nnnn* must be a power of two. For instance, **/A=16** indicates that segments should be aligned on a paragraph boundary.

The default segment alignment size is 512. For efficiency, you should use the smallest value that still allows for correct segment offsets in the segment table. The file addresses in the segment table are multiplied by the alignment factor in order to be used as byte offsets into the executable file. Since the offsets are stored as 16-bit words, 65536 times the alignment factor is the limit of segment offsets that can be represented in the segment table. If you get this message, increase the segment alignment value.

/c (case sensitivity)

The **/c** option forces the case to be significant in public and external symbols.

/C (case sensitive exports)

By default, TLINK treats the EXPORTS and IMPORTS sections of the module definition file as case-insensitive. The **/C** or **/C+** option turns on case-sensitivity; **/C-** turns off case-sensitivity.

/d (duplicate symbols)

Normally, TLINK will not warn you if a symbol appears in more than one library file. If the symbol must be included in the program, TLINK will use the copy of that symbol in the first file on the command line in which it is found. Since this is a commonly used feature, TLINK does not normally warn about the duplicate symbols. The following hypothetical situation illustrates how you might want to use this feature.

Suppose you have two libraries: one called SUPPORT.LIB, and a supplemental one called DEBUGSUP.LIB. Suppose also that DEBUGSUP.LIB contains duplicates of some of the routines in SUPPORT.LIB (but the duplicate routines in DEBUGSUP.LIB include slightly different functionality, such as debugging versions of the routines). If you include DEBUGSUP.LIB *first* in the link command, you will get the debugging routines and *not* the routines in SUPPORT.LIB.

If you are not using this feature or are not sure which routines are duplicated, you may include the **/d** option. TLINK will list all symbols duplicated in libraries, even if those symbols are not going to be used in the program.

Given this option, TLINK will also warn about symbols that appear both in an .OBJ and a .LIB file. In this case, since the symbol that appears in the first (left-most) file listed on the command line is the one linked in, the symbol in the .OBJ file is the one that will be used.

The exception to this rule is OVERLAY.LIB. OVERLAY.LIB does duplicate some symbols found in other libraries; that's why OVERLAY.LIB must be the first standard library specified on the TLINK command line.

With Borland C++, the distributed libraries you would use in any given link command do not contain any duplicated symbols. So while EMU.LIB and FP87.LIB (or CS.LIB and CL.LIB) obviously have duplicate symbols, they would never rightfully be used together in a single link. There are no symbols duplicated between EMU.LIB, MATHS.LIB, and CS.LIB, for example.

/e (no extended dictionary)

The library files that are shipped with Borland C++ all contain an *extended dictionary* with information that enables TLINK to link faster with those libraries. This extended dictionary can also be added to any other library file using the **/E** option with TLIB (see the section on TLIB starting on page 45). The TLINK **/e** option disables the use of this dictionary.

Although linking with libraries that contain an extended dictionary is faster, you might want to use the **/e** option if you have a program that needs slightly more memory to link when an extended dictionary is used.



Unless you use **/e** to turn off extended dictionary use, TLINK will ignore any debugging information contained in a library that has an extended dictionary.

/i (uninitialized trailing segments)

The **/i** option causes uninitialized trailing segments to be output into the executable file even if the segments do not contain data records. This option is not normally necessary.

/l (line numbers)

The **/l** option creates a section in the .MAP file for source code line numbers. To use it, you must have created the .OBJ files by compiling with the **-y** or **-v** option. If you use the **/x** to tell TLINK to create no map at all, this option will have no effect.

/L (library search paths)

The **/L** option lets you specify a list of directories that TLINK searches for libraries if an explicit path is not specified. TLINK searches the current directory before those specified with the **/L** option. For example,

```
TLINK /Lc:\BORLANDC\lib;c:\mylibs splash logo,,,utils .\logolib
```

With this command line, TLINK first searches the current directory for UTILS.LIB, then searches C:\BORLANDC\LIB and C:\MYLIBS. Because .\LOGOLIB explicitly names the current directory, TLINK does not search the libraries specified with the **/L** option to find LOGOLIB.LIB.

TLINK also searches for the C or C++ initialization module (C0x.OBJ, C0Wx.OBJ, C0Dx.OBJ) on the specified library search path.

`/m`, `/s`, and `/x` (map options)

By default, TLINK always creates a map of the executable file. This default map includes only the list of the segments in the program, the program start address, and any warning or error messages produced during the link. If you don't want to create a map, turn it off with the `/x` option.

If you want to create a more complete map, the `/m` option will add a list of public symbols to the map file, sorted alphabetically as well as in increasing address order. This kind of map file is useful in debugging. Many debuggers can use the list of public symbols to allow you to refer to symbolic addresses when you are debugging.

The `/s` option creates a map file with segments, public symbols and the program start address just like the `/m` option did, but also adds a detailed segment map. Figure 4.1 is an example of a detailed segment map.

For each segment in each module, this map includes the address, length in bytes, class, segment name, group, module, and ACBP information.

If the same segment appears in more than one module, each module will appear as a separate line (for example, SYMB.C). Except for the ACBP field, the information in the detailed segment map is self-explanatory.

Figure 4.1
Detailed map of segments

Address	Length (Bytes)	Class	Segment Name	Group	Module	Alignment/ Combining
0000:0000	0E5B	C=CODE	S=SYMB_TEXT	G=(none)	M=SYMB.C	ACBP=28
00E5:000B	2735	C=CODE	S=QUAL_TEXT	G=(none)	M=QUAL.C	ACBP=28
0359:0000	002B	C=CODE	S=SCOPY_TEXT	G=(none)	M=SCOPY	ACBP=28
035B:000B	003A	C=CODE	S=LRSH_TEXT	G=(none)	M=LRSH	ACBP=20
035F:0005	0083	C=CODE	S=PADA_TEXT	G=(none)	M=PADA	ACBP=20
0367:000B	005B	C=CODE	S=PADD_TEXT	G=(none)	M=PADD	ACBP=20
036D:0003	0025	C=CODE	S=PSBP_TEXT	G=(none)	M=PSBP	ACBP=20
036F:0008	05CE	C=CODE	S=BRK_TEXT	G=(none)	M=BRK	ACBP=28
03CC:0006	066F	C=CODE	S=FLOAT_TEXT	G=(none)	M=FLOAT	ACBP=20
0433:0006	000B	C=DATA	S=_DATA	G=DGROUP	M=SYMB.C	ACBP=48
0433:0012	00D3	C=DATA	S=_DATA	G=DGROUP	M=QUAL.C	ACBP=48
0433:00E6	000E	C=DATA	S=_DATA	G=DGROUP	M=BRK	ACBP=48
0442:0004	0004	C=BSS	S=_BSS	G=DGROUP	M=SYMB.C	ACBP=48
0442:0008	0002	C=BSS	S=_BSS	G=DGROUP	M=QUAL.C	ACBP=48
0442:000A	000E	C=BSS	S=_BSS	G=DGROUP	M=BRK	ACBP=48

The ACBP field encodes the A (*alignment*), C (*combination*), and B (*big*) attributes into a set of four bit fields, as defined by Intel. TLINK uses only three of the fields, the A, C, and B fields. The ACBP value in the map is printed in hexadecimal: The following values of the fields must be OR'ed together to arrive at the ACBP value printed.

Field	Value	Description
The A field (alignment)	00	An absolute segment.
	20	A byte-aligned segment.
	40	A word-aligned segment.
	60	A paragraph-aligned segment.
	80	A page-aligned segment.
	A0	An unnamed absolute portion of storage.
The C field (combination)	00	May not be combined.
	08	A public combining segment.
The B field (big)	00	Segment less than 64K.
	02	Segment exactly 64K.

When you request a detailed map with the */s* option, the list of public symbols (if it appears) has public symbols flagged with "idle" if there are no references to that symbol. For example, this fragment from the public symbol section of a map file indicates that symbols *Symbol1* and *Symbol3* are not referenced by the image being linked:

```
0C7F:031E  idle  Symbol1
0000:3EA2                Symbol2
0C7F:0320  idle  Symbol3
```

/n (ignore default libraries)

The **/n** option causes the linker to ignore default libraries specified by some compilers. You may want to use this option when linking modules written in another language.

/o (overlays)

The **/o** option causes the code in all modules or libraries specified after the option to be overlaid. It remains in effect until the next comma (explicit or implicit) or **/o-** on the command line. **/o-** turns off overlaying. (Chapter 9, "DOS memory management," in the *Programmer's Guide* covers overlays in more detail.)

The **/o** option can be optionally followed by a segment class name; this will cause all segments of that class to be overlaid. When no such name is specified, all segments of classes ending with CODE will be overlaid. Multiple **/o** options can be given, thus overlaying segments of several classes; all **/o** options remain in effect until the next comma or **/o-** is encountered.

The syntax **/o#xx**, where *xx* is a two-digit hexadecimal number, overrides the overlay interrupt number, which by default is 3FH.

Here are some examples of **/o** options:

Table 4.5
TLINK overlay options

Option	Result
/o	Overlay all code segments until next comma or /o- .
/o-	Stop overlaying.
/oOVY	Overlay segments of class OVY until the next comma or /o- .
/oCODE /oOVLY	Overlay segments of class CODE or class OVLY until next comma or /o- .
/o#F0	Use interrupt vector 0F0H for overlays.

If you use the **/o** option, it will be turned off automatically before the libraries are processed. If you want to overlay a library, you must use another **/o** right before all the libraries or right before the library you want to overlay.



You can't use the **/o** option with any **/Tw** option; Windows applications can't be overlaid. However, in order to achieve essentially the same results under Windows, use discardable code

segments (see page 72 for information on defining code segments attributes in the module definition file).

/P (pack code segments)



When you use **/P**, when TLINK links Windows executables, TLINK combines as many code segments as possible in one physical segment up to the code segment packing limit. Code segment packing never creates segments greater than this limit; TLINK starts a new segment if it needs to.

The default code segment packing limit is 8,192 bytes (8K). To change it, use

`/P=n`

where *n* specifies the number of bytes between 1 and 65,536. You would probably want the limit to be a multiple of 4K under 386 enhanced mode.

Although the optimum segment size in 386 enhanced mode is 4K, the default code segment packing size is 8K. Because typical code segments are likely to be from 4K to 8K, an 8K packing size will probably result in more effective packing.

Because there is a certain amount of system overhead for every segment maintained, code segment packing, by reducing the number of segments to maintain, typically increases performance. The **/P** option is on by default. **/P-** turns off code segment packing (useful if you've turned it on in the configuration file and want to disable it for a particular link).

/t (tiny model .COM file)

If you compile your file in the tiny memory model and link it with this option toggled on, TLINK will generate a .COM file instead of the usual .EXE file. Also, when you use **/t**, the default extension for the executable file is .COM. This works the same as the **/Tdc** option. Neither **/t** or **/Tdc** is compatible with the Windows option, **/Tw**.

Note: .COM files may not exceed 64K in size, cannot have any segment-relative fixups, cannot define a stack segment, and must have a starting address equal to 0:100H. When an extension other than .COM is used for the executable file (.BIN, for example), the starting address may be either 0:0 or 0:100H.

TLINK can't generate debugging information for a .COM file. If you need to debug your program, create and debug it as an .EXE file, then relink it as a .COM file. Alternatively, if you have Turbo Debugger, you can use the TDSTRIP utility with the **-c** option; this creates a .COM file from an .EXE.

/Td and /Tw (target options)

These options are called target options. You use them (with **c**, **e**, or **d**) to produce a .COM, .EXE, or .DLL file.

- **/Td** creates a DOS .EXE file.
- **/Tdc** creates a DOS .COM file.
- **/Tde** creates a DOS .EXE file.



- **/Tw** tells TLINK to create a Windows executable file. This option is not necessary if you include a module definition file with an EXETYPE Windows statement. With or without the **/Tw** option, if the included module definition file has a NAME statement, TLINK creates an application (.EXE); if the module definition file has a LIBRARY statement, TLINK creates a DLL.

If no module definition file is included in the link, you must specify the **/Tw** or **/Twe** option for a Windows .EXE, or the **/Twd** option for a Windows DLL.

None of the **/Tw** options are compatible with the **b** option (overlay modules).

- **/Twe** creates Windows .EXE files. The **/Twe** option overrides a LIBRARY statement in the module definition file (which normally causes TLINK to create a DLL).
- **/Twd** creates Windows DLLs. The **/Twd** option overrides a NAME statement in the module definition file (which normally causes TLINK to create an .EXE file).

/v (debugging information)

The **/v** option directs TLINK to include debugging information in the executable file. If this option is found anywhere on the command line, debugging information will be included executable for all object modules that contained debugging information. You can use the **/v+** and **/v-** options to selectively enable or disable inclusion of debugging information on a module-by-module basis (but not on the same command line as **/v**). For example, this command

```
tlink mod1 /v+ mod2 mod3 /v- mod4
```

includes debugging information for modules *mod2* and *mod3*, but not for *mod1* and *mod4*.

TLINK can't generate debugging information for a .COM file (one created with the **/t** or **/Tdc** options). If you need to debug your program, create and debug it as an .EXE file, then relink it as a .COM file. Alternatively, if you have Turbo Debugger, you can use the TDSTRIP utility with the **-c** option; this creates a .COM file from an .EXE.

/ye (expanded memory)

This option controls TLINK's use of expanded memory for I/O buffering. If, while reading object files or while writing the executable file, TLINK needs more memory for active data structures, it will either purge buffers or swap them to expanded memory.

In the case of input file buffering, purging simply means throwing away the input buffer so that its space can be used for other data structures. In the case of output file buffering, purging means writing the buffer to its correct place in the executable file. In either case, you can substantially increase the speed of a link by allowing these buffers to be swapped to expanded memory.

TLINK's capacity is not increased by swapping; only its performance is improved. By default, swapping to extended memory is enabled, while swapping to expanded memory is disabled. If swapping is enabled and no appropriate memory exists in which to swap, then swapping does not occur.

This option has several forms, shown below

/ye or /ye+	enable expanded memory swapping (default)
/ye-	disable expanded memory swapping

/yx (extended memory)

The **/yx** option controls TLINK's use of extended memory for I/O buffering. By default, TLINK will take up to 8MB of extended memory. You can change TLINK's use of extended memory with one of the following forms of this option:

/yx+	Use all available extended memory.
/yxn	Use only up to <i>n</i> KB extended memory.

The module definition file



The module definition file provides information to the linker about the contents and system requirements of a Windows application. More specifically, it

- names the application or dynamic link library (DLL)
- identifies the type of application as Windows or OS/2
- lists imported functions and exported functions
- describes the code and data segment attributes; allows you to specify attributes for additional code and data segments
- specifies the size of the heap and stack
- provides for the inclusion of a DOS stub program

Note that the IMPLIB utility can use a module definition file to create an import library (see page 6). The IMPDEF utility can actually create a module definition file for use with IMPLIB (see page 3).

Module definition file defaults

The module definition file is not strictly necessary to produce a Windows executable under Borland C++.

If no module definition file is specified, the following defaults are assumed.

CODE	PRELOAD MOVEABLE DISCARDABLE
DATA	PRELOAD MOVEABLE MULTIPLE (for applications) or PRELOAD MOVEABLE SINGLE (for DLLs)
HEAPSIZE	4096
STACKSIZE	5120

To replace the EXETYPE statement, the Borland C++ linker can discover what kind of executable you want to produce by checking settings in the IDE or options on the command line.

You can include an import library to substitute for the IMPORTS section of the module definition.

You can use the **_export** keyword in the definitions of export functions in your C and C++ source code to remove the need for an EXPORTS section. Note, however, that if **_export** is used to

export a function, that function will be exported by name rather than by ordinal (ordinal is usually more efficient).

If you want to change various attributes from the default, you'll need to have a module definition file.

A quick example

Here's a module definition from the WHELLO example, discussed in Chapter 8, "Building a Windows application," in the *Programmer's Guide*:

```
NAME            WHELLO
DESCRIPTION     'C++ Windows Hello World'
EXETYPE        WINDOWS
CODE           MOVEABLE
DATA          MOVEABLE MULTIPLE
HEAPSIZE      1024
STACKSIZE     5120
EXPORTS       MainWindowProc
```

Let's take this file apart, statement by statement:

- **NAME** specifies a name for an application. If you want to build a DLL instead of an application, you would use the **LIBRARY** statement instead. Every module definition file should have either a **NAME** statement or a **LIBRARY** statement, but never both. The name specified must be the same name as the executable file.
- **DESCRIPTION** lets you specify a string that describes your application or library.
- **EXETYPE** can be either **WINDOWS** or **OS2**. Only **WINDOWS** is supported in this version of Borland C++.
- **CODE** defines the default attributes of code segments. The **MOVEABLE** option means that the code segment can be moved in memory at run-time.
- **DATA** defines the default attributes of data segments. **MOVEABLE** means that it can be moved in memory at run-time. Windows lets you run more than one instance of an application at the same time. In support of that, the **MULTIPLE** options ensures that each instance of the application has its own data segment.
- **HEAPSIZE** specifies the size of the application's local heap.

- `STACKSIZE` specifies the size of the application's local stack. You can't use the `STACKSIZE` statement to create a stack for a DLL.
- `EXPORTS` lists those functions in the `WHELLO` application that will be called by other applications or by Windows. Functions that are intended to be called by other modules are called callbacks, callback functions, or export functions.
- To help you avoid the necessity of creating and maintaining long `EXPORTS` sections, Borland C++ provides the `_export` keyword. Functions flagged with `_export` will be identified by the linker and entered into an export table for the module. If the Smart Callbacks option is used at compile time (`/WS` on the `BCC` command-line, or Options | Compiler | Entry/Exit Code | Windows Smart Callbacks), then callback functions do *not* need to be listed either in the `EXPORTS` statement or flagged with the `_export` keyword. Borland C++ compiles them in such a way so that they can be callback functions.

This application doesn't have an `IMPORTS` statement, because the only functions it calls from other modules are those from the Windows API; those functions are imported via the automatic inclusion of the `IMPORT.LIB` import library. When an application needs to call other external functions, these functions must be listed in the `IMPORTS` statement, or included via an import library (see page 6 for a discussion of import libraries).

This application doesn't include a `STUB` statement. Borland C++ uses a built-in stub for Windows applications. The built-in stub simply checks to see if the application was loaded under Windows, and, if not, terminates the application with a message that Windows is required. If you want to write and include a custom stub, specify the name of that stub with the `STUB` statement.

Module definition reference

This section describes each statement in a module definition file.

CODE

The `CODE` statement defines the default attributes of code segments. Code segments can have any name, but must belong to segment classes whose name ends in `CODE`. For instance, valid segment class names are `CODE` or `MYCODE`. The syntax is

CODE [FIXED | MOVEABLE]
[DISCARDABLE | NONDISCARDABLE]
[PRELOAD | LOADONCALL]

FIXED means that the segment remains at a fixed memory location; MOVEABLE means that the segment can be moved.

DISCARDABLE means that the segment can be discarded if it is no longer needed. DISCARDABLE implies MOVEABLE. NONDISCARDABLE means that the segment can't be discarded.

PRELOAD means that the segment is loaded when the module is first loaded; LOADONCALL means that the segment is loaded when code in this segment is called. The Resource Compiler and the Windows loader set the code segment containing the initial program entry point to PRELOAD regardless of the setting in the module definition file.

Default attributes for code segments are

- FIXED
- NONDISCARDABLE
- LOADONCALL

DATA

The DATA statement defines the default attributes of data segments.

The syntax of the DATA statement is

DATA [NONE | SINGLE | MULTIPLE] [FIXED | MOVEABLE]

NONE means that there is no data segment. If you specify NONE, do not include any other options. This option is available only for libraries.

SINGLE means that a single segment is shared by all instances of the module. MULTIPLE means that each instance of an application has a segment. SINGLE is only valid for libraries; MULTIPLE is only valid for applications.

FIXED means that the segment remains at a fixed memory location. MOVEABLE means that the segment can be moved.

The default attributes for data segments in applications are FIXED MULTIPLE. For libraries the default attributes are FIXED SINGLE.

The automatic data segment is the segment whose group is DGROUP. This physical segment also contains the local heap and stack (see the HEAPSIZE and STACKSIZE module definition file statements). The Resource Compiler and the Windows loader set the automatic data segment to be PRELOAD, regardless of the setting in the module definition file.

DESCRIPTION

The DESCRIPTION statement inserts text into the application module. The DESCRIPTION statement is typically used to embed author, date, or copyright information. DESCRIPTION is an optional statement. The syntax is

```
DESCRIPTION 'Text'
```

Text specifies an ASCII string delimited with single quotes.

EXETYPE

The EXETYPE statement specifies the default executable file (.EXE) header type (Windows or OS/2). You can only specify WINDOWS in this version of Borland C++, so the syntax is

```
EXETYPE WINDOWS
```

EXPORTS

The EXPORTS statement defines the names and attributes of the functions to be exported. The EXPORTS keyword marks the beginning of the definitions. It can be followed by any number of export definitions, each on a separate line. The syntax is

```
EXPORTS  
  ExportName [Ordinal] [RESIDENTNAME] [NODATA] [Parameter]
```

ExportName specifies an ASCII string that defines the symbol to be exported. It has the following form:

```
EntryName [=InternalName]
```

InternalName is the name used within the application to refer to this entry. *EntryName* is the name listed in the executable file's entry table is externally visible.

Ordinal defines the function's ordinal value. It has the following form:

@ordinal

where *ordinal* is an integer value that specifies the function's ordinal value.

When an application module or DLL module calls a function exported from a DLL, the calling module can refer to the function by name or by ordinal value. In terms of speed, referring to the function by ordinal is faster since string comparisons are not required to locate the function. In terms of memory allocation, exporting a function by ordinal (from the point of view of that function's DLL) and importing/calling a function by ordinal (from the point of view of the calling module) is more efficient. When a function is exported by ordinal, the name resides in the non-resident name table. When a function is exported by name, the name resides in the resident name table. The resident name table for a module is resident in memory whenever the module is loaded; the non-resident name table isn't.

The RESIDENTNAME option lets you specify that the function's name must be resident at all times. This is useful only when exporting by ordinal (when the name wouldn't be resident by default).

The NODATA option lets you specify that the function is not bound to a specific data segment. The function will use the current data segment.

Parameter is an optional integer value that specifies the number of words the function expects to be passed as parameters.

HEAPSIZE

The HEAPSIZE statement defines the number of bytes needed by the application for its local heap. An application uses the local heap whenever it allocates local memory. The syntax is

HEAPSIZE *bytes*

bytes is an integer value that specifies the heap size in bytes. It must not exceed 65,536 (the physical segment size).

The default heap size is zero. The minimum size is 256 bytes. The sum total of the automatic data segment (DGROUP), the local heap, and the stack must not exceed 65,536.

IMPORTS

The **IMPORTS** statement defines the names and attributes of the functions to be imported from dynamic link libraries. Instead of listing imported DLL functions in the **IMPORTS** statement, you can either specify an import library for the DLL in the **TLINK** command line, or—in the IDE—include the import library for the DLL in the project.

The **IMPORTS** key word marks the beginning of the definitions. It can be followed by any number of import definitions, each on a separate line. The syntax is

```
IMPORTS
    [InternalName=]ModuleName.Entry
```

InternalName is an ASCII string that specifies the unique name that the application will use to call the function.

ModuleName specifies one or more uppercase ASCII characters that define the name of the executable module that contains the function. The module name must match the name of the executable file. For example, the file **SAMPLE.DLL** has the the module name **SAMPLE**.

Entry specifies the function to be imported. It can be either an ASCII string that names the function, or an integer that gives the function's ordinal value.

LIBRARY

The **LIBRARY** statement defines the name of a DLL module. A module definition file can contain either a **NAME** statement to indicate an application or a **LIBRARY** statement to indicate a DLL, but not both.

Like an application's module name, a library's module name must match the name of the executable file. For example, the library **MYLIB.DLL** has the module name **MYLIB**. The syntax is

```
LIBRARY LibraryName
```

LibraryName specifies an ASCII string that defines the name of the library module.

The start address of the library module is determined by the library's object files; it is an internally defined function.

LibraryName is optional. If the parameter is not included, TLINK uses the filename part of the executable file (that is, the name with the extension removed).

If the module definition file includes neither a NAME nor a LIBRARY statement, TLINK assumes a NAME statement without a *ModuleName* parameter.

NAME

The NAME statement defines the name of the application's executable module. The module name identifies the module when exporting functions. The syntax is

```
NAME ModuleName
```

ModuleName specifies one or more uppercase ASCII characters that define the name of the executable module. The module name must match the name of the executable file. For example, an application with the executable file SAMPLE.EXE has the module name "SAMPLE".

The *ModuleName* parameter is optional. If the parameter is not included, TLINK assumes that the module name matches the filename of the executable file. For example, if you do not specify a module name and the executable file is named MYAPP.EXE, TLINK assumes that the module name is "MYAPP".

If the module definition file includes neither a NAME nor a LIBRARY statement, TLINK assumes a NAME statement without a *ModuleName* parameter.

SEGMENTS

The SEGMENTS statement defines the segment attributes of additional code and data segments. The syntax is

```
SEGMENTS  
  SegmentName [CLASS 'ClassName'] [MinAlloc]  
  [FIXED | MOVEABLE]  
  [DISCARDABLE | NONDISCARDABLE]  
  [SHARED | NONSHARED]  
  [PRELOAD | LOADONCALL]
```

SegmentName specifies a character string that names the new segment. It can be any name, including the standard segment names `_TEXT` and `_DATA`, which represent the standard code and data segments.

ClassName is an optional key word that specifies the class name of the specified segment. If no class name is specified, TLINK uses the class name CODE by default.

MinAlloc is an optional integer value that specifies the minimum allocation size for the segment. Currently, TLINK ignores this value.

FIXED means that the segment remains at a fixed memory location. The MOVEABLE option means that the segment can be moved if necessary, in order to compact memory.

DISCARDABLE means that the segment can be discarded if it is no longer needed; NONDISCARDABLE means that the segment can not be discarded.

PRELOAD means that the segment is loaded immediately; LOADONCALL means that the segment is loaded when it is accessed or called. The Resource Compiler may override the LOADONCALL option and preload segments instead.

Default attributes for additional segments are as described for CODE and DATA segments (depending on the type of additional segment).

STACKSIZE

The STACKSIZE statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it makes function calls. Do not use the STACKSIZE statement for dynamic link libraries. The syntax is

`STACKSIZE bytes`

bytes is an integer value that specifies the stack size in bytes.

If the application makes no function calls, STACKSIZE defaults to 0. If your application does make function calls the minimum size is 5120 bytes (if you specify less, it will be changed to 5120). The sum total of the automatic data segment (DGROUP), the local heap, and the stack must not exceed 65,536.

STUB

The STUB statement appends a DOS executable file specified by *FileName* to the beginning of the module. The executable stub should display a warning message and terminate if the user doesn't have Windows loaded.

Borland C++ adds a built-in stub to the beginning of a Windows application unless a different stub is specified with the STUB statement. Therefore, you should not use the STUB statement merely to include WINSTUB.EXE, because the linker will do this for you automatically.

The syntax is

```
STUB "FileName"
```

FileName specifies the name of the DOS executable file that will be appended to the module. The name must have the DOS file name format.

If the file named by *FileName* is not in the current directory, TLINK searches for the file in the directories specified by the PATH environment variable.

Using WinSight

WinSight is a debugging tool that gives you information about windows, window classes, and messages. You can use it to study a Windows application—yours or others—to see how windows and window classes are created and used, and what messages the windows receive.

You can configure WinSight to trace messages

- by window
- by window class
- by message type
- by a combination of these

Remember that WinSight is a passive observer: It intercepts and displays information about messages, but it does not keep messages from getting to other applications.

Getting started

Double-clicking on the WinSight icon brings up the main window in its default configuration, which is a list of all the windows currently active on the desktop.

In general, you will find that you want to use a mouse to manipulate items in the WinSight window. Table 5.1 summarizes the mouse actions and their keyboard and menu equivalents.

Table 5.1: Mouse and keyboard actions

Desired action	With the mouse	With the keyboard	With menus
Select an item	Left click	↑ or ↓	
Move selection bar		Ctrl+↑ or Ctrl+↓	
Toggle highlighted item	Ctrl+Left click	Spacebar	
Show details of item	Left Double-click	Enter	Spy Open detail
Expand window tree		+	Tree Expand one level
Expand a branch	Right click on <->	*	Tree Expand branch
Collapse window tree	Left click on <+>	-	Tree Collapse branch
Expand tree completely		Ctrl+*	Tree Expand all
Activate following pane		Tab or F6	
Activate previous pane		Shift+Tab or Shift+F6	

Getting out

When you are through with WinSight, exit using the Spy | Exit menu command.

Choosing a view

You can select various views and degrees of information for your message tracing.

Picking a pane

WinSight offers three panes that can appear within its main window: a window tree, a class list, and a message trace. You can choose to look at any or all of the panes. WinSight will automatically tile the panes within the main window, but you can resize them to suit your needs.

- The window tree displays the hierarchy of windows on the desktop. This is the default display when you start WinSight.
- The class list pane shows all the currently registered window classes.
- The message trace pane displays information about messages received by selected windows or window classes.

You can hide or display panes at any time, using the View menu. Information and selections are not lost when a pane is hidden.

Arranging the panes

When you have two or more panes displayed in the main window, you can choose to display them either stacked on top of one another or arrayed side by side. The Split Vertical and Split Horizontal commands on the view menu toggle between these configurations.

Getting more detail

Within the window tree and class list panes, you can get more detailed information about a selected window or class. Choosing Open Detail from the Spy menu will bring up detail windows on selected windows or selected classes, depending on which pane is focused.

Class detail Double-clicking or pressing *Enter* on an item in the class list pane brings up a Class Detail window that shows full detailed information about that window class.

Window detail Double-clicking or pressing *Enter* on an item in the window tree brings up a Window Detail window that shows full detailed information on that window, in addition to information about that window's class.

Using the window tree

The Window Tree pane shows an outline of the hierarchy of all existing windows. You can use this display in several ways:

- to determine what windows are actually present
- to see the status of windows, including hidden windows
- to see which windows are receiving messages
- to select the windows you want to trace messages for

Next to each entry in the Window Tree pane is a small diamond. An empty diamond means the window has no child windows. A plus sign (+) in the diamond indicates that the window has children, but they are hidden. A minus sign (-) in the diamond means that the window's children are currently shown.

Pruning the tree

Within the Window Tree pane, you can show or hide lists of child windows.

Showing child windows

Notice that the plus sign in the diamond changes to a minus sign, to show that the window's children are shown.

To show a window's children, click on the diamond next to the window with the left mouse button or press +. All windows which are children of the window appear. To show *all* the levels of child windows (children of children, etc.), click with the right mouse button.

Hiding child windows

Notice that the minus sign in the diamond changes back to a plus sign, showing that the window's children are hidden.

To hide all of a window's child windows, click (or double-click) on the diamond next to that window or press -. All child windows (and their child windows, if any) will disappear from the Window Tree.

Finding a window

WinSight has a special mode for locating windows. It can work in two ways, either identifying the line in the Window Tree that corresponds to a window you point at, or highlighting a window you select in the Window Tree.

Important! *All other applications are suspended while you're in Find Window mode.*

In either case, you enter Find Window mode by choosing **Spy | Find Window**. In this mode, whenever the mouse passes into the boundaries of a window, a thick border appears around that window, and the window is selected in the Window Tree pane.

Alternatively, once in Find Window mode, you can select windows in the Window Tree with the mouse or cursor keys, and WinSight will put the thick border around the selected window or windows.

Leaving Find Window mode

Once you have located the window you want, you can leave Find Window mode by clicking the mouse button, or by pressing the **Esc** or **Enter** keys. This removes the the border from the screen, leaving the current window selected in the Window Tree.

Spying on windows

Once you have selected a window from the Window Tree, you can trace the messages going to that window by choosing Messages | Selected Windows. Changing the selection in the Window Tree will immediately change which windows have messages traced.

You can also choose to spy on all windows, regardless of what is selected in the Class List or the Window Tree, by choosing Messages | All Windows.

Choosing Messages | Selected Windows or Windows | All Windows when the Message Trace pane is hidden causes the Message Trace pane to become visible.

Choosing Messages | Trace off will disable message tracing without causing the Message Trace pane to become hidden.

Working with classes

Sometimes, instead of choosing specific windows to trace messages for, you might want to look at messages for entire classes of windows. WinSight helps you do this using the Class List pane.



A “Class” in WinSight refers to the class name with which the window class was registered with Windows, not C++ classes as used in ObjectWindows.

Using the Class List pane

The Class List pane works much the same way as the Window Tree pane, but it’s much simpler, since classes are not hierarchical. The Class List pane shows all the currently-registered window classes. You can get full details about a class by double-clicking on it or pressing *Enter* when it’s selected. The diamonds to the left work the same way as they do in the Window Tree pane.

Spying on classes

Once you have selected a class from the Class List pane, you can choose Messages | Selected Classes to trace only messages going to

that particular class. If the Message Trace pane is hidden when you choose Messages | Selected Classes, it will become visible.

Note that tracing messages to a class enables you to see all messages to windows of that class, including creation messages, which would otherwise not be accessible.

Changing the selection in the Class List while tracing messages by class immediately changes which classes have their messages traced.

Taking time out

Several parts of WinSight can be disabled and reenabled at your control.

Turning off tracing

Choosing Messages | Trace Off turns off message tracing. The Message Trace pane remains visible, and tracing resumes when you choose another of the message-tracing menu options, Selected Classes, Selected Windows, or All Windows.

Suspending screen updates

The Stop! command on the main menu bar turns off all real-time updating in WinSight. Normally, all the panes are kept current as classes are registered, windows are created and destroyed, and messages are received. Choosing Stop! suspends all of these, and changes the menu command to Start!. Choosing Start! resumes normal operation.

Using Stop! has two main purposes: It gives you a chance to study a particular situation, and it removes the overhead of having WinSight update itself constantly.

Choosing messages to trace

WinSight gives you several ways to narrow down the tracing of messages. Watching the messages to specific windows and window classes is described in “Spying on a window” and “Spying on a class,” elsewhere in this chapter.

You may also want to specify which types of messages you want to trace, regardless of which windows you're spying on. This is done with the Message Trace Options dialog box, brought up by the Messages | Options... command.

Filtering out messages

By default, WinSight traces all messages. If you uncheck the All Messages box, you can then select any or all of ten subgroups of messages. These subgroups are described in Tables 5.2 through 5.10. Checking All Messages again disables all the separate subgroups and will trace all messages.

Message tracing options

The Message Trace Options dialog box gives two other useful options, one which determines the format of the Message Trace pane's display, and the other which logs traced messages to a file.

Formatting message parameters

Normally, the Message Trace pane interprets each message's parameters and displays them in a readable format. You can disable this by checking Hex Only in the Message Trace Options dialog box. When Hex Only is checked, message parameters appear only as hex values of wParam and lParam.

Logging traced messages

Information on traced messages usually goes only to the Message Trace pane. By checking Log File in the Message Trace Options dialog box and typing in a file name, you can capture the message trace to a log file. If the file already exists, messages are appended to the end. To stop logging message traces to the file, uncheck Log File.

To send logged messages to a printer or other device, type in the name of the device instead of a file name. For example, typing PRN for the log file would send output to the printer port.

Table 5.2
Mouse messages

WM_HSCROLL	WM_MOUSEACTIVATE
WM_LBUTTONDOWN	WM_MOUSEMOVE
WM_LBUTTONUP	WM_RBUTTONDOWN
WM_MBUTTONDOWN	WM_RBUTTONUP
WM_MBUTTONUP	WM_SETCURSOR
	WM_VSCROLL

Table 5.3
Window messages

WM_ACTIVATE	WM_KILLFOCUS
WM_ACTIVATEAPP	WM_MOVE
WM_CANCELMODE	WM_PAINT
WM_CHILDACTIVATE	WM_PAINTICON
WM_CLOSE	WM_QUERYDRAGICON
WM_CREATE	WM_QUERYENDSESSION
WM_CTLCOLOR	WM_QUERYNEWPALETTE
WM_DESTROY	WM_QUERYOPEN
WM_ENABLE	WM_QUIT
WM_ENDSESSION	WM_SETFOCUS
WM_ERASEBKGND	WM_SETFONT
WM_GETDLGCODE	WM_SETREDRAW
WM_GETMINMAXINFO	WM_SETTEXT
WM_GETTEXT	WM_SHOWWINDOW
WM_GETTEXTLENGTH	WM_SIZE
WM_ICONERASEBKGND	

Table 5.4
Input messages

WM_CHAR	WM_MENUSELECT
WM_CHARTOITEM	WM_PARENTNOTIFY
WM_COMMAND	WM_SYSCHAR
WM_DEADCHAR	WM_SYSDEADCHAR
WM_KEYDOWN	WM_SYSKEYDOWN
WM_KEYLAST	WM_SYSKEYUP
WM_KEYUP	WM_TIMER
WM_MENUCHAR	WM_VKEYTOITEM

Table 5.5
System messages

WM_COMPACTING	WM_QUEUESYNC
WM_DEVMODECHANGE	WM_SPOOLERSTATUS
WM_ENTERIDLE	WM_SYSCOLORCHANGE
WM_FONTCHANGE	WM_SYSCOMMAND
WM_NULL	WM_TIMECHANGE
WM_PALETTECHANGED	WM_WININICHANGE
WM_PALETTEISCHANGING	

Table 5.6
Initialization messages

WM_INITDIALOG	WM_INITMENUPOPUP
WM_INITMENU	

Table 5.7
Clipboard messages

WM_ASKCBFORMATNAME	WM_PASTE
WM_CHANGECHAIN	WM_PAINTCLIPBOARD
WM_CLEAR	WM_RENDERALLFORMATS
WM_CUT	WM_RENDERFORMAT
WM_COPY	WM_SIZECLIPBOARD
WM_DESTROYCLIPBOARD	WM_UNDO
WM_DRAWCLIPBOARD	WM_VSCROLLCLIPBOARD
WM_HSCROLLCLIPBOARD	

Table 5.8
DDE messages

WM_DDE_ACK	WM_DDE_POKE
WM_DDE_ADVISE	WM_DDE_REQUEST
WM_DDE_DATA	WM_DDE_TERMINATE
WM_DDE_EXECUTE	WM_DDE_UNADVISE
WM_DDE_INITIATE	

Table 5.9
Non-client messages

WM_NCACTIVATE	WM_NCLBUTTONUP
WM_NCCREATE	WM_NCMBUTTONDOWN
WM_NCCALCSIZE	WM_NCMBUTTONUP
WM_NCDESTROY	WM_NCMOUSEMOVE
WM_NCHITTEST	WM_NCPAINT
WM_NCLBUTTONDBLCLK	WM_NCRBUTTONDBLCLK
WM_NCLBUTTONDOWN	WM_NCRBUTTONDOWN
WM_NCMBUTTONDBLCLK	WM_NCRBUTTONUP

Table 5.10
Control messages

BM_GETCHECK	CB_DIR
BM_SETCHECK	CB_GETCOUNT
BM_GETSTATE	CB_GETCURSEL
BM_SETSTATE	CB_GETLBTEXT
BM_SETSTYLE	CB_GETLBTEXTLEN
BN_CLICKED	CB_INSERTSTRING
BN_PAINT	CB_RESETCONTENT
BN_HILITE	CB_FINDSTRING
BN_UNHILITE	CB_SELECTSTRING
BN_DISABLE	CB_SETCURSEL
BN_DOUBLECLICKED	CB_SHOWDROPDOWN
CB_GETEDITSEL	CB_GETITEMDATA
CB_LIMITTEXT	CB_SETITEMDATA
CB_SETEXTSEL	CB_GETDROPPEDCONTROLRECT
CB_ADDSTRING	CB_MSGMAX
CB_DELETESTRING	CBN_SELCHANGE
	CBN_DBLCLK

Table 5.10: Control messages (continued)

CBN_SETFOCUS	EN_CHANGE
CBN_KILLFOCUS	EN_UPDATE
CBN_EDITCHANGE	EN_ERRSPACE
CBN_EDITUPDATE	EN_MAXTEXT
CBN_DROPDOWN	EN_HSCROLL
	EN_VSCROLL
DM_GETDEFID	
DM_SETDEFID	LB_ADDSTRING
	LB_INSERTSTRING
EM_GETSEL	LB_DELETESTRING
EM_SETSEL	LB_RESETCONTENT
EM_GETRECT	LB_SETSEL
EM_SETRECT	LB_SETCURSEL
EM_SETRECTNP	LB_GETSEL
EM_SCROLL	LB_GETCURSEL
EM_LINESCROLL	LB_GETTEXT
EM_GETMODIFY	LB_GETTEXTLEN
EM_SETMODIFY	LB_GETCOUNT
EM_GETLINECOUNT	LB_SELECTSTRING
EM_LINEINDEX	LB_DIR
EM_SETHANDLE	LB_GETTOPINDEX
EM_GETHANDLE	LB_FINDSTRING
EM_GETTHUMB	LB_GETSELCOUNT
EM_LINELENGTH	LB_GETSELITEMS
EM_REPLACESEL	LB_SETTABSTOPS
EM_SETFONT	LB_GETHORIZONTALEXTENT
EM_GETLINE	LB_SETHORIZONTALEXTENT
EM_LIMITTEXT	LB_SETCOLUMNWIDTH
EM_CANUNDO	LB_SETTOPINDEX
EM_UNDO	LB_GETITEMRECT
EM_FMTLINES	LB_GETITEMDATA
EM_LINEFROMCHAR	LB_SETITEMDATA
EM_SETWORDBREAK	LB_SELITEMRANGE
EM_SETTABSTOPS	LB_MSGMAX
EM_SETPASSWORDCHAR	
EM_EMPTYUNDOBUFFER	LBN_SELCHANGE
EM_MSGMAX	LBN_DBLCLK
	LBN_SELCANCEL
EN_SETFOCUS	LBN_SETFOCUS
EN_KILLFOCUS	LBN_KILLFOCUS

Table 5.11
Other messages

*Messages not documented
by Microsoft are shown in
lowercase.*

wm_alttabactive	wm_dropobject
wm_begindrag	wm_entermenuloop
WM_COMPAREITEM	wm_entersizemove
wm_convertrequest	wm_exitmenuloop
wm_convertresult	wm_exitsizemove
WM_DELETEITEM	wm_filesyschange
wm_dragloop	WM_GETFONT
wm_dragmove	wm_isactiveicon
wm_dragselect	wm_lbtrackpoint
WM_DRAWITEM	WM_MDIACTIVATE

Table 5.11: Other messages (continued)

WM_MDICASCADE	WM_NEXTDLGCTL
WM_MDICREATE	wm_nextmenu
WM_MDIDESTROY	wm_querydropobject
WM_MDIGETACTIVE	wm_queryparkicon
WM_MDIICONARRANGE	wm_setvisible
WM_MDIMAXIMIZE	wm_systemerror
WM_MDINEXT	wm_syncpaint
WM_MDIRESTORE	wm_synctask
WM_MDISETMENU	wm_systimer
WM_MDITILE	wm_testing.
WM_MEASUREITEM	

WinSight windows

This section describes the various windows and window panes WinSight provides.

Class List pane

Displays all registered window classes.

Display format

Class (Module) Function Styles

The diamonds have one more purpose: whenever the window receives any messages, they invert color momentarily. This gives you an overview of which windows are currently receiving messages. If a window's children are collapsed in the tree, the diamond for that window will invert to show message activity in the children.

Class is the name of the class. Some predefined Windows classes have numeric names. For example, the popup menu class uses the number 32768 as its name. These classes are shown with both the number and a name, such as #32768:PopupMenu. However, the actual class name is only the number itself, in the MAKEINTRESOURCE format also used for resource ID's.

Module is the name of the executable module (.EXE or .DLL) that registered the class.

Function is the address of the class window function.

Styles is a list of the CS_ styles for the class. The names are the same as the CS_ definitions in windows.h, except the CS_ is removed and the name is in mixed case.

Window Tree pane

Displays all windows in existence, showing their parent-child relationships.

Display format

Tree Handle {Class} Module Position "Title"

The lines on the left show the tree structure. Each window is connected to its parent, siblings, and children with these lines. The lines are in the same fashion as the File Manager. The diamond next to each window shows whether the window has any children. If it is empty, there are no children. If it contains a plus sign, there are children but they are collapsed out in the tree display. If it contains a minus sign, there are children and they are visible in the tree display (at least one level of child windows is visible; further levels may be collapsed).

Handle is the window handle as returned by `CreateWindow`.

Class is the window class name, as described in the Class List pane.

Module is the name of the executable module (.EXE or .DLL) that created the window. Strictly speaking, this is the name of the module owning the data segment passed as the *hInstance* parameter to `CreateWindow`.

Position is either (hidden) if the window is hidden, or (xBegin,yBegin)-(xEnd,yEnd) if the window is visible. For top-level windows, these are screen coordinates. For child windows, they are coordinates within the parent window's client area, as used in `CreateWindow` for a child window.

Title is the window title or text, as returned by `GetWindowText` or a `WM_GETTEXT` message. If the title is the null string, the quotes are omitted.

Message Trace pane

Displays messages received by selected window classes or windows. Messages received via `SendMessage` are shown twice, once when they are sent and again when they return to show the return value. Dispatched messages are shown once only, since their return value is meaningless. The message display is indented to show how messages are nested within other messages.

Format Handle ["Title" | {Class}] Message Status

Handle is the window handle receiving the message.

Title is the window's title. If the title is the null string, the class name is displayed instead, in curly braces.

Message is the message name as defined in WINDOWS.H. Known undocumented Windows messages are shown in lower case.

Unknown message numbers (user-defined) are shown as WM_USER+0XXXX if they are greater-than or equal to WM_USER, or WM_0XXXX if they are less than WM_USER.

Registered message numbers (from **RegisterWindowsMessage**) are shown with their registered name in single quotes.

Status is one or more of the following:

- *Dispatched* indicates the message was received via **DispatchMessage**
- *Sent [from XXXX]* indicates the message was received via **SendMessage**. If it was sent from another window, *from XXXX* gives that window's handle. If it was sent from the same window receiving it, this is shown with *from self*. If it was sent from Windows itself, the "from" phrase is omitted.
- *Returns* indicates the message was received via **SendMessage** and is now returning.
- Additional information specific to each message. In the case of a returning message, this gives the return value, either in numeric form, or with more specific information for messages such as WM_GETTEXT. For sent and dispatched messages, this gives the message parameters. WinSight interprets the parameters to give a readable display, and for messages that have associated data structures (WM_CREATE, for example) it grabs those structures and includes them in the display.

RC: The Windows resource compiler

Most Windows programs are easy to use because they provide a standard user interface. For example, most Windows programs use menus to let you implement program commands, and change cursors to let the mouse pointer represent a wide variety of tools, such as arrows or paint brushes.

Menus and cursors are two examples of a Windows program's resources. Resources are data stored in a program's executable (.EXE) file separate from the program's normal data. Resources are designed and specified outside the program code, then added to the program's compiled code to create a program's executable file.

These are the resources you will create and use most often:

- Menus
- Dialog boxes
- Icons
- Cursors
- Keyboard accelerators
- Bitmaps
- Character strings

Creating resources

You can create resources using a resource editor or the Resource Compiler. In most cases, it's easier to use a resource editor and visually create your resources. However, it is sometimes

convenient to use the Resource Compiler to compile resource script files that appear in books or magazines.

Regardless of which approach you take, you normally create a resource file (.RES) for each application. This resource file contains binary information for all of the menus, dialogs, bitmaps, and other resources used by your application.

The binary resource file (.RES) is added to your executable file (.EXE) by using the Resource Compiler as described later in this chapter. You must also write code that loads the resources into memory. Each resource must be loaded into memory separately. This gives you flexibility, since your program will only use memory for the resources that are currently required.

Adding resources to an executable

Once the resources are stored in binary format in a .RES file, they must be added to the program's executable (.EXE) file. The result is a file that contains the application's compiled code as well as its resources.

There are two ways to add resources to an executable file:

- Use a resource editor to copy resources from a .RES file into the program's already-compiled .EXE file.
- Use the Resource Compiler (RC) to copy resources from a .RES file into the program's already-compiled .EXE file.

Resource compiling from the IDE

From the IDE, the Resource Compiler is invoked by the Project Manager when building a Windows project. Any .RC file (source file) included in a project causes Borland C++ to invoke the Resource Compiler to compile it to a .RES file. Then, after TLINK has linked the project's application or DLL, the Resource Compiler marks and binds the resources to it.

Resource compiling from the command line

From the command line, you can compile the resource files you want to use in your Windows application with the Resource Compiler. When you're ready to build the application, you use the Resource Compiler to bind the .RES file to the .EXE or .DLL.

Resource compiling from a makefile

In a make file, add the .RES file to the list of files in the explicit rule that governs the build of the final .EXE. In that rule, also add the command to invoke the Resource Compiler with the correct .RES file. You can also add a rule to invoke the Resource Compiler on an out-of-date .RES file.

Resource Compiler syntax

See Table 6.1 for a description of the Resource Compiler options.

This is how you invoke the Resource Compiler from the command line:

```
RC [options] ResourceFile [ModuleFile]
```

For example, to compile WHELLO.RC file and add it to WHELLO.EXE, you would give this command line:

```
rc whello
```

This simplest form only works if the resource file and the executable file share the same name. If WHELLO.RC was instead named WHELLORS.RC, you would type

```
rc whellors whello
```

To compile only the WHELLO.RC resource file (and not add the resulting WHELLO.RES to WHELLO.EXE), use the **-R** option, like this:

```
rc -r whello
```

You would then have a WHELLO.RES file. To add WHELLO.RES to WHELLO.EXE, type

```
rc whello.res
```

To mark a module as Windows-compatible, but not add any resources to it, simply invoke the Resource Compiler with the

module name (note that the file name must have one of these extensions: .EXE, .DLL, or .DRV). For example,

```
rc whello.exe
```

The following table describes the Resource Compiler options. Note that Resource Compiler options are not case sensitive (**-e** is the same as **-E**). Also, options that take no arguments can be combined (for instance, **-kpr** is legal).

Table 6.1: Resource Compiler options

Option	What it does
-?	Lists help on Resource Compiler options (also -H).
-d <i>Symbol</i>	Defines <i>Symbol</i> for the preprocessor.
-e	Changes location of global memory for a DLL to above the EMS bank line
-fe <i>FileName</i>	Renames the .EXE file to <i>FileName</i> .
-fo <i>FileName</i>	Renames the .RES file to <i>FileName</i> .
-h	Lists help on Resource Compiler options (also -?).
-i <i>Path</i>	After searching the current directory for include files and resource files, RC searches the directory named in <i>Path</i> . The -i option can be repeated if you want to specify more than one search path. Also see the description for the -x option.
-k	Turns off load optimization for segments and resources. (Normally, the Resource Compiler preloads all data segments, nondiscardable code segments, and the entry-point code segment, even if the segments were not marked as PRELOAD in the module definition file. In addition, the Resource Compiler normally places all preloaded segments in a contiguous area in the executable file.)
-l	Informs Windows that the application will be using expanded memory, according to the LIM 3.2 specification.
-lim32	Same as -l option.
-m	Assigns each instance of a task to a different EMS bank, if the expanded memory under Windows is configured under EMS 4.0.
-multinst	Same as -m option.
-p	Makes a DLL private to one or more instances of a single application, which might result in performance gains.
-r	Compile the .RC file into a .RES file, but do not add it to an .EXE.
-t	Creates application to be run only in standard mode or 386 enhanced mode (protected mode). If the user tries to run in real mode, a message will be displayed.
-v	Display all compiler progress messages (compile verbose).
-x	Excludes searching in the directories named in the INCLUDE environment variable. Also see the description for the -i option.

HC: The Windows Help compiler

A Help system provides users with online information about an application. Creating the system requires the efforts of both a Help writer and a Help programmer. The Help writer plans, writes, codes, builds, and keeps track of Help topic files, which are text files that describe various aspects of the application. The Help programmer ensures that the Help system works properly with the application.

This chapter describes the following topics:

- Providing and creating the Help system
- Planning the Help system
- Creating Help topic files
- Building the Help file
- Help examples and compiler error messages

This section and those that follow assume you are familiar with Windows Help. The sections use examples from sample applications provided on your disks. If you are unfamiliar with Windows Help, take a moment to run the sample application.

Creating a Help system: The development cycle

The creation of a Help system for a Windows application comprises the following major tasks:

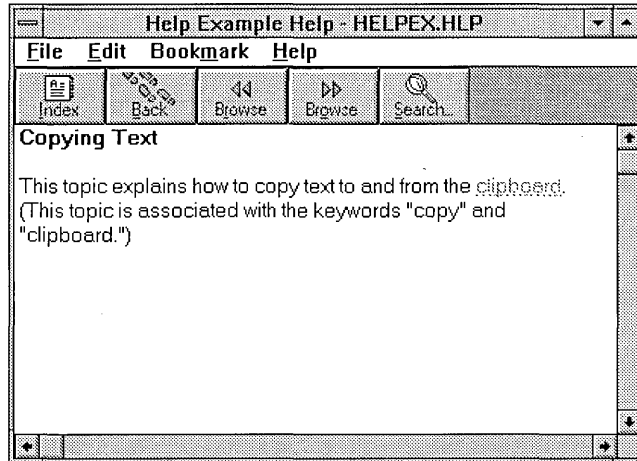
1. Gathering information for the Help topics.
2. Planning the Help system. The section, "Planning the Help system," describes considerations you should keep in mind when planning your Help system.
3. Writing the text for the Help topics.
4. Entering all required control codes into the text files. Control codes determine how the user can move around the Help system. The section titled, "How Help Appears to the Writer," includes an example of several control codes. A later section, "Creating the Help topic files," describes the codes in detail.
5. Creating a project file for the build. The Help project file provides information that the Help Compiler needs to build a Help resource file. A later section, "Building the Help files," describes the Help project file.
6. Building the Help resource file. The Help resource file is a compiled version of the topic files the writer creates. Later in this chapter, the section "Building the Help files" describes how to compile a Help resource file.
7. Testing and debugging the Help system.
8. Programming the application so that it can access Windows Help.

How Help appears to the user

To the user, the Help system appears to be part of the application, and is made up of text and graphics displayed in the Help window in front of the application. Figure 7.1 illustrates the Help window that appears when the user asks for help with copying text in Helpex.

The Help window displays one sample Help topic, a partial description of how to perform one task. In Figure 7.1, the first sentence includes a definition of the word "clipboard." By pressing the mouse button when the cursor is on the word (denoted by dotted underlined text), the user can read the definition, which appears in an overlapping box as long as the mouse button is held down.

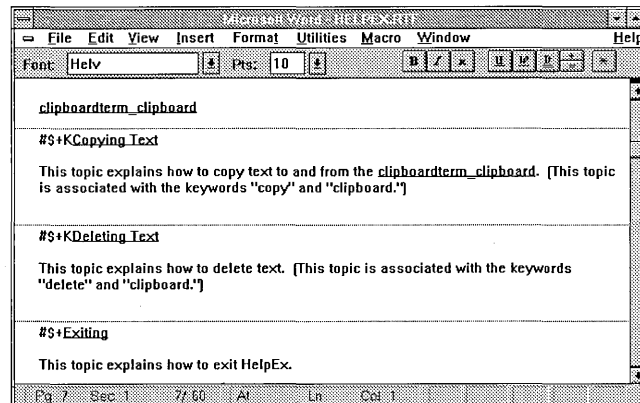
Figure 7.1
Helpex help window



Cross-references to related topics are called jumps. By clicking on a jump term for a related topic (denoted by underlined text), the user changes the content of the Help window to a description of the new topic or command. Figure 7.1 includes a look-up to the definition of “clipboard.”

How Help appears to the help writer

Figure 7.2
Topic file



To create this topic, the Help writer describes the task, formats the text, and inserts codes using strikethrough text, underlined text, and footnotes. In place of strikethrough, the writer can use double

underlining if the word processor does not support strikethrough formatting. Footnotes in the text contain linking information required by the Help Compiler. The section "Planning the Help system" discusses formatting considerations. Another section, "Creating the Help topic files," describes how to create topics and enter the special codes that the Help system uses.

How Help
appears to the
help programmer
*See "Building the Help files"
for details about the Help
application programming
interface (API).*

To the programmer, Windows Help is a stand alone Windows application which the user can run like any other application. Your application can call the **WinHelp** function to ask Windows to run the Help application and specify which topic to display in the Help window.

Planning the Help system

The initial task for the Help writer is to develop a plan for creating the system. This section discusses planning the Help system for a particular application; it covers these topics:

- Developing a plan
- Determining the topic file structure
- Designing the visual appearance of Help topics

Developing a plan

Before you begin writing Help topics using the information you have gathered, you and the other members of the Help team should develop a plan that addresses the following issues:

- The audience for your application
- The content of the Help topics
- The structure of topics
- The use of context-sensitive topics

You might want to present your plan in a design document that includes an outline of Help information, a diagram of the structure of topics, and samples of the various kinds of topics your system will include. Keep in mind that context-sensitive Help requires increased development time, especially for the application programmer.

Defining the audience The audience you address determines what kind of information you make available in your Help system and how you present the information.

Users of Help systems might be classified as follows:

Table 7.1
Your application audience

User	Background
Computer novice	Completely new to computing.
Application novice	Some knowledge of computing, but new to your kind of application. For example, if you are providing Help for a spreadsheet program, the application novice might be familiar only with word-processing packages.
Application intermediate	Knowledgeable about your kind of application.
Application expert	Experienced extensively with your type of application.

Keep in mind that one user may have various levels of knowledge. For example, the expert in word processors may have no experience using spreadsheets.

Planning the contents You should create topics that are numerous enough and specific enough to provide your users with the help they need.

Novice users need help learning tasks and more definitions of terms. More sophisticated users occasionally seek help with a procedure or term, but most often refresh their memory of commands and functions. The expert user tends only to seek help with command or function syntax, keyboard equivalents, and shortcut keys.

There are no rules for determining the overall content of your Help system. If you are providing Help for all types of users, you will want to document commands, procedures, definitions, features, functions, and other relevant aspects of your application. If you are providing help for expert users only, you might want to omit topics that describe procedures. Let your audience definition guide you when deciding what topics to include.

Keep in mind that the decision to implement context-sensitive Help is an important one. Context-sensitive Help demands a close working relationship between the Help author and the

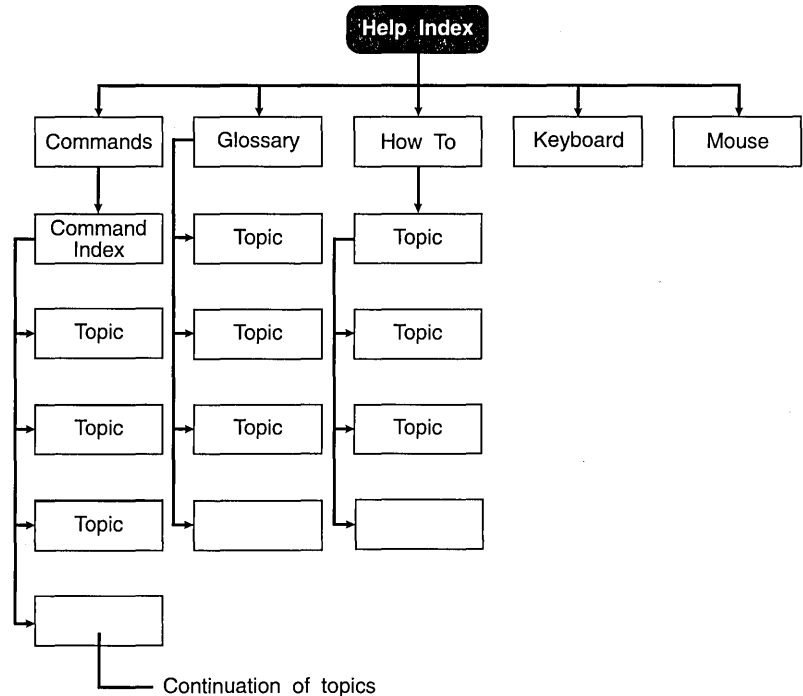
application programmer, and will therefore increase the development time necessary to create a successful Help system.

Planning the structure

Many Help systems structure topics hierarchically. At the top of the hierarchy is an index or a table of contents, or both. The index and table of contents list individual topics or categories of topics available to the user.

Topics themselves can be related hierarchically. Each successive step takes the user one level down in the hierarchy of the Help system until the user reaches topic information. The hierarchical relationship of Help topics determines in part how the user navigates through the Help system. Figure 7.3 illustrates a possible hierarchy:

Figure 7.3
Example of a help hierarchy



Helpex contains an index that lists several categories of topics. Each category includes a secondary index, which lists topics in the category, and the topics themselves.

Moving from the index to a topic, the user goes from the general to the specific.

The hierarchical structure provides the user a point of reference within Help. Users are not constrained to navigate up and down the hierarchy; they can jump from one topic to another, moving across categories of topics. The effect of jumps is to obscure hierarchical relationships. For example, the Windows Help application contains a search feature that lets the user enter a keyword into a dialog box and search for topics associated with that keyword. The Help application then displays a list of titles to choose from in order to access information that relates to the keyword.

For more about the search feature, see page 118.

Because users often know which feature they want help with, they can usually find what they want faster using the search feature than they can by moving through the hierarchical structure.

For more about browse sequences, see page 113.

In addition to ordering topics hierarchically, you can order them in a logical sequence that suits your audience. The logical sequence, or "browse sequence," lets the user choose the Browse button to move from topic to topic. Browse sequences are especially important for users who like to read several related topics at once, such as the topics covering the commands on the File menu.

Whichever structure you decide to use, try to minimize the number of lists that users must traverse in order to obtain information. Also, avoid making users move through many levels to reach a topic. Most Help systems function quite well with only two or three levels.

Displaying context-sensitive Help topics

Windows Help supports context-sensitive Help. When written in conjunction with programming of the application, context-sensitive Help lets the user press *F1* in an open menu to get help with the selected menu item. Alternatively, the user can press *Shift+F1* and then click on a screen region or command to get help on that item.

Developing context-sensitive Help requires coordination between the Help writer and the application programmer so that Help and the application pass the correct information back and forth.

For information on creating a Help project file, see page 129.

To plan for context-sensitive Help, the Help author and the application programmer should agree on a list of context numbers. Context numbers are arbitrary numbers that correspond to each menu command or screen region in the application, and are used to create the links to the corresponding Help topics. You

can then enter these numbers, along with their corresponding context-string identifiers, in the Help project file, which the Help Compiler uses to build a Help resource file.

For more on assigning context numbers, see page 141.

The context numbers assigned in the Help project file must correspond to the context numbers that the application sends at run time to invoke a particular topic.

See page 138 for more on context-sensitive Help.

If you do not explicitly assign context numbers to topics, the Help Compiler generates default values by converting topic context strings into context numbers.

Page 127 provides you with more information about using a tracker.

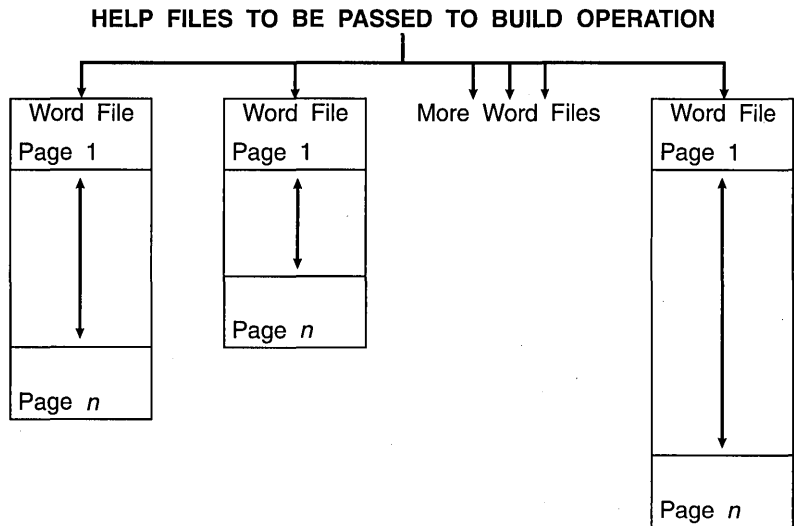
To manage context numbers and file information, you might want to create a Help tracker to list the context numbers for your context-sensitive topics.

Determining the topic file structure

The Help file structure remains essentially the same for all applications even though the context and number of topic files differ. Topic files are segmented into the different topics by means of page breaks. When you build the Help system, the compiler uses these topic files to create the information displayed for the user in the application's Help window.

Figure 7.4 shows this basic file structure.

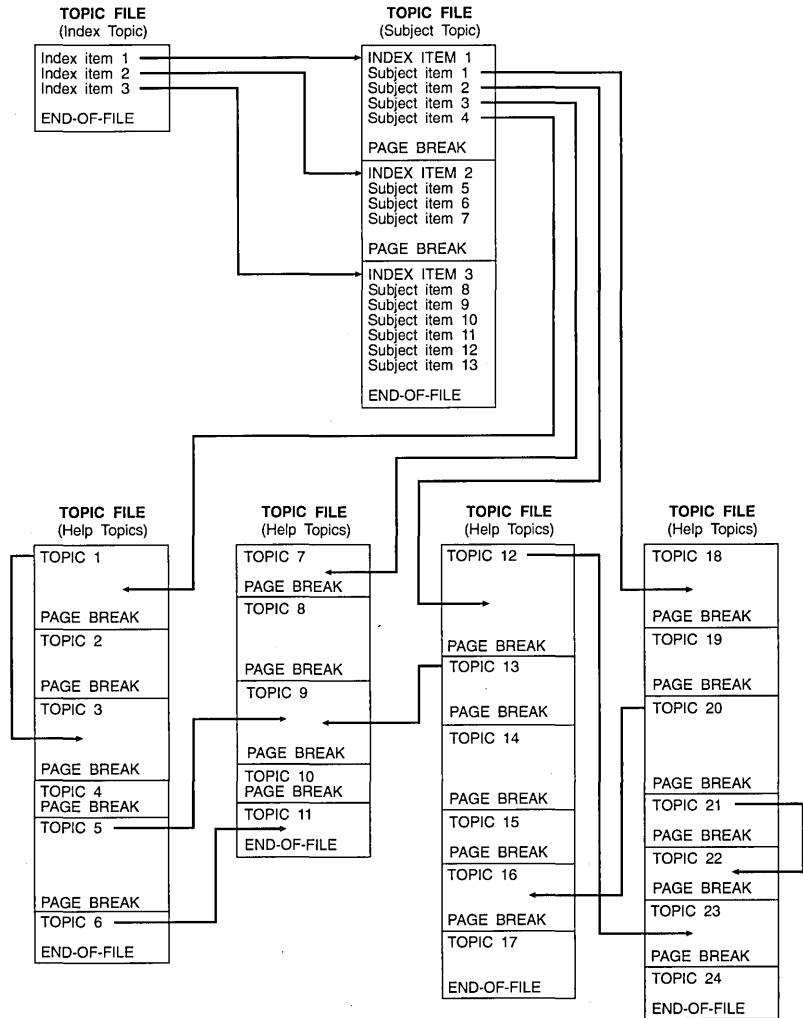
Figure 7.4
Basic help file structure



Choosing a file structure for your application

When choosing a file structure for your Help system, consider the scope and content of the Help system you are planning. For example, you could place all Help topics in a single large topic file. Or, you could place each Help topic in a separate file. Neither of these file structures is generally acceptable. An enormous single file or too many individual files can present difficulties during the creation of the Help resource file.

Figure 7.5
Help file structure showing
hypertext jumps



The number of topics relates to the number of features covered by the Help system. Consequently, you cannot make extensive changes to one without making changes to the other. For instance, if a number of additional product features are added to Help, then additional topics must be created to accommodate the new information.

Figure 7.5 illustrates the file structure of a possible Help system. The number of topics and topic files is limited to simplify the diagram and more clearly show the concept of linking the topics together through jumps, shown in the figure as arrows. The figure is not intended to show the number of files that can be included in the Help file system. Moreover, the figure does not show how topic files are ordered using the browse feature.

Designing Help topics

How the information in the Help window appears to the user is primarily a function of the layout of the Help topic. The Windows Help application supports a number of text attributes and graphic images you can use to design your Help window.

This section provides general guidelines for designing a window and describes fonts and graphic images that Windows Help supports.

Layout of the Help text

Help text files are not limited to plain, unformatted text. You can use different fonts and point sizes, include color and graphics to emphasize points, indent paragraphs to present complex information, and use a variety of other visual devices to present your information.

Research on screen format and Help systems has produced general guidelines for presenting information to users. Table 7.2 summarizes the findings of these studies.

Table 7.2: Help design issues

Design Issue	Guideline
Language	<i>Use language appropriate for the audience you have defined.</i> Language that is too sophisticated for your audience can frustrate users by requiring them to learn the definition of unfamiliar terms and concepts.
Amount of text	<i>Use a minimum of text.</i> Studies indicate that reading speed decreases by 30 percent when users read online text rather than printed text. Minimal, concise text helps users compensate for the decreased reading speed.

Table 7.2: Help design issues (continued)

Paragraph length	<i>Use short paragraphs.</i> Online users become overloaded with text more easily than do readers of printed material. Breaking your text into short paragraphs helps avoid this problem.
Whitespace	<i>Use it to help group information visually.</i> Whitespace is important to making online text more readable. Use it liberally, while also considering the overall size that a topic will occupy on the screen. Users tend to think there is more information on a screen than exists. For example, if the ratio of whitespace to text is 50:50, users perceive the ratio to be 40:60.
Highlighting	<i>Use highlighting techniques judiciously.</i> Windows Help provides a variety of highlighting devices, such as font sizes, font types, and color. Using a few devices can help users find information quickly. Using many devices will decrease the effectiveness of your visual presentation and frustrate users. As with print-based documentation, use only one or two fonts at a time.
Graphics and icons	<i>Use graphics to support the explanation of visual events.</i> Windows Help supports the use of bitmapped graphic images. Use appropriate images to help explain the function of icons and screen elements in your application. Remember that graphics will draw the user's eye before the accompanying text. Be sure to crop your images to remove distracting information. Use color images only if you are certain that all your users' systems have color capability. As with context-sensitive Help, consider the additional time necessary to create accurate and meaningful graphic images.
Design consistency	<i>Be rigorously consistent in your design.</i> Users expect the appearance of Help topics to be the same, regardless of the information presented. Consistent titling, highlighting, fonts, and positioning of text in the window is essential to an effective Help system.

Type fonts and sizes The Windows Help application can display text in any font and size available to the system. When the topic files are passed to the build process, the Help Compiler attempts to use the fonts and sizes found in the topic files. If a font or point size cannot be matched exactly when the Help file is displayed by Windows Help, the closest available font and size on the user's system will be used.

Windows ships with only certain fonts in specific font sizes. If you write Help files using these fonts and sizes, the displayed Help file will closely match the printed word-processed file. Because fonts other than those shipped with Windows may not be available on users' machines, you might want to limit your font selection to the shipped Windows fonts.

The fonts included with Windows are shown in Table 7.3:

Table 7.3
Windows fonts

Font	Sizes
Courier	10,12,15
Helv	8,10,12,14,18,24
Modern	
Roman	
Script	
Symbol	8,10,12,14,18,24
Tms Rmn	8,10,12,14,18,24

Since Windows Help supports any Windows font, special symbols such as arrows can be included in your topics by using the Symbol font.

Graphic images

The Windows Help application allows you to embed graphics in the Help file. Graphics can be placed and displayed anywhere on the page. Text can appear next to the graphic.

Color graphic images can be included, provided you use only the available Windows system colors. If you use graphics tools that support an enhanced color palette to create or capture images, these images may not always display with the intended colors. And since you cannot control the color capabilities on a user's machine, you might want to limit your graphic images to black and white bitmaps.

For more information on placing graphics into your Help files, see page 124.

Keep in mind that graphics are most effective when they contribute to the learning process. Graphics not tied to the information are usually distracting rather than helpful and should be avoided.

For additional information about screen design, refer to the following books and journals:

- Bradford, Annette Norris. "Conceptual Differences Between the Display Screen and the Printed Page." *Technical Communication* (Third Quarter 1984): 13-16.
- Galitz, Wilbert O. *Handbook of Screen Format Design*. 3d ed. Wellesley, MA: QED Information Sciences, Inc., 1989.
- Houghton, Raymond C., Jr. "Online Help Systems: A Conspectus." *Communications of the ACM* 27(February 1984): 126-133.

- Queipo, Larry. "User Expectations of Online Information." *IEEE Transactions on Professional Communications* PC 29 (December 1986): 11-15.

Creating the Help topic files

Probably the most time-consuming task in developing a Help system for your application is creating the Help topic files from which you compile the Help system. Help topic files are text files that define what the user sees when using the Help system. The topic files can define various kinds of information, such as an index to information on the system, a list of commands, or a description of how to perform a task.

Creating topic files entails writing the text that the user sees when using Help, and entering control codes that determine how the user can move from one topic to another. This section describes the following topics:

- Choosing an authoring tool
- Structuring Help topic files
- Coding Help topic files
- Managing Help topic files

Choosing an authoring tool

To write your text files, you will need a Rich Text Format (RTF) editor, which lets you create the footnotes, underlined text, and strikethrough or double-underlined text that indicate the control codes. These codes are described in the section titled "Coding Help Topic Files" on page 114. RTF capability allows you to insert the coded text required to define Help terms, such as jumps, keywords, and definitions.

Structuring Help topic files

A Help topic file typically contains multiple Help topics. To identify each topic within a file:

- Topics are separated by hard page breaks.
- Each topic accessible via a hypertext link must have a unique identifier, or context string.

- Each topic can have a title.
- Each topic can have a list of keywords associated with it.
- Each topic can have a build-tag indicator.
- Any topic can have an assigned browse sequence.

For information about inserting page breaks between topics, see the documentation for the editor you are using. For information about assigning context strings and titles to topics, see the following sections.

Coding Help topic files

Table 7.4
Help control codes

The Help system uses control codes for particular purposes:

Control Code	Purpose
Asterisk (*) footnote	Build tag—Defines a tag that specifies topics the compiler conditionally builds into the system. Build tags are optional, but they must appear first in a topic when they are used.
Pound sign (#)	Context string—Defines a context string that uniquely identifies a topic. Because hypertext relies on links provided by context strings, topics without context strings can only be accessed using keywords or browse sequences.
Dollar sign (\$) footnote	Title—Defines the title of a topic. Titles are optional.
Letter “K” footnote	Keyword—Defines a keyword the user uses to search for a topic. Keywords are optional.
Plus sign (+) footnote	Browse sequence number—Defines a sequence that determines the order in which the user can browse through topics. Browse sequences are optional. However, if you omit browse sequences, the Help window will still include the Browse buttons, but they will be grayed.
Strikethrough or double-underlined text	Cross-reference—Indicates the text the user can choose to jump to another topic.
Underlined text	Definition—Specifies that a temporary or “look-up” box be displayed when the user holds down the mouse button or <i>Enter</i> key. The box can include such information as the definition of a word or phrase, or a hint about a procedure.

Table 7.4: Help control codes (continued)

Hidden text	Cross-reference context string—Specifies the context string for the topic that will be displayed when the user chooses the text that immediately precedes it.
-------------	---

If you are using build tags, footnote them at the very beginning of the topic. Place other footnotes in any order you want. For information about assigning specific control codes, see the following sections.

Assigning build tags

Build tags are strings that you assign to a topic in order to conditionally include or exclude that topic from a build. Each topic can have one or more build tags. Build tags are not a necessary component of your Help system. However, they do provide a means of supporting different versions of a Help system without having to create different source files for each version. Topics without build tags are always included in a build.

*For information about the **BUILD** option, the (BuildTags) section and the Help project file, see "Building the Help files."*

You insert build tags as footnotes using the asterisk (*). When you assign a build tag footnote to a topic, the compiler includes or excludes the topic according to build information specified in the **BUILD** option and [BuildTags] section of the Help project file.

To assign a build tag to a topic:

1. Place the cursor at the beginning of the topic heading line, so that it appears before all other footnotes for that topic.
2. Insert the asterisk (*) as a footnote reference mark.
Note that a superscript asterisk (^{*}) appears next to the heading.
3. Type the build tag name as the footnote.
Be sure to allow only a single space between the asterisk (*) and the build tag.

Build tags can be made up of any alphanumeric characters. The build tag is not case-sensitive. The tag may not contain spaces. You can specify multiple build tags by separating them with a semicolon, as in the following example:

```
* AppVersion1; AppVersion2; Test_Build
```

Including a build tag footnote with a topic is equivalent to setting the tag to true when compared to the value set in the project file. The compiler assumes all other build tags to be false for that topic.

After setting the truth value of the build tag footnotes, the compiler evaluates the build expression in the Options section of the Help project file. Note that all build tags must be declared in the project file, regardless of whether a given conditional compilation declares the tags. If the evaluation results in a true state, the compiler includes the topic in the build. Otherwise, the compiler omits the topic.

The compiler includes in all builds topics that do not have a build tag footnote regardless of the build tag expressions defined in the Help project file. For this reason, you may want to use build tags primarily to exclude specific topics from certain builds. If the compiler finds any build tags not declared in the Help project file, it displays an error message.

By allowing conditional inclusion and exclusion of specific topics, you can create multiple builds using the same topic files. This saves time and effort for the Help development team. It also means that you can develop Help topics that will help you maintain a higher level of consistency across your product lines.

Assigning context strings

Context strings identify each topic in the Help system. Each context string must be unique. A given context string may be assigned to only one topic within the Help project; it cannot be used for any other topic.

For information about assigning jumps, see page 122; for assigning browse sequences, see page 120; for assigning keywords, see page 118.

The context string provides the means for creating jumps between topics or for displaying look-up boxes, such as word and phrase definitions. Though not required, most topics in the Help system will have context-string identifiers. Topics without context strings may not be accessed through hypertext jumps. However, topics without context-string identifiers can be accessed through browse sequences or keyword searches, if desired. It is up to the Help writer to justify the authoring of topics that can be accessed only in these manners.

To assign a context string to a Help topic:

1. Place the cursor to the left of the topic heading.
2. Insert the pound sign (#) as the footnote reference mark.
Note that a superscript pound sign (#) appears next to the heading.
3. Type the context string as the footnote.

Be sure to allow only a single space between the pound sign (#) and the string.

Context strings are not case-sensitive.

Valid context strings may contain the alphabetic characters A – Z, the numeric characters 0 – 9, and the period (.) and underscore (_) characters. The following example shows the context string footnote that identifies a topic called “Opening an Existing Text File”:

```
#OpeningExistingTextFile
```

Although a context string has a practical limitation of about 255 characters, there is no good reason for approaching this value. Keep the strings sensible and short so that they’re easier to enter into the text files.

Assigning titles Title footnotes perform the following functions within the Help system:

- They appear on the Bookmark menu.
- They appear in the “Topics found” list that results from a keyword search. (Topics that do not have titles, but are accessible via keywords are listed as >>>>Untitled Topic<<<< in the Topics found list.)

Although not required, most topics have a title. You might not assign a title to topics containing low-level information that Help’s search feature, look-up boxes, and system messages do not access.

To assign a title to a topic:

1. Place the cursor to the left of the topic heading.
2. Insert a footnote with a dollar sign (\$) as the footnote reference mark.

Note that a superscript dollar sign (^{\$}) appears next to the heading.

3. Type the title as the footnote.

Be sure to allow only a single space between the dollar sign (\$) and the title.

The following is an example of a footnote that defines the title for a topic:

```
$ Help Keys
```

When adding titles, keep in mind the following restrictions:

Table 7.5
Restrictions of Help titles

Item	Restrictions
Characters	Titles can be up to 128 characters in length. The Help compiler truncates title strings longer than 128 characters. The help system displays titles in a list box when the user searches for a keyword or enters a bookmark.
Formatting	Title footnote entries cannot be formatted.

Assigning keywords

Help allows the user to search for topics with the use of keywords assigned to the topics. When the user searches for a topic by keyword, Help matches the user-entered word to keywords assigned to specific topics. Help then lists matching topics by their titles in the Search dialog box. Because a keyword search is often a fast way for users to access Help topics, you'll probably want to assign keywords to most topics in your Help system.

Note You should specify a keyword footnote only if the topic has a title footnote, since the title of the topic will appear in the search dialog when the user searches for the keyword.

To assign a keyword to a topic:

1. Place the cursor to the left of the topic heading.
2. Insert an uppercase K as the footnote reference mark.
Note that a superscript K (^K) appears next to the heading.
3. Type the keyword, or keywords, as the footnote.
Be sure to allow only a single space between the K and the first keyword.
If you add more than one keyword, separate each with a semicolon.

The following is an example of keywords for a topic:

```
K open;opening;text file;ASCII;existing;text only;documents;
```

Whenever the user performs a search on any of these keywords, the corresponding titles appear in a list box. More than one topic may have the same keyword.

When adding keywords, keep in mind the following restrictions:

Table 7.6
Help keyword restrictions

Item	Restrictions
Characters	Keywords can include any ANSI character, including accented characters. The maximum length for keywords is 255 characters. A space embedded in a key phrase is considered to be a character, permitting phrases to be keywords.
Phrases	Help searches for any word in the specified phrase.
Formatting	Keywords are unformatted.
Case sensitivity	Keywords are not case-sensitive.
Punctuation	Except for semicolon delimiters, you can use punctuation.

Creating multiple keyword tables

Multiple keyword tables are useful to allow a program to look up topics that are defined in alternate keyword tables. You can use an additional keyword table to allow users familiar with keywords in a different application to discover the matching keywords in your application.

*For information on the **MULTIKEY** option, see page 136.*

Authoring additional keyword tables is a two-part process. First, the **MULTIKEY** option must be placed in the [Options] section of the project file.

Second, the topics to be associated with the additional keyword table must be authored and labeled. Footnotes are assigned in the same manner as the normal keyword footnotes, except that the letter specified with the **MULTIKEY** option is used. With this version of the Help Compiler, the keyword footnote used is case-sensitive. Therefore, care should be taken to use the same case, usually uppercase, for your keyword footnote. Be sure to associate only one topic with a keyword. Help does not display the normal search dialog box for a multiple keyword search. Instead it displays the first topic found with the specified keyword. If you want the topics in your additional keyword table to appear in the normal Help keyword table, you must also specify a "K" footnote and the given keyword.

The application you are developing Help for can then display the Help topic associated with a given string in a specified keyword table. Keywords are sorted without regard to case for the keyword table. For information on the parameters passed by the

application to call a topic found in alternate keyword table, see page 147.

Assigning browse sequence numbers

The Browse >>>> and Browse <<<< buttons on the icon bar in the Help window let users move back and forth between related topics. The order of topics that users follow when moving from topic to topic is called a "browse sequence." A browse sequence is determined by sequence numbers, established by the Help writer.

To build browse sequences into the Help topics, the writer must

1. Decide which topics should be grouped together and what order they should follow when viewed as a group.
Help supports multiple, discontinuous sequence lists.
2. Code topics to implement the sequence.



In this version of Help, topics defined in browse sequences are accessed using the Browse buttons at the top of the Help window. Future versions of Help will not normally display browse buttons for the user. However, if your Help resource file includes browse sequences authored in the format described here, these future versions will support the feature by automatically displaying browse buttons for the user.

Organizing browse sequences

When organizing browse sequences, the writer must arrange the topics in an order that will make sense to the user. Topics can be arranged in alphabetical order within a subject, in order of difficulty, or in a sensible order that seems natural to the application. The following example illustrates browse sequences for the menu commands used in a given application. The Help writer has subjectively defined the order that makes the most sense from a procedural point of view. You may, of course, choose a different order.

```
SampleApp Commands
  File Menu - commands:005
    New Command - file_menu:005
    Open Command - file_menu:010
    Save Command - file_menu:015
    Save As Command - file_menu:020
    Print Command - file_menu:025
    Printer Setup Command - file_menu:030
    Exit Command - file_menu:035
```

```

Edit Menu - commands:010
  Undo Command - edit_menu:025
  Cut Command - edit_menu:015
  Copy Command - edit_menu:010
  Paste Command - edit_menu:020
  Clear Command - edit_menu:005
  Select All Command - edit_menu:030
  Word Wrap Command - edit_menu:035
  Type Face Command - edit_menu:040
  Point Size Command - edit_menu:045
Search Menu - commands:015
  Find Command - search_menu:005
  Find Next Command - search_menu:010
  Previous Command - search_menu:015
Window Menu - commands:020
  Tile Command - window_menu:005
  Cascade Command - window_menu:010
  Arrange Icons Command - window_menu:015
  Close All Command - window_menu:020
  Document Names Command - window_menu:025

```

Each line consists of a sequence list name followed by a colon and a sequence number. The sequence list name is optional. If the sequence does not have a list name, as in the following example, the compiler places the topic in a "null" list:

```
Window Menu - 120
```

Note that the numbers used in the browse sequence example begin at 005 and advance in increments of 005. Generally, it is good practice to skip one or more numbers in a sequence so you can add new topics later if necessary. Skipped numbers are of no consequence to the Help Compiler; only their order is significant.

Sequence numbers establish the order of topics within a browse sequence list. Sequence numbers can consist of any alphanumeric characters. During the compiling process, strings are sorted using the ASCII sorting technique, not a numeric sort.

Both the alphabetic and numeric portions of a sequence can be several characters long; however, their lengths should be consistent throughout all topic files. If you use only numbers in the strings make sure the strings are all the same length; otherwise a higher sequence number could appear before a lower one in certain cases. For example, the number 100 is numerically higher than 99, but 100 will appear before 99 in the sort used by Help, because Help is comparing the first two digits in the strings.

In order to keep the topics in their correct numeric order, you would have to make 99 a three-digit string: 099.

Coding browse sequences

After determining how to group and order topics, code the sequence by assigning the appropriate sequence list name and number to each topic, as follows:

1. Place the cursor to the left of the topic heading.
2. Insert the plus sign (+) as the footnote reference mark.
Note that a superscript plus sign (⁺) appears next to the heading.
3. Type the sequence number using alphanumeric characters.

For example, the following footnote defines the browse sequence number for the Edit menu topic in the previous browse sequence example:

```
+ commands:010
```

While it may be easier to list topics within the file in the same order that they appear in a browse sequence, it is not necessary. The compiler orders the sequence for you.

Creating cross-references between topics

Cross-references, or “jumps,” are specially-coded words or phrases that are linked to other topics. Although you indicate jump terms with strikethrough or double-underlined text in the topic file, they appear underlined in the Help window. In addition, jump terms appear in color on color systems. For example, the strikethrough text (double-underlined in Word for Windows) ~~New Command~~ appears as New Command in green text to the user.

To code a word or phrase as a jump in the topic file :

1. Place the cursor at the point in the text where you want to enter the jump term.
2. Select the strikethrough (or double-underline) feature of your editor.
3. Type the jump word or words in strikethrough mode.
4. Turn off strikethrough and select the editor’s hidden text feature.

5. Type the context string assigned to the topic that is the target of the jump.

When coding jumps, keep in mind that:

- No spaces can occur between the strikethrough (or double-underlined) text and the hidden text.
- Coded spaces before or after the jump term are not permitted.
- Paragraph marks must be entered as plain text.

Defining terms

Most topic files contain words or phrases that require further definition. To get the definition of a word or phrase, the user first selects the word and then holds down the mouse button or *Enter* key, causing the definition to appear in a box within the Help window. The Help writer decides which words to define, considering the audience that will be using the application and which terms might already be familiar.



The look-up feature need not be limited to definitions. With the capability of temporarily displaying information in a box, you might want to show a hint about a procedure, or other suitable information for the user.

Defining a term requires that you

- Create a topic that defines the term.
The definition topic must include a context string. See the section titled "Assigning Context Strings." on page 116.
- Provide a cross-reference for the definition topic whenever the term occurs.

You don't need to define the same word multiple times in the same topic, just its first occurrence. Also, consider the amount of colored text you are creating in the Help window. See the following "Coding definitions" section.

Creating definition topics

You can organize definition topics any way you want. For example, you can include each definition topic in the topic file that mentions the term. Or you can organize all definitions in one topic file and provide the user with direct access to it. Helpex uses the latter method, with all definitions residing in the TERMS.RTF file. Organizing definition topics into one file provides you with a glossary and lets you make changes easily.

Coding definitions

After creating definition topics, code the terms as they occur, as follows:

1. Place the insertion point where you want to place the term that requires definition.
2. Select the underline feature of your editor.
3. Type the term.
4. Turn off the underline feature, and select the editor's hidden-text feature.
5. Type the context string assigned to the topic that contains the definition of the term.

Inserting graphic images

Bitmapped graphic images can be placed in Help topics using either of two methods. If your word processor supports the placement of Windows 2.1 or Windows 3.0 graphics directly into a document, you can simply paste your bitmaps into each topic file. Alternatively, you can save each bitmap in a separate file and specify the file by name where you want it to appear in the Help topic file. The latter method of placing graphics is referred to as "bitmaps by reference." The following sections describe the process of placing bitmaps directly or by reference into your Help topics.

Creating and capturing bitmaps

You can create your bitmaps using any graphical tools, as long as the resulting images can be displayed in the Windows environment. Each graphic image can then be copied to the Windows clipboard. Once on the clipboard, a graphic can be pasted into a graphics editor such as Paint, and modified or cleaned up as needed.

Windows Help 3.0 supports color bitmaps. However, for future compatibility, you might want to limit graphics to monochrome format. If you are producing monochrome images, you might have to adjust manually the elements of your source graphic that were originally different colors to either black, white, or a pattern of black and white pixels.

When you are satisfied with the appearance of your bitmap, you can either save it as a file, to be used as a bitmap by reference, or you can copy it onto the clipboard and paste it into your word

processor. If you save the graphic as a file, be sure to specify its size in your graphics editor first, so that only the area of interest is saved for display in the Help window. The tighter you crop your images, the more closely you will be able to position text next to the image. Always save (or convert and save if necessary) graphics in Windows' .BMP format.

Bitmap images should be created in the same screen mode that you intend Help to use when topics are displayed. If your Help files will be displayed in a different mode, bitmaps might not retain the same aspect ratio or information as their source images.

Placing bitmaps using a graphical word processor

The easiest way to precisely place bitmaps into Help topics is to use a graphical word processor. Microsoft Word for Windows supports the direct importation of bitmaps from the clipboard. Simply paste the graphic image where you want it to appear in the Help topic. You can format your text so that it is positioned below or alongside the bitmap. When you save your Help topic file in RTF, the pasted-in bitmap is converted as well and will automatically be included in the Help resource file.

Placing bitmaps by reference

If your word processor cannot import and display bitmaps directly, you can specify the location of a bitmap that you have saved as a file. To insert a bitmap reference in the Help topic file, insert one of the following statements where you want the bitmap to appear in the topic:

```
{bmc filename.bmp}  
{bml filename.bmp}  
{bmr filename.bmp}
```



Do not specify a full path for *filename*. If you need to direct the compiler to a bitmap in a location other than the root directory for the build, specify the absolute path for the bitmap in the [Bitmaps] section of the project file.

The argument **bmc** stands for "bitmap character," indicating that the bitmap referred to will be treated the same as a character placed in the topic file at the same location on a line. Text can precede or follow the bitmap on the same line, and line spacing will be determined based upon the size of the characters (including the bitmap character) on the line. Don't specify negative line spacing for a paragraph with a bitmap image, or the image may inadvertently overwrite text above it when it's displayed in Help. When you use the argument **bmc**, there is no

automatic text wrapping around the graphic image. Text will follow the bitmap, positioned at the baseline.

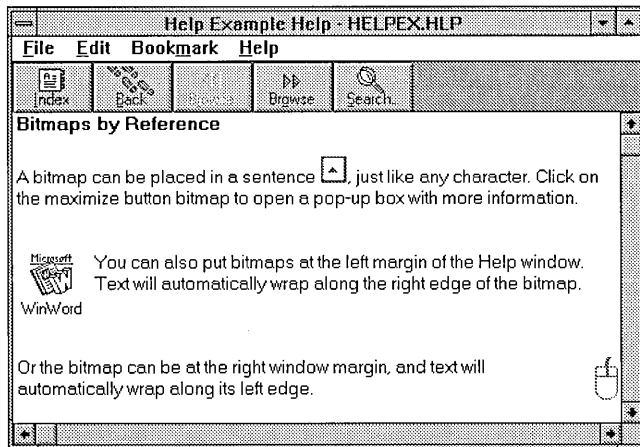
The argument **bml** specifies that the bitmap appear at the left margin, with text wrapping automatically along the right edge of the image. The argument **bmr** specifies that the bitmap appear at the right margin, with text to its left. Bitmap filenames must be the same as those listed in the [Bitmaps] section of the Help project file. The [Bitmaps] section is described in the section "Building the Help files."



Multiple references to a bitmap of the same name refer to the same bitmap when the Help file is displayed. This means that bitmap references can be repeated in your Help system without markedly increasing the size of the Help resource file.

Figure 7.6 shows the placement of three bitmaps with related text in a topic as displayed in Help.

Figure 7.6
Help topic display showing
bitmaps by reference



Managing topic files

Help topic files can be saved in the default word-processor format or in RTF. If you always save your files in RTF, and later need to make a change, the word processor may take additional time to interpret the format as it reloads the file. If you anticipate making numerous changes during Help development, you might want to minimize this delay by saving topic files in both default and RTF formats, with different file extensions to distinguish them. The compiler needs only the RTF files, and you will have faster access

to the source files for changes. On a large project, this practice can save a considerable amount of development time.

Keeping track of files and topics

It is important to keep track of all topic files for the following reasons:

- ▣ To ensure that no topics are left out of the build process
- ▣ To ensure that each topic has been assigned a unique context string
- ▣ To double-check browse sequencing within general and specific lists
- ▣ To show keyword and title matches
- ▣ To allow writers to see where the text for each of the topics is located
- ▣ To keep track of changes to files and the current status
- ▣ To track any other aspect of the Help development process that you think essential

At a minimum, writers must keep track of their own topic files, and must pass the filenames to the person who is responsible for creating the Help project file.

Creating a help tracker

While it is important that you track topic files throughout the development cycle, the tracking tool can be anything that suits your needs. You can maintain a current list of topics in an ASCII text file, in a Quattro Pro spreadsheet, or in another format.

When you or another writer creates or revises a topic, you should update the Help tracking file to reflect the change. The contents of the tracking file are not rigidly defined, but should contain entries for filename, context string, title, browse sequence, and keywords. If your application makes use of the context-sensitive feature of Help, you may want to keep track of the context-sensitive information as well. This entry is necessary only if you are assigning context numbers to topics in the Help project file. You can also include optional information, such as date created, date modified, status, and author, if you want to keep track of all aspects of the Help development process. How you organize this information is entirely up to you.

The following sample text file and worksheet illustrate how the tracker might be organized for the Help system topics. The examples show both Help menu and context-sensitive Help

entries for the topic files. Typically, the same topics that the user accesses when choosing commands from the Help menus can be accessed by the context-sensitive Help feature. The topics with entries in the context ID column are used for context-sensitive help as well as for the Help menus. Notice that some topics have more than one context-sensitive help number. This enables the topic to be displayed when the user clicks on different regions of the screen. Of course, you're free to keep track of your topic files in any manner you choose.

Figure 7.7
Help tracker text file example

Ctx. String	Title	Browse Seq.	Key Words	Ctx. No.	Filename	Modified	Status
hlpidx_id_mp	Multipad Help Index	commands:0001	commands; menus	0xFFFF	helpex.idx	5/16/89	Done
mc_cmd_mp	Multipad Commands	commands:0004	commands; menus; files;	0x1000	helpex.cmd	5/16/89	Done
fm_cmd_mp	File Menu		documents	0x1001	helpex.cmd	5/16/89	Done
nc_cmd_mp	New Command	commands:0008	commands; new files;	0x1002	helpex.cmd	5/16/89	Done
oc_cmd_mp	Open Command	commands:0012	new documents				
			commands; file; open;	0x1003	helpex.cmd	5/16/89	Test
			documents; read only				
sac_cmd_mp	Save Command	commands:0016	commands; file; save; save as;	0x1004	helpex.cmd	5/16/89	Done
			documents; files				
sasc_cmd_mp	Save As Command	commands:0020	commands; file; save as; save;	0x1005	helpex.cmd	5/16/89	Done
			documents; files				
ptc_cmd_mp	Print Command	commands:0024	commands; file; print;	0x1006	helpex.cmd	5/16/89	Done
			documents; files				
psc_cmd_mp	Print Setup Command	commands:0028	commands; file; printer setup;	0x1007	helpex.cmd	5/16/89	Debug
			print				
ec_cmd_mp	Exit Command	commands:0032	commands; file; exit; exiting;	0x1008	helpex.cmd	5/16/89	Done
			close; closing; quit; quitting				
em_cmd_mp	Edit Menu	commands:0036	commands; menus; editing;	0x1009	helpex.cmd	5/16/89	Done
			documents				
uc_cmd_mp	Undo Command	commands:0040	commands; edit; editing; undo;	0x100A	helpex.cmd	5/16/89	Done
			typing				
cic_cmd_mp	Cut Command	commands:0044	commands; edit; editing; cut;	0x100B	helpex.cmd	5/16/89	Done
			cutting; text; Clipboard				
cyc_cmd_mp	Copy Command	commands:0048	commands; edit; editing; copy;	0x100C	helpex.cmd	5/16/89	Done
			copying; text; Clipboard				
pec_cmd_mp	Paste Command	commands:0052	commands; edit; editing; paste;	0x100D	helpex.cmd	5/16/89	Done
			insert; inserting; text; Clipboard				
erc_cmd_mp	Clear Command	commands:0056	commands; edit; editing; clear;	0x100E	helpex.cmd	5/16/89	Done
			text				
salc_cmd_mp	Select All Command	commands:0060	commands; edit; editing; select all;	0x100F	helpex.cmd	5/16/89	Done
			select; selecting; text				
wwc_cmd_mp	Word Wrap Command	commands:0064	commands; edit; editing; word wrap;	0x1010	helpex.cmd	5/16/89	Done
			wrapping; text; format				
ffc_cmd_mp	Type Face Command	commands:0068	commands; edit; editing; type face;	0x1011	helpex.cmd	5/16/89	Test
			font; text; format				
ptsc_cmd_mp	Point Size Command	commands:0072	commands; edit; editing; point size;	0x1012	helpex.cmd	5/16/89	Test
			text; format				

Figure 7.8
Help tracker worksheet
example

	A	B	C	D	E	F	G	H
1	Ctx. String	Text	Browse Seq.	Key Words	Ctx. No.	Filename	Modified	Status
2	hlpidx_id_mp	Multipad Help Index	commands:0001	commands; menus	0xFFFF	helpex.idx	5/16/89	Done
3	mc_cmd_mp	Multipad Commands	commands:0004	commands; menus; files;	0x1000	helpex.cmd	5/16/89	Done
4	fm_cmd_mp	File Menu		documents	0x1001	helpex.cmd	5/16/89	Done
5								
6	nc_cmd_mp	New Command	commands:0008	commands; new files;	0x1002	helpex.cmd	5/16/89	Done
7				new documents				
8	oc_cmd_mp	Open Command	commands:0012	commands; file; open;	0x1003	helpex.cmd	5/16/89	Test
9				documents; read only				
10	sac_cmd_mp	Save Command	commands:0016	commands; file; save; save as;	0x1004	helpex.cmd	5/16/89	Done
11				documents; files				
12	sasc_cmd_mp	Save As Command	commands:0020	commands; file; save as; save;	0x1005	helpex.cmd	5/16/89	Done
13				documents; files				
14	ptc_cmd_mp	Print Command	commands:0024	commands; file; print;	0x1006	helpex.cmd	5/16/89	Done
15				documents; files				
16	psc_cmd_mp	Print Setup Command	commands:0028	commands; file; printer setup;	0x1007	helpex.cmd	5/16/89	Debug
17				print				
18	ec_cmd_mp	Exit Command	commands:0032	commands; file; exit; exiting;	0x1008	helpex.cmd	5/16/89	Done
19				close; closing; quit; quitting				
20	em_cmd_mp	Edit Menu	commands:0036	commands; menus; editing;	0x1009	helpex.cmd	5/16/89	Done
21				documents				
22	uc_cmd_mp	Undo Command	commands:0040	commands; edit; editing; undo;	0x100A	helpex.cmd	5/16/89	Done
23				typing				
24	cic_cmd_mp	Cut Command	commands:0044	commands; edit; editing; cut;	0x100B	helpex.cmd	5/16/89	Done
25				cutting; text; Clipboard				
26	cyc_cmd_mp	Copy Command	commands:0048	commands; edit; editing; copy;	0x100C	helpex.cmd	5/16/89	Done
27				copying; text; Clipboard				
28	pec_cmd_mp	Paste Command	commands:0052	commands; edit; editing; paste;	0x100D	helpex.cmd	5/16/89	Done
29				insert; inserting; text; Clipboard				
30	erc_cmd_mp	Clear Command	commands:0056	commands; edit; editing; clear;	0x100E	helpex.cmd	5/16/89	Done
31				text				
32	salc_cmd_mp	Select All Command	commands:0060	commands; edit; editing; select all;	0x100F	helpex.cmd	5/16/89	Done
33				select; selecting; text				
34	wwc_cmd_mp	Word Wrap Command	commands:0064	commands; edit; editing; word wrap;	0x1010	helpex.cmd	5/16/89	Done
35				wrapping; text; format				
36	ffc_cmd_mp	Type Face Command	commands:0068	commands; edit; editing; type face;	0x1011	helpex.cmd	5/16/89	Test
37				font; text; format				
38	ptsc_cmd_mp	Point Size Command	commands:0072	commands; edit; editing; point size;	0x1012	helpex.cmd	5/16/89	Test
39				text; format				

Building the Help file

While the examples in this chapter are in C, you can also do the same tasks in Turbo Pascal. Code examples for both languages are included on your Borland product disks.

After the topic files for your Help system have been written, you are ready to create a Help project file and run a build to test the Help file. The Help project file contains all information the compiler needs to convert help topic files into a binary Help resource file.

You use the Help project file to tell the Help Compiler which topic files to include in the build process. Information in the Help project file also enables the compiler to map specific topics to context numbers (for the context-sensitive portion of Help).

After you have compiled your Help file, the development team programs the application so the user can access it.

This section describes the following:

- Creating a Help project file
- Compiling the Help file
- Programming the application to access Help

Creating the Help project file

You use the Help project file to control how the Help Compiler builds your topic files. The Help project file can contain up to six sections that perform the following functions:

Table 7.7
Help project file sections

Section	Function
[Files]	Specifies topic files to be included in the build. This section is mandatory.
[Options]	Specifies the level of error reporting, topics to be included in the build, the directory in which to find the files, and the location of your Help index. This section is optional.
[BuildTags]	Specifies valid build tags. This section is optional.
[Alias]	Assigns one or more context strings to the same topic. This section is optional.
[Map]	Associates context strings with context numbers. This section is optional.
[Bitmaps]	Specifies bitmap files to be included in the build. This section is optional.

You can use any ASCII text editor to create your Help project file. The extension of a Help project file is .HPJ. If you do not use the extension .HPJ on the **HC** command line, the compiler looks for a project file with this extension before loading the file. The .HLP output file will have the same name as the .HPJ file.

The order of the sections within the Help project file is arbitrary, except: that an [Alias] section must always precede the [Map] section (if an [Alias] section is used).

Section names are placed within square brackets using the following syntax:

```
[section-name]
```

You can use a semicolon to indicate a comment in the project file. The compiler ignores all text from the semicolon to the end of the line on which it occurs.

Specifying topic files

Use the [Files] section of the Help project file to list all topic files that the Help Compiler should process to produce a Help resource file. A Help project file must have a [Files] section.

The following sample shows the format of the [Files] section:

```
[FILES]
HELPEX.RTF ;Main topics for HelpEx application
TERMS.RTF ;Lookup terms for HelpEx application
```

*For more information about the **ROOT** option, see the section titled "Specifying the root directory."*

Using the path defined in the **ROOT** option, the Help Compiler finds and processes all files listed in this section of the Help project file. If the file is not on the defined path and cannot be found, the compiler generates an error.

You can include files in the build process using the C **#include** directive command. The **#include** directive uses this syntax:

```
#include <filename>
```

You must include the angle brackets around the filename. The pound sign (#) must be the first character in the line. The filename must specify a complete path, either the path defined by the **ROOT** option or an absolute directory path to the file.

You may find it easier to create a text file that lists all files in the Help project and include that file in the build, as in this example:

```
[FILES]
#include <hlpfiles.inc>
```

Specifying build tags

If you code build tags in your topic files, use the [BuildTags] section of the Help project file to define all the valid build tags for a particular Help project. The [BuildTags] section is optional.

The following example shows the format of the [BuildTags] section in a sample Help project file:

```
[BUILDTAGS]
WINENV           ;topics to include in Windows build
DEBUGBUILD      ;topics to include in debugging build
TESTBUILD       ;topics to include in a mini-build for testing
```

For information about coding build tags in topic files, see page 115.

The [BuildTags] section can include up to 30 build tags. The build tags are not case-sensitive and may not contain space characters. Only one build tag is allowed per line in this section. The compiler will generate an error message if anything other than a comment is listed after a build tag in the [BuildTags] section.

Specifying options

Use the [Options] section of the Help project file to specify the following options:

Table 7.8
The Help (Options) options

Option	Meaning
BUILD	Determines what topics the compiler includes in the build.
COMPRESS	Specifies compression of the Help resource file.
FORCEFONT	Specifies the creation of a Help resource file using only one font.
INDEX	Specifies the context string of the Help index.
MAPFONT SIZE	Determines the mapping of specified font sizes to different sizes.
MULTIKEY	Specifies alternate keyword mapping for topics.
ROOT	Designates the directory to be used for the Help build.
TITLE	Specifies the title shown for the Help system.
WARNING	Indicates the kind of error message the compiler reports.

These options can appear in any order within the [Options] section. The [Options] section is not required.

Detailed explanations of the available options follow.

Specifying error reporting

Use the **WARNING** option to specify the amount of debugging information that the compiler reports. The **WARNING** option has the following syntax:

`WARNING = level`

You can set the **WARNING** option to any of the following levels:

Table 7.9
WARNING levels

Level	Information Reported
1	Only the most severe warnings.
2	An intermediate level of warnings.
3	All warnings. This is the default level if no WARNING option is specified.

The following example specifies an intermediate level of error reporting:

```
[OPTIONS]  
WARNING=2
```

Use the DOS Ctrl+PrtSc accelerator key before you begin your compilation to echo errors which appear on the screen to your printer. Type Ctrl+PrtSc again to stop sending information to the printer.

The compiler reports errors to the standard output file, typically the screen. You may want to redirect the errors to a disk file so that you can browse it when you are debugging the Help system. The following example shows the redirection of compiler screen output to a file.

```
HC HELPEX >> errors.out
```

Specifying build topics

If you have included build tags in your topic files, use the **BUILD** option to specify which topics to conditionally include in the build. If your topic files have no build tags, omit the **BUILD** option from the [Options] section.



All build tags must be listed in the [BuildTags] section of the project file, regardless whether or not a given conditional compilation declares the tags.

See "Creating the Help topic files" on page 113 for information on assigning build tags to topics in the Help topic files.

The **BUILD** option line uses the following syntax:

`BUILD = expression`

Build expressions cannot exceed 255 characters in length, and must be entered on only one line. Build expressions use Boolean

logic to specify which topics within the specified Help topic files the compiler will include in the build. The compiler evaluates all build expressions from left to right. The tokens of the language (listed in order of precedence from highest to lowest) are:

Table 7.10
Build tag order of
precedence

Token	Description
<tag>	Build tag
()	Parentheses
~	NOT operator
&	AND operator
	OR operator

For example, if you coded build tags called WINENV, APP1, and TEST_BUILD in your topic files, you could include one of the following build expressions in the [Options] section:

Table 7.11: Build expression examples

Build expression	Topics built
BUILD = WINENV	Only topics that have the WINENV tag
BUILD = WINENV & APP1	Topics that have both the WINENV and APP1 tags
BUILD = WINENV APP1	Topics that have either the WINENV tag or the APP1 tag
BUILD = (WINENV APP1) & TESTBUILD	Topics that have either the WINENV or the APP1 tags and that also have the TESTBUILD tag
BUILD =~ APP1	Topics that do not have an APP1 tag

Specifying the root directory

Use the **ROOT** option to designate the root directory of the Help project. The compiler searches for files in the specified root directory.

The **ROOT** option uses the following syntax:

```
ROOT = pathname
```

For example, the following root option specifies that the root directory is \BUILD\TEST on drive D:

```
[OPTIONS]  
ROOT=D:\BUILD\TEST
```

The **ROOT** option allows you to refer to all relative paths off the root directory of the Help project. For example, the following entry in the [Files] section refers to a relative path off the root directory:

```
TOPICS\FILE.RTF
```

To refer to a file in a fixed location, independent of the project root, you must specify a fully qualified or “absolute” path, including a drive letter, if necessary, as in the following line:

```
D:\HELPTTEST\TESTFILE.RTF
```

If you do not include the **ROOT** option in your Help project file, all paths are relative to the current DOS directory.

Specifying the Index

Use the **INDEX** option to identify the context string of the Help index. Because the Index button gives the user access to the index from anywhere in the Help system, you will probably not want to author terms to jump to the index. Users access this general index either from the Help menu of the application or by choosing the Index button from the Help window.

Assigning a context string to the index topic in the [Options] section lets the compiler know the location of the main index of Help topics for the application’s Help file. If you do not include the **INDEX** option in the [Options] section, the compiler assumes that the first topic it encounters is the index.

The **INDEX** option uses the following syntax:

```
INDEX = context-string
```

For information on assigning context strings, see page 116.

The context string specified must match the context string you assigned to the Help index topic. In the following example, the writer informs the compiler that the context string of the Help index is “main_index”:

```
[OPTIONS]  
INDEX=main_index
```

Assigning a title to the Help system

You can assign a title to your Help system with the **TITLE** option. The title appears in the title bar of the Help window with the word “Help” automatically appended, followed by the DOS filename of the Help resource file.

The **TITLE** option uses the following syntax:

```
TITLE = Help-system-title-name
```

Titles are limited to 32 characters in length. If you do not specify a title using the **TITLE** option, only the word Help followed by the Help system filename will be displayed in the title bar. Because the compiler always inserts the word Help, don’t duplicate it in your title.

Converting fonts You can use the **FORCEFONT** option to create a Help resource file that is made up of only one typeface or font. This is useful if you must compile a Help system using topic files that include fonts not supported by your users' systems.

The **FORCEFONT** option uses the following syntax:

FORCEFONT = *fontname*

See page 112 for a list of the fonts Windows ships with.

The *fontname* parameter is any Windows system font. Note that the *fontname* used in the **FORCEFONT** option cannot contain spaces. Therefore, Tms Rmn font cannot be used with **FORCEFONT**.

Font names must be spelled the same as they are in the Fonts dialog box in Control Panel. Font names do not exceed 20 characters in length. If you designate a font that is not recognized by the compiler, an error message is generated and the compilation continues using the default Helvetica (Helv) font.

Changing font sizes The font sizes specified in your topic files can be mapped to different sizes using the **MAPFONTSIZE** option. In this manner, you can create and edit text in a size chosen for easier viewing in the topic files and have them sized by the compiler for the actual Help display. This may be useful if there is a large size difference between your authoring monitor and your intended display monitor.

The **MAPFONTSIZE** option uses the following syntax:

MAPFONTSIZE = *m*[-*n*]:*p*

The *m* parameter is the size of the source font, and the *p* parameter is the size of the desired font for the Help resource file. All fonts in the topic files that are size *m* are changed to size *p*. The optional parameter *n* allows you to specify a font range to be mapped. All fonts in the topic files falling between *m* and *n*, inclusive, are changed to size *p*. The following examples illustrate the use of the **MAPFONTSIZE** option:

```
MAPFONTSIZE=12-24:16 ;make fonts from 12 to 24 come out 16.  
MAPFONTSIZE=8:12 ;make all size 8 fonts come out size 12.
```

Note that you can map only one font size or range with each **MAPFONTSIZE** statement used in the Options section. If you use more than one **MAPFONTSIZE** statement, the source font size or

range specified in subsequent statements cannot overlap previous mappings. For instance, the following mappings would generate an error when the compiler encountered the second statement:

```
MAPFONTSIZE=12-24:16 MAPFONTSIZE=14:20
```

Because the second mapping shown in the first example contains a size already mapped in the preceding statement, the compiler will ignore the line. There is a maximum of five font ranges that can be specified in the project file.

Multiple keyword tables

The **MULTIKEY** option specifies a character to be used for an additional keyword table.

The **MULTIKEY** option uses the following syntax:

```
MULTIKEY = footnote-character
```

The *footnote-character* parameter is the case-sensitive letter to be used for the keyword footnote. The following example illustrates the enabling of the letter *L* for a keyword-table footnote:

```
MULTIKEY=L
```



You must be sure to limit your keyword-table footnotes to one case, usually uppercase. In the previous example, topics with the footnote *L* would have their keywords incorporated into the additional keyword table, whereas those assigned the letter *l* would not.

You may use any alphanumeric character for a keyword table except *K* and *k*, which are reserved for Help's normal keyword table. There is an absolute limit of five keyword tables, including the normal table. However, depending upon system configuration and the structure of your Help system, a practical limit of only two or three may actually be the case. If the compiler cannot create an additional table, the excess table is ignored in the build.

Compressing the file

You can use the **COMPRESS** option to reduce the size of the Help resource file created by the compiler. The amount of file compression realized will vary according to the number, size and complexity of topics that are compiled. In general, the larger the Help files, the more they can be compressed.

The **COMPRESS** option uses the following syntax:

```
COMPRESS = TRUE | FALSE
```

Because the Help application can load compressed files quickly, there is a clear advantage in creating and shipping compressed Help files with your application. Compiling with compression turned on, however, may increase the compile time, because of the additional time required to assemble and sort a key-phrase table. Thus, you may want to compile without compression in the early stages of a project.

The **COMPRESS** option causes the compiler to compress the system by combining repeated phrases found within the source file(s). The compiler creates a phrase-table file with the .PH extension if it does not find one already in existence. If the compiler finds a file with the .PH extension, it will use the file for the current compilation. This is in order to speed compression when not a lot of text has changed since the last compilation.

Deleting the key-phrase file before each compilation will prevent the compiler from using the previous file. Maximum compression will result only by forcing the compiler to create a new phrase table.

Specifying alternate context strings

Use the [Alias] section to assign one or more context strings to the same topic alias. Because context strings must be unique for each topic and cannot be used for any other topic in the Help project, the [Alias] section provides a way to delete or combine Help topics without recoding your files. The [Alias] section is optional.

For example, if you create a topic that replaces the information in three other topics, and you delete the three, you will have to search through your files for invalid cross-references to the deleted topics. You can avoid this problem by using the [Alias] section to assign the name of the new topic to the deleted topics. You can also use the [Alias] section when your application program has multiple context identifiers for which you have only one topic. This can be the case with context-sensitive Help.

Each expression in the [Alias] section has the following format:

context_string=alias

In the alias expression, the *alias* parameter is the alternate string, or alias name, and the *context_string* parameter is the context string identifying a particular topic. An alias string has the same format and follows the same conventions as the topic context string. That is, it is not case-sensitive and may contain the

alphabetic characters A – Z, numeric characters 0 – 9, and the period and underscore characters.

The following example illustrates an [Alias] section:

```
[ALIAS]
sm_key=key_shrtcuts
cc_key=key_shrtcuts
st_key=key_shrtcuts;combined into keyboard shortcuts topic
clskey=us_dlog_bxs
maakey=us_dlog_bxs;covered in using dialog boxes topic
chk_key=dlogprts
drp_key=dlogprts
lst_key=dlogprts
opt_key=dlogprts
tbx_key=dlogprts;combined into parts of dialog box topic
frmtxt=edittxt
wrptxt=edittxt
seltxt=edittxt;covered in editing text topic
```



You can use alias names in the [Map] section of the Help project file. If you do, however, the [Alias] section must precede the [Map] section.

Mapping context-sensitive topics

*For more information on
context-sensitive Help, see
page 107.*

If your Help system supports context-sensitive Help, use the [Map] section to associate either context strings or aliases to context numbers. The context number corresponds to a value the parent application passes to the Help application in order to display a particular topic. This section is optional.

When writing the [Map] section, you can do the following:

- Use either decimal or hexadecimal numbers formatted in standard C notation to specify context numbers.
- Assign no more than one context number to a context string or alias.
Assigning the same number to more than one context string will generate a compiler error.
- Separate context numbers and context strings by an arbitrary amount of whitespace using either space characters or tabs.

You can use the C **#include** directive to include other files in the mapping. In addition, the Map section supports an extended format that lets you include C files with the .H extension directly.

Entries using this format must begin with the **#define** directive and may contain comments in C format, as in this example:

```
#define context_string context_number /* comment */
```

The following example illustrates several formats you can use in the [Map] section:

```
[MAP]
```

These eight entries give hexadecimal equivalents for the context numbers.

```
Edit_Window      0x0001
Control_Menu     0x0002
Maximize_Icon    0x0003
Minimize_Icon    0x0004
Split_Bar        0x0005
Scroll_Bar       0x0006
Title_Bar        0x0007
Window_Border    0x0008
```

These five entries show decimal context numbers.

```
dcmb_scr        30; Document Control-menu Icon
dmxi_scr        31; Document Maximize Icon
dmni_scr        32; Document Minimize Icon
dri_scr         33; Document Restore Icon
dtb_scr         34; Document Title Bar
```

These five entries show how you might include topics defined in a C include file.

```
#define vscroll  0x010A /* Vertical Scroll Bar */
#define hscroll  0x010E /* Horizontal Scroll Bar */
#define scrollthm 0x0111 /* Scroll Thumb */
#define upscroll 0x0112 /* Up Scroll Arrow */
#define dncroll  0x0113 /* Down Scroll Arrow */
```

*This entry shows a C **#include** directive for some generic topics.*

```
#include <sample.h>
```

If context numbers use the **#define** directive, and the file containing the **#define** statements is included in both the application code and the Help file, then updates made to the context numbers by the application programmers will automatically be reflected in the next Help build.

You can define the context strings listed in the [Map] section either in a Help topic or in the [Alias] section. The compiler generates a warning message if a context string appearing in the [Map] section is not defined in any of the topic files or in the [Alias] section.



If you use an alias name, the [Alias] section must precede the [Map] section in the Help project file.

Including bitmaps by reference

If your Help system uses bitmaps by reference, the filenames of each of the bitmaps must be listed in the [Bitmaps] section of the project file. The following example illustrates the format of the [Bitmaps] section.

```
[BITMAPS]
DUMP01.BMP
DUMP02.BMP
DUMP03.BMP
c:\PROJECT\HELP\BITMAPS\DUMP04.BMP
```



The [Bitmaps] section uses the same rules as the [Files] section for locating bitmap files.

Compiling Help files

After you have created a Help project file, you are ready to build a Help file using the Help Compiler. The compiler generates the binary Help resource file from the topic files listed in the Help project file. When the build process is complete, your application can access the Help resource file that results.

Before initiating a build operation to create the Help file, consider the locations of the following files:

- The Help Compiler, HC.EXE. The compiler must be in a directory from which it can be executed. This could be the current working directory, on the path set with the PATH environment variable, or a directory specified by a full pathname, as follows:

```
C:\BIN\HC_HELPEX.HPJ
```

- The Help project file, *filename*.HPJ. The project file can be located either in the current directory or specified by a path, as follows:

```
C:\BIN\HC D:\MYPROJ\HELPEX.HPJ
```

- The topic files named in the Help project file, saved as RTF. The topic files may be located in the current working directory, a subdirectory of the current working directory specified in the [Files] section, or the location specified in the Root option.
- Files included with the **#include** directive in the Help project file. Since the **#include** directive can take pathnames, then any number of places will work for these files.

- All bitmap files listed by reference in the topic files.

You must also place any files named in an **#include** directive in the path of the project root directory or specify their path using the **ROOT** option. The compiler searches only the directories specified in the Help project file. For information about the **ROOT** option, see the section titled "Specifying the root directory," on page 133.



If you use a RAM drive for temporary files (set with the DOS environment variable TMP), it must be large enough to hold the compiled Help resource file. If your Help file is larger than the size of the available RAM drive, the compiler will generate an error message and the compilation will be aborted.

Using the Help Compiler

To run the Help Compiler, use the **HC** command. There are no options for **HC**. All options are specified in the Help project file.

The **HC** command uses the following syntax:

```
HC filename.HPJ
```

As the compiler program runs, it displays sequential periods on the screen, indicating its progress in the process. Error messages are displayed when each error condition is encountered. When the Help Compiler has finished compiling, it writes a Help resource file with an .HLP extension in the current directory and returns to the DOS prompt. The Help resource file that results from the build has the same name as does the Help project file.

Compiler errors and status messages can be redirected to a file using standard DOS redirection syntax. This is useful for a lengthy build where you may not be monitoring the entire process. The redirected file is saved as a text file that can be viewed with any ASCII editor.

Programming the application to access Help

The application development team must program the application so that the user can access both the Windows Help application and your Help file. The Help application is a stand alone Windows application, and your application can ask Windows to run the Help application and specify the topic that Help is to show the user. To the user, Help appears to be part of your application, but it acts like any other Windows application.

Calling WinHelp from an application

An application makes a Help system available to the user by calling the **WinHelp** function.

The **WinHelp** function uses the following syntax:

The C and Pascal samples are provided on disk.

```
BOOL WinHelp (hWnd, lpHelpFile, wCommand, dwData)
```

The *hWnd* parameter identifies the window requesting Help. The Windows Help application uses this identifier to keep track of which applications have requested Help.

The *lpHelpFile* parameter specifies the name (with optional directory path) of the Help file containing the desired topic.

The *wCommand* parameter specifies either the type of search that the Windows Help application is to use to locate the specified topic, or that the application no longer requires Help. It may be set to any one of the following values:

Table 7.12
wCommand values

Value	Meaning
HELP_CONTEXT	Displays Help for a particular topic identified by a context number.
HELP_HELPONHELP	Displays the Using Help index topic.
HELP_INDEX	Displays the main Help index topic.
HELP_KEY	Displays Help for a topic identified by a keyword.
HELP_MULTIKEY	Displays Help for a topic identified by a keyword in an alternate keyword table.
HELP_QUIT	Informs the Help application that Help is no longer needed. If no other applications have asked for Help, Windows closes the Help application.
HELP_SETINDEX	Displays a designated Help index topic.

The *dwData* parameter specifies the topic for which the application is requesting Help. The format of *dwData* depends upon the value of *wCommand* passed when your application calls **WinHelp**. The following list describes the format of *dwData* for each value of *wCommand*.

Table 7.13
dwData formats

<i>wCommand</i> value	<i>dwData</i> format
HELP_CONTEXT	An unsigned long integer containing the context number for the topic. Instead of using HELP_INDEX, HELP_CONTEXT can use the value -1.
HELP_HELPOPHELP	Ignored.
HELP_INDEX	Ignored.
HELP_KEY	A long pointer to a string which contains a keyword for the desired topic.
HELP_MULTIKKEY	A long pointer to the MULTIKEYHELP structure, as defined in WINDOWS.H. This structure specifies the table footnote character and the keyword.
HELP_QUIT	Ignored.
HELP_SETINDEX	An unsigned long integer containing the context number for the topic.

Because it can specify either a context number or a keyword, **WinHelp** supports both context-sensitive and topical searches of the Help file.



To ensure that the correct index remains set, the application should call **WinHelp** with *wCommand* set to HELP_SETINDEX (with *dwData* specifying the corresponding context identifier) following each call to **WinHelp** with a command set to HELP_CONTEXT. HELP_INDEX should never be used with HELP_SETINDEX.

Getting context-sensitive Help

Context-sensitive Help should be made available when a user wants to learn about the purpose of a particular window or control. For example, the user might pull down the File menu, select the Open command (by using the arrow keys), and then press *F1* to get Help for the command.

Implementing certain types of context-sensitive help requires advanced programming techniques. The Helpex sample application illustrates the use of two techniques. These techniques are described in the following sections.

Shift+F1 support

To implement a *Shift+F1* mode, Helpex responds to the *Shift+F1* accelerator key by calling **SetCursor** to change the shape of the cursor to an arrow pointer supplemented by a question mark.

```
case WM_KEYDOWN:
    if (wParam == VK_F1) {
        /* If Shift-F1, turn help mode on and set help
           cursor */

        if (GetKeyState(VK_SHIFT)) {
            bHelp = TRUE;
            SetCursor(hHelpCursor);
            return (DefWindowProc(hWnd, message,
                                   wParam, lParam));
        }
        /* If F1 without shift, then call up help main
           index topic */
        else {
            WinHelp(hWnd, szHelpFileName, HELP_INDEX, 0L);
        }
    }
    else if (wParam == VK_ESCAPE && bHelp) {
        /* Escape during help mode: turn help mode off */
        bHelp = FALSE;
        SetCursor((HCURSOR)GetClassWord(hWnd, GCW_HCURSOR));
    }
    break;
```

As long as the user is in Help mode (that is, until the user clicks the mouse or presses *Esc*), Helpex responds to **WM_SETCURSOR** messages by resetting the cursor to the arrow and question mark combination.

```
case WM_SETCURSOR:
    /* In help mode it is necessary to reset the cursor
       in response to every WM_SETCURSOR message.
       Otherwise, by default, Windows will reset the
       cursor to that of the window class. */

    if (bHelp) {
        SetCursor(hHelpCursor);
        break;
    }
    return (DefWindowProc(hWnd, message, wParam, lParam));
    break;

case WM_INITMENU:
    if (bHelp) {
        SetCursor(hHelpCursor);
```

```

    }
    return (TRUE);

```

When the user is in *Shift+F1* Help mode and clicks the mouse button, Helpex will receive a `WM_NCLBUTTONDOWN` message if the click is in a nonclient area of the application window. By examining the *wParam* value of this message, the program can determine which context ID to pass to **WinHelp**.

```

case WM_NCLBUTTONDOWN:
    /* If we are in help mode (Shift+F1), then display
       context-sensitive help for nonclient area. */
    if (bHelp) {
        dwHelpContextId =
            (wParam == HTCAPTION)      ? (DWORD)HELPID_TITLE_BAR:
            (wParam == HTSIZE)        ? (DWORD)HELPID_SIZE_BOX:
            (wParam == HTREDUCE)      ? (DWORD)HELPID_MINIMIZE_ICON:
            (wParam == HTZOOM)        ? (DWORD)HELPID_MAXIMIZE_ICON:
            (wParam == HTSYSTEMMENU)  ? (DWORD)HELPID_SYSTEM_MENU:
            (wParam == HTBOTTOM)      ? (DWORD)HELPID_SIZING_BORDER:
            (wParam == HTBOTTOMLEFT) ? (DWORD)HELPID_SIZING_BORDER:
            (wParam == HTBOTTOMRIGHT) ? (DWORD)HELPID_SIZING_BORDER:
            (wParam == HTTOP)         ? (DWORD)HELPID_SIZING_BORDER:
            (wParam == HTLEFT)        ? (DWORD)HELPID_SIZING_BORDER:
            (wParam == HTRIGHT)       ? (DWORD)HELPID_SIZING_BORDER:
            (wParam == HTTOPLEFT)     ? (DWORD)HELPID_SIZING_BORDER:
            (wParam == HTTOPRIGHT)    ? (DWORD)HELPID_SIZING_BORDER:
            (DWORD)0L;

        if (!(BOOL)dwHelpContextId)
            return (DefWindowProc(hWnd, message, wParam, lParam));

        bHelp = FALSE;
        WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, dwHelpContextId);
        break;
    }

    return (DefWindowProc(hWnd, message, wParam, lParam));

```

F1 support

Context-sensitive *F1* support for menus is relatively easy to implement in your application. If a menu is open and the user presses *F1* while one of the menu items is highlighted, Windows sends a `WM_ENTERIDLE` message to the application to indicate that the system is going back into an idle state after having determined that *F1* was not a valid key stroke for choosing a

menu item. You can take advantage of this idle state by looking at the keyboard state at the time of the WM_ENTERIDLE message.

If the *F1* key is down, then you can simulate the user's pressing the *Enter* key by posting a WM_KEYDOWN message using VK_RETURN. You don't really want your application to execute the menu command. What you do is set a flag (bHelp=TRUE) so that when you get the WM_COMMAND message for the menu item, you don't execute the command. Instead, the topic for the menu item is displayed by Windows Help.

The following code samples illustrate *F1* sensing for menu items.

```
case WM_ENTERIDLE:
    if ((wParam == MSGF_MENU) &&
        (GetKeyState(VK_F1) & 0x8000)) {
        bHelp = TRUE;
        PostMessage(hWnd, WM_KEYDOWN, VK_RETURN, 0L);
    }
    break;

case WM_COMMAND:
    /* Was F1 just pressed in a menu, or are we in help mode
       (Shift+F1)? */
    if (bHelp) {
        dwHelpContextId =
            (wParam == IDM_NEW)?(DWORD)HELPID_FILE_NEW:
            (wParam == IDM_OPEN)?(DWORD)HELPID_FILE_OPEN:
            (wParam == IDM_SAVE)?(DWORD)HELPID_FILE_SAVE:
            (wParam == IDM_SAVEAS)?(DWORD)HELPID_FILE_SAVE_AS:
            (wParam == IDM_PRINT)?(DWORD)HELPID_FILE_PRINT:
            (wParam == IDM_EXIT)?(DWORD)HELPID_FILE_EXIT:
            (wParam == IDM_UNDO)?(DWORD)HELPID_EDIT_UNDO:
            (wParam == IDM_CUT)?(DWORD)HELPID_EDIT_CUT:
            (wParam == IDM_CLEAR)?(DWORD)HELPID_EDIT_CLEAR:
            (wParam == IDM_COPY)?(DWORD)HELPID_EDIT_COPY:
            (wParam == IDM_PASTE)?(DWORD)HELPID_EDIT_PASTE:
            (DWORD)0L;

        if (!dwHelpContextId) {
            MessageBox( hWnd, "Help not available for Help Menu item",
                "Help Example", MB_OK);
            return (DefWindowProc(hWnd, message, wParam, lParam));
        }

        bHelp = FALSE;
        WinHelp(hWnd, szHelpFileName, HELP_CONTEXT, dwHelpContextId);
        break;
    }
}
```

Detecting *F1* in dialog boxes is somewhat more difficult than in menus. You must install a message filter, using the `WH_MSGFILTER` option of the **SetWindowsHook** function. Your message filter function responds to `WM_KEYDOWN` and `WM_KEYUP` messages for `VK_F1` when they are sent to a dialog box, as indicated by the `MSGF_DIALOGBOX` code. By examining the message structure passed to the filter, you can determine the context of the *F1* help—what the dialog box is, and the specific option or item. You should not call **WinHelp** while processing the filtered message, but rather post an application-defined message to your application to call **WinHelp** at the first available opportunity.

Getting help on items on the Help menu

Sometimes users may want information about a general concept in the application rather than about a particular control or window. In these cases, the application should provide Help for a particular topic that is identified by a keyword rather than a context identifier.

For example, if the Help file for your application contains a topic that describes how the keyboard is used, you could place a “Keyboard” item in your Help menu. Then, when the user selects that item, your application calls **WinHelp** and requests that topic:

```
case IDM_HELP_KEYBOARD:
    WinHelp (hWnd, lpHelpFile, HELP_KEY, (LPSTR) "Keyboard");
    break;
```

Accessing additional keyword tables

Your application may have commands or terms that correspond to terms in a similar, but different, application. Given a keyword, the application can call **WinHelp** and look up topics defined in an alternate keyword table. This multikey functionality is accessed through the **WinHelp** hook with the *wCommand* parameter set to `HELP_MULTIKEY`.

You specify the footnote character for the alternate keyword table, and the keyword or phrase, via a **MULTIKEYHELP** structure which is passed as the *dwData* parameter in the call to **WinHelp**. This structure is defined in `WINDOWS.H` as:

```
typedef struct tagMULTIKEYHELP {
    WORD mdSize;
    BYTE mkKeyList;
    BYTE szKeyPhrase[1];
} MULTIKEYHELP;
```

The following table lists the format of the fields of the **MULTIKEYHELP** structure:

Table 7.14
MULTIKEYHELP structure
formats

Parameter	Format
mkSize	The size of the structure, including the keyword (or phrase) and the associated key-table letter.
mkKeyList	A single character which defines the footnote character for the alternate keyword table to be searched.
szKeyPhrase	A null-terminated keyword or phrase to be looked up in the alternate keyword table.

The following example illustrates a keyword search for the word "frame" in the alternate keyword table designated with the footnote character "L":

```
MULTIKEYHELP mk;
char szKeyword[]="frame";
mk.mkSize = sizeof(MULTIKEYHELP) + (WORD)lstrlen(szKeyword);
mk.mkKeylist = 'L';
mk.szKeyphrase = szKeyword;
WinHelp(hWnd, lpHelpfile, HELP_MULTIKEY, (LPSTR)&mk);
```

Canceling Help

The Windows Help application is a shared resource that is available to all Windows applications. In addition, since it is a stand alone application, the user can execute it like any other application. As a result, your application has limited control over the Help application. While your application cannot directly close the Help application window, your application can inform the Help application that Help is no longer needed. Before closing its main window, your application should call **WinHelp** with the *wCommand* parameter set to **HELP_QUIT**, as shown in the following example, to inform the Help application that your application will not need it again.

```
case WM_DESTROY:
    WinHelp (hWnd, lpHelpFile, HELP_QUIT, NULL);
```

An application that has called **WinHelp** at some point during its execution must call **WinHelp** with the *wCommand* parameter set to **HELP_QUIT** before the application exits from **WinMain** (typically during the **WM_DESTROY** message processing).

If an application opens more than one Help file, then it must call **WinHelp** to quit help for each file.

If an application or DLL has opened a Help file but no longer wants the associated instance of the Help engine (WINHELP.EXE) to remain active, then the application or DLL should call **WinHelp** with the *wCommand* parameter set to HELP_QUIT to destroy the instance of the Help engine.

Under no circumstances should an application or DLL terminate without calling **WinHelp** for any of the opened Help files. A Help file is opened if any other **WinHelp** call has been previously made using the Help filename.

The Windows Help application does not exit until all windows that have called **WinHelp** have called it with *wCommand* set to HELP_QUIT. If an application fails to do so, then the Help application will continue running after all applications that requested Help have terminated.

Help examples

This section contains some examples of Help source files and their corresponding topics as displayed in Help. Each example shows a topic (or part of a topic) as it appears to the Help writer in the RTF-capable word processor and as it appears to the user in the Help window. You can use these examples as guides when creating your own topic files. The examples should help you predict how a particular topic file created in a word processor will appear to the user.

Figure 7.9
Word for Windows topic

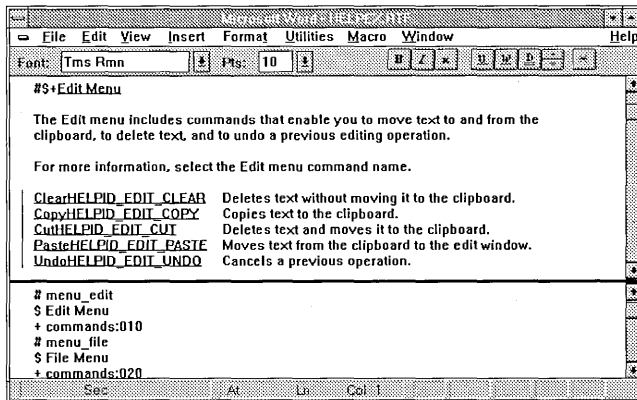


Figure 7.10
Help topic display

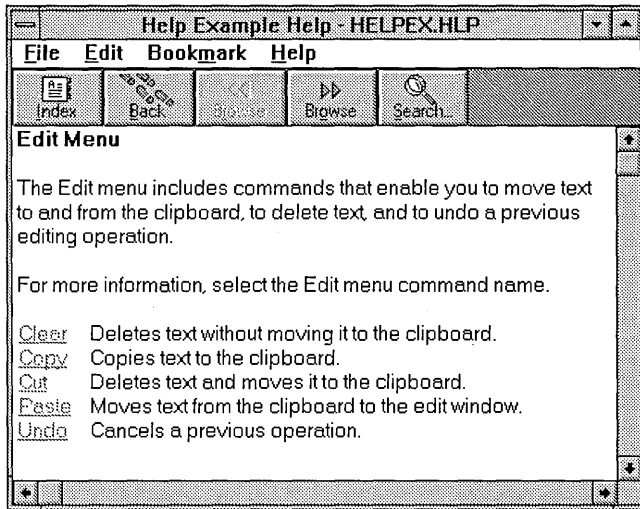


Figure 7.11
Bitmap by reference in topic

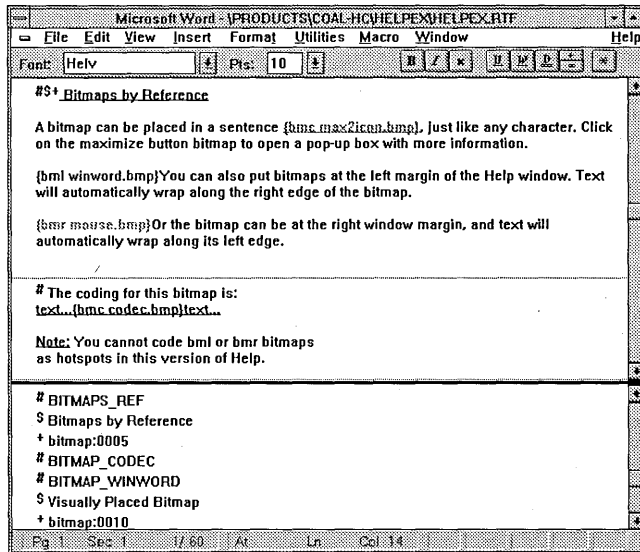
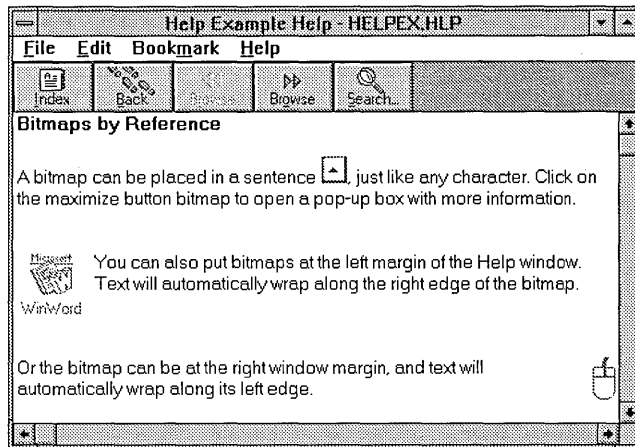


Figure 7.12
Help topic display



The Helpex project file

The following is the Helpex (sample Help) project file:

```
[OPTIONS]
ROOT=c:\help
INDEX=main_index
TITLE=Help Example
COMPRESS=true

[FILES]
helpex.rtf ; jump topics
terms.rtf ; look-up terms

[MAP]
main_index 0xFFFF
#define HELPID_EDIT_CLEAR 100
#define HELPID_EDIT_COPY 101
#define HELPID_EDIT_CUT 102
#define HELPID_EDIT_PASTE 103
#define HELPID_EDIT_UNDO 104
#define HELPID_FILE_EXIT 200
#define HELPID_FILE_NEW 201
#define HELPID_FILE_OPEN 202
#define HELPID_FILE_PRINT 203
#define HELPID_FILE_SAVE 204
#define HELPID_FILE_SAVE_AS 205
#define HELPID_EDIT_WINDOW 300
#define HELPID_MAXIMIZE_ICON 301
#define HELPID_MINIMIZE_ICON 302
#define HELPID_SPLIT_BAR 303
#define HELPID_SIZE_BOX 304
```



```
#define HELPID_SYSTEM_MENU 305
#define HELPID_TITLE_BAR 306
#define HELPID_SIZING_BORDER 307
```

Error messages

Borland C++ error messages include: compile-time, DPMI server, Help, MAKE, run-time, TLIB, and TLINK. We explain them here; user-interface error messages are explained in online Help.

The *type* of message (such as *Compile-time* or *Help*) is noted in the column to the left. Most explanations provide a probable cause and remedy for the error or warning message.

Finding a message in this appendix

The messages are listed in ASCII alphabetic order; messages beginning with symbols normally come first, followed by numbers and letters of the alphabet.

Since messages that begin with a variable cannot be alphabetized by what you will actually see when you receive such a message, all such messages are alphabetized by the word following the variable.

For example, if you have a C++ function **goforit**, you might receive the following actual message:

```
goforit must be declared with no arguments
```

In order to look this error message up, you would need to find

function must be declared with no arguments

alphabetized starting with the word “must”.

If the variable occurs later in the text of the error message (for example, “Address of overloaded function *function* doesn’t match

Type”), you can find the message in correct alphabetical order; in this case, under the letter A.

Types of messages

The kinds of messages you get are different, depending on where they come from. This section lists each category with a table of variables that it may contain.

Compile-time messages

The Borland C++ compiler diagnostic messages fall into three classes: fatal errors, errors, and warnings.

Fatal errors are rare. Some of them indicate an internal compiler error. When a fatal error occurs, compilation stops immediately. You must take appropriate action and then restart compilation.

Errors indicate program syntax errors, command-line errors, and disk or memory access errors. The compiler completes the current phase of the compilation and then stop. The compiler attempts to find as many real errors in the source program as possible during each phase (preprocessing, parsing, optimizing and code-generating).

Warnings do not prevent the compilation from finishing. They indicate conditions that are suspicious, but are usually legitimate as part of the language. The compiler also produces warnings if you use some machine-dependent constructs in your source files.

The compiler prints messages with the message class first, then the source file name and line number where the compiler detected the condition, and finally the text of the message itself.

Line numbers are not exact

You should be aware of one detail about line numbers in error messages: the compiler only generates messages as they are detected. Because C and C++ do not force any restrictions on placing statements on a line of text, the true cause of the error may be one or more lines before or after the line number mentioned.

The following variable names and values are some of those that appear in the compiler messages listed in this appendix (most are self-explanatory). When you get an error message, the appropriate name or value is substituted.

Table A.1
Compile-time message
variables

What you'll see in the manual	What you'll see on your screen
<i>argument</i>	An argument (command-line or other)
<i>class</i>	A class name
<i>filename</i>	A file name (with or without extension)
<i>function</i>	A function name
<i>group</i>	A group name
<i>identifier</i>	An identifier (variable name or other)
<i>language</i>	The name of a programming language
<i>member</i>	The name of a data member or member function
<i>message</i>	A message string
<i>module</i>	A module name
<i>number</i>	An actual number
<i>option</i>	An option (command-line or other)
<i>parameter</i>	A parameter name
<i>segment</i>	A segment name
<i>specifier</i>	A type specifier
<i>symbol</i>	A symbol name
<i>type</i>	A type name
XXXXh	A 4-digit hexadecimal number, followed by <i>h</i>

DPMI server messages

All Dos Protected Mode Interface (DPMI) server error messages but one relate to conditions that are out of the scope of Borland's software control. If the program can't find your machine-type, the message directs you to a solution. Otherwise there is a problem with the configuration of your system, your disks, or the hardware itself. Normally you will receive one of the common messages but if your disk has been damaged and you receive an undocumented message, call Technical Support.

The error messages are all fatal. They contain no variables and there are no warnings.

Help compiler messages

The Help Compiler displays messages when it encounters errors or warnings in building the Help resource file. Messages during processing of the project file are numbered beginning with the letter *P* and appear as in the following examples:

```
Error P1025: line...7 of filename.HPJ : Section heading sectionname
unrecognized.
```

```
Warning P1039: line...38 of filename.HPJ : [BUILDTAGS] section
missing.
```

Messages that occur during processing of the RTF topic file(s) are numbered beginning with the letter *R* and appear as in the following examples:

Error R2025: File environment error.

Warning R2501: Using old key-phrase table.

Topic numbers

Whenever possible, the compiler will display the topic number or file name that contains the error. If you numbered your topics, the topic number given with an error message refers to that topic's sequential position in your RTF file (first, second, and so on). These numbers may be identical to the page number shown by your word processor, depending on the number of lines you have assigned to the hypothetical printed page. Remember that topics are separated by hard page breaks, even though there is no such thing as a "page" in online Help.

Messages beginning with the word "Error" (on your screen) are fatal errors. Errors are always reported, and no usable Help resource file will result from the build. Messages beginning with the word "Warning" (on your screen) are less serious in nature. A build with warnings will produce a valid Help resource file that will load under Windows, but the file may contain operational errors. You can specify the amount of warning information to be reported by the compiler.

During processing of the project file, the compiler ignores lines that contain errors and attempts to continue with the build. This means that errors encountered early in a build may result in many more errors being reported as the build continues. Similarly, errors during processing of the RTF topic files will be reported and if not serious, the compiler will continue with the build. A single error condition in the topic file may result in more than one error message being reported by the compiler. For instance, a misidentified topic will cause an error to be reported every time jump terms refer to the correct topic identifier. Such a mistake is easily rectified by simply correcting the footnote containing the wrong context string.

Table A.2
Help message variables

What you'll see in the manual	What you'll see on your screen
<i>contextname</i>	A context string alias
<i>context-string</i>	A context string
<i>filename</i>	A file name (with or without extension)
<i>fontname</i>	The name of a font
<i>optionname</i>	An option name

Table A.2: Help message variables (continued)

<i>sectionname</i>	A section heading
<i>tagname</i>	A build tag
<i>topicnumber</i>	A topic number

MAKE messages

MAKE diagnostic messages fall into two classes: errors and fatal errors.

- Errors indicate some sort of syntax or semantic error in the source makefile.
- Fatal errors prevent processing of the makefile. You must take appropriate action and then restart MAKE.

The following generic names and values appear in the messages listed in this section. When you get an error message, the appropriate name or value is substituted.

Table A.3
MAKE error message
variables

What you'll see in the manual	What you'll see on your screen
<i>argument(s)</i>	An argument (command-line or other)
<i>expression</i>	An expression
<i>filename</i>	A file name (with or without extension)
<i>line number</i>	A line number
<i>message</i>	A message string
<i>target</i>	A receiver

Run-time error messages

Borland C++ has a small number of run-time error messages. These errors occur after the program has successfully compiled and while it is running.

TLIB messages

TLIB has error and warning messages. The following generic names and values appear in TLIB messages. When you get a message, the variable is substituted.

Table A.4
TLIB message variables

What you'll see in the manual	What you'll see on your screen
<i>filename</i>	A file name (with or without extension)
<i>function</i>	A function name
<i>len</i>	An actual number
<i>module</i>	A module name
<i>num</i>	An actual number

Table A.4: TLIB message variables (continued)

<i>path</i>	A path name
<i>reason</i>	Reason given in warning message
<i>size</i>	An actual number
<i>type</i>	A type name

TLINK messages

The linker has three types of messages: fatal errors, errors, and warnings.

- A fatal error causes TLINK to stop immediately; the .EXE file is deleted.
- An error (also called a nonfatal error) does not delete .EXE or .MAP files, but you shouldn't try to execute the .EXE file. Errors are treated as fatal errors in the IDE.
- Warnings are just that: warnings of conditions that you probably want to fix. When warnings occur, .EXE and .MAP files are still created.

The following generic names and values appear in the error messages listed in this section. When you get an error message, the appropriate name or value is substituted.

Table A.5
TLINK error message variables

What you'll see in the manual	What you'll see on your screen
<i>errorcode</i>	Error code number for internal errors
<i>filename</i>	A file name (with or without extension)
<i>group</i>	A group name
<i>linenum</i>	The line number within a file
<i>module</i>	A module name
<i>segment</i>	A segment name
<i>symbol</i>	A symbol name
XXXXh	A 4-digit hexadecimal number, followed by <i>h</i>

Message explanations

- MAKE fatal error* **)' missing in macro invocation in command *command***
 A left parenthesis is required to invoke a macro.
- Compile-time error* **(expected**
 A left parenthesis was expected before a parameter list.

- Compile-time error* **) expected**
A right parenthesis was expected at the end of a parameter list.
- Compile-time error* **, expected**
A comma was expected in a list of declarations, initializations, or parameters.
- Compile-time error* **: expected after private/protected/public**
When used to begin a **private/protected/public** section of a C++ class, these reserved words must be followed by a colon.
- Compile-time error* **< expected**
The keyword **template** was not followed by a left angle bracket (<). Every template declaration must include the template formal parameters enclosed within angle brackets (< >), immediately following the **template** keyword.
- TLIB error* **@ seen, expected a response-files name**
The response file is not given immediately after @.
- Compile-time error* **{ expected**
A left brace ({) was expected at the start of a block or initialization.
- Compile-time error* **} expected**
A right brace (}) was expected at the end of a block or initialization.
- TLINK fatal error* **32-bit record encountered**
An object file that contains 80386 32-bit records was encountered, and the **/3** option had not been used.
- Compile-time error* **286/287 instructions not enabled**
Use the **-2** command-line compiler option or the 80286 options from the Options | Compiler | Code Generation | Advanced Code Generation dialog box to enable 286/287 opcodes. Be aware that the resulting code cannot be run on 8086- and 8088-based machines.
- DPMIINST error* **A20 line already enabled, so test is meaningless**
DPMIINST generates this message while you are running it to locate and add information about your machine to the kernel's database. If you encounter a *series* of these messages, boot clean (that is, with a plain generic CONFIG.SYS and AUTOEXEC.BAT) and try again. See the message **Machine not in database (run DPMIINST)** on page 198.

- Run-time error* **Abnormal program termination**
The program called **abort** because there was not enough memory to execute. Can happen through memory overwrites.
- Compile-time error* **Access can only be changed to public or protected**
A C++ derived class may modify the access rights of a base class member, but only to **public** or **protected**. A base class member cannot be made **private**.
- TLIB warning* **added file *filename* does not begin correctly, ignored**
The librarian has decided that in no way, shape, or form is the file being added an object module, so it will not try to add it to the library. The library is created anyway.
- Compile-time error* **Address of overloaded function *function* doesn't match *type***
A variable or parameter is assigned/initialized with the address of an overloaded function, and the type of the variable/parameter doesn't match any of the overloaded functions with the specified name.
- TLIB warning* ***module* already in LIB, not changed!**
An attempt to use the + action on the library has been made, but there is already a object with the same name in the library. If an update of the module is desired, the action should be +-. The library has not been modified.
- Compile-time error* **Ambiguity between *function1* and *function2***
Both of the named overloaded functions could be used with the supplied parameters. This ambiguity is not allowed.
- Compile-time error* **Ambiguous member name *name***
A structure member name used in inline assembly must be unique. If it is defined in more than one structure all of the definitions must agree in type and offset within the structures. The member name in this case is ambiguous. Use the syntax `(struct xxx).yyy` instead.
- Compile-time warning* **Ambiguous operators need parentheses**
This warning is displayed whenever two shift, relational, or bitwise-Boolean operators are used together without parentheses. Also, an addition or subtraction operator that appears unparenthesized with a shift operator will produce this warning. Programmers frequently confuse the precedence of these operators.

- Command line fatal error* **Application load & execute error 0001**
Application load & execute error FFE0
 There was insufficient extended memory available for the protected mode command line tool to load.
- Compile-time error* **Array allocated using new may not have an initializer**
 When initializing a vector (array) of classes, you must use the constructor that has no arguments. This is called the *default constructor*, which means that you may not supply constructor arguments when initializing such a vector.
- Compile-time error* **Array bounds missing]**
 Your source file declared an array in which the array bounds were not terminated by a right bracket.
- Compile-time error* **Array must have at least one element**
 ANSI C and C++ require that an array be defined to have at least one element (objects of zero size are not allowed). An old programming trick declares an array element of a structure to have zero size, then allocates the space actually needed with **malloc**. You can still use this trick, but you must declare the array element to have (at least) one element if you are compiling in strict ANSI mode. Declarations (as opposed to definitions) of arrays of unknown size are still allowed, of course.
- For example,
- ```
char ray[]; /* definition of unknown size -- illegal */
char ray[0]; /* definition of 0 size -- illegal */
extern char ray[]; /* declaration of unknown size -- ok */
```
- Compile-time error* **Array of references is not allowed**  
 It is illegal to have an array of references, since pointers to references are not allowed and array names are coerced into pointers.
- Compile-time warning* **Array size for 'delete' ignored**  
 With the latest specification of C++, it is no longer necessary to specify the array size when deleting an array; to allow older code to compile, the compiler ignores this construct, and issues this warning.
- Compile-time error* **Array size too large**  
 The declared array is larger than 64K.

- Compile-time warning* **Array variable *identifier* is near**  
Whenever you use either the **-Ff** or **-Fm** command-line or the IDE Options | Compiler | Advanced Code Generation... | Far Data Threshold selection to set threshold limit, global variables larger than the threshold size are automatically made far by the compiler. However, when the variable is an initialized array with an unspecified size, its total size is not known when the decision whether to make it near or far has to be made by the compiler, and so it is made near. If the number of initializers given for the array causes the total variable size to exceed the data size threshold, the compiler issues this warning. If the fact that the variable is made near by the compiler causes problems (for example, the linker reports a group overflow due to too much global data), you must make the offending variable explicitly far by inserting the keyword **far** immediately to the left of the variable name in its definition.
- Compile-time error* **Assembler statement too long**  
Inline assembly statements may not be longer than 480 bytes.
- Compile-time warning* **Assigning *type* to *enumeration***  
Assigning an integer value to an **enum** type. This is an error in C++, but is reduced to a warning to give existing programs a chance to work.
- Compile-time error* **Assignment to this not allowed, use *X::operator new* instead**  
In early versions of C++, the only way to control allocation of class of objects was by assigning to the **this** parameter inside a constructor. This practice is no longer allowed, since a better, safer, and more general technique is to define a member function **operator new** instead.
- TLINK warning* **Attempt to export non-public symbol *symbol***  
A symbol name was listed in the EXPORTS section of the module definition file, but no symbol of this name was found as public in the modules linked. This either implies a mistake in spelling or case, or that a procedure of this name was not defined.
- Compile-time error* **Attempt to grant or reduce access to *identifier***  
A C++ derived class can modify the access rights of a base class member, but only by restoring it to the rights in the base class. It cannot add or reduce access rights.

- Compile-time error* **Attempting to return a reference to a local object**  
 In a function returning a reference type, you attempted to return a reference to a temporary object (perhaps the result of a constructor or a function call). Since this object will disappear when the function returns, the reference will then be illegal.
- Compile-time error* **Attempting to return a reference to local variable *identifier***  
 This C++ function returns a reference type, and you are trying to return a reference to a local (auto) variable. This is illegal, since the variable referred to disappears when the function exits. You may return a reference to any static or global variable, or you may change the function to return a value instead.
- TLINK error* **Automatic data segment exceeds 64K**  
 The sum of the DGROUP physical segment, local heap, and stack exceeded 64K. Either specify smaller values for the HEAPSIZE and STACKSIZE statements in the module definition file, or decrease the size of your near data in DGROUP. The map file will show the sizes of the component segments in DGROUP. The */s* TLINK command-line option is useful to help you find out how much each module contributes to DGROUP.
- Compile-time fatal error* **Bad call of intrinsic function**  
 You have used an intrinsic function without supplying a prototype, or you supplied a prototype for an intrinsic function that was not what the compiler expected.
- TLINK fatal error* **Bad character in parameters**  
 One of the following characters was encountered in the command line or in a response file:  
 “ \* < = > ? [ ] |  
 or any control character other than horizontal tab, line feed, carriage return, or *Ctrl-Z*.
- Compile-time error* **Bad define directive syntax**  
 A macro definition starts or ends with the ## operator, or contains the # operator that is not followed by a macro argument name.
- DPMI server fatal error* **bad environment params**  
 The value for the environmental variable DPMIMEM had incorrect syntax.

- Compile-time error* **Bad file name format in include directive**  
Include file names must be surrounded by quotes ("FILENAME.H") or angle brackets (<FILENAME.H>). The file name was missing the opening quote or angle bracket. If a macro was used, the resulting expansion text is incorrect; that is, not surrounded by quote marks.
- MAKE error* **Bad file name format in include statement**  
Include file names must be surrounded by quotes or angle brackets. The file name was missing the opening quote or angle bracket.
- Compile-time error* **Bad file name format in line directive**  
Line directive file names must be surrounded by quotes ("FILENAME.H") or angle brackets (<FILENAME.H>). The file name was missing the opening quote or angle bracket. If a macro was used, the resulting expansion text is incorrect; that is, not surrounded by quote marks.
- TLIB warning* **bad GCD type in GRPDEF, extended dictionary aborted**  
**bad GRPDEF type encountered, extended dictionary aborted**  
The librarian has encountered an invalid entry in a group definition (GRPDEF) record in an object module while creating an extended dictionary. The only type of GRPDEF record that the librarian (and linker) supports are segment index type. If any other type of GRPDEF is encountered, the librarian won't be able to create an extended dictionary. It's possible that an object module created by products other than Borland tools may create GRPDEF records of other types. It's also possible for a corrupt object module to generate this warning.
- TLIB error* **Bad header in input LIB**  
When adding object modules to an existing library, the librarian has determined that it has a bad library header. Rebuild the library.
- Compile-time error* **Bad ifdef directive syntax**  
An **#ifdef** directive must contain a single identifier (and nothing else) as the body of the directive.
- MAKE error* **Bad macro output translator**  
Invalid syntax for substitution within macros. For example:  
`$(MODEL:=s) or $(MODEL:) or $(MODEL:s)`

- TLINK fatal error* **Bad object file record in library file *filename* in module *module* near module file offset 0xxxxxxx**  
**Bad object file record in module *filename* near module file offset 0xxxxxxx**  
An ill-formed object file was encountered. This is most commonly caused by naming a source file or by naming an object file that was not completely built. This can occur if the machine was rebooted during a compile, or if a compiler did not delete its output object file when a *Ctrl-Brk* was pressed.
- TLIB error* **bad OMF record type *type* encountered in module *module***  
The librarian encountered a bad Object Module Format (OMF) record while reading through the object module. The librarian has already read and verified the header records on the *module*, so this usually indicates that the object module has become corrupt in some way and should be recreated.
- Compile-time error* **Bad syntax for pure function definition**  
Pure virtual functions are specified by appending “= 0” to the declaration. You wrote something similar, but not quite the same.
- Compile-time error* **Bad undef directive syntax**  
An **#undef** directive must contain a single identifier (and nothing else) as the body of the directive.
- MAKE error* **Bad undef statement syntax**  
An **!undef** statement must contain a single identifier and nothing else as the body of the statement.
- TLINK fatal error* **Bad version number in parameter block**  
This error indicates an internal inconsistency in the IDE. If it occurs, exit and restart the IDE. This error will not occur in the standalone version.
- Compile-time error* **Base class *class* contains dynamically dispatchable functions**  
Currently, dynamically dispatched virtual tables do not support the use of multiple inheritance. This error occurs because a class which contains DDVT function attempted to inherit DDVT functions from multiple parent classes.
- Compile-time warning* **Base class *class* is inaccessible because also in *class***  
It is not legal to use a class as both a direct and indirect base class, since the members are automatically ambiguous. Try making the base class virtual in both locations.

- Compile-time error* **Base class *class* is included more than once**  
A C++ class may be derived from any number of base classes, but may be directly derived from a given class only once.
- Compile-time error* **Base class *class* is initialized more than once**  
In a C++ class constructor, the list of initializations following the constructor header includes base class *class* more than once.
- Compile-time error* **Base initialization without a class name is now obsolete**  
Early versions of C++ provided for initialization of a base class by following the constructor header with just the base class constructor parameter list. It is now recommended to include the base class name.
- This makes the code much clearer, and is required when there are multiple base classes.
- Old way:
- ```
derived::derived(int i) : (i,10) { ... }
```
- New way:
- ```
derived::derived(int i) : base(i, 10) { ... }
```
- Compile-time error* **Bit field cannot be static**  
Only ordinary C++ class data members can be declared **static**, not bit fields.
- Compile-time error* **Bit field too large**  
This error occurs when you supply a bit field with more than 16 bits.
- Compile-time error* **Bit fields must be signed or unsigned int**  
In ANSI C, bit fields may only be signed or unsigned **int** (not **char** or **long**, for example).
- Compile-time warning* **Bit fields must be signed or unsigned int**  
In ANSI C, bit fields may not be of type signed char or unsigned char; when not compiling in strict ANSI mode, though, the compiler will allow such constructs, but flag them with this warning.
- Compile-time error* **Bit fields must contain at least one bit**  
You cannot declare a named bit field to have 0 (or less than 0) bits. You can declare an unnamed bit field to have 0 bits, a convention used to force alignment of the following bit field to a byte boundary (or word boundary, if you select the **-a** alignment option or IDE Options | Compiler | Code

Generation | Word Alignment). In C++, bit fields must have an integral type; this includes enumerations.

*Compile-time error* **Bit fields must have integral type**

In C++, bit fields must have an integral type; this includes enumerations.

*Compile-time error* **Body has already been defined for function *function***

A function with this name and type was previously supplied a function body. A function body can only be supplied once.

*Compile-time warning* **Both return and return with a value used**

The current function has **return** statements with and without values. This is legal in C, but almost always an error. Possibly a **return** statement was omitted from the end of the function.

*Compile-time error* **Call of nonfunction**

The name being called is not declared as a function. This is commonly caused by incorrectly declaring the function or misspelling the function name.

*Compile-time warning* **Call to function *function* with no prototype**

The "Prototypes required" warning was enabled and you called function *function* without first giving a prototype for that function.

#### CHAR 34

*Compile-time error* **Cannot add or subtract relocatable symbols**

The only arithmetic operation that can be performed on a relocatable symbol in an assembler operand is addition or subtraction of a constant. Variables, procedures, functions, and labels are relocatable symbols. Assuming that *Var* is a variable and *Const* is a constant, then the instructions

```
MOV AX,Const+Const
```

and

```
MOV AX,Var+Const
```

are valid, but `MOV AX,Var+Var` is not.

*Compile-time error* **Cannot allocate a reference**

An attempt to create a reference using the **new** operator has been made; this is illegal, as references are not objects and cannot be created through **new**.



- Compile-time error* **identifier cannot be declared in an anonymous union**  
The compiler found a declaration for a member function or static member in an anonymous union. Such unions can only contain data members.
- Compile-time error* **function1 cannot be distinguished from function2**  
The parameter type lists in the declarations of these two functions do not differ enough to tell them apart. Try changing the order of parameters or the type of a parameter in one declaration.
- Compile-time error* **Cannot call near class member function with a pointer of type type**  
Member functions of near classes (remember that classes are near by default in the tiny, small, and medium memory models) cannot be called using far or huge member pointers. (Note that this also applies to calls using pointers to members.) Either change the pointer to be near, or declare the class as far.
- Compile-time error* **Cannot cast from type1 to type2**  
A cast from type *type1* to type *type2* is not allowed. In C, a pointer may be cast to an integral type or to another pointer. An integral type may be cast to any integral, floating, or pointer type. A floating type may be cast to an integral or floating type. Structures and arrays may not be cast to or from. You cannot cast from a **void** type.
- C++ checks for user-defined conversions and constructors, and if one cannot be found, then the preceding rules apply (except for pointers to class members). Among integral types, only a constant zero may be cast to a member pointer. A member pointer may be cast to an integral type or to a similar member pointer. A similar member pointer points to a data member if the original does, or to a function member if the original does; the qualifying class of the type being cast to must be the same as or a base class of the original.
- Compile-time error* **Cannot convert type1 to type2**  
An assignment, initialization, or expression requires the specified type conversion to be performed, but the conversion is not legal.
- Compile-time error* **Cannot create instance of abstract class class**  
Abstract classes—those with pure virtual functions—cannot be used directly, only derived from.


- Compile-time error*    **Cannot define a pointer or reference to a reference**  
It is illegal to have a pointer to a reference or a reference to a reference.
- Compile-time error*    **Cannot find `class::class (class &)` to copy a vector**  
When a C++ class `class1` contains a vector (array) of class `class2`, and you want to construct an object of type `class1` from another object of type `class1`, there must be a constructor **`class2::class2(class2&)`** so that the elements of the vector can be constructed. This constructor takes just one parameter (which is a reference to its class) and is called a *copy constructor*.  
  
Usually the compiler supplies a copy constructor automatically. However, if you have defined a constructor for class `class2` that has a parameter of type `class2&` and has additional parameters with default values, the copy constructor cannot be created by the compiler. (This is because `class2::class2(class2&)` and `class2::class2(class2&, int = 1)` cannot be distinguished.) You must redefine this constructor so that not all parameters have default values. You can then define a copy constructor or let the compiler create one.
- Compile-time error*    **Cannot find `class::operator=(class&)` to copy a vector**  
When a C++ class `class1` contains a vector (array) of class `class2`, and you wish to copy a class of type `class1`, there must be an assignment operator **`class2::operator=(class2&)`** so that the elements of the vector can be copied. Usually the compiler supplies such an operator automatically. However, if you have defined an **`operator=`** for class `class2`, but not one that takes a parameter of type `class2&`, the compiler will not supply it automatically—you must supply one.
- Compile-time error*    **Cannot find default constructor to initialize array element of type `class`**  
When declaring an array of a class that has constructors, you must either explicitly initialize every element of the array, or the class must have a default constructor (it will be used to initialize the array elements that don't have explicit initializers). The compiler will define a default constructor for a class unless you have defined any constructors for the class.
- Compile-time error*    **Cannot find default constructor to initialize base class `class`**  
Whenever a C++ derived class `class2` is constructed, each base class `class1` must first be constructed. If the constructor for `class2` does not specify a constructor for `class1` (as part of `class2`'s header), there must be a constructor **`class1::class1()`** for the

base class. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class *class1*; in that case, the compiler will not supply the default constructor automatically—you must supply one.

- Compile-time error* **Cannot find default constructor to initialize member identifier**  
When a C++ class *class1* contains a member of class *class2*, and you wish to construct an object of type *class1* but not from another object of type *class1*, there must be a constructor **class2::class2()** so that the member can be constructed. This constructor without parameters is called the default constructor. The compiler will supply a default constructor automatically unless you have defined any constructor for class *class2*; in that case, the compiler will not supply the default constructor automatically—you must supply one.
- TLINK fatal error* **Cannot generate COM file: data below initial CS:IP defined**  
This error results from trying to generate data or code below the starting address (usually 100) of a .COM file. Be sure that the starting address is set to 100 by using the (ORG 100H) instruction. This error message should not occur for programs written in a high-level language. If it does, ensure that the correct startup (COx) object module is being linked in.
- TLINK fatal error* **Cannot generate COM file: invalid initial entry point address**  
You used the */Tdc* or */t* option, but the program starting address is not equal to 100H, which is required with .COM files.
- TLINK fatal error* **Cannot generate COM file: program exceeds 64K**  
You used the */Tdc* or */t* option, but the total program size exceeds the .COM file limit.
- TLINK fatal error* **Cannot generate COM file: segment-relocatable items present**  
You used the */Tdc* or */t* option, but the program contains segment-relative fixups, which are not allowed with .COM files.
- TLINK fatal error* **Cannot generate COM file: stack segment present**  
You used the */Tdc* or */t* option, but the program declares a stack segment, which is not allowed with .COM files.

- Compile-time error* **Cannot generate function from template function template**  
A call to a template function was found, but a matching template function cannot be generated from the function template.
- Compile-time error* **Cannot have a near class member in a far class**  
All members of a C++ **far** class must be far. This member is in a class that was declared (or defaults to) **near**.
- Compile-time error* **Cannot have a non-inline function in a local class**  
**Cannot have a static data member in a local class**  
All members of classes declared local to a function must be entirely defined in the class definition. This means that such local classes may not contain any static data members, and all of their member functions must have bodies defined within the class definition.
- MAKE error* **Cannot have multiple paths for implicit rule**  
You can only have a single path for each of the extensions in an implicit rule. Multiple path lists are only allowed for dependents in an explicit rule. For example:
- ```
{path1;path2}.c.obj:    # Invalid
{path}.c.obj           # Valid
```
- MAKE error* **Cannot have path list for target**
You can only specify a path list for dependents of an explicit rule. For example:
- ```
{path1;path2}prog.exe: prog.obj # Invalid
prog.exe: {path1;path2}prog.obj # Valid
```
- Compile-time error* **Cannot initialize a class member here**  
Individual members of **structs**, **unions**, and C++ **classes** may not have initializers. A **struct** or **union** may be initialized as a whole using initializers inside braces. A C++ **class** may only be initialized by the use of a constructor.
- Compile-time error* **Cannot initialize type1 with type2**  
You are attempting to initialize an object of type *type1* with a value of type *type2*, which is not allowed. The rules for initialization are essentially the same as for assignment.
- Compile-time error* **Cannot modify a const object**  
This indicates an illegal operation on an object declared to be **const**, such as an assignment to the object.



- Compile-time error*    **Cannot overload 'main'**  
**main** is the only function which cannot be overloaded.
- Compile-time error*    **function cannot return a value**  
 A function with a return type **void** contains a **return** statement that returns a value; for example, an **int**.
- Compile-time error*    **identifier cannot start an argument declaration**  
 Undefined *identifier* found at the start of an argument in a function declarator. Often the type name is misspelled or the type declaration is missing (usually caused by not including the appropriate header file).
- Compile-time error*    **Cannot use tiny or huge memory model with Windows**  
 This message is self-explanatory. Use small, medium, compact, or large instead.
- TLIB error*    **cannot write GRPDEF list, extended dictionary aborted**  
 The librarian cannot write the extended dictionary to the tail end of the library file. This usually indicates lack of space on the disk.
- TLIB error*    **can't grow LE/LIDATA record buffer**  
 Command-line error. See **out of memory reading LE/LIDATA record from object module**.
- Compile-time error*    **Case bypasses initialization of a local variable**  
 In C++ it is illegal to bypass the initialization of a local variable in any way. In this case, there is a case label which can transfer control past this local variable.
- Compile-time error*    **Case outside of switch**  
 The compiler encountered a **case** statement outside a **switch** statement. This is often caused by mismatched braces.
- Compile-time error*    **Case statement missing :**  
 A **case** statement must have a constant expression followed by a colon. The expression in the **case** statement either is missing a colon or has an extra symbol before the colon.
- MAKE or compile-time error*    **Character constant must be one or two characters long**  
 Character constants can be only one or two characters long.
- MAKE fatal error*    **Circular dependency exists in makefile**  
 The makefile indicates that a file needs to be up-to-date BEFORE it can be built. Take, for example, the explicit rules:  
 filea: fileb

fileb: filec  
filec: filea

This implies that filea depends on fileb, which depends on filec, and filec depends on filea. This is illegal, since a file cannot depend on itself, indirectly or directly.

*Compile-time error* **Class *class* may not contain pure functions**

The class being declared cannot be abstract, and therefore it may not contain any pure functions.

*Compile-time error* **Class member *member* declared outside its class**

C++ class member functions can be declared only inside the class declaration. Unlike nonmember functions, they cannot be declared multiple times or at other locations.

*Compile-time warning* **Code has no effect**

The compiler encountered a statement with operators that have no effect. For example the statement

```
a + b;
```

has no effect on either variable. The operation is unnecessary and probably indicates a bug in your file.

*MAKE error* **Colon expected**

You have forgotten to put the colon at the end of your implicit rule.

```
.c.obj: # Correct
.c.obj # Incorrect
```

*MAKE error* **Command arguments too long**

The arguments to a command were more than the 127-character limit imposed by DOS.

*MAKE error* **Command syntax error**

This message occurs if

- The first rule line of the makefile contained any leading whitespace.
- An implicit rule did not consist of *.ext.ext:*.
- An explicit rule did not contain a name before the *:* character.
- A macro definition did not contain a name before the *=* character.

*MAKE error* **Command too long**

The length of a command has exceeded 128 characters. You might want to use a response file.

- TLINK error* **Common segment exceeds 64K**  
The program had more than 64K of near uninitialized data. Try declaring some uninitialized data as far.
- Compile-time error* **Compiler could not generate copy constructor for class *class***  
The compiler cannot generate a needed copy constructor due to language rules.
- Compile-time error* **Compiler could not generate default constructor for class *class***  
The compiler cannot generate a needed default constructor due to language rules.
- Compile-time error* **Compiler could not generate operator= for class *class***  
The compiler cannot generate a needed assignment operator due to language rules.
- Compile-time fatal error* **Compiler table limit exceeded**  
One of the compiler's internal tables overflowed. This usually means that the module being compiled contains too many function bodies. Making more memory available to the compiler will not help with such a limitation; simplifying the file being compiled is usually the only remedy.
- Compile-time error* **Compound statement missing }**  
The compiler reached the end of the source file and found no closing brace. This is often caused by mismatched braces.
- Compile-time warning* **Condition is always false**  
**Condition is always true**  
The compiler encountered a comparison of values where the result is always true or false. For example:
- ```
void proc(unsigned x)
{
    if (x >= 0)      /* always 'true' */
    {
        :
    }
}
```
- Compile-time error* **Conflicting type modifiers**
This occurs when a declaration is given that includes, for example, both **near** and **far** keywords on the same pointer. Only one addressing modifier may be given for a single pointer, and only one language modifier (**cdecl**, **pascal**, or **interrupt**) may be given for a function.

- TLINK warning* **symbol conflicts with module *module* in module *module***
This indicates an inconsistency in the definition of *symbol*; TLINK found one virtual function and one common definition with the same name.
- Compile-time error* **Constant expression required**
Arrays must be declared with constant size. This error is commonly caused by misspelling a **#define** constant.
- Compile-time warning* **Constant is long**
The compiler encountered either a decimal constant greater than 32767 or an octal (or hexadecimal) constant greater than 65535 without a letter *l* or *L* following it. The constant is treated as a **long**.
- Compile-time error* **Constant member *member* in class without constructors**
A class that contains constant members must have at least one user-defined constructor; otherwise, there would be no way to ever initialize such members.
- Compile-time warning* **Constant member *member* is not initialized**
This C++ class contains a constant member *member*, which does not have an initialization. Note that constant members may be initialized only, not assigned to.
- Compile-time warning* **Constant out of range in comparison**
Your source file includes a comparison involving a constant sub-expression that was outside the range allowed by the other sub-expression's type. For example, comparing an **unsigned** quantity to -1 makes no sense. To get an **unsigned** constant greater than 32767 (in decimal), you should either cast the constant to **unsigned** (for example, **(unsigned)65535**) or append a letter *u* or *U* to the constant (for example, **65535u**).
- Whenever this message is issued, the compiler will still generate code to do the comparison. If this code ends up always giving the same result, such as comparing a **char** expression to 4000, the code will still perform the test.
- Compile-time error* **Constant variable *variable* must be initialized**
This C++ object is declared **const**, but is not initialized. Since no value may be assigned to it, it must be initialized at the point of declaration.
- Compile-time error* **constructor cannot be declared **const** or **volatile****
A constructor has been declared as **const** and/or **volatile**, and this is not allowed.

- Compile-time error* **constructor cannot have a return type specification**
C++ constructors have an implicit return type used by the compiler, but you cannot declare a return type or return a value.
- Compile-time warning* **Conversion may lose significant digits**
For an assignment operator or some other circumstance, your source file requires a conversion from **long** or **unsigned long** to **int** or **unsigned int** type. Since **int** type and **long** type variables don't have the same size, this kind of conversion may alter the behavior of a program.
- Compile-time error* **Conversion of near pointer not allowed**
This message used only by IDE debugger.
A near pointer cannot be converted to a far pointer in the expression evaluation box when a program is not currently running. This is because the conversion needs the current value of DS in the user program, which doesn't exist.
- Compile-time error* **Conversion operator cannot have a return type specification**
This C++ type conversion member function specifies a return type different from the type itself. A declaration for conversion function **operator** may not specify any return type.
- Compile-time error* **Conversion to *type* will fail for members of virtual base *class***
This warning is issued in some cases when a member pointer is cast to another member pointer type, if the class of the member pointer contains virtual bases, and only if the **-Vv** option or IDE Options | Compiler | Advanced Compiler | Deep Virtual Bases has been used. It means that if the member pointer being cast happens to point (at the time of the cast) to a member of **class**, the conversion cannot be completed, and the result of the cast will be a NULL member pointer. (See the *User's Guide* for details).
- TLIB error* **could not allocate memory for per module data**
The librarian has run out of memory.
- TLIB error* **could not create list file *filename***
The librarian could not create a list file for the library. This could be due to lack of disk space.
- Compile-time error* **Could not find a match for *argument(s)***
No C++ function could be found with parameters matching the supplied arguments.

<i>Compile-time error</i>	Could not find file <i>filename</i> The compiler is unable to find the file supplied on the command line.
<i>TLIB error</i>	Could not write output. The librarian could not write the output file.
<i>TLIB error</i>	couldn't alloc memory for per module data The librarian has run out of memory.
<i>TLIB warning</i>	<i>filename</i> couldn't be created, original won't be changed An attempt has been made to extract an object ('*' action) but the librarian cannot create the object file to extract the module into. Either the object already exists and is read only, or the disk is full.
<i>TLIB error</i>	couldn't get LE/LIDATA record buffer Command-line error. See out of memory reading LE/LIDATA record from object module.
<i>TLINK warning</i>	Debug info switch ignored for .COM files Borland C++ does not include debug information for .COM files. See the description of the <i>/v</i> option on page 70.
<i>TLINK warning</i>	Debug information in module <i>module</i> will be ignored Object files compiled with debug information now have a version record. The major version of this record is higher than what TLINK currently supports and TLINK did not generate debug information for the module in question.
<i>Compile-time error</i>	Declaration does not specify a tag or an identifier This declaration doesn't declare anything. This may be a struct or union without a tag or a variable in the declaration. C++ requires that something be declared.
<i>Compile-time error</i>	Declaration is not allowed here Declarations cannot be used as the control statement for while , for , do , if , or switch statements.
<i>Compile-time error</i>	Declaration missing ; Your source file contained a declaration that was not followed by a semicolon.
<i>Compile-time error</i>	Declaration syntax error Your source file contained a declaration that was missing some symbol or had some extra symbol added to it.



- Compile-time error* **Declaration terminated incorrectly**
A declaration has an extra or incorrect termination symbol, such as a semicolon placed after a function body. A C++ member function declared in a class with a semicolon between the header and the opening left brace also generates this error.
- Compile-time error* **Declaration was expected**
A declaration was expected here but not found. This is usually caused by a missing delimiter such as a comma, semicolon, right parenthesis, or right brace.
- Compile-time error* **Declare operator delete (void*) or (void*, size_t)**
Declare the operator **delete** with a single **void*** parameter, or with a second parameter of type **size_t**. If you use the second version, it will be used in preference to the first version. The global operator **delete** can only be declared using the single-parameter form.
- Compile-time warning* **Declare type *type* prior to use in prototype**
When a function prototype refers to a structure type that has not previously been declared, the declaration inside the prototype is not the same as a declaration outside the prototype. For example,
- ```
int func(struct s *ps);
struct s { /* ... */};
```
- Since there is no struct *s* in scope at the prototype for **func**, the type of parameter *ps* is pointer to undefined struct *s*, and is not the same as the struct *s* which is later declared. This will result in later warning and error messages about incompatible types, which would be very mysterious without this warning message. To fix the problem, you can move the declaration for struct *s* ahead of any prototype which references it, or add the incomplete type declaration **struct s**; ahead of any prototype which references struct *s*. If the function parameter is a **struct**, rather than a pointer to **struct**, the incomplete declaration is not sufficient; you must then place the struct declaration ahead of the prototype.
- Compile-time warning* ***identifier* is declared but never used**  
Your source file declared the named variable as part of the block just ending, but the variable was never used. The warning is indicated when the compiler encounters the closing brace of the compound statement or function. The declaration

of the variable occurs at the beginning of the compound statement or function.

*Compile-time error* **Default argument value redeclared for parameter *parameter***  
When a parameter of a C++ function is declared to have a default value, this value cannot be changed, redeclared, or omitted in any other declaration for the same function.

*Compile-time error* **Default expression may not use local variables**  
A default argument expression is not allowed to use any local variables or other parameters.

*Compile-time error* **Default outside of switch**  
The compiler encountered a **default** statement outside a **switch** statement. This is most commonly caused by mismatched braces.

*Compile-time error* **Default value missing**  
When a C++ function declares a parameter with a default value, all of the following parameters must also have default values. In this declaration, a parameter with a default value was followed by a parameter without a default value.

*Compile-time error* **Default value missing following parameter *parameter***  
All parameters following the first parameter with a default value must also have defaults specified.

*Compile-time error* **Define directive needs an identifier**  
The first non-whitespace character after a **#define** must be an identifier. The compiler found some other character.

*TLINK error or warning* ***symbol* defined in module *module* is duplicated in module *module***  
There is a conflict between two symbols (either public or communal). This usually means that a symbol is defined in two modules. An error occurs if both are encountered in the .OBJ file(s), because TLINK doesn't know which is valid. A warning results if TLINK finds one of the duplicated symbols in a library and finds the other in an .OBJ file; in this case, TLINK uses the one in the .OBJ file.

*Compile-time error* **Delete array size missing ]**  
The array specifier in an operator date is missing a right bracket.

*Compile-time error* **Destructor cannot be declared const or volatile**  
A destructor has been declared as **const** and/or **volatile**, and this is not allowed.

- Compile-time error*    **Destructor cannot have a return type specification**  
It is illegal to specify the return type for a destructor.
- Compile-time error*    **Destructor for *class* is not accessible**  
The destructor for this C++ class is **protected** or **private**, and cannot be accessed here to destroy the class. If a class destructor is **private**, the class cannot be destroyed, and thus can never be used. This is probably an error. A **protected** destructor can be accessed only from derived classes. This is a useful way to ensure that no instance of a base class is ever created, but only classes derived from it.
- Compile-time error*    **Destructor for *class* required in conditional expression**  
If the compiler must create a temporary local variable in a conditional expression, it has no good place to call the destructor, since the variable may or may not have been initialized. The temporary variable can be explicitly created, as with classname (val, val), or implicitly created by some other code. Recast your code to eliminate this temporary value.
- Compile-time error*    **Destructor name must match the class name**  
In a C++ class, the tilde (~) introduces a declaration for the class destructor. The name of the destructor must be the same as the class name. In your source file, the tilde (~) preceded some other name.
- Run-time error*    **Divide error**  
You've tried to divide an integer by zero. For example,
- ```
int n = 0;  
n = 2 / n;
```
- You can trap this error with the signal function. Otherwise, Borland C++ calls **abort** and your program terminates.
- Compile-time error* **Division by zero**
Your source file contained a division or remainder operator in a constant expression with a zero divisor.
- Compile-time warning* **Division by zero**
A division or remainder operator expression had a literal zero as a divisor.
- MAKE error* **Division by zero**
A division or remainder operator in an **!if** statement has a zero divisor.

- Compile-time error* **do statement must have while**
Your source file contained a **do** statement that was missing the closing **while** keyword.
- MAKE fatal error* **filename does not exist – don't know how to make it**
There's a nonexistent file name in the build sequence, and no rule exists that would allow the file name to be built.
- TLINK fatal error* **DOS error, ax = number**
This occurs if a DOS call returned an unexpected error. The *ax* value printed is the resulting error code. This could indicate a TLINK internal error or a DOS error. The only DOS calls TLINK makes where this error could occur are read, write, seek, and close.
- Compile-time error* **do-while statement missing (**
In a **do** statement, the compiler found no left parenthesis after the **while** keyword.
- Compile-time error* **do-while statement missing)**
In a **do** statement, the compiler found no right parenthesis after the test expression.
- Compile-time error* **do-while statement missing ;**
In a **do** statement test expression, the compiler found no semi-colon after the right parenthesis.
- Compile-time error* **Duplicate case**
Each **case** of a **switch** statement must have a unique constant expression value.
- TLINK warning* **filename (linenum): Duplicate external name in exports**
Two export functions listed in the EXPORTS section of a module definition file defined the same external name. For instance,
EXPORTS
AnyProc=MyProc1
AnyProc=MyProc2
- TLINK warning* **filename (linenum): Duplicate internal name in exports**
Two export functions listed in the EXPORTS section of the module definition file defined the same internal name. For example,
EXPORTS
AnyProc1=MyProc
AnyProc2=MyProc



TLINK warning **filename (linenum): Duplicate internal name in imports**
Two import functions listed in the IMPORTS section of the module definition file defined the same internal name. For instance,

```
IMPORTS
    AnyProc=MyMod1.MyProc1
    AnyProc=MyMod2.MyProc2
```

or

```
IMPORTS
    MyMod1.MyProc
    MyMod2.MyProc]
```

TLINK warning **Duplicate ordinal number in exports**
This warning occurs when TLINK encounters two exports with the same ordinal value. First check the module definition file to ensure that there are no duplicate ordinal values specified in the EXPORTS section. If not, then you are linking with modules which specify exports by ordinals and one of two things happened: either two export records specify the same ordinal, or the exports section in the module definition file duplicates an ordinal in an export record.

Export records (EXPDEF) are comment records found in object files and libraries which specify that particular variables are to be exported. Optionally, these records can specify ordinal values when exporting by ordinal (rather than by name).

Compile-time error **Enum syntax error**
An **enum** declaration did not contain a properly formed list of identifiers.

TLIB error **error changing file buffer size**
TLIB is attempting to adjust the size of a buffer used while reading or writing a file but there is not enough memory. It is likely that quite a bit of system memory will have to be freed up to resolve this error.

Compile-time fatal error **Error directive: message**
The text of the **#error** directive being processed in the source file is displayed.

MAKE fatal error **Error directive: message**
MAKE has processed an **#error** directive in the source file, and the text of the directive is displayed in the message.



TLIB error **error opening filename**
TLIB cannot open the specified file for some reason.

TLIB error **error opening filename for output**
TLIB cannot open the specified file for output. This is usually due to lack of disk space for the target library, or a listing file. Additionally this error will occur when the target file exists but is marked as a read only file.

TLIB error **error renaming filename to filename**
TLIB builds a library into a temporary file and then renames the temporary file to the target library file name. If there is an error, usually due to lack of disk space, this message will be posted.

Compile-time fatal error **Error writing output file**
A DOS error that prevents Borland C++ from writing an .OBJ, .EXE, or temporary file. Check the **-n** or Options | Directories | Output directory and make sure that this is a valid directory. Also check that there is enough free disk space.

Compile-time error **Expression expected**
An expression was expected here, but the current symbol cannot begin an expression. This message may occur where the controlling expression of an **if** or **while** clause is expected or where a variable is being initialized. It is often due to an accidentally inserted or deleted symbol in the source code.

Compile-time error **Expression of scalar type expected**
The not (!), increment (++), and decrement (—) operators require an expression of scalar type—only types **char**, **short**, **int**, **long**, **enum**, **float**, **double**, **long double**, and pointer types are allowed.

Compile-time error **Expression syntax**
This is a catchall error message when the compiler parses an expression and encounters some serious error. This is most commonly caused by two consecutive operators, mismatched or missing parentheses, or a missing semicolon on the previous statement.

MAKE error **Expression syntax error in !if statement**
The expression in an **!if** statement is badly formed—it contains a mismatched parenthesis, an extra or missing operator, or a missing or extra constant.

<i>TLIB warning</i>	reason – extended dictionary not created TLIB could not produce the extended dictionary because of the <i>reason</i> given in the warning message.
<i>Compile-time error</i>	extern variable cannot be initialized The storage class extern applied to a variable means that the variable is being declared but not defined here—no storage is being allocated for it. Therefore, you can't initialize the variable as part of the declaration.
<i>Compile-time error</i>	Extra argument in template class name <i>template</i> A template class name specified too many actual values for its formal parameters.
<i>Compile-time error</i>	Extra parameter in call A call to a function, via a pointer defined with a prototype, had too many arguments given.
<i>Compile-time error</i>	Extra parameter in call to <i>function</i> A call to the named function (which was defined with a prototype) had too many arguments given in the call.
<i>Command line fatal error</i>	Failed to locate DPMI server (DPMI16BI.OVL) Failed to locate protected mode loader (DPMILOAD.EXE) Make sure that DPMI16BI.OVL and DPMILOAD.EXE are somewhere on your path or in the same directory as the protected mode command line tool you were attempting to use.
<i>Compile-time error</i>	File must contain at least one external declaration This compilation unit was logically empty, containing no external declarations. ANSI C and C++ require that something be declared in the compilation unit.
<i>Compile-time error</i>	File name too long The file name given in an #include directive was too long for the compiler to process. Path names in DOS must be no more than 79 characters long.
<i>MAKE error</i>	File name too long The path name in an !include directive overflowed MAKE's internal buffer (512 bytes).
<i>TLIB warning</i>	filename file not found The command-line librarian attempted to add a nonexisting object but created the library anyway.

TLIB error **filename file not found**

The IDE creates the library by first removing the existing library and then rebuilding. If any objects do not exist, the library is considered incomplete and thus an error. If the IDE reports that an object does not exist, either the source module has not been compiled or there were errors during compilation. Performing either a Compile | Make or Compile | Build should resolve the problem or indicate where the errors have occurred.

TLINK fatal error **filename (linenum): File read error**

A DOS error occurred while TLINK read the module definition file. This usually means that a premature end of file occurred.

TLINK error **Fixup overflow at segment:xxxxh, target = segment:xxxxh in module module**

Fixup overflow at segment:xxxxh, target = symbol in module module

Either of these messages indicate an incorrect data or code reference in an object file that TLINK must fix up at link time.

The cause is often a mismatch of memory models. A **near** call to a function in a different code segment is the most likely cause. This error can also result if you generate a **near** call to a data variable or a data reference to a function. In either case the symbol named as the *target* in the error message is the referenced variable or function. The reference is in the named module, so look in the source file of that module for the offending reference.

In an assembly language program, a fixup overflow frequently occurs if you have declared an external variable within a segment definition, but this variable actually exists in a different segment.

If this technique does not identify the cause of the failure, or if you are programming in assembly language or a high-level language besides Borland C++, there may be other possible causes for this message. Even in Borland C++, this message could be generated if you are using different segment or group names than the default values for a given memory model.

Run-time error **Floating point error: Divide by 0.**

Floating point error: Domain.

Floating point error: Overflow.

These fatal errors result from a floating-point operation for which the result is not finite.

- “Divide by 0” means the result is +INF or -INF exactly, such as 1.0/0.0.
- “Domain” means the result is NAN (not a number), like 0.0/0.0.
- “Overflow” means the result is +INF (infinity) or -INF with complete loss of precision, such as assigning 1e200*1e200 to a **double**.

Run-time error **Floating point error: Partial loss of precision.**
Floating point error: Underflow.

These exceptions are masked by default, and the error messages do not occur. Underflows are converted to zero and losses of precision are ignored. They can be unmasked by calling **_control87**.

Run-time error **Floating point error: Stack fault.**

The floating-point stack has been overrun. This error does not normally occur and may be due to assembly code using too many registers or due to a misdeclaration of a floating-point function.

These floating-point errors can be avoided by masking the exception so that it doesn’t occur, or by catching the exception with **signal**. See the functions **_control87** and **signal** (in the *Library Reference*) for details.

In each of the above cases, the program prints the error message and then calls **abort**, which prints

Abnormal program termination

and calls **_exit(3)**. See **abort** and **_exit** for more details.

Compile-time error **For statement missing (**

In a **for** statement, the compiler found no left parenthesis after the **for** keyword.

Compile-time error **For statement missing)**

In a **for** statement, the compiler found no right parenthesis after the control expressions.

Compile-time error **For statement missing ;**

In a **for** statement, the compiler found no semicolon after one of the expressions.

Compile-time error **Friends must be functions or classes**


A **friend** of a C++ class must be a function or another class.

- Compile-time error* **Function call missing)**
The function call argument list had some sort of syntax error, such as a missing or mismatched right parenthesis.
- Compile-time error* **Function calls not supported**
This message used only by IDE debugger. In integrated debugger expression evaluation, calls to functions (including implicit conversion functions, constructors, destructors, overloaded operators, and inline functions) are not supported.
- Compile-time error* **Function defined inline after use as extern**
Functions cannot become inline after they have already been used. Either move the inline definition forward in the file or delete it entirely.
- Compile-time error* **Function definition cannot be a Typedef'ed declaration**
In ANSI C a function body cannot be defined using a typedef with a function Type.
- Compile-time error* **Function *function* cannot be static**
Only ordinary member functions and the operators **new** and **delete** can be declared static. Constructors, destructors and other operators must not be static.
- Compile-time error* **Function *function* should have a prototype**
A function was called with no prototype in scope.

In C, `int foo();` is not a prototype, but `int foo(int);` is, and so is `int foo(void);`. In C++, `int foo();` is a prototype, and is the same as `int foo(void);`. In C, prototypes are *recommended* for all functions. In C++, prototypes are *required* for all functions. In all cases, a function definition (a function header with its body) serves as a prototype if it appears before any other mention of the function.
- Compile-time warning* **Function should return a value**
This function was declared (maybe implicitly) to return a value. A **return** statement was found without a return value or the end of the function was reached without a **return** statement being found. Either return a value or declare the function as **void**.
- Compile-time error* **Function should return a value**
Your source file declared the current function to return some type other than **void** in C++ (or **int** in C), but the compiler encountered a return with no value. This is usually some sort of error. In C **int** functions are exempt, since in old versions of




- C there was no **void** type to indicate functions which return nothing.
- Compile-time error* **Functions *function1* and *function2* both use the same dispatch number**
Dynamically dispatched virtual table (DDVT) problem.
- Compile-time warning* **Functions containing local destructors are not expanded inline in function *function***
You've created an inline function for which Borland C++ turns off inlining. You can ignore this warning if you like; the function will be generated out of line.
- Compile-time warning* **Functions containing *reserved word* are not expanded inline**
Functions containing any of the reserved words **do**, **for**, **while**, **goto**, **switch**, **break**, **continue**, and **case** cannot be expanded inline, even when specified as **inline**. The function is still perfectly legal, but will be treated as an ordinary static (not global) function.
- Compile-time error* **Functions may not be part of a struct or union**
This C **struct** or **union** field was declared to be of type function rather than pointer to function. Functions as fields are allowed only in C++.
- TLINK fatal error* **General error**
General error in library file *filename* in module *module* near module file offset 0xyyyyyyyy.
General error in module *module* near module file offset 0xyyyyyyyy
TLINK gives as much information as possible about what processing was happening at the time of the unknown fatal error.
- Compile-time error* **Global anonymous union not static**
In C++, a global anonymous union at the file level must be static.
- Compile-time error* **Goto bypasses initialization of a local variable**
In C++ it is illegal to bypass the initialization of a local variable in any way. In this case, there is a **goto** which can transfer control past this local variable.
- Compile-time error* **Goto statement missing label**
The **goto** keyword must be followed by an identifier.

- TLINK fatal error* **Group *group* exceeds 64K**
A group exceeded 64K bytes when the segments of the group were combined.
- Compile-time error* **Group overflowed maximum size: *group***
The total size of the segments in a group (for example, DGROUP) exceeded 64K.
- TLINK warning* **Group *group1* overlaps group *group2***
 This means that TLINK has encountered nested groups. This warning only occurs when overlays are used or when linking a Windows program.
- Compile-time error* ***specifier* has already been included**
This type specifier occurs more than once in this declaration. Delete or change one of the occurrences.
- Compile-time warning* **Hexadecimal value contains more than 3 digits**
Under older versions of C, a hexadecimal escape sequence could contain no more than three digits. The ANSI standard allows any number of digits to appear as long as the value fits in a byte. This warning results when you have a long hexadecimal escape sequence with many leading zero digits (such as “\x00045”). Older versions of C would interpret such a string differently.
- Compile-time warning* ***function1* hides virtual function *function2***
A virtual function in a base class is usually overridden by a declaration in a derived class. In this case, a declaration with the same name but different argument types makes the virtual functions inaccessible to further derived classes.
- Compile-time error* **Identifier expected**
An identifier was expected here, but not found. In C, this is in a list of parameters in an old-style function header, after the reserved words **struct** or **union** when the braces are not present, and as the name of a member in a structure or union (except for bit fields of width 0). In C++, an identifier is also expected in a list of base classes from which another class is derived, following a double colon (::), and after the reserved word **operator** when no operator symbol is present.
- Compile-time error* **Identifier *identifier* cannot have a type qualifier**
A C++ qualifier *class::identifier* may not be applied here. A qualifier is not allowed on **typedef** names, on function declarations (except definitions at the file level), on local variables or


parameters of functions, or on a class member except to use its own class as a qualifier (redundant but legal).

- Compile-time error* **If statement missing (**
In an **if** statement, the compiler found no left parenthesis after the **if** keyword.
- Compile-time error* **If statement missing)**
In an **if** statement, the compiler found no right parenthesis after the test expression.
- MAKE error* **If statement too long**
ifdef statement too long
ifndef statement too long
An **if**, **ifdef**, or **ifndef** statement has exceeded 4,096 characters.
- TLIB warning* **ignored module, path is too long**
The path to a specified **.obj** or **.lib** file is greater than 64 characters. The max path to a file for TLIB is 64 characters.
- Compile-time error* **Illegal character character (0xvalue)**
The compiler encountered some invalid character in the input file. The hexadecimal value of the offending character is printed. This can also be caused by extra parameters passed to a function macro.
- MAKE error* **Illegal character in constant expression expression**
MAKE encountered a character not allowed in a constant expression. If the character is a letter, this probably indicates a misspelled identifier.
- TLINK fatal error* **Illegal group definition: group in module module**
This error results from a malformed GRPDEF record in an **.OBJ** file. This latter case could result from custom-built **.OBJ** files or a bug in the translator used to generate the **.OBJ** file. If this occurs in a file created by Borland C++, recompile the file. If the error persists, contact Borland.
- Compile-time error* **Illegal initialization**
In C, initializations must be either a constant expression, or else the address of a global **extern** or **static** variable plus or minus a constant.
- MAKE or compile-time error* **Illegal octal digit**
An octal constant was found containing a digit of 8 or 9.
- Compile-time error* **Illegal parameter to __emit__**
You supplied an argument to **emit** which is not a constant or an address.

- Compile-time error* **Illegal pointer subtraction**
This is caused by attempting to subtract a pointer from a non-pointer.
- Compile-time error* **Illegal structure operation**
In C or C++, structures may be used with dot (`.`), address-of (`&`), or assignment (`=`) operators, or be passed to or from functions as parameters. In C or C++, structures can also be used with overloaded operators. The compiler encountered a structure being used with some other operator.
- Compile-time error* **Illegal to take address of bit field**
It is not legal to take the address of a bit field, although you can take the address of other kinds of fields.
- Compile-time error* **Illegal use of floating point**
Floating-point operands are not allowed in shift, bitwise Boolean, indirection (`*`), or certain other operators. The compiler found a floating-point operand with one of these prohibited operators.
- Compile-time error* **Illegal use of member pointer**
Pointers to class members can only be used with assignment, comparison, the `.*`, `->*`, `?:`, `&&` and `||` operators, or passed as arguments to functions. The compiler has encountered a member pointer being used with a different operator.
- Compile-time error* **Illegal use of pointer**
Pointers can only be used with addition, subtraction, assignment, comparison, indirection (`*`) or arrow (`->`). Your source file used a pointer with some other operator.
- Compile-time warning* **Ill-formed pragma**
A pragma does not match one of the pragmas expected by the Borland C++ compiler.
- Compile-time error* **Implicit conversion of *type1* to *type2* not allowed**
When a member function of a class is called using a pointer to a derived class, the pointer value must be implicitly converted to point to the appropriate base class. In this case, such an implicit conversion is illegal.
- TLINK error* **Imported reference from a VIRDEF to *symbol***
 The linker does not currently support references from VIRDEFs to symbols that are imported from dynamically-linked libraries. If you have an inline function in an executable which

references a static data member of a class in a DLL, take the function out of the line.

- Compile-time error* **Improper use of typedef identifier**
Your source file used a **typedef** symbol where a variable should appear in an expression. Check for the declaration of the symbol and possible misspellings.
- TLINK fatal error* **filename (linenum): Incompatible attribute**
TLINK encountered incompatible segment attributes in a CODE or DATA statement. For instance, both PRELOAD and LOADONCALL can't be attributes for the same segment.
- Compile-time error* **Incompatible type conversion**
The cast requested can't be done. Check the types.
- MAKE fatal error* **Incorrect command-line argument: argument**
You've used incorrect command-line arguments.
- Compile-time error* **Incorrect command-line option: option**
The compiler did not recognize the command-line parameter as legal.
- Compile-time error* **Incorrect configuration file option: option**
The compiler did not recognize the configuration file parameter as legal; check for a preceding hyphen (-).
- Compile-time error* **Incorrect number format**
The compiler encountered a decimal point in a hexadecimal number.
- Compile-time error* **Incorrect use of default**
The compiler found no colon after the **default** keyword.
- Compile-time warning* **Initializing enumeration with type**
You're trying to initialize an **enum** variable to a different type. For example,
- ```
enum count { zero, one, two } x = 2;
```
- will result in this warning, because 2 is of type **int**, not type **enum count**. It is better programming practice to use an **enum** identifier instead of a literal integer when assigning to or initializing **enum** types.
- This is an error, but is reduced to a warning to give existing programs a chance to work.

- Compile-time error* **Inline assembly not allowed in inline and template functions**  
The compiler cannot handle inline assembly statements in a C++ inline or template function. You could eliminate the inline assembly code or, in case of an inline function, make this a macro, or remove the **inline** storage class.
- MAKE error* **Int and string types compared**  
You have tried to compare an integer operand with a string operand in an **!if** or **!elif** expression.
- TLINK fatal error* **Internal linker error *errorcode***  
An error occurred in the internal logic of TLINK. This error shouldn't occur in practice, but is listed here for completeness in the event that a more specific error isn't generated. If this error persists, write down the *errorcode* number and contact Borland.
- Compile-time error* **Invalid combination of opcode and operands**  
The built-in assembler does not accept this combination of operands. Possible causes are:
- There are too many or too few operands for this assembler opcode; for example, **INC AX,BX**, or **MOV AX**.
  - The number of operands is correct, but their types or order do not match the opcode; for example **DEC 1**, **MOV AX,CL**, or **MOV 1,AX**.
- TLINK warning* **Invalid entry at *xxxxh***  
 This error indicates that a necessary entry was missing from the entry table of a Windows executable file. The application may not work in real mode unless you fix the code and data.
- TLINK error* **Invalid entry point offset**  
This message occurs only when modules with 32-bit records are linked. It means that the initial program entry point offset exceeds the DOS limit of 64K.
- Compile-time error* **Invalid indirection**  
The indirection operator (**\***) requires a non-**void** pointer as the operand.
- TLINK fatal error* **Invalid initial stack offset**  
This message occurs only when modules with 32-bit records are linked. It means that the initial stack pointer value exceeds the DOS limit of 64K.

*TLINK fatal error*

**Invalid limit specified for code segment packing**

This error occurs if you used the */P* option or IDE Options | Linker | Settings | Pack code segments and specified a size limit that was out of range. To be valid, the size limit must be between 1 and 65536 bytes; the default is 8192.

*Compile-time error*

**Invalid macro argument separator**

In a macro definition, arguments must be separated by commas. The compiler encountered some other character after an argument name.

*TLIB warning*

**invalid page size value ignored**

Invalid page size is given. The page size must be a power of 2, and it may not be smaller than 16 or larger than 32,768.

*Compile-time error*

**Invalid pointer addition**

Your source file attempted to add two pointers together.

*Compile-time error*

**Invalid register combination (e.g. [BP+BX])**

The built-in assembler detected an illegal combination of registers in an instruction. Valid index register combinations are [BX], [BP], [SI], [DI], [BX+SI], [BX+DI], [BP+SI], and [BP+DI]. Other index register combinations (such as [AX], [BP+BX], and [SI+DX]) are not allowed.



Local variables (variables declared in procedures and functions) are usually allocated on the stack and accessed via the BP register. The assembler automatically adds [BP] in references to such variables, so even though a construct like **Local[BX]** (where **Local** is a local variable) appears valid, it is not since the final operand would become **Local[BP+BX]**.

*TLINK fatal error*

**Invalid segment definition in module *module***

The compiler produced a flawed object file. If this occurs in a file created by Borland C++, recompile the file. If the problem persists, contact Borland.

*TLINK error*



**Invalid size specified for segment alignment**

This error occurs if an invalid value is specified for the Options | Linker | Settings | Segment alignment (or */A*) option. The value specified must be an integral multiple of 2 and less than 64K. Common values are 16 and 512. This error only occurs when linking for Windows.

- Compile-time error* **Invalid template argument list**  
 In a template declaration, the keyword **template** must be followed by a list of formal arguments enclosed within the < and > delimiters; an illegal template argument list was found.
- Compile-time error* **Invalid template qualified name `template::name`**  
 When defining a template class member, the actual arguments in the template class name that is used as the left operand for the :: operator must match the formal arguments of the template class. For example:
- ```
template <class T> class X
{
    void f();
};

template <class T> void X<T>::f(){}
```
- The following would be illegal:
- ```
template <class T> void X<int>::f(){}
```
- Compile-time error* **Invalid use of dot**  
 An identifier must immediately follow a period operator (.).
- Compile-time error* **Invalid use of template `template`**  
 Outside of a template definition, it is illegal to use a template class name without specifying its actual arguments. For example, you can use **vector<int>** but not **vector**.
- Compile-time fatal error* **Irreducible expression tree**  
 This is a sign of some form of compiler error. Some expression on the indicated line of the source file has caused the code generator to be unable to generate code. Whatever the offending expression is, it should be avoided. Notify Borland if the compiler ever encounters this error.
- Compile-time error* **base is an indirect virtual base class of class**  
 A pointer to a C++ member of the given virtual base class cannot be created; an attempt has been made to create such a pointer (either directly, or through a cast). See the **-Vv** switch in the *User's Guide*.
- Compile-time warning* **identifier is assigned a value that is never used**  
 The variable appears in an assignment, but is never used anywhere else in the function just ending. The warning is indicated only when the compiler encounters the closing brace.

- Compile-time warning* **identifier is declared as both external and static**  
This identifier appeared in a declaration that implicitly or explicitly marked it as global or external, and also in a static declaration. The identifier is taken as static. You should review all declarations for this identifier.
- TLINK error or warning* **symbol is duplicated in module *module***  
There is a conflict between two symbols (either public or communal) defined in the same module. An error occurs if both are encountered in an .OBJ file. A warning is issued if TLINK finds the duplicates in a library; in this case, TLINK uses the first definition.
- Compile-time error* **constructor is not a base class of *class***  
A C++ class constructor *class* is trying to call a base class constructor *constructor*, or you are trying to change the access rights of *class::constructor*. *constructor* is not a base class of *class*. Check your declarations.
- Compile-time error* **identifier is not a member of *struct***  
You are trying to reference *identifier* as a member of ***struct***, but it is not a member. Check your declarations.
- Compile-time error* **identifier is not a non-static data member and can't be initialized here**  
Only data members can be initialized in the initializers of a constructor. This message means that the list includes a static member or function member.
- Compile-time error* **identifier is not a parameter**  
In the parameter declaration section of an old-style function definition, *identifier* is declared but is not listed as a parameter. Either remove the declaration or add *identifier* as a parameter.
- Compile-time error* **identifier is not a public base class of *classtype***  
The right operand of a ***.\****, ***->\****, or ***::operator*** was not a pointer to a member of a class that is either identical to or an unambiguous accessible base class of the left operand's class type.
- Compile-time error* **member is not accessible**  
You are trying to reference C++ class member *member*, but it is **private** or **protected** and cannot be referenced from this function. This sometimes happens when attempting to call one accessible overloaded member function (or constructor), but the arguments match an inaccessible function. The check for

overload resolution is always made before checking for accessibility. If this is the problem, try an explicit cast of one or more parameters to select the desired accessible function.

*Compile-time error*

**Last parameter of *operator* must have type *int***

When a postfix **operator++** or **operator--** is declared, the last parameter must be declared with the type **int**.

*TLIB warning*

**library contains COMDEF records – extended dictionary not created**

An object record being added to a library contains a COMDEF record. This is not compatible with the extended dictionary option.

*TLIB error*

**library too large, please restart with */P size***

**library too large, restart with library page size *size***

The library being created could not be built with the current library page size. You can set the library page size with the */P* command-line switch detailed in Chapter 3, “TLIB: The Turbo librarian” in the *Tools and Utilities Guide*. In the IDE, the library page size can be set from the Options | Librarian dialog box.

*TLINK fatal error*



**Limit of 254 segments for new executable file exceeded**

The new executable file format only allows for 254 segments. Examine the map file. Usually, one of two things cause the problem. If the application is large model, the code segment packing size could be so small that there are too many code segments. Increasing the code segment packing size with the */P* option could help.

The other possibility is that you have a lot of far data segments with only a few bytes of data in them. The map file will tell you if this is happening. In this case, reduce the number of far data segments.

*Compile-time error*

**Linkage specification not allowed**

Linkage specifications such as **extern “C”** are only allowed at the file level. Move this function declaration out to the file level.

*TLINK fatal error*

**Linker stack overflow**

TLINK uses a recursive procedure for marking modules to be included in an executable image from libraries. This procedure can cause stack overflows in extreme circumstances. If you get this error message, remove some modules from libraries, include them with the object files in the link, and try again.

- Compile-time error* **Lvalue required**  
The left hand side of an assignment operator must be an addressable expression. These include numeric or pointer variables, structure field references or indirection through a pointer, or a subscripted array element.
- DPMI server fatal error* **Machine not in database (run DPMIINST)**  
The Dos Protected Mode Interface (DPMI) server searched the kernel's database and could not locate information about your machine. Run DPMIINST (several times, if necessary) to update the database. DPMIINST also generates a .DB file for you to send to Borland. See also **A20 line already enabled, so test is meaningless** on page 159.
- Compile-time error* **Macro argument syntax error**  
An argument in a macro definition must be an identifier. The compiler encountered some non-identifier character where an argument was expected.
- Compile-time error* **Macro expansion too long**  
A macro cannot expand to more than 4,096 characters.
- MAKE error* **Macro expansion too long**  
A macro cannot expand to more than 4,096 characters. This error often occurs if a macro recursively expands itself. A macro cannot legally expand to itself.
- MAKE fatal errors* **Macro substitute text *string* is too long**  
**Macro replace text *string* is too long**  
The macro substitution or replacement text *string* overflowed MAKE's internal buffer of 512 bytes.
- Compile-time error* **main must have a return type of int**  
In C++, function **main** has special requirements, one of which is that it cannot be declared with any return type other than **int**.
- Compile-time error* **Matching base class function for *function* has different dispatch number.**  
If a DDVT function is declared in a derived class, the matching base class function must have the same dispatch number as the derived function.
- Compile-time error* **Matching base class function for *function* is not dynamic**  
If a DDVT function is declared in a derived class, the matching base class function must also be dynamic.

- Compile-time warning*    **Maximum precision used for member pointer type *type***  
When a member pointer type is declared, its class has not been fully defined, and the **-Vmd** option has been used, the compiler has to use the most general (and the least efficient) representation for that member pointer type. This may not only cause less efficient code to be generated (and make the member pointer type unnecessarily large), but it can also cause problems with separate compilation; see the **-Vm** compiler switch discussion in Chapter 5, "The command-line compiler" in the *User's Guide* for details.
- Compile-time error*    **Member function must be called or its address taken**  
When a member function is used in an expression, either it must be called, or its address must be taken using the **&** operator. In this case, a member function has been used in an illegal context.
- Compile-time error*    **Member identifier expected**  
The name of a structure or C++ class member was expected here, but not found. The right side of a dot (.) or arrow (->) operator must be the name of a member in the structure or class on the left of the operator.
- Compile-time error*    **Member is ambiguous: *member1* and *member2***  
You must qualify the member reference with the appropriate base class name. In C++ class *class*, member *member* can be found in more than one base class, and was not qualified to indicate which was meant. This happens only in multiple inheritance, where the member name in each base class is not hidden by the same member name in a derived class on the same path. The C++ language rules require that this test for ambiguity be made before checking for access rights (**private**, **protected**, **public**). It is therefore possible to get this message even though only one (or none) of the members can be accessed.
- Compile-time error*    **Member *member* cannot be used without an object**  
This means that the user has written *class::member* where *member* is an ordinary (non-static) member, and there is no class to associate with that member. For example, it is legal to write *obj.class::member*, but not to write *class::member*.
- Compile-time error*    **Member *member* has the same name as its class**  
A static data member, enumerator, member of an anonymous union, or nested type may not have the same name as its class.





Only a member function or a non-static member may have a name that is identical to its class.

*Compile-time error*

**Member *member* is initialized more than once**

In a C++ class constructor, the list of initializations following the constructor header includes the same member name more than once.

*Compile-time error*

**Member pointer required on right side of *.* or *->*\***

The right side of a C++ dot-star (*.*) or an arrow-star (*->*\*) operator must be declared as a pointer to a member of the class specified by the left side of the operator. In this case, the right side is not a member pointer.

*TLIB warning*

**Memory full listing truncated!**

The librarian has run out of memory creating a library listing file. A list file will be created but is not complete.

*Compile-time error*

**Memory reference expected**

The built-in assembler requires a memory reference. Most likely you have forgotten to put square brackets around an index register operand; for example, **MOV AX,BX+SI** instead of **MOV AX,[BX+SI]**.

*Compile-time error*

**Misplaced break**

The compiler encountered a **break** statement outside a **switch** or looping construct.

*Compile-time error*

**Misplaced continue**

The compiler encountered a **continue** statement outside a looping construct.

*Compile-time error*

**Misplaced decimal point**

The compiler encountered a decimal point in a floating-point constant as part of the exponent.

*Compile-time error*

**Misplaced elif directive**

The compiler encountered an **#elif** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.

*MAKE error*

**Misplaced elif statement**

An **!elif** directive is missing a matching **!if** directive.

*Compile-time error*


**Misplaced else**



The compiler encountered an **else** statement without a matching **if** statement. An extra **else** statement could cause this message, but it could also be caused by an extra semicolon, missing braces, or some syntax error in a previous **if** statement.

- Compile-time error*    **Misplaced else directive**  
The compiler encountered an **#else** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.
- MAKE error*    **Misplaced else statement**  
There's an **!else** directive without any matching **!if** directive.
- Compile-time error*    **Misplaced endif directive**  
The compiler encountered an **#endif** directive without any matching **#if**, **#ifdef**, or **#ifndef** directive.
- MAKE error*    **Misplaced endif statement**  
There's an **!endif** directive without any matching **!if** directive.
- TLINK fatal error*    **filename (linenum): Missing internal name**  
In the IMPORTS section of the module definition file there was a reference to an entry specified via module name and ordinal number. When an entry is specified by ordinal number an internal name must be assigned to this import definition. It is this internal name that your program uses to refer to the imported definition. The syntax in the module definition file should be:
- ```
<internalname>=<modulename>.<ordinal>
```
- Compile-time warning* **Mixing pointers to signed and unsigned char**
You converted a **signed char** pointer to an **unsigned char** pointer, or vice versa, without using an explicit cast. (Strictly speaking, this is incorrect, but it is often harmless.)
- Compile-time error* **Multiple base classes require explicit class names**
In a C++ class constructor, each base class constructor call in the constructor header must include the base class name when there is more than one immediate base class.
- Compile-time error* **Multiple declaration for identifier**
This identifier was improperly declared more than once. This might be caused by conflicting declarations such as `int a;` `double a;`, a function declared two different ways, or a label repeated in the same function, or some declaration repeated other than an **extern** function or a simple variable (in C).
- Compile-time error* **identifier must be a member function**
Most C++ operator functions may be members of classes or ordinary nonmember functions, but certain ones are required to be members of classes. These are **operator =**, **operator ->**, **operator ()**, and type conversions. This operator function is not a member function but should be.



- Compile-time error* **identifier must be a member function or have a parameter of class type**
Most C++ operator functions must have an implicit or explicit parameter of class type. This operator function was declared outside a class and does not have an explicit parameter of class type.
- Compile-time error* **identifier must be a previously defined class or struct**
You are attempting to declare *identifier* to be a base class, but either it is not a class or it has not yet been fully defined. Correct the name or rearrange the declarations.
- Compile-time error* **identifier must be a previously defined enumeration tag**
This declaration is attempting to reference *identifier* as the tag of an **enum** type, but it has not been so declared. Correct the name, or rearrange the declarations.
- Compile-time error* **function must be declared with no parameters**
This C++ operator function was incorrectly declared with parameters.
- Compile-time error* **function must be declared with one parameter**
This C++ operator function was incorrectly declared with more than one parameter.
- Compile-time error* **operator must be declared with one or no parameters**
When **operator++** or **operator --** is declared as a member function, it must be declared to take either no parameters (for the prefix version of the operator) or one parameter of type **int** (for the postfix version).
- Compile-time error* **operator must be declared with one or two parameters**
When **operator++** or **operator --** is declared as a nonmember function, it must be declared to take either one parameter (for the prefix version of the operator) or two parameters (the postfix version).
- Compile-time error* **function must be declared with two parameters**
This C++ operator function was incorrectly declared with other than two parameters.
- Compile-time error* **Must take address of a memory location**
Your source file used the address-of operator (**&**) with an expression which cannot be used that way; for example, a register variable (in C).

- Compile-time error* **Need an identifier to declare**
In this context, an identifier was expected to complete the declaration. This might be a **typedef** with no name, or an extra semicolon at file level. In C++, it might be a class name improperly used as another kind of identifier.
- IDE debugger error* **'new' and 'delete' not supported**
In integrated debugger expression evaluation, the **new** and **delete** operators are not supported.
- TLINK fatal error* **New executable header overflowed 64K**
The size of all the components of the new executable header of a Windows application is greater than 64K. Usually this is caused by a very large RESIDENTNAME table. If your application exports many functions, try exporting more of them by ordinal, rather than by name.
- Compile-time error* **No : following the ?**
The question mark (?) and colon (:) operators do not match in this expression. The colon may have been omitted, or parentheses may be improperly nested or missing.
- TLINK warning* **No automatic data segment**

No group named DGROUP was found. Because the Borland C++ initialization files define DGROUP, you will only see this error if you don't link with an initialization file and your program doesn't define DGROUP. Windows uses DGROUP to find the local data segment. The DGROUP is required for Windows applications (but not DLLs) unless DATA NONE is specified in the module definition file.
- Compile-time error* **No base class to initialize**
This C++ class constructor is trying to implicitly call a base class constructor, but this class was declared with no base classes. Check your declarations.
- MAKE error* **No closing quote**
There is no closing quote for a string expression in a !if or !elif expression.
- Compile-time warning* **No declaration for function *function***
You called a function without first declaring that function. In C, you can declare a function without presenting a prototype, as in "int func();". In C++, every function declaration is also a prototype; this example is equivalent to "int func(void);". The declaration can be either classic or modern (prototype) style.

- Compile-time error* **No file name ending**
The file name in an **#include** statement was missing the correct closing quote or angle bracket.
- MAKE error* **No file name ending**
The file name in an **!include** statement is missing the correct closing quote or angle bracket.
- Compile-time error* **No file names given**
The command line of the Borland C++ command-line compiler (BCC) contained no file names. You have to specify a source file name.
- MAKE error* **No macro before =**
You must give a macro a name before you can assign it a value.
- MAKE error* **No match found for wildcard *expression***
There are no files matching the wildcard *expression* for MAKE to expand. For example, if you write
- ```
prog.exe: *.obj
```
- MAKE sends this error message if there are no files with the extension .OBJ in the current directory.
- TLINK warning* **No module definition file specified: using defaults**  
 TLINK was invoked with one of the Windows options, but no module definition file was specified. See page 72 for more information about module definition file defaults.
- TLINK warning* **No program starting address defined**  
 This warning means that no module defined the initial starting address of the program. This is almost certainly caused by forgetting to link in the initialization module C0x.OBJ. This warning should not occur when linking a Windows DLL.
- TLINK warning* **No stack**  
This warning is issued if no stack segment is defined in any of the object files or in any of the libraries included in the link. This is a normal message for the tiny memory model in Borland C++, or for any application program that will be converted to a .COM file. For other programs (except DLLs), this indicates an error.
- If a Borland C++ program produces this message for any but the tiny memory model, make sure you are using the correct C0x startup object files.

|                                                                             |                                                                                                                                                                                                                                                                                                                            |
|-----------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>TLINK warning</i>                                                        | <p><b>No stub for fixup at <i>segment:xxxxh</i> in module <i>module</i></b></p> <p>This error occurs when the target for a fixup is in an overlay segment, but no stub is found for a target external. This is usually the result of not making public a symbol in an overlay that is referenced from the same module.</p> |
| <i>MAKE fatal error</i>                                                     | <p><b>No terminator specified for in-line file operator</b></p> <p>The makefile contains either the <b>&amp;&amp;</b> or <b>&lt;&lt;</b> command-line operators to start an in-line file, but the file is not terminated.</p>                                                                                              |
| <i>Compile-time error</i><br><i>This message used only by IDE debugger.</i> | <p><b>No type information</b></p> <p>Debugger has no type information for this variable. Module may have been compiled without debug switch turned on, or by another compiler or assembler.</p>                                                                                                                            |
| <i>Compile-time warning</i>                                                 | <p><b>Non-const function <i>function</i> called for const object</b></p> <p>A non-<b>const</b> member function was called for a <b>const</b> object. This is an error, but was reduced to a warning to give existing programs a chance to work.</p>                                                                        |
| <i>Compile-time warning</i>                                                 | <p><b>Nonportable pointer comparison</b></p> <p>Your source file compared a pointer to a non-pointer other than the constant zero. You should use a cast to suppress this warning if the comparison is proper.</p>                                                                                                         |
| <i>Compile-time error</i>                                                   | <p><b>Nonportable pointer conversion</b></p> <p>An implicit conversion between a pointer and an integral type is required, but the types are not the same size. This cannot be done without an explicit cast. This conversion may not make any sense, so be sure this is what you want to do.</p>                          |
| <i>Compile-time warning</i>                                                 | <p><b>Nonportable pointer conversion</b></p> <p>A nonzero integral value is used in a context where a pointer is needed or where an integral value is needed; the sizes of the integral type and pointer are the same. Use an explicit cast if this is what you really meant to do.</p>                                    |
| <i>Compile-time error</i>                                                   | <p><b>Nontype template argument must be of scalar type</b></p> <p>A nontype formal template argument must have scalar type; it can have an integral, enumeration, or pointer type.</p>                                                                                                                                     |
| <i>Compile-time error</i>                                                   | <p><b>Non-virtual function <i>function</i> declared pure</b></p> <p>Only virtual functions can be declared pure, since derived classes must be able to override them.</p>                                                                                                                                                  |
| <i>Compile-time warning</i>                                                 | <p><b>Non-volatile function <i>function</i> called for volatile object</b></p> <p>In C++, a class member function was called for a volatile object of the class type, but the function was not declared with</p>                                                                                                           |



“volatile” following the function header. Only a volatile member function may be called for a volatile object.

*Compile-time error*  
*This message used only by*  
*IDE debugger.*

**Not a valid expression format type**

Invalid format specifier following expression in the debug evaluate or watch window. A valid format specifier is an optional repeat value followed by a format character (c, d, f[n], h, x, m, p, r, or s).

*Compile-time error*

**Not an allowed type**

Your source file declared some sort of forbidden type; for example, a function returning a function or array.

*MAKE fatal error*

**Not enough memory**

All your working storage has been exhausted.

*TLINK fatal error*

**Not enough memory**

There is not enough memory to run TLINK. Try reducing the size of any RAM disk or disk cache currently active. Then run TLINK again. If you are running real mode, try using the MAKE **-S** option, removing TSRs and network drivers. If you are using protected mode MAKE, try reducing the size of any ram disk or disk cache you may have active.

*TLIB error*

**Not enough memory for command-line buffer**

This error occurs when TLIB runs out of memory.

*DPMI server fatal error*

**not enough memory for PM init**

There was not enough extended memory available for the DPMI server to initialize protected mode.

*TLIB warning*

**module not found in library**

An attempt to perform either a ‘\_’ or ‘\*’ on a library has been made and the indicated object does not exist in the library.

*Run-time error*

**Null pointer assignment**

When a small or medium memory model program exits, a check is made to determine if the contents of the first few bytes within the program’s data segment have changed. These bytes would never be altered by a working program. If they have been changed, the message “Null pointer assignment” is displayed to inform you that (most likely) a value was stored to an uninitialized pointer. The program may appear to work properly in all other respects; however, this is a serious bug which should be attended to immediately. Failure to correct an uninitialized pointer can lead to unpredictable behavior (including “locking” the computer up in the large, compact,

and huge memory models). You can use the integrated debugger to track down null pointers.

*Compile-time error*

**Numeric constant too large**

String and character escape sequences larger than hexadecimal `\xFF` or octal `\377` cannot be generated. Two-byte character constants may be specified by using a second backslash. For example, `\x0D\x0A` represents a two-byte constant. A numeric literal following an escape sequence should be broken up like this:

```
printf("\x0D" "12345");
```

This prints a carriage return followed by 12345.

*TLIB error*

**object module *filename* is invalid**

The librarian could not understand the header record of the object module being added to the library and has assumed that it is an invalid module.

*Compile-time error*

**Objects of type *type* cannot be initialized with {}**

Ordinary C structures can be initialized with a set of values inside braces. C++ classes can only be initialized with constructors if the class has constructors, private members, functions or base classes which are virtual.

*MAKE error*

**Only <<KEEP or <<NOKEEP**

You have specified something besides KEEP or NOKEEP when closing a temporary inline file.

*Compile-time error*

**Only member functions may be 'const' or 'volatile'**

Something other than a class member function has been declared **const** and/or **volatile**.

*Compile-time error*

**Only one of a set of overloaded functions can be "C"**

C++ functions are by default overloaded, and the compiler assigns a new name to each function. If you wish to override the compiler's assigning a new name by declaring the function extern "C", you can do this for only one of a set of functions with the same name. (Otherwise the linker would find more than one global function with the same name.)

*Compile-time error*

**Operand of delete must be non-const pointer**

It is illegal to delete a constant pointer value using operator **delete**.

*Compile-time error*


**Operator [ ] missing ]**

The C++ **operator[ ]** was declared as **operator [**. You must add the missing **]** or otherwise fix the declaration.





- Compile-time error*    **operator -> must return a pointer or a class**  
The C++ **operator->** function must be declared to either return a class or a pointer to a **class** (or **struct** or **union**). In either case, it must be something to which the **->** operator can be applied.
- Compile-time error*    **operator delete must return void**  
This C++ overloaded operator **delete** was declared in some other way.
- Compile-time error*    **Operator must be declared as function**  
An overloaded operator was declared with something other than function type.
- Compile-time error*    **operator new must have an initial parameter of type size\_t**  
Operator **new** can be declared with an arbitrary number of parameters, but it must always have at least one, which is the amount of space to allocate.
- Compile-time error*    **operator new must return an object of type void \***  
The C++ overloaded operator **new** was declared another way.
- Compile-time error*    **Operators may not have default argument values**  
It is illegal for overloaded operators to have default argument values.
- Compile-time fatal error*    **Out of memory**  
The total working storage is exhausted. Compile the file on a machine with more memory. When running under Windows, close one or more applications to free up memory.
- TLIB error*    **Out of memory**  
For any number of reasons, TLIB or Borland C++ ran out of memory while building the library. For many specific cases a more detailed message is reported, leaving "Out of memory" to be the basic catchall for general low memory situations. When running under Windows, close one or more applications to free up memory.
- If this message occurs when public symbol tables grow too large, you must free up memory. For the command line this could involve removing TSR's or device drivers using real mode memory. In the IDE, some additional memory can be gained by closing editors. When running under Windows, close one or more applications to free up memory.

- TLINK fatal error* **Out of memory**  
TLINK has run out of dynamically allocated memory needed during the link process. This error is a catchall for running into a TLINK limit on memory usage. This usually means that too many modules, externals, groups, or segments have been defined by the object files being linked together. You can try reducing the size of RAM disks and/or disk caches that may be active. If running under Windows, close one or more applications to free up memory.
- TLIB error* **out of memory creating extended dictionary**  
The librarian has run out of memory creating an extended dictionary for a library. The library is created but will not have an extended dictionary.
- TLIB error* **out of memory reading LE/LIDATA record from object module**  
The librarian is attempting to read a record of data from the object module, but it cannot get a large enough block of memory. If the module that is being added has a large data segment or segments, it is possible that adding the module before any other modules might resolve the problem. By adding the module first, there will be memory available for holding public symbol and module lists later.
- TLIB error* **Out of space allocating per module debug struct**  
The librarian ran out of memory while allocating space for the debug information associated with a particular object module. Removing debugging information from some modules being added to the library might resolve the problem.
- TLIB error* **Output device is full**  
The output device is full, usually no space left on the disk.
- TLINK warning* **Overlays generated and no overlay manager included**  
This warning is issued if overlays are created but the symbol `__OVRTRAP__` is not defined in any of the object modules or libraries linked in. The standard overlay library (OVERLAY.LIB) defines this symbol.
- TLINK warning*  **Overlays ignored in new executable image**  
This error occurs if you attempt to link a Windows program with the `/b` option on. Windows executables can't be overlaid, although, with discardable code segments, you should be able to achieve a similar effect.

- Compile-time error* **Overlays only supported in medium, large, and huge memory models**  
As explained in Chapter 9, “DOS memory management” of the *Programmer’s Guide*, only non-Windows programs using the medium, large, or huge memory models may be overlaid.
- Compile-time warning* **overload is now unnecessary and obsolete**  
Early versions of C++ required the reserved word **overload** to mark overloaded function names. C++ now uses a “type-safe linkage” scheme, whereby all functions are assumed overloaded unless marked otherwise. The use of **overload** should be discontinued.
- Compile-time error* **Overloadable operator expected**  
Almost all C++ operators can be overloaded. The only ones that can’t be overloaded are the field-selection dot (`.`), dot-star (`.*`), double colon (`::`), and conditional expression (`?:`). The preprocessor operators (`#` and `##`) are not C or C++ language operators and thus cannot be overloaded. Other nonoperator punctuation, such as semicolon (`;`), of course, cannot be overloaded.
- Compile-time error* **Overloaded function name ambiguous in this context**  
The only time an overloaded function name can be used without actually calling the function is when a variable or parameter of an appropriate type is initialized or assigned. In this case an overloaded function name has been used in some other context.
- Compile-time error* **Overloaded function resolution not supported**  
*This message used only by IDE debugger.*  
In integrated debugger expression evaluation, resolution of overloaded functions or operators is not supported, not even to take an address.
- Compile-time warning* **Overloaded prefix ‘operator operator’ used as a postfix operator**  
With the latest specification of C++, it is now possible to overload both the prefix and postfix versions of the `++` and `--` operators. To allow older code to compile, whenever only the prefix operator is overloaded, but is used in a postfix context, Borland C++ uses the prefix operator and issues this warning.
- Help project message* **P1001 Unable to read file filename**  
The file specified in the project file is unreadable. This is a DOS file error.

- Help project message* **P1003 Invalid path specified in Root option**  
The path specified by the Root option cannot be found. The compiler uses the current working directory.
- Help project message* **P1005 Path and filename exceed limit of 79 characters**  
The absolute pathname, or the combined root and relative pathname, exceed the DOS limit of 79 characters. The file is skipped.
- Help project message* **P1007 Root path exceeds maximum limit of 66 characters**  
The specified root pathname exceeds the DOS limit of 66 characters. The pathname is ignored and the compiler uses the current working directory.
- Help project message* **P1009 [FILES] section missing**  
The [Files] section is required. The compilation is aborted.
- Help project message* **P1011 Option *optionname* previously defined**  
The specified option was defined previously. The compiler ignores the attempted redefinition.
- Help project message* **P1013 Project file extension cannot be .HLP**  
You cannot specify that the compiler use a project file with the .HLP extension. Normally, project files are given the .HPJ extension.
- Help project message* **P1015 Unexpected end-of-file**  
The compiler has unexpectedly come to the end of the project file. There might be an open comment in the project file or an included file.
- Help project message* **P1017 Parameter exceeds maximum length of 128 characters**  
An option, context name or number, build tag, or other parameter on the specified line exceeds the limit of 128 characters. The line is ignored.
- Help project message* **P1021 Context number already used in [MAP] section**  
The context number on the specified line in the project file was previously mapped to a different context string. The line is ignored.
- Help project message* **P1023 Include statements nested too deeply**  
The **#include** statement on the specified line has exceeded the maximum of five include levels.
- Help project message* **P1025 Section heading *sectionname* unrecognized**  
A section heading that is not supported by the compiler has been used. The line is skipped.

- Help project message* **P1027 Bracket missing from section heading `sectionname`**  
The right bracket (]) is missing from the specified section heading. Insert the bracket and compile again.
- Help project message* **P1029 Section heading missing**  
The section heading on the specified line is not complete. This error is also reported if the first entry in the project file is not a section heading. The compiler continues with the next line.
- Help project message* **P1030 Section `sectionname` previously defined**  
A duplicate section has been found in the project file. The lines under the duplicated section heading are ignored and the compiler continues from the next valid section heading.
- Help project message* **P1031 Maximum number of build tags exceeded**  
The maximum number of build tags that can be defined is 30. The excess tags are ignored.
- Help project message* **P1033 Duplicate build tag in [BUILDTAGS] section**  
A build tag in the [BUILDTAGS] section has been repeated unnecessarily.
- Help project message* **P1035 Build tag length exceeds maximum**  
The build tag on the specified line exceeds the maximum of 32 characters. The compiler skips this entry.
- Help project message* **P1037 Build tag `tagname` contains invalid characters**  
Build tags can contain only alphanumeric characters or the underscore (\_) character. The line is skipped.
- Help project message* **P1039 [BUILDTAGS] section missing**  
The **BUILD** option declared a conditional build, but there is no [BuildTags] section in the project file. All topics are included in the build.
- Help project message* **P1043 Too many tags in Build expression**  
The Build expression on the specified line has used more than the maximum of 20 build tags. The compiler ignores the line.
- Help project message* **P1045 [ALIAS] section found after [MAP] section**  
When used, the [Alias] section must precede the [Map] section in the project file. The [Alias] section is skipped otherwise.
- Help project message* **P1047 Context string `contextname` already assigned an alias**  
You cannot do: `a=b` then `a=c<_>` (A context string can only have one alias.) The specified context string has previously been aliased in the [Alias] section. The attempted reassignment on this line is ignored.

- Help project message* **P1049 Alias string aliasname already assigned**  
You cannot do: a=b then b=c. An alias string cannot, in turn, be assigned another alias.
- Help project message* **P1051 Context string *contextname* cannot be used as alias string**  
You cannot do: a=b then c=a. A context string that has been assigned an alias cannot be used later as an alias for another context string.
- Help project message* **P1053 Maximum number of font ranges exceeded**  
The maximum number of font ranges that can be specified is five. The rest are ignored.
- Help project message* **P1055 Current font range overlaps previously defined range**  
A font size range overlaps a previously defined mapping. Adjust either font range to remove any overlaps. The second mapping is ignored.
- Help project message* **P1056 Unrecognized font name in Forcefont option**  
A font name not supported by the compiler has been encountered. The font name is ignored and the compiler uses the default Helvetica font.
- Help project message* **P1057 Font name too long**  
Font names cannot exceed 20 characters. The font is ignored.
- Help project message* **P1059 Invalid multiple-key syntax**  
The syntax used with a **MULTIKEY** option is unrecognized. See "Building the Help files" for the proper syntax.
- Help project message* **P1061 Character already used**  
The specified keyword-table identifier is already in use. Choose another character.
- Help project message* **P1063 Characters 'K' and 'k' cannot be used**  
These characters are reserved for Help's normal keyword table. Choose another character.
- Help project message* **P1065 Maximum number of keyword tables exceeded**  
The limit of five keyword tables has been exceeded. Reduce the number. The excess tables are ignored.
- Help project message* **P1067 Equal sign missing**  
An option is missing its required equal sign on the specified line. Check the syntax for the option.
- Help project message* **P1069 Context string missing**  
The line specified is missing a context string before an equal sign.

- Help project message* **P1071 Incomplete line in *sectionname* section**  
The entry on the specified line is not complete. The line is skipped.
- Help project message* **P1073 Unrecognized option in [OPTIONS] section**  
An option has been used that is not supported by the compiler. The line is skipped.
- Help project message* **P1075 Invalid build expression**  
The syntax used in the build expression on the specified line contains one or more logical or syntax errors.
- Help project message* **P1077 Warning level must be 1, 2, or 3**  
The **WARNING** reporting level can only be set to 1, 2, or 3. The compiler will default to full reporting (level 3).
- Help project message* **P1079 Invalid compression option**  
The **COMPRESS** option can only be set to TRUE or FALSE. The compilation continues without compression.
- Help project message* **P1081 Invalid title string**  
The **TITLE** option defines a string that is null or contains more than 32 characters. The title is truncated.
- Help project message* **P1083 Invalid context identification number**  
The context number on the specified line is null or contains invalid characters.
- Help project message* **P1085 Unrecognized text**  
The unrecognizable text that follows valid text in the specified line is ignored.
- Help project message* **P1086 Invalid font-range syntax**  
The font-range definition on the specified line contains invalid syntax. The compiler ignores the line. Check the syntax for the **MAPFONTSIZE** option.
- Help project message* **P1089 Unrecognized sort ordering**  
You have specified an ordering that is not supported by the compiler. Contact Borland Technical Support for clarification of the error.
- Compile-time error* **Parameter names are used only with a function body**  
When declaring a function (not defining it with a function body), you must use either empty parentheses or a function prototype. A list of parameter names only is not allowed.  
Example declarations include:

```

int func(); // declaration without prototype--OK
int func(int, int); // declaration with prototype--OK
int func(int i, int j); // parameter names in prototype--OK
int func(i, j); // parameter names only--illegal

```

*Compile-time error* **Parameter *number* missing name**

In a function definition header, this parameter consisted only of a type specifier *number* with no parameter name. This is not legal in C. (It is allowed in C++, but there's no way to refer to the parameter in the function.)

*Compile-time warning* **Parameter *parameter* is never used**

The named parameter, declared in the function, was never used in the body of the function. This may or may not be an error and is often caused by misspelling the parameter. This warning can also occur if the identifier is redeclared as an automatic (local) variable in the body of the function. The parameter is masked by the automatic variable and remains unused.

*TLIB error* ***path* – path is too long**

This error occurs when the length of any of the library file or module file's *path* is greater than 64.

*Compile-time error* **Pointer to structure required on left side of  $\rightarrow$  or  $\rightarrow^*$**

Nothing but a pointer is allowed on the left side of the arrow ( $\rightarrow$ ) in C or C++. In C++ a  $\rightarrow^*$  operator is allowed.

*Compile-time warning* **Possible use of *identifier* before definition**

Your source file used the named variable in an expression before it was assigned a value. The compiler uses a simple scan of the program to determine this condition. If the use of a variable occurs physically before any assignment, this warning will be generated. Of course, the actual flow of the program may assign the value before the program uses it.

*Compile-time warning* **Possibly incorrect assignment**

This warning is generated when the compiler encounters an assignment operator as the main operator of a conditional expression (that is, part of an **if**, **while** or **do-while** statement). Usually, this is a typographical error for the equality operator. To suppress this warning, enclose the assignment in parentheses and compare the whole thing to zero explicitly. Thus,

```
if (a = b) ...
```

should be rewritten as

```
if ((a = b) != 0) ...
```





- TLINK error* **Program entry point may not reside in an overlay**  
 Although almost all of an application can be overlaid, the initial starting address cannot reside in an overlay. This error usually means that an attempt was made to overlay the initialization module C0x.OBJ, for instance, by specifying the **o** option before the startup module.
- TLIB error* **public symbol in module *module1* clashes with prior module *module2***  
 A public symbol may only appear once in a library file. A module which is being added to the library contains a public *symbol* that is already in a module of the library and cannot be added. The command-line message reports the *module2* name.
- TLIB error* **public symbol in module *filename* clashes with prior module**  
 A public symbol may only appear once in a library file. A module which is being added to the library contains a public *symbol* that is already in a module of the library and cannot be added.
- Help RTF message* **R2001 Unable to open bitmap file *filename***  
 The specified bitmap file is unreadable. This is a DOS file error.
- Help RTF message* **R2003 Unable to include bitmap file *filename***  
 The specified bitmap file could not be found or is unreadable. This is a DOS file error or an out-of-memory condition.
- Help RTF message* **R2005 Disk full**  
 The Help resource file could not be written to disk. Create more space on the destination drive.
- Help RTF message* **R2009 Cannot use reserved DOS device name for file *filename***  
 A file has been referred to as COM1, LPT2, PRN, etc. Rename the file.
- Help RTF message* **R2013 Output file *filename* already exists as a directory**  
 There is a subdirectory in the Help project root with the same name as the desired Help resource file. Move or rename the subdirectory.
- Help RTF message* **R2015 Output file *filename* already exists as read-only**  
 The specified filename cannot be overwritten by the Help resource file because the file has a read-only attribute. Rename the project file or change the file's attribute.

- Help RTF message* **R2017 Path for file *filename* exceeds limit of 79 characters**  
The absolute pathname, or the combined root and relative pathname, to the specified file exceed the DOS limit of 79 characters. The file is ignored.
- Help RTF message* **R2019 Cannot open file *filename***  
The specified file is unreadable. This is a DOS file error.
- Help RTF message* **R2021 Cannot find file *filename***  
The specified file could not be found or is unreadable. This is a DOS file error or an out-of-memory condition.
- Help RTF message* **R2023 Not enough memory to build Help file**  
To free up memory, unload any unneeded applications, device drivers, and memory-resident programs.
- Help RTF message* **R2025 File environment error**  
The compiler has insufficient available file handles to continue. Increase the values for FILES= and BUFFERS= in your CONFIG.SYS file and reboot.
- Help RTF message* **R2027 Build tag *tagname* not defined in [BUILDTAGS] section of project file**  
The specified build tag has been assigned to a topic, but not declared in the project file. The tag is ignored for the topic.
- Help RTF message* **R2033 Context string in Map section not defined in any topic**  
There are one or more context strings defined in the project file that the compiler could not find topics for.
- Help RTF message* **R2035 Build expression missing from project file**  
The topics have build tags, but there is no Build= expression in the project file. The compiler includes all topics in the build.
- Help RTF message* **R2037 File *filename* cannot be created, due to previous error(s)**  
The Help resource file could not be created because the compiler has no topics remaining to be processed. Correct the errors that preceded this error and recompile.
- Help RTF message* **R2039 Unrecognized table formatting in topic *topicnumber* of file *filename***  
The compiler ignores table formatting that is unsupported in Help. Reformat the entries as linear text if possible.

- Help RTF message* **R2041 Jump *context\_string* unresolved in topic *topicnumber* of file *filename***  
The specified topic contains a context string that identifies a nonexistent topic. Check spelling, and that the desired topic is included in the build.
- Help RTF message* **R2043 Hotspot text cannot spread over paragraphs**  
A jump term spans two paragraphs. Remove the formatting from the paragraph mark.
- Help RTF message* **R2045 Maximum number of tab stops reached in topic *topicnumber* of file *filename***  
The limit of 32 tab stops has been exceeded in the specified topic. The default stops are used after the 32nd tab.
- Help RTF message* **R2047 File *filename* not created**  
There are no topics to compile, or the build expression is false for all topics. There is no Help resource file created.
- Help RTF message* **R2049 Context string text too long in topic *topicnumber* of file *filename***  
Context string hidden text cannot exceed 64 characters. The string is ignored.
- Help RTF message* **R2051 File *filename* is not a valid RTF topic file**  
The specified file is not an RTF file. Check that you have saved the topic as RTF from your word processor.
- Help RTF message* **R2053 Font *fontname* in file *filename* not in RTF font table**  
A font not defined in the RTF header has been entered into the topic. The compiler uses the default system font.
- Help RTF message* **R2055 File *filename* is not a usable RTF topic file**  
The specified file contains a valid RTF header, but the content is not RTF or is corrupted.
- Help RTF message* **R2057 Unrecognized graphic format in topic *topicnumber* of file *filename***  
The compiler supports only Windows bitmaps. Check that metafiles or Macintosh formats have not been used. The graphic is ignored.
- Help RTF message* **R2059 Context string identifier already defined in topic *topicnumber* of file *filename***  
There is more than one context-string identifier footnote for the specified topic. The compiler uses the identifier defined in the first # footnote.

- Help RTF message* **R2061 Context string *contextname* already used in file *filename***  
The specified context string was previously assigned to another topic. The compiler ignores the latter string and the topic has no identifier.
- Help RTF message* **R2063 Invalid context-string identifier for topic *topicnumber* of file *filename***  
The context-string footnote contains nonalphanumeric characters or is null. The topic is not assigned an identifier.
- Help RTF message* **R2065 Context string defined for index topic is unresolved**  
The index topic defined in the project file could not be found. The compiler uses the first topic in the build as the index.
- Help RTF message* **R2067 Footnote text too long in topic *topicnumber* of file *filename***  
Footnote text cannot exceed the limit of 1000 characters. The footnote is ignored.
- Help RTF message* **R2069 Build tag footnote not at beginning of topic *topicnumber* of file *filename***  
The specified topic contains a build tag footnote that is not the first character in the topic. The topic is not assigned a build tag.
- Help RTF message* **R2071 Footnote text missing in topic *topicnumber* of file *filename***  
The specified topic contains a footnote that has no characters.
- Help RTF message* **R2073 Keyword string is null in topic *topicnumber* of file *filename***  
A keyword footnote exists for the specified topic, but contains no characters.
- Help RTF message* **R2075 Keyword string too long in topic *topicnumber* of file *filename***  
The text in the keyword footnote in the specified topic exceeds the limit of 255 characters. The excess characters are ignored.
- Help RTF message* **R2077 Keyword(s) defined without title in topic *topicnumber* of file *filename***  
Keyword(s) have been defined for the specified topic, but the topic has no title assigned. Search Topics Found displays Untitled Topic<< for the topic.
- Help RTF message* **R2079 Browse sequence string is null in topic *topicnumber* of file *filename***  
The browse-sequence footnote for the specified topic contains no sequence characters.


- Help RTF message* **R2081 Browse sequence string too long in topic *topicnumber* of file *filename***  
The browse-sequence footnote for the specified topic exceeds the limit of 128 characters. The sequence is ignored.
- Help RTF message* **R2083 Missing sequence number in topic *topicnumber* of file *filename***  
A browse-sequence number ends in a colon (:) for the specified topic. Remove the colon, or enter a “minor” sequence number.
- Help RTF message* **R2085 Sequence number already defined in topic *topicnumber* of file *filename***  
There is already a browse-sequence footnote for the specified topic. The latter sequence is ignored.
- Help RTF message* **R2087 Build tag too long**  
A build tag for the specified topic exceeds the maximum of 32 characters. The tag is ignored for the topic.
- Help RTF message* **R2089 Title string null in topic *topicnumber* of file *filename***  
The title footnote for the specified topic contains no characters. The topic is not assigned a title.
- Help RTF message* **R2091 Title too long in topic *topicnumber* of file *filename***  
The title for the specified topic exceeds the limit of 128 characters. The excess characters are ignored.
- Help RTF message* **R2093 Title titlename in topic *topicnumber* of file *filename* used previously**  
The specified title has previously been assigned to another topic.
- Help RTF message* **R2095 Title defined more than once in topic *topicnumber* of file *filename***  
There is more than one title footnote in the specified topic. The compiler uses the first title string.
- Help RTF message* **R2501 Using old key-phrase table**  
Maximum compression can only result by deleting the .PH file before each recompilation of the Help topics.
- Help RTF message* **R2503 Out of memory during text compression**  
The compiler encountered a memory limitation during compression. Compilation continues with the Help resource file not compressed. Unload any unneeded applications, device drivers, and memory-resident programs.

- Help RTF message* **R2505 File environment error during text compression**  
The compiler has insufficient available file handles for compression. Compilation continues with the Help resource file not compressed. Increase the values for FILES= and BUFFERS= in your CONFIG.SYS file and reboot.
- Help RTF message* **R2507 DOS file error during text compression**  
The compiler encountered a problem accessing a disk file during compression. Compilation continues with the Help resource file not compressed.
- Help RTF message* **R2509 Error during text compression**  
One of the three compression errors—R2503, R2505, or R2507—has occurred. Compilation continues with the Help resource file not compressed.
- Help RTF message* **R2701 Internal error**  
**R2703 Internal error**  
**R2705 Internal error**  
**R2707 Internal error**  
**R2709 Internal error**  
Contact Borland Technical Support for clarification of the error.
- TLIB error* **record kind *num* found, expected theadr or lheadr in module *filename***  
The librarian could not understand the header record of the object module being added to the library and has assumed that it is an invalid module.
- TLIB error* **record length *len* exceeds available buffer in module *module***  
This error occurs when the record length *len* exceeds the available buffer to load the buffer in module *module*. This occurs when TLIB runs out of dynamic memory.
- TLIB error* **record type *type* found, expected theadr or lheadr in *module***  
TLIB encountered an unexpected type *type* instead of the expected THEADR or LHEADER record in module *module*.
- Compile-time warning* **Redefinition of *macro* is not identical**  
Your source file redefined the named macro using text that was not exactly the same as the first definition of the macro. The new text replaces the old.
- MAKE error* **Redefinition of target *filename***  
The named file occurs on the left side of more than one explicit rule.

- Compile-time error* **Reference initialized with *type1*, needs lvalue of type *type2***  
A reference variable or parameter that is not declared constant must be initialized with an lvalue of the appropriate type. In this case, the initializer either wasn't an lvalue, or its type didn't match the reference being initialized.
- Compile-time error* **Reference member *member* in class without constructors**  
A class that contains reference members must have at least one user-defined constructor; otherwise, there would be no way to ever initialize such members.
- Compile-time error* **Reference member *member* is not initialized**  
References must always be initialized. A class member of reference type must have an initializer provided in all constructors for that class. This means that you cannot depend on the compiler to generate constructors for such a class, since it has no way of knowing how to initialize the references.
- Compile-time error* **Reference member *member* needs a temporary for initialization**  
You provided an initial value for a reference type which was not an lvalue of the referenced type. This requires the compiler to create a temporary for the initialization. Since there is no obvious place to store this temporary, the initialization is illegal.
- Compile-time error* **Reference variable *variable* must be initialized**  
This C++ object is declared as a reference but is not initialized. All references must be initialized at the point of their declaration.
- Compile-time fatal error* **Register allocation failure**  
This is a sign of some form of compiler error. Some expression in the indicated function was so complicated that the code generator could not generate code for it. Try to simplify the offending function. Notify Borland Technical Support if the compiler encounters this error.
- TLINK fatal error* **Relocation item exceeds 1MB DOS limit**  
The DOS executable file format doesn't support relocation items for locations exceeding 1MB. Although DOS could never *load* an image this big, DOS extenders can, and thus TLINK supports generating images greater than DOS could load. Even if the image is loaded with a DOS extender, the DOS executable file format is limited to describing relocation items in the first 1MB of the image.

- TLINK fatal error*     **Relocation offset overflow**  
 This error only occurs for 32-bit object modules and indicates a relocation (segment fixup) offset greater than the DOS limit of 64K.
- TLINK fatal error*     **Relocation table overflow**  
 This error only occurs for 32-bit object modules. The file being linked contains more base fixups than the standard DOS relocation table can hold (base fixups are created mostly by calls to far functions).
- Compile-time error*  
*This message used only by IDE debugger.*     **Repeat count needs an lvalue**  
 The expression before the comma (,) in the Watch or Evaluate window must be a manipulable region of storage. For example, expressions like this one are not valid:
- ```

i++,10d
x = y, 10m

```
- TLIB warning* **results are safe in file *filename***
 The librarian has successfully built the library into a temporary file, but cannot rename the file to the desired library name. The temporary file will not be removed (so that the library can be preserved).
- MAKE error* **Rule line too long**
 An implicit or explicit rule was longer than 4,096 characters.
- TLINK fatal error* **Segment alignment factor too small**

 This error occurs if the segment alignment factor (set with Options | Linker | ... Segment Alignment the **/A** option) is too small to represent the file addresses of the segments in the .EXE file. This error only occurs when linking for Windows. See the documentation for the **/A** option on page 63 for more information.
- TLINK fatal error* **Segment *segment* exceeds 64K**
 This message occurs if too much data is defined for a given data or code segment when TLINK combines segments with the same name from different source files.
- TLINK warning* **Segment *segment* is in two groups: *group1* and *group2***
 The linker found conflicting claims by the two named groups. Usually, this only happens in assembly language programs. It means that two modules assigned the segment to two different groups.

TLINK fatal error



Segment too large for segment table

This error should never occur in practice. It means that a segment was bigger than 64K and its size cannot be represented in the executable file. This error can only occur when linking for Windows; the format of the executable file used for Windows does not support segments greater than 64K.

Compile-time error

*This message used only by
IDE debugger.*

Side effects are not allowed

Side effects such as assignments, **++**, or **--** are not allowed in the debugger watch window. A common error is to use $x = y$ (not allowed) instead of $x == y$ to test the equality of x and y .

Compile-time error

Size of *identifier* is unknown or zero

This identifier was used in a context where its size was needed. A **struct** tag may only be declared (the **struct** not defined yet), or an **extern** array may be declared without a size. It's illegal then to have some references to such an item (like **sizeof**) or to dereference a pointer to this type. Rearrange your declaration so that the size of *identifier* is available.

Compile-time error

sizeof may not be applied to a bit field

sizeof returns the size of a data object in bytes, which does not apply to a bit field.

Compile-time error

sizeof may not be applied to a function

sizeof may be applied only to data objects, not functions. You may request the size of a pointer to a function.

Compile-time error

Size of the type is unknown or zero

This type was used in a context where its size was needed. For example, a **struct** tag may only be declared (the **struct** not defined yet). It's illegal then to have some references to such an item (like **sizeof**) or to dereference a pointer to this type. Rearrange your declarations so that the size of this type is available.

Compile-time error

***identifier* specifies multiple or duplicate access**

A base class may be declared **public** or **private**, but not both. This access specifier may appear no more than once for a base class.

Run-time error

Stack overflow

The default stack size for Borland C++ programs is 5120 bytes. This should be enough for most programs, but those which execute recursive functions or store a great deal of local data can overflow the stack. You will only get this message if you

have stack checking enabled. If you do get this message, you can try increasing the stack size or decreasing your program's dependence on the stack. For Windows, to increase your stack size, use `STACKSIZE` in your module definition `.DEF` file. See Chapter 3, "Global variables" in the *Library Reference* for information on changing the stack size by altering the global variable `_stklen`. Try switching to a larger memory model to fit the larger stack.

To decrease the amount of local data used by a function, look at the example below. The variable `buffer` has been declared static and does not consume stack space like `list` does.

```
void anyfunction(void)
{
    static int buffer[2000]; /* resides in the data segment */
    int list[2000];         /* resides on the stack */
}
```

There are two disadvantages to declaring local variables as static.

1. It now takes permanent space away from global variables and the heap. (You have to rob Peter to pay Paul.) This is usually only a minor disadvantage.
2. The function may no longer be reentrant. What this means is that if the function is called recursively or asynchronously and it is important that each call to the function have its own unique copy of the variable, you cannot make it static. This is because every time the function is called, it will use the same exact memory space for the variable, rather than allocating new space for it on each call. You could have a sharing problem if the function is trying to execute from within itself (recursively) or at the same time as itself (asynchronously). For most DOS programs this is not a problem.


TLINK warning



Stack size is less than 1400h. It has been reset to 1400h.

Windows 3.0 requires the stack size of an application to be at least 1400h. If the automatic data segment (ADS) is near 64K, but your stack is less than 1400h, this can cause the ADS to overflow at load time, but not at link time. To protect against this, TLINK forces the stack size to be at least 1400h for a Windows application.

S

- Compile-time error* **Statement missing ;**
The compiler encountered an expression statement without a semicolon following it.
- Compile-time error* **Storage class *storage class* is not allowed here**
The given storage class is not allowed here. Probably two storage classes were specified, and only one may be given.
- MAKE error* **String type not allowed with this operand**
You have tried to use an operand which is not allowed for comparing string types. Valid operands are ==, !=, <, >, <=, and >=.
- Compile-time warning* **Structure passed by value**
A structure was passed by value as an argument to a function without a prototype. It is a frequent programming mistake to leave an address-of operator (&) off a structure when passing it as an argument. Because structures can be passed by value, this omission is acceptable. This warning provides a way for the compiler to warn you of this mistake.
- Compile-time error* **Structure required on left side of . or .***
The left side of a dot (.) operator (or C++ dot-star operator) must evaluate to a structure type. In this case it did not.
- Compile-time error* **Structure size too large**
Your source file declared a structure larger than 64K.
- TLINK fatal error* **Stub program exceeds 64K**
 This error occurs if a DOS stub program written for a Windows application exceeds 64K. Stub programs are specified via the STUB module definition file statement; TLINK only supports stub programs up to 64K.
- Compile-time warning* **Style of function definition is now obsolete**
In C++, this old C style of function definition is illegal:
- ```
int func(p1, p2)
int p1, p2;
{
:
}
```
- This practice may not be allowed by other C++ compilers.

- Compile-time error*    **Subscripting missing ]**  
 The compiler encountered a subscripting expression which was missing its closing bracket. This could be caused by a missing or extra operator, or mismatched parentheses.
- Compile-time warning*    **Superfluous & with function**  
 An address-of operator (&) is not needed with function name; any such operators are discarded.
- Compile-time warning*    **Suspicious pointer conversion**  
 The compiler encountered some conversion of a pointer which caused the pointer to point to a different type. You should use a cast to suppress this warning if the conversion is proper.
- Compile-time error*    **Switch selection expression must be of integral type**  
 The selection expression in parentheses in a **switch** statement must evaluate to an integral type (**char, short, int, long, enum**). You may be able to use an explicit cast to satisfy this requirement.
- Compile-time error*    **Switch statement missing (**  
 In a **switch** statement, the compiler found no left parenthesis after the **switch** keyword.
- Compile-time error*    **Switch statement missing )**  
 In a **switch** statement, the compiler found no right parenthesis after the test expression.
- TLINK fatal error*    **filename (linenum): Syntax error**  
 TLINK found a syntax error in the module definition file. The filename and line number tell you where the syntax error occurred.
- TLINK fatal error*    **Table limit exceeded**  
 One of linker's internal tables overflowed. This usually means that the programs being linked have exceeded the linker's capacity for public symbols, external symbols, or for logical segment definitions. Each instance of a distinct segment name in an object file counts as a logical segment; if two object files define this segment, then this results in two logical segments.
- Compile-time error*    **Template argument must be a constant expression**  
 A non-type actual template class argument must be a constant expression (of the appropriate type); this includes constant integral expressions, and addresses of objects or functions with external linkage or members.



Compile-time error

### Template class nesting too deep: 'class'

The compiler imposes a certain limit on the level of template class nesting; this limit is usually only exceeded through a recursive template class dependency. When this nesting limit is exceeded, the compiler will issue this error message for all of the nested template classes, which usually makes it easy to spot the recursion. This is always followed by the fatal error **Out of memory**.

For example, consider the following set of template classes:

```
template<class T> class A
{
 friend class B<T*>;
};

template<class T> class B
{
 friend class A<T>;
};

A<int> x;
```

This snippet will be flagged with the following errors:

```
Error: Template class nesting too deep: 'B<int * * * * *>'
Error: Template class nesting too deep: 'A<int * * * * *>'
Error: Template class nesting too deep: 'B<int * * * * *>'
Error: Template class nesting too deep: 'A<int * * * * *>'
Error: Template class nesting too deep: 'B<int * * * * *>'
Error: Template class nesting too deep: 'A<int * * * * *>'
Error: Template class nesting too deep: 'B<int * * * * *>'
Error: Template class nesting too deep: 'A<int * * * * *>'
Error: Template class nesting too deep: 'B<int * * * * *>'
Error: Template class nesting too deep: 'A<int * * * * *>'
Fatal: Out of memory
```

Compile-time error

### Template function argument *argument* not used in argument types

The given argument was not used in the argument list of the function. The argument list of a template function must use all of the template formal arguments; otherwise, there is no way to generate a template function instance based on actual argument types.

Compile-time error

### Template functions may only have type-arguments

A function template was declared with a non-type argument. This is not allowed with a template function, as there is no way to specify the value when calling it.

- Compile-time error* **Templates can only be declared at file level**  
 Templates cannot be declared inside classes or functions, they are only allowed in the global scope (file level).
- Compile-time error* **Templates must be classes or functions**  
 The declaration in a template declaration must specify either a class type or a function.
- Compile-time warnings* **Temporary used to initialize *identifier***  
**Temporary used for parameter *number* in call to function**  
**Temporary used for parameter *parameter* in call to function**  
**Temporary used for parameter *number***  
**Temporary used for parameter *parameter***  
 In C++, a variable or parameter of reference type must be assigned a reference to an object of the same type. If the types do not match, the actual value is assigned to a temporary of the correct type, and the address of the temporary is assigned to the reference variable or parameter. The warning means that the reference variable or parameter does not refer to what you expect, but to a temporary variable, otherwise unused.
- For example, here function **f** requires a reference to an **int**, and **c** is a **char**:
- ```
f(int&);
char c;
f(c);
```
- Instead of calling **f** with the address of **c**, the compiler generates code equivalent to the C++ source code:
- ```
int X = c, f(X);
```
- TLINK fatal error* **Terminated by user**  
 You canceled the link.
- TLIB error* **The combinations '+\*' or '\*+' are not allowed**  
 It is not legal to add and extract an object module from a library in one action. The action probably desired is a '+-'.
- Compile-time error* **The constructor *constructor* is not allowed**  
 Constructors of the form **X::(X)** are not allowed. The correct way to write a copy constructor is **X::(const X&)**.
- Compile-time error* **The value for *identifier* is not within the range of an int**  
 All enumerators must have values which can be represented as an integer. You attempted to assign a value which is out of the

range of an integer. In C++ if you need a constant of this value, use a **const** integer.

*Compile-time error* **'this' can only be used within a member function**

In C++, **this** is a reserved word that can be used only within class member functions.

*Compile-time warning* **This initialization is only partly bracketed**

Result of IDE Options | Compiler | Messages | ANSI violations selection. Initialization is only partially bracketed. When structures are initialized, braces can be used to mark the initialization of each member of the structure. If a member itself is an array or structure, nested pairs of braces may be used. This ensures that your idea and the compiler's idea of what value goes with which member are the same. When some of the optional braces are omitted, the compiler issues this warning.

*Compile-time error* **Too few arguments in template class name *template***

A template class name was missing actual values for some of its formal parameters.

*Compile-time error* **Too few parameters in call**

A call to a function with a prototype (via a function pointer) had too few arguments. Prototypes require that all parameters be given.

*Compile-time error* **Too few parameters in call to *function***

A call to the named function (declared using a prototype) had too few arguments.

*Compile-time error* **Too many decimal points**

The compiler encountered a floating-point constant with more than one decimal point.

*Compile-time error* **Too many default cases**

The compiler encountered more than one **default** statement in a single **switch**.

*Compile-time error* **Too many error or warning messages**

A maximum of 255 errors and warnings can be set before the compiler stops.

*TLINK error* **Too many error or warning messages**

The number of messages reported by the compiler has exceeded its limit. This error indicates that TLINK reached its limit.

- Compile-time error*    **Too many exponents**  
The compiler encountered more than one exponent in a floating-point constant.
- Compile-time error*    **Too many initializers**  
The compiler encountered more initializers than were allowed by the declaration being initialized.
- Compile-time error*    **Too many storage classes in declaration**  
A declaration may never have more than one storage class.
- MAKE error*    **Too many suffixes in .SUFFIXES list**  
You have exceeded the 255 allowable suffixes in the suffixes list.
- Compile-time error*    **Too many types in declaration**  
A declaration may never have more than one of the basic types: **char**, **int**, **float**, **double**, **struct**, **union**, **enum**, or **typedef-name**.
- Compile-time error*    **Too much global data defined in file**  
The sum of the global data declarations exceeds 64K bytes. Check the declarations for any array that may be too large. Also consider reorganizing the program or using **far** variables if all the declarations are needed.
- Compile-time error*    **Trying to derive a far class from the huge base *base***  
If a class is declared (or defaults to) **huge**, all derived classes must also be **huge**.
- Compile-time error*    **Trying to derive a far class from the near base *base***  
If a class is declared (or defaults to) **near**, all derived classes must also be **near**.
- Compile-time error*    **Trying to derive a huge class from the far base *base***  
If a class is declared (or defaults to) **far**, all derived classes must also be **far**.
- Compile-time error*    **Trying to derive a huge class from the near base *base***  
If a class is declared (or defaults to) **near**, all derived classes must also be **near**.
- Compile-time error*    **Trying to derive a near class from the far base *base***  
If a class is declared (or defaults to) **far**, all derived classes must also be **far**.
- Compile-time error*    **Trying to derive a near class from the huge base *base***  
If a class is declared (or defaults to) **huge**, all derived classes must also be **hugh**.





*Compile-time error* **Two consecutive dots**  
Because an ellipsis contains three dots (...), and a decimal point or member selection operator uses one dot (.), there is no way two consecutive dots can legally occur in a C program.

*Compile-time error* **Two operands must evaluate to the same type**  
The types of the expressions on both sides of the colon in the conditional expression operator (?:) must be the same, except for the usual conversions like **char** to **int** or **float** to **double**, or **void\*** to a particular pointer. In this expression, the two sides evaluate to different types that are not automatically converted. This may be an error or you may merely need to cast one side to the type of the other.

*Type mismatch family*



When compiling C++ programs, the following messages that refer to this note are always preceded by another message that explains the exact reason for the type mismatch; this is usually "Cannot convert 'type1' to 'type2'", but the mismatch may be due to many other reasons.

*Compile-time error* **Type mismatch in default argument value**  
**Type mismatch in default value for parameter *parameter***

The default parameter value given could not be converted to the type of the parameter. The first message is used when the parameter was not given a name.

*Compile-time error* **Type mismatch in parameter *number***

The function called, via a function pointer, was declared with a prototype; the given parameter *number* (counting left to right from 1) could not be converted to the declared parameter type. See the previous note on Type mismatch family.

*Compile-time error* **Type mismatch in parameter *number* in call to *function***

Your source file declared the named function with a prototype, and the given parameter *number* (counting left to right from 1) could not be converted to the declared parameter type. See the previous note on Type mismatch family.

*Compile-time error* **Type mismatch in parameter *parameter***

Your source file declared the function called via a function pointer with a prototype, and the named parameter could not be converted to the declared parameter type. See the previous note on Type mismatch family.

- Compile-time error* **Type mismatch in parameter *parameter* in call to function**  
Your source file declared the named function with a prototype, and the named parameter could not be converted to the declared parameter type. See entry for **Type mismatch in parameter *parameter***.
- Compile-time error* **Type mismatch in parameter *parameter* in template class name *template***  
**Type mismatch in parameter *number* in template class name *template***  
The actual template argument value supplied for the given parameter did not exactly match the formal template parameter type. See the previous note on Type mismatch family.
- Compile-time error* **Type mismatch in redeclaration of *identifier***  
Your source file redeclared with a different type than was originally declared. This can occur if a function is called and subsequently declared to return something other than an integer. If this has happened, you must declare the function before the first call to it.
- Compile-time error* **Type name expected**  
One of these errors has occurred:
- In declaring a file-level variable or a **struct** field, neither a type name nor a storage class was given.
  - In declaring a **typedef**, no type for the name was supplied.
  - In declaring a destructor for a C++ class, the destructor name was not a type name (it must be the same name as its class).
  - In supplying a C++ base class name, the name was not the name of a class.
- Compile-time error* **Type qualifier *identifier* must be a struct or class name**  
The C++ qualifier in the construction *qual::identifier* is not the name of a **struct** or **class**.
- Compile-time fatal error* **Unable to create output file *filename***  
The work disk is full or write-protected or the output directory does not exist. If the disk is full, try deleting unneeded files and restarting the compilation. If the disk is write-protected, move the source files to a writable disk and restart the compilation.
- Compile-time error* **Unable to create turboc.\$ln**  
The compiler cannot create the temporary file TURBOC.\$LN because it cannot access the disk or the disk is full.



*MAKE fatal error* **Unable to execute command**  
A command failed to execute; this may be because the command file could not be found, it was misspelled, there was no disk space left in the specified swap directory, swap directory does not exist, or (less likely) because the command itself exists but has been corrupted.

*Compile-time error* **Unable to execute command *command***  
TLINK or TASM cannot be found, or possibly the disk is bad.

*TLINK fatal error and TLIB error* **Unable to open file *filename***  
**unable to open *filename***  
This occurs if the named file does not exist or is misspelled.

*TLIB error* **unable to open *filename* for output**  
TLIB cannot open the specified file for output. This is usually due to lack of disk space for the target library, or a listing file. Additionally this error will occur if the target file exists but is marked as a read only file.

*Compile-time error* **Unable to open include file *filename***  
The compiler could not find the named file. This could also be caused if an **#include** file included itself, or if you do not have FILES set in CONFIG.SYS on your root directory (try FILES=20). Check whether the named file exists.

*MAKE error* **Unable to open include file *filename***  
The compiler could not find the named file. This could also be caused if an **!include** file included itself, or if you do not have FILES set in CONFIG.SYS on your root directory (try FILES=20). Check whether the named file exists.

*Compile-time error* **Unable to open input file *filename***  
This error occurs if the source file cannot be found. Check the spelling of the name and whether the file is on the proper disk or directory.

*Command line fatal error* **unable to open 'dpmimem.dll'**  
Make sure that DPMIMEM.DLL is somewhere on your path or in the same directory as the protected mode command line tool you were attempting to use.

*MAKE fatal error* **Unable to open makefile**  
The current directory does not contain a file named MAKEFILE, MAKEFILE.MAK, or does not contain the file you specified with **-f**.

- MAKE fatal error*    **Unable to redirect input or output**  
 MAKE was unable to open the temporary files necessary to redirect input or output. If you are on a network, make sure you have rights to the current directory.
- TLIB error*    **unable to rename filename to filename**  
 TLIB builds a library into a temporary file and then renames the temporary file to the target library file name. If there is an error, usually due to lack of disk space, this message will be posted.
- Compile-time error*    **Undefined label identifier**  
 The named label has a **goto** in the function, but no label definition.
- Compile-time warning*    **Undefined structure identifier**  
 The named structure was used in the source file, probably on a pointer to a structure, but had no definition in the source file. This is probably caused by a misspelled structure name or a missing declaration.
- Compile-time error*    **Undefined structure structure**  
 Your source file used the named structure on some line before where the error is indicated (probably on a pointer to a structure) but had no definition for the structure. This is probably caused by a misspelled structure name or a missing declaration.
- Compile-time error*    **Undefined symbol identifier**  
 The named identifier has no declaration. This could be caused by a misspelling either at this point or at the declaration. This could also be caused if there was an error in the declaration of the identifier.
- TLINK error*    **Undefined symbol symbol in module module**  
 The named symbol is referenced in the given module but is not defined anywhere in the set of object files and libraries included in the link. Check to make sure the symbol is spelled correctly.
- You will usually see this error from TLINK for Borland C++ symbols if you did not properly match a symbol's declarations of **pascal** and **cdecl** type in different source files, or if you have omitted the name of an .OBJ file your program needs. If you are linking C++ code with C modules, you might have forgotten to wrap C external declarations in `extern "C" {...}`.



You might have a case mismatch between two symbols. See the /C and /c switches.

- Compile-time error* **Unexpected }**  
An extra right brace was encountered where none was expected. Check for a missing {.
- TLIB error* **Unexpected char X in command line**  
TLIB encountered a syntactical error while parsing the command line.
- MAKE error* **Unexpected end of file**  
The end of the makefile was reached without a temporary inline file having been closed.
- Compile-time error* **Unexpected end of file in comment started on line number**  
The source file ended in the middle of a comment. This is normally caused by a missing close of comment (\*/\*).
- MAKE or compile-time error* **Unexpected end of file in conditional started on line line number**  
The source file ended before the compiler (or MAKE) encountered an !endif. The !endif was either missing or misspelled.
- Compile-time error* **union cannot be a base type**  
A union cannot be used as a base type for another class type.
- Compile-time error* **union cannot have a base type**  
A union cannot be derived from any other class.
- Compile-time error* **Union member member is of type class with constructor**  
**Union member member is of type class with destructor**  
**Union member member is of type class with operator=**  
A union may not contain members that are of type **class** with user-defined constructors, destructors, or operator=.
- Compile-time error* **unions cannot have virtual member functions**  
A union may not have virtual functions as its members.
- Compile-time warning* **Unknown assembler instruction**  
The compiler encountered an inline assembly statement with a disallowed opcode. Check the spelling of the opcode. This warning is off by default.
- See Chapter 12 in the Programmer's Guide for more on opcode spelling.*
- TLIB warning* **unknown command line switch X ignored**  
A forward slash character (/) was encountered on the command line or in a response file without being followed by one of the allowed options.

*Compile-time error*    **Unknown language, must be C or C++**

In the C++ construction

```
extern "name" type func(/*...*/);
```

The name given in quotes must be "C" or "C++"; other language names are not recognized. For example, you can declare an external Pascal function without the compiler's renaming like this:

```
extern "C" int pascal func(/*...*/);
```

A C++ (possibly overloaded) function may be declared Pascal and allow the usual compiler renaming (to allow overloading) like this:

```
extern int pascal func(/*...*/);
```

*TLINK fatal error*    **Unknown option**

A forward slash character (/), hyphen (-), or DOS switch character was encountered on the command line or in a response file without being followed by one of the allowed options. This might mean that you used the wrong case to specify an option.

*Compile-time error*    **Unknown preprocessor directive: *identifier***

The compiler encountered a # character at the beginning of a line, and the name following was not a legal directive name, or the rest of the directive was not well-formed.

*MAKE error*    **Unknown preprocessor statement**

A ! character was encountered at the beginning of a line, and the statement name following was not **error**, **undef**, **if**, **elif**, **include**, **else**, or **endif**.

*Compile-time warning*    **Unreachable code**

A **break**, **continue**, **goto** or **return** statement was not followed by a label or the end of a loop or function. The compiler checks **while**, **do** and **for** loops with a constant test condition, and attempts to recognize loops which cannot fall through.

*Compile-time error*    **Unterminated string or character constant**

The compiler found no terminating quote after the beginning of a string or character constant.

*Compile-time error*    **Use . or -> to call function**

You tried to call a member function without giving an object.



- Compile-time error*    **Use . or -> to call *member*, or & to take its address**  
A reference to a non-static class member without an object was encountered. Such a member may not be used without an object, or its address must be taken using the & operator.
- Compile-time error*    **Use :: to take the address of a member function**  
If *f* is a member function of class *c*, you take its address with the syntax **&c::f**. Note the use of the class type name, not the name of an object, and the **::** separating the class name from the function name. (Member function pointers are not true pointer types, and do not refer to any particular instance of a class.)
- TLIB warning*    **use /e with TLINK to obtain debug information from library**  
The library was built with an extended dictionary and also includes debugging information. TLINK will not extract debugging information if it links using an extended dictionary, so in order to obtain debugging information in an executable from this library, the linker must be told to ignore the extended dictionary using the /e switch. NOTE: The IDE linker does NOT support extended dictionaries therefore no settings need be altered in the IDE.
- MAKE error*    **Use of : and :: dependants for target *target***  
You have tried to use the target in both single and multiple description blocks (using both the : and :: operators).  
Examples:  

```
filea: fileb
filea:: filec
```
- Compile-time warning*    **Use qualified name to access nested type *type***  
In older versions of the C++ specification, typedef and tag names declared inside classes were directly visible in the global scope. With the latest specification of C++, these names must be prefixed with a **class::** qualifier if they are to be used outside of their class' scope. To allow older code to compile, whenever such a name is uniquely defined in one single class, Borland C++ will allow its usage without **class::** and issues this warning.
- TLINK or compile-time error*    **User break**  
You pressed *Ctrl-Break* while compiling or linking in the IDE, aborting the process. (This is not an error, just a confirmation.)

|                                |                                                                                                                                                                                                                                                                                                                                                            |
|--------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <i>DPMI server fatal error</i> | <b>v86 task without vcpi</b><br>Another application is running, preventing the DPMI server from switching to protected mode. Remove the interfering application, such as a desktop manager or debugger, then reboot.                                                                                                                                       |
| <i>Compile-time error</i>      | <b>Value of type void is not allowed</b><br>A value of type <b>void</b> is really not a value at all, and thus may not appear in any context where an actual value is required. Such contexts include the right side of an assignment, an argument of a function, and the controlling expression of an <b>if</b> , <b>for</b> , or <b>while</b> statement. |
| <i>Compile-time error</i>      | <b>Variable <i>variable</i> has been optimized.</b><br>You have tried to inspect, watch, or otherwise access a variable which the optimizer removed. This variable is never assigned a value and has no stack location.                                                                                                                                    |
| <i>Compile-time error</i>      | <b>Variable <i>identifier</i> is initialized more than once</b><br>This variable has more than one initialization. It is legal to declare a file level variable more than once, but it may have only one initialization (even if two are the same).                                                                                                        |
| <i>Compile-time error</i>      | <b>'virtual' can only be used with member functions</b><br>A data member has been declared with the <b>virtual</b> specifier; only member functions may be declared <b>virtual</b> .                                                                                                                                                                       |
| <i>Compile-time error</i>      | <b>Virtual function <i>function1</i> conflicts with base class <i>base</i></b><br>A virtual function has the same argument types as one in a base class, but a different return type. This is illegal.                                                                                                                                                     |
| <i>Compile-time error</i>      | <b>virtual specified more than once</b><br>The C++ reserved word <b>virtual</b> may appear only once in a member function declaration.                                                                                                                                                                                                                     |
| <i>Compile-time error</i>      | <b>void &amp; is not a valid type</b><br>A reference always refers to an object, but an object cannot have the type void. Thus the type void is not allowed.                                                                                                                                                                                               |
| <i>Compile-time warning</i>    | <b>Void functions may not return a value</b><br>Your source file declared the current function as returning <b>void</b> , but the compiler encountered a return statement with a value. The value of the return statement will be ignored.                                                                                                                 |
| <i>Compile-time error</i>      | <b>function was previously declared with the language <i>language</i></b><br>Only one language can be used with <b>extern</b> for a given function. This function has been declared with different languages in different locations in the same module.                                                                                                    |





- Compile-time error*    **While statement missing (**  
In a **while** statement, the compiler found no left parenthesis after the **while** keyword.
- Compile-time error*    **While statement missing )**  
In a **while** statement, the compiler found no right parenthesis after the test expression.
- TLINK fatal error*    **Write failed, disk full?**  
This occurs if TLINK could not write all of the data it attempted to write. This is almost certainly caused by the disk being full.
- Compile-time error*    **Wrong number of arguments in call of macro *macro***  
Your source file called the named macro with an incorrect number of arguments.

**\*\*** (all dependents macro) 34  
**?** (all out of date dependents macro) 34  
 - + and + - (TLIB action symbols) 49  
**\*** (base file name macro) 32  
**.** (file name and extension macro) 33  
**&** (file name only macro) 33  
**:** (file name path macro) 33  
**<** (full file name macro) 32  
**@** (full name with path macro) 33  
 -? MAKE help option 12  
 /3 TLINK option (32-bit code) 63  
 & (ampersand) MAKE command (multiple dependents) 18  
 -\* and \*- (TLIB action symbols) 48  
 32-bit code 63  
 - (hyphen) MAKE command (ignore exit status) 18  
 # (MAKE comment character) 17  
 && operator, MAKE 18  
 << operator, MAKE 18  
 >> operator, MAKE 18  
 -? RC help option 100  
 \* (TLIB action symbol) 48  
 + (TLIB action symbol) 48  
 - (TLIB action symbol) 48  
 \$ editor macros *See* individual names of macros  
 @ MAKE command 18  
 80x86 processors  
     32-bit code 63  
 #include directive *See* include files  
 .LST files *See* listfile (TLIB option)  
 /P TLINK option (pack code segments) 69

## A

-a MAKE option (autodependency check) 12, 25  
 /A TLINK option (align segments) 63  
 ACBP field 67

action symbols *See* TLIB (librarian)  
 add (TLIB action symbol) 48  
 Alias section 137  
 alignment attribute 67  
 ampersand (&) MAKE command (multiple dependents) 18  
 applications *See* Microsoft Windows applications  
 attributes 67  
 .autodepend MAKE directive 36  
 automatic dependencies  
     checking, MAKE (program manager) 12, 25  
     MAKE option 36

## B

-B MAKE option (build all) 12  
 base file name macro (MAKE) 32  
 batch files, MAKE 20  
 BBS segment *See* segments  
 BCC.EXE *See* command-line compiler  
 BGI OBJ *See* The online document UTIL.DOC  
 big attribute 67  
 Bitmaps section 140  
 bugs *See* debugging  
 BUILD option 132  
 BuildTags section 131  
 BUILTINS.MAK 13

## C

/C TLIB option (case sensitivity) 47, 50  
 /C TLINK option (case sensitive imports) 64  
 /c TLINK option (case sensitivity) 63  
 C0Fx.OBJ 59  
 C0x.OBJ 58  
 callbacks  
     smart *See* smart callbacks  
 case sensitivity  
     module definition file and 64

- TLIB option 47, 50
- TLINK and 63
- classes, DLLs and 4
- code segment, discardable 68
- .COM files
  - generating 58
  - TLINK 69
  - limitations 69
- combining attribute 67
- command-line compiler
  - MAKE and 36
  - options
    - v (debugging information) 65
  - TLINK and 62
- commands
  - printing, MAKE option 36
- comments, in makefiles 17
- compatibility
  - initialization modules 59
  - MAKE 12
- compilers
  - command line *See* command-line compiler
  - diagnostic messages 154-240
  - memory models 59-61, *See* memory models
- COMPRESS option 136
- conditional execution directives (MAKE) 38
  - expressions in 40
- coprocessors 60
- CPP (preprocessor) *See* The online document
- UTIL.DOC
- CWx.LIB 60
- Cx.LIB 60

## D

- \$d MAKE macro (defined test) 31
  - expressions and 41
- D MAKE option (define identifier) 12, 28
- d RC option (define symbol) 100
- /d TLINK option (duplicate symbols) 64
- data types
  - floating point 60
- debugging
  - information 65
  - MAKE 12
  - map files 66
  - TLINK and 70
- defined test macro (MAKE) 31

- dependencies, automatic 25
- diagnostic messages
  - compiler 154-158
- directional delimiters *See* delimiters
- directives 17
  - MAKE *See* MAKE(program manager), directives
  - MAKE (program manager) 35-42
- directories
  - include files, MAKE 12
- DLLs *See also* import libraries
  - classes and 4
  - export functions, hiding 5
  - extended and expanded memory 100
  - import libraries (IMPLIB) and 5, 6
  - MAKE and 60
  - mangled names and 4
  - packing code segments 69
  - private 100
  - TLINK option 70

## DOS

- commands
  - MAKE and 21
- environment strings, macros 30
- paths
  - MAKE 36
- dot directives (MAKE) 36
- DPMI server messages 155
- duplicate symbols 64
- dynamic link libraries *See* DLLs

## E

- e MAKE option 12
- e RC option (EMS) 100
- /E TLIB option (extended dictionary) 47, 49
- /e TLINK option 65
- editor macros *See* MAKE (program manager)
- !elif MAKE directive 38
  - defined test macro and 31
  - macros and 30
- !else MAKE directive 38
- EMS *See* extended and expanded memory
- EMU.LIB 60, 61
- !endif MAKE directive 38
- environment, DOS
  - macros and 30
- !error MAKE directive 41

## errors

- command line
    - defined 154
  - compiler 154-240
  - disk access 154
  - DPMI server 155
  - fatal 154
  - Help compiler 155
  - MAKE 157
  - memory access
    - defined 154
  - messages
    - list 154-240
  - run-time 157
  - syntax
    - defined 154
  - TLIB 157
  - TLINK (list) 158
  - undocumented 155
- examples
- MAKE (program manager) 13
    - batch files 20
- .EXE files
- .COM files and 71
  - debugging information 70
  - renaming 100
  - TLINK and 70
- executable files *See* .EXE files
- exit codes
- MAKE and 18
- expanded memory
- TLINK and 71
- explicit
- rules (MAKE) 16, 22
- expressions *See* debugging
- MAKE and 40, 41
- extended and expanded memory
- DLLs and 100
  - Resource Compiler and 100
- extended dictionary
- TLIB and 47, 49
- extended memory
- TLINK and 71
- extensions, file, supplied by TLINK 56
- extract and remove (TLIB action) 48

## F

- f MAKE option (MAKE file name) 10, 12
- fatal errors *See* errors
  - Compile-time 154
- fe RC option (rename .EXE file) 100
- file-inclusion directive (!include) 38
- file name macros (MAKE) 33, 34
- files
  - batch 20
  - .COM 58, 69
    - .EXE files and 71
    - TLINK and 70, 71
  - executable *See* .EXE files
  - extensions 56
  - include *See* include files
  - library *See* libraries
  - make *See* MAKE (program manager)
  - map *See* map files
  - names
    - extensions (meanings) 56
  - .RES 98
  - response 48, 56, *See* response files
  - updating 9
- Files section
  - Help project file 130
- fo RC option (rename .RES file) 100
- FORCEFONT option 134
- FP87.LIB 61
- full file name macro (MAKE) 32

## G

- graphics, library
  - TLINK and 60
- GREP *See* The online document UTIL.DOC

## H

- h MAKE option (help) 12
- h RC option (help on options) 100
- header files *See* include files
- help, MAKE 12
- hyphen (-) MAKE command (ignore exit status) 18

## I

- i MAKE option (ignore exit status) 12

- I MAKE option (include files directory) *12, 13*
- i RC option (include files) *100*
- /i TLINK option (uninitialized trailing segments) *65*
- identifiers
  - defining *28*
- !if MAKE directive *38*
  - defined test macro and *31*
  - macros and *30*
- !ifdef MAKE directive *38*
- !ifndef MAKE directive *38*
- ignore exit status (MAKE command) *18*
- .ignore MAKE directive *36*
- IMPDEF (module definition files) *3-5*
  - IMPLIB and *3*
- IMPLIB (import libraries) *6-7*
  - DLLs and *6*
  - IMPDEF and *3*
  - warnings *6*
- \$IMPLIB *See* import libraries
- implicit
  - rule (MAKE) *16*
- import libraries *6-7, See also* DLLs
  - creating new *7*
  - customizing *3*
  - DLLs and *5, 6*
- !include directive (MAKE) *13, 38*
- include files
  - automatic dependency checking (MAKE) *25*
  - MAKE *13, 38*
    - directories *12*
    - Resource Compiler and *100*
- INDEX option *134*
- initialization modules
  - compatibility *59*
  - used with TLINK *58, 59*
- integrated debugger *See* debugging

## K

- K MAKE option (keep temporary files) *12, 19*
- k RC option (disable load optimization) *100*

## L

- l RC option (expanded memory) *100*
- /l TLINK option (line numbers) *65*
- .LIB files *See* libraries

- libname (TLIB option) *47*
- librarian *See* TLIB
- libraries
  - duplicate symbols in *64*
  - dynamic link (DLL) *See* DLLs
  - floating point
    - TLINK and *60*
  - graphics
    - TLINK and *60*
  - import *See* import libraries
  - memory models and *59-61*
  - numeric coprocessor *60*
  - object files *45, 46*
    - creating *48*
    - order of use *60*
    - page size *50*
    - run time
      - TLINK and *61*
      - TLINK and *58, 59*
      - ignoring *68*
    - utility *See* TLIB
    - Windows applications and *60*
- lim32 RC option (expanded memory) *100*
- line numbers, TLINK and *65*
- linker *See* TLINK
- listfile (TLIB option) *47*
- load optimization
  - disabling (Resource Compiler) *100*
- .LST files *47*

## M

- m MAKE option (display time/date stamp) *12*
- m RC option (expanded memory) *100*
- macros *See* MAKE (program manager)
  - DOS
    - environment strings and *30*
    - path (MAKE) *36*
  - invocation
    - defined *29*
  - Turbo editor *See* The online document
  - UTIL.DOC

## MAKE

- NMAKE vs. *42*
- \_\_MAKE\_\_ macro *30*
- MAKE (program manager)
  - automatic dependency checking *12, 25*
  - batching files and *20*

- BUILTINS.MAK file 13
- clocks and 10
- commands
  - @ (hide commands) 18
  - ampersand (&) (multiple dependents) 18
  - hiding (@) 18
  - hyphen (-) (ignore exit status) 18
  - num (stop on exit status num) 18
- compatibility 12
- debugging 12
- directives
  - .autodepend 36
  - command-line compiler options and 36
  - conditional execution 38
    - expressions in 40
  - defined 35
  - dot 36
  - !elif 38
    - macros and 30
  - !else 38
  - !endif 38
  - !error 41
  - file inclusion 38
  - !if 38
    - macros and 30
  - !ifdef 38
  - !ifndef 38
  - .ignore 36
  - !include 38
  - .noautodepend 36
  - .noignore 36
  - .nosilent 36
  - .noswap 36
  - .silent 36
  - .swap 36
  - !undef 42
- DLLs and 60
- DOS commands and 21
- example 13
- exit codes and 18
- explicit rules *See* MAKE (program manager)
- external commands and 21
- functionality 10
- hide commands 18
- implicit rules *See* MAKE (program manager)
- !include directive 13
- macros 18, 26, 28, 30
  - \$? 18
  - \*\* 18
  - all dependents (\*\*) 34
  - all out of date dependents (\$) 34
  - base file name (\*) 32
  - defined test 31
  - !elif directive and 30, 31
  - example 26
  - file name and extension (\$) 33
  - file name only (&) 33
  - file name path (\$) 33
  - full file name (<) 32
  - full name with path (&) 33
  - !if directive and 30, 31
  - in expressions \$d 41
  - \_\_MAKE\_\_ 30
  - predefined 30
  - undefining 42
  - version number 30
- makefiles
  - comments in 17
  - creating 16
  - defined 14
  - naming 16
  - parts of 16
- multiple dependents and 18
- operators 41
- options 11
  - ? (help) 12
  - automatic dependency checking (-a) 25
  - build all (-B) 12
  - default (-w) 12
  - define identifier (-D) 12
    - conditional execution 38
  - display rules (-p) 12
  - display time/date stamp (-m) 12
  - don't print commands (-s) 12
  - environment variables(-e) 12
  - file name (-f) 10, 12
  - help (-? and -h) 12
  - ignore BUILTINS.MAK (-r) 12
  - ignore exit status (-i) 12
  - include files directory (-I) 12, 13
  - keep files (-K) 12, 19
  - N (increase compatibility) 12
  - n (print commands but don't execute) 12
  - saving (-w) 12

- swap MAKE out of memory (-S) 12
- undefine (-U) 12
- using 11
- W (save options) 12
- .path directive 36
- .precious directive 36
- printing commands 36
- redirection operators 18
- rules
  - explicit
    - considerations 23
    - defined 22
    - example 17, 24
  - implicit 15
    - discussion 25
    - example 17
- swapping in memory 36
- syntax 11
- wildcards and 22
- Windows applications and 60
- MAKE.EXE (protected-mode MAKE) 10
- makefiles *See* MAKE (program manager)
- MAKER.EXE (real-mode MAKE) 10
- mangled names
  - DLLs and 4
- map 66
- map files
  - debugging 66
  - generated by TLINK 66
- Map section 138
- MAPFONTSIZE option 135
- math coprocessors 60
- MATHx.LIB 60
- memory
  - extended and expanded *See* extended and expanded memory
  - swapping MAKE in 36
- memory models 59-61
- messages *See also* errors; warnings
  - Compile-time 154
  - DPMI 155
  - Help compiler 155
  - MAKE 157
  - run-time 157
  - TLIB 157
  - TLINK (list) 158
  - tracing 88

- Microsoft Windows applications
  - code segments 69
  - import libraries 3
  - MAKE and 60
  - modes 100
  - overlays and 68
  - TLINK and 62
  - TLINK option 70
- Microsoft Windows Help
  - appearance to programmer 104
  - appearance to user 102, 103
  - appearance to writer 103, 104
  - audience definition 104, 105
  - bitmaps 124-126
  - calling WinHelp 142, 143
  - canceling 148, 149
  - compiler 140-141
  - context-sensitive 107-108, 143-147
  - control codes 114
  - development cycle described 101, 102
  - F1 support 146, 147
  - file structure 108-110
  - graphics 112
  - keywords 106, 118-120
  - keywords table
    - accessing 147, 148
  - on Help menu item 147
  - planning, overview 104
  - tracker 127-128
- Microsoft Windows Help Project file
  - accessing from an application 142
  - Alias section 137, 138
  - bitmaps, including by reference 140
  - Bitmaps section 140
  - BUILD option 132, 133
  - BuildTags section 131
  - compiling 140, 141
  - COMPRESS option 136, 137
  - context-sensitive Help 143, 144, 145, 146, 147
  - context-sensitive topics 138, 139, 140
  - context strings, alternate 137, 138
  - creating 129, 130
  - F1 support 146, 147
  - Files section 130, 131
  - FORCEFONT option 135
  - INDEX option 134

- keyword table, accessing 147, 148
- Map section 138, 139, 140
- MAPFONTSIZE option 135, 136
- MULTIKEY option 136
- on Help menu item 147
- Options section 131, 132
- ROOT option 133, 134
- TITLE option 134
- WARNING option 132-133
- Microsoft Windows Help text
  - fonts 111, 112
  - layout 110, 111
- Microsoft Windows Help topic files
  - authoring tool 113
  - browse sequence numbers 120, 121, 122
  - build tags 115, 116
  - context strings 116, 117
  - control codes 114
  - cross references 122
  - definitions 123, 124
  - graphics 124
  - jumps 122
  - keywords 118, 119, 120
  - managing 127
  - title footnotes 117, 118
  - tracking 127, 128
- Microsoft Windows Help topics
  - content 105
  - context-sensitivity 107, 108
  - cross-references 122
  - definitions 123, 124
  - file structure 108, 109, 110
  - jumps 122
  - structure of 106, 107
- models, memory *See* memory models
- module definition file
  - case sensitivity and 64
- module definition files 72-81
  - IMPDEF and 3
  - /Tw TLINK option and 70
- module names, TLIB 48
- MULTIKEY option 136
- multinst RC option (expanded memory) 100
- multiple dependents
  - MAKE and 18

## N

- N MAKE option (increase compatibility) 12
- n MAKE option (print commands but don't execute) 12
- /n TLINK option (ignore default libraries) 68
- names *See* identifiers
- NMAKE
  - MAKE vs. 42
  - .noautodepend MAKE directive 36
  - .noignore MAKE directive 36
  - nondirectional delimiters *See* delimiters
  - nonfatal errors *See* errors
  - .nosilent MAKE directive 36
  - .noswap MAKE directive 36
  - num MAKE command 18
  - numeric coprocessors
    - TLINK and 60

## O

- /o TLINK option (overlays) 68
  - /Tw option and 68
- .OBJ files (object files)
  - duplicate symbols in 64
  - libraries
    - advantages of using 46
    - creating 48
    - TLIB and 45
- OBJXREF *See* The online document UTIL.DOC
- operations
  - precedence 47
  - operations (TLIB option) 47
- operators
  - MAKE 18, 41
- optimizations, Resource Compiler 100
- options *See* specific entries (such as command-line compiler, options)
- Options section 131
- overlays
  - TLINK and 68
  - Windows applications and 68

## P

- p MAKE option 12
- p RC option (private DLLs) 100
- /P TLIB option (page size) 50
- page size (libraries) 50



pane

- classes 84

- messages 84

- windows 84

.path directive (MAKE) 36

PF87.LIB 60

precedence, TLIB commands 47

.precious directive (MAKE) 36

PRJ2MAK *See* The online document UTIL.DOC

PRJCFG *See* The online document UTIL.DOC

PRJCNVT *See* The online document UTIL.DOC

program manager (MAKE) *See* MAKE  
(program manager)

## R

-r RC option (compile .RC to .RES) 100

.RC files *See* Resource Compiler

real numbers *See* floating point

redirection operators, MAKE 18

remove (TLIB action) 48

replace (TLIB action) 49

.RES files 98, *See* resources

Resource Compiler 97

- .EXE files

  - renaming 100

- include files 100

- messages 100

- options 100

  - help 100

- resources, renaming files 100

- syntax 99

resources 97

- adding to executable 98

- compiling 97

- creating 97

- files 98

  - renaming 100

- loading 98

- types of 97

response files

- defined 56

- TLIB 49

- TLINK and 56

ROOT option 133

## S

-s MAKE option (don't print commands) 12

-S MAKE option (swap MAKE out of memory)  
12

segments

- aligning 63

- code

  - discardable 68

  - minimizing 69

  - packing 69

- map of

  - ACBP field and 67

  - TLINK and 66

- uninitialized, TLINK and 65

.silent MAKE directive 36

source files

- separately compiled 46

standalone utilities 1, *See also* MAKE (program manager); TLIB (librarian); TLINK (linker)

standard library files *See* libraries

startup code (TLINK) 59

startup modules for memory models 61

.swap MAKE directive 36

switches *See* command-line compiler

symbolic

- constants *See* macros

symbols

- action *See* TLIB

- duplicate warning (TLINK) 64

syntax

- MAKE 11

- Resource Compiler 99

- TLIB 46

- TLINK 54

## T

-t RC option (standard/386 mode) 100

/t TLINK option 58, 69

TASM *See* Turbo Assembler

/Td and /Tw TLINK options (target file) 70

TDSTRIP

- TLINK and 71

TEML *See* The online document UTIL.DOC

THELP *See* The online document UTIL.DOC

32-bit code 63

/3 TLINK option (32-bit code) 63

thunks *See* callbacks  
 TITLE option 134  
 TLIB (librarian) 45-52  
   action symbols 47-49  
   capabilities 45  
   examples 51  
   extended dictionary (/e)  
     TLINK and 50  
   module names 48  
   operations 47, 48  
     precedence 47  
   options  
     case sensitivity (/c) 47, 50  
     /E 47, 49  
     extended dictionary (/e) 47, 49  
     libname 47  
     listfile 47  
     operations 47  
     page size (/P) 50  
     using 46  
   response files  
     using 49  
   syntax 46  
 TLINK (linker)  
   ACBP field and 67  
   assembler code and 63  
   .COM files and 69, 71  
   command-line compiler and 62  
   debugging information 70, 71  
   executable file map generated by 66  
   floating-point libraries 60  
   graphics library and 60  
   initialization modules 59  
   invoking 53  
   libraries 59  
   memory models and 58  
   numeric coprocessor libraries 60  
   options 62  
     align segments (/A) 63  
     case sensitive imports (/C) 64  
     case sensitivity (/c) 63  
     .COM files (/t) 58, 69  
     .COM files (/Td and /Tw) 70  
     debugging information (/v) 70  
     DLLs (/Twe) 70  
     duplicate symbols warning (/d) 64  
     executable files (/Td and /Tw) 70  
     expanded memory (/ye) 71  
     extended dictionary (/e) 65  
     extended memory (/yx) 71  
     file extension 56, 58  
     /i (uninitialized trailing segments) 65  
     /l (source code line numbers) 65  
     libraries, ignoring (/n) 68  
     line numbers (/l) 65  
     map files (/m)  
       debugging 66  
       public symbols in 66  
       segments in 66  
     /n (ignore default libraries) 68  
     overlays (/o) 68  
     pack code segments (/P) 69  
     /s (map files) 66  
     source code line numbers (/l) 65  
     target files 70  
     /Td (Windows executable) 68  
     /Td and /Tw (target files) 70  
     32-bit assembler code and (/3) 63  
     tiny model .COM files (/t) 58, 69  
     /Tw (Windows executable) 68  
     uninitialized trailing segments (/i) 65  
     /v (debugging information) 70  
     Windows executable (/Td and /Tw) 68, 70  
     /x (map files) 66  
     /ye (expanded memory) 71  
     /yx (extended memory) 71  
   response files 56  
     example 57  
   segment limit 209  
   starting 53  
   startup code 59  
   syntax 54  
   target file options (/Td and /Tw) 70  
   TLIB extended dictionary and 50  
   Windows applications and 62  
   topic numbers, Help compiler 156  
   tracing, messages 88  
   trailing segments, uninitialized 65  
   TRIGRAPH *See* The online document  
     UTIL.DOC  
   Turbo Assembler  
     TLINK and 63  
   Turbo Editor Macro Language compiler *See*  
     The online document UTIL.DOC

## U

-U MAKE option (undefine) 12  
!undef MAKE directive 42  
utilities *See also* The online document  
    UTIL.DOC  
    standalone 1  
    TLIB 45-52

## V

-v option (debugging information) 65  
-v RC option (display compiler messages) 100  
/v TLINK option (debugging information) 70

## W

-W MAKE option (save options) 12  
WARNING option 132

warnings *See also* errors

    Compile-time 154  
    defined 154  
    Help compiler 155  
    IMPLIB 6  
    TLIB 157  
    TLINK 158

wildcards

    MAKE and 22

Windows *See* Microsoft Windows applications

## X

-x RC option (exclude include directories) 100

## Y

/ye TLINK option (expanded memory) 71  
/yx TLINK option (extended memory) 71

3.1

# BORLAND® C++

**B O R L A N D**

Corporate Headquarters: 1800 Green Hills Road, P.O. Box 660001, Scotts Valley, CA 95067-0001, (408) 438-5300. Offices in: Australia, Belgium, Canada, Denmark, France, Germany, Hong Kong, Italy, Japan, Korea, Malaysia, Netherlands, New Zealand, Singapore, Spain, Sweden, Taiwan and United Kingdom ■ Part #14MN-BCP02-31 ■ BOR 3859