Microsoft® QuickC®

Tool Kit

Microsoft®

Microsoft® QuickC®

TOOL KIT

VERSION 2.0

MICROSOFT CORPORATION

Printed and bound in the United States of America.

Document No. 410840031-200-R00-1088
Part No. 04320
10 9 8

Table of Contents Overview

Table of Contents

Introduction

PART 1 Tool Kit Tutorial

PART 2 Reference to QuickC Tools

Appendixes

Figures

Tables

Introduction

The QuickC® Tool Kit is a set of utility programs that you can use to develop your own programs outside the QuickC integrated environment. These tools include

- QCL, the Microsoft QuickC Compiler, which compiles QuickC source programs and invokes LINK (see below) to link object files
- LINK, the Microsoft Overlay Linker, which combines object files that you've created with the Microsoft® QuickC Compiler (or any other Microsoft language product) into executable programs
- LIB, the Microsoft Library Manager, which combines object files into libraries
- NMAKE, the Microsoft Program-Maintenance Utility, which maintains large programs that consist of separate modules
- HELPMAKE, the Microsoft Help-File-Creation Utility, which lets you create on-line-help files
- The special-purpose utilities, including MSHERC (which provides support for Hercules® graphics adapters) and FIXSHIFT (which fixes a bug in certain keyboards that makes them incompatible with QuickC and some other programs)

Why use the Tool Kit when you can perform many of these same operations within the QuickC environment? The main reason is flexibility. The QuickC environment uses the same tools as the Tool Kit but provides access to only the most commonly used options. When you use the utilities from the Tool Kit, all their powerful and flexible options are available to you. You may find that it's easiest to use the integrated environment during the early stages of program development, when you're still tinkering with programs and you need to compile, run, and debug programs fast. Then, when you're fine-tuning and maintaining your code, use the tools from the Tool Kit for additional control and flexibility.

About This Manual

If you're new to Microsoft language products, this book will teach you how to get the most out of the tools provided in this package. Experienced users of Microsoft languages will be able to find information about existing utilities quickly, as well as learn about the new utilities provided with QuickC (including the new NMAKE and HELPMAKE utilities and the hardware-specific support utilities documented in Appendix C, "Hardware-Specific Utilities.")

Part 1 of the manual is a tutorial that illustrates the ways you'll use the QCL, LINK, LIB, and NMAKE utilities for everyday programming work. Each chapter describes the most common options of each utility.

Part 2 is a reference to the Tool Kit. Each chapter describes a tool in detail, showing the exact syntax of the command line and describing all of the tool's options and their effects. Chapter 8 comprises a complete reference to HELPMAKE, the Microsoft Help-File-Creation Utility. Consult this chapter if you want to customize your on-line help.

Appendixes of this manual list the exit codes returned by each tool, explain the use of QuickC memory models, describe the MSHERC and FIXSHIFT utilities, and describe the error messages associated with each tool.

Following the appendixes is a glossary, which defines all the terms introduced in this manual, as well as other C-specific terms you may find helpful.

NOTE *Microsoft documentation uses the term "OS/2" to refer to the OS/2 systems—Microsoft Operating System/2 (MS® OS/2) and IBM® OS/2. Similarly, the term "DOS" refers to both the MS-DOS® and IBM Personal Computer DOS operating systems. The name of a specific operating system is used when it is necessary to note features that are unique to that system.*

Elsewhere in This Package

As you're reading this manual, you may want to refer to other manuals or on-line documentation for information about other parts of the product. This manual assumes that you've installed the QuickC compiler software as described in the manual titled *Up and Running*. If you haven't yet installed the software, install it now.

Read the manual titled *C for Yourself* if you're new to C programming and want to learn how to write C programs. That manual includes an appendix that summarizes the C language and common C library routines.

Insert the disk titled "Learning the QuickC Environment" and type `learn` if you want to learn how to use the QuickC integrated environment. The lesson titled "Basic Skills" shows how to get on-line help for any command or option within the environment or for any part of the C language or run-time library.

Key to Document Conventions

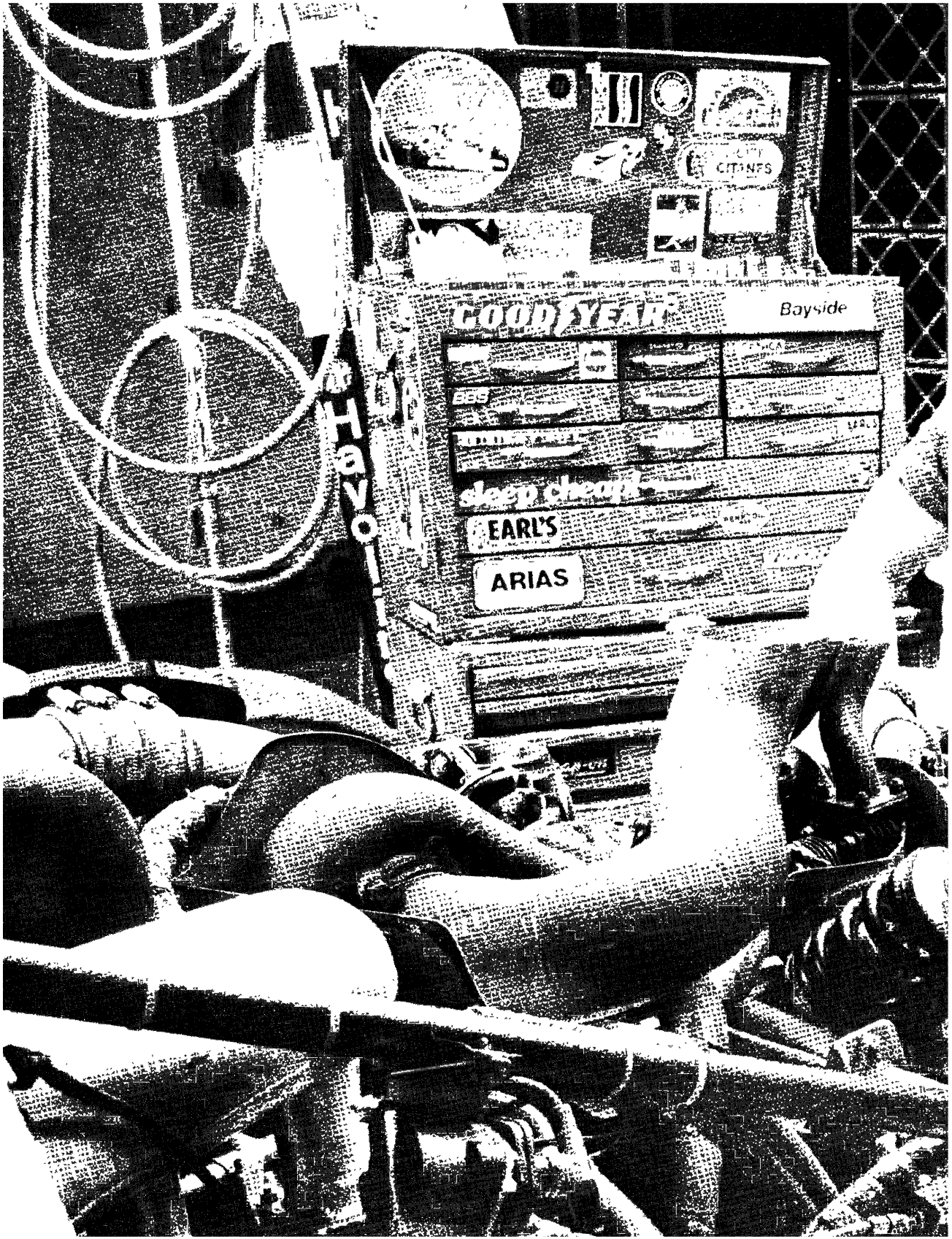
This book uses the following document conventions:

Examples	Description
STDIO.H, PATH, C:\BIN, QCL, NMAKE, DX, _TEXT	UPPERCASE LETTERS indicate file names, segment names, registers, and terms used at the DOS-command level.
cdecl, int, printf, alloc_text, #undef, &&, DosCreateThread	**Boldface letters** indicate C keywords, operators, language-specific characters, and library functions, as well as OS/2 functions.
`QCL A.C B.C C.OBJ`	`This font` is used for examples, user input, program output, and error messages in text.
CTRL+ENTER	SMALL CAPITAL LETTERS are used for the names of keys on the keyboard. When you see a plus sign (+) between two key names, you should hold down the first key while pressing the second.
	The carriage-return key, sometimes appearing as a bent arrow on the keyboard, is called ENTER.
"argument"	Quotation marks enclose a new term the first time it is defined in text.
Color Graphics Adapter (CGA)	The first time an acronym is used, it is often spelled out.
if (*expression*) *statement1*	*Italic letters* indicate placeholders for information you must supply, such as a file name. Italics are also occasionally used for emphasis in the text.
[[*option*]]	Items inside double square brackets are optional.
#pragma pack {1|2}	Braces and a vertical bar indicate a choice among two or more items. You must choose one of these items unless double square brackets surround the braces.
QCL *options* [[*file...*]]	Three dots following an item indicate that more items having the same form may appear.

`while()`
`{`
`  .`
`  .`
`  .`
`}`
	A column of three dots tells you that part of the example program has been intentionally omitted.

PART 1

Tool Kit Tutorial


GOODYEAR
Bayside
EARL'S
ARIAS

PART 1

Tool Kit Tutorial

The QuickC Tool Kit is divided into two parts. Part 1 is a tutorial designed to get you started with the QuickC tools. It does not cover the tools in detail, but instead gives you a "quick start" on the options and techniques that you are most likely to need. If you are [illegible] Part 1 to learn how to use the Tool Kit and [illegible] familiar with the most useful options of each tool. After [illegible] and some experience with the tools, turn to Part 2, "Reference Tools," for the "nuts and bolts" of each tool.

[illegible] the fundamental tools you must [illegible] stable programs outside the QuickC environment. It goes into the Library Manager (LIB), with which you can create libraries of compiled code. The part concludes with NMAKE, a program-maintenance utility that helps you automate the process of rebuilding software.

CHAPTERS

CHAPTER 1

1

Creating Executable Programs

This chapter shows you how to create executable programs from QuickC source files. The QuickC Tool Kit has two programs for this purpose: QCL and LINK.

Although you can create executable programs within the QuickC environment, the QCL and LINK commands give you more power and flexibility in this process. For example, QCL gives you greater control over the QuickC preprocessor, allows you to generate special code for an 8087-family coprocessor or an 80286 processor, and allows you to rename output files.

This chapter introduces the basic concepts and the most common options of the QCL and LINK commands. For a complete description of all the QCL options, listed alphabetically, see Chapter 4, "QCL Command Reference," in Part 2, "Reference to QuickC Tools." For a complete explanation of how LINK works, see Chapter 5, "LINK," also in Part 2.

Compiling and Linking: an Overview

The first step in creating a QuickC program is to enter the source code using an editor and save it in a file. This file is known as a C "source file." You may enter separate parts of the program in different source files and compile these source files separately.

Once you've saved your C source file(s), two steps are required to convert it to an executable file:

1. *Compiling.* During this step, the QuickC compiler converts the C source files to object files. An object file contains binary code but is not yet in executable form.

2. *Linking.* During this step, the linker takes the object files created during compilation, combines them with standard libraries plus any other object files and libraries you specify, and creates an executable file that can be run under DOS.

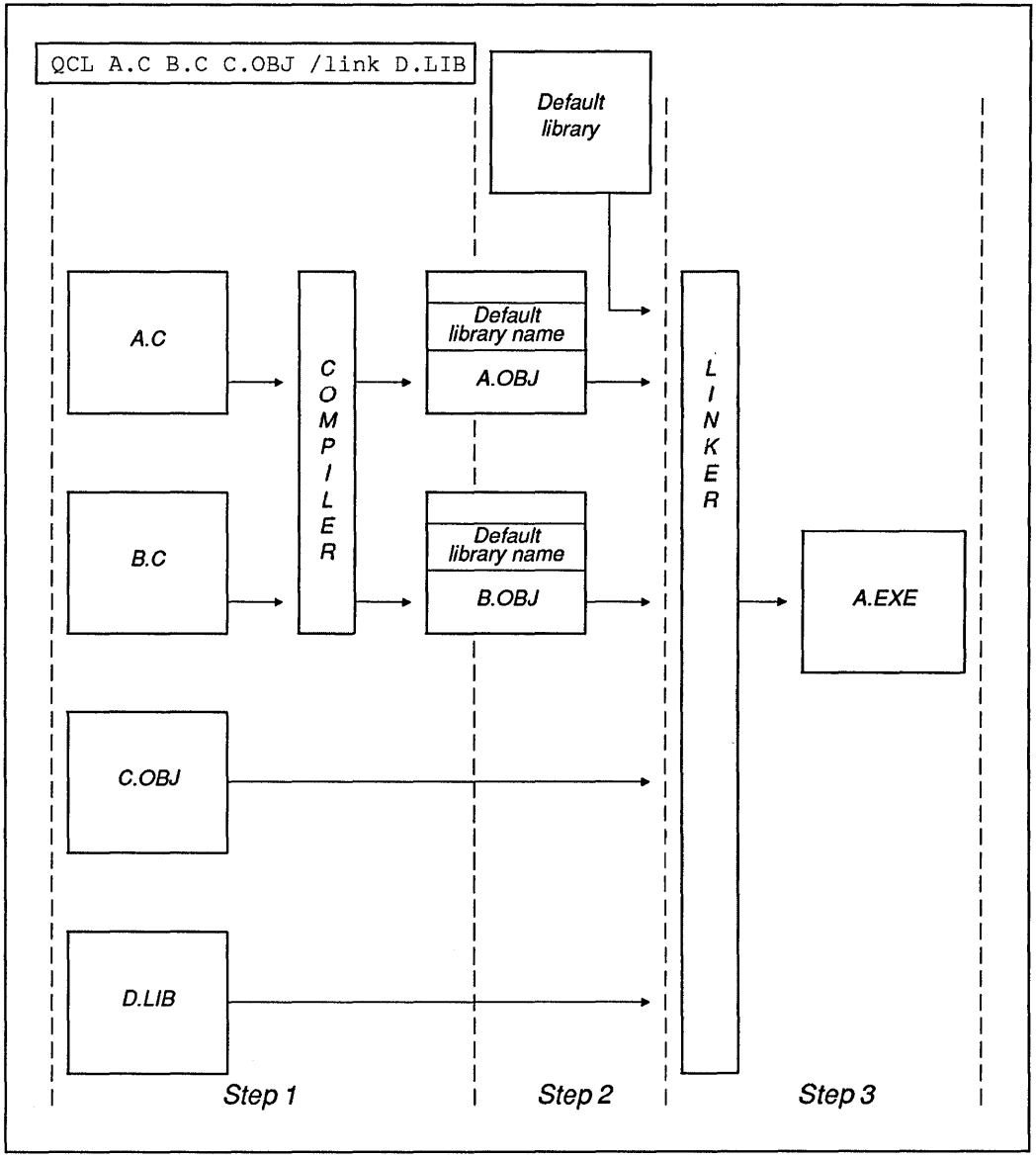


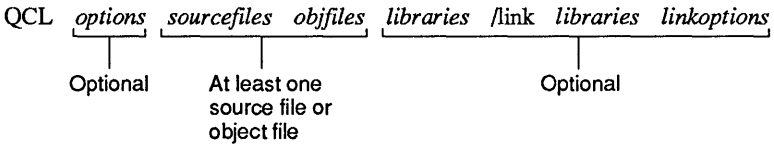
Figure 1.1 The Compiling and Linking Process

You'll use the QCL command to perform both compiling and linking. On the QCL command line, give the names of any C source files that you want to compile, and the names of any additional object files or libraries that you want to link. The compile/link procedure is described below and illustrated in Figure 1.1:

1. QCL compiles the source files you named, creating object files. All object files created by QCL from source files have the extension .OBJ. They also contain the name of the combined library needed to create the executable file.
2. QCL calls the linker and passes the object files created in the compiling step plus any object files and libraries that you specified on the QCL command line.
3. The linker links the object files, the libraries named in the object files, and libraries passed by QCL to create the executable file.

Using the QCL Command

The QCL command, which you'll use for most compiling and linking operations, has the format shown below:



The items in italics are different pieces of input (described below) that you must give on the QCL command line:

- The *options* are QCL options, which control some aspect of the compiling or linking process. They may appear anywhere on the command line and in most cases affect any files that appear later on the command line. The most commonly used QCL options are described in the section titled "Controlling Compiling and Linking with QCL Options." For complete information on all QCL options, see Chapter 4, "QCL Command Reference."
- The *sourcefiles* are the names of the C source files that you are compiling. Normally, these file names have .C extensions.

- The *objfiles* are the names of additional object files that you want to link with. QCL compiles the source files, then links the resulting object files with *objfiles*. For example, given the command line

  ```
  QCL MAIN.C AUX.OBJ
  ```

  QCL compiles `MAIN.C`, creating the object file `MAIN.OBJ`, then passes `MAIN.OBJ` and `AUX.OBJ` to the linker, which creates an executable file named `MAIN.EXE`.

  If you give a file name with any extension other than .C or .LIB, or with no extension, QCL assumes you are giving it the name of an object file. For example, in the command line

  ```
  QCL OBJECT1 OBJECT2.OBJ
  ```

  the QCL command assumes the .OBJ extension for `OBJECT1` and passes it and `OBJECT2.OBJ` to the linker for processing.

- The *libraries* are the names of libraries that you want to link with. These names must have .LIB extensions.

  Ordinarily, you don't need to give a library name unless your program calls functions that are stored in libraries other than the standard combined C libraries (which you created during installation). For example, if you use libraries created by a company other than Microsoft, or if you have created a private library of functions and your program calls functions in this library, you must give the private-library name on the QCL command line. For example, the command line

  ```
  QCL MAIN.C GRAPHICS.LIB
  ```

  tells QCL to compile `MAIN.C`, creating the object file `MAIN.OBJ`, then to pass `MAIN.OBJ` to the linker, which links `MAIN.OBJ` with functions in the default combined library SLIBCE.LIB and the GRAPHICS.LIB library given on the command line.

- The *linkoptions* are linker options, which control some aspect of the linking process. Linker options are described in the section in this chapter titled "Controlling the Linking Process."

- If you're not sure that your program will fit in available memory, you can indicate that certain parts of the program will become "overlays"; that is, they will be stored on disk and read into memory—overlaid—only when needed. To specify overlays, enclose the modules you want to overlay in parentheses on the QCL command line. For example,

  ```
  QCL RESIDNT.C (ONCALL.C) MAIN.C
  ```

  creates a program named `RESIDNT.EXE` with an overlay module named `ONCALL.OBJ`. Whenever control passes to `ONCALL.OBJ`, it is read into memory from disk. (See Section 5.6, "Using Overlays," for more information about overlays and restrictions on their use.)

Specifying File Names

File-name extensions

A DOS file name has two parts: the "base name," which includes everything before the period (.), and the "extension," which includes the period and up to three characters following the period. The extension identifies the type of the file. The QCL command uses the extension of each file name to determine how to process the corresponding file, as explained in the following list:

Extension	Processing
.C	QCL assumes the file is a C source file and compiles it.
.OBJ	QCL assumes the file is an object file and passes it to the linker.
.LIB	QCL assumes the file is a library and passes it to the linker. The linker links this library with the object files QCL created from source files and the object files given on the command line.
Any other extension or no extension	QCL assumes the file is an object file and passes it to the linker. You must end the file name with a period (.) if the file has no extension. Otherwise, QCL assumes the extension .OBJ.

Uppercase and lowercase letters

In file names, any combination of uppercase and lowercase letters is legal. For example, `SHUTTLE.C` and `Shuttle.c` represent the same file.

Path names

Any file name can include a path name. When a file name includes a path name, QCL assumes the file to be in that path. You may supply either a full path name or a partial path name. A full path name includes a drive name and one or more directory names. A partial path name is the same as a full path name but omits the drive name, which QCL assumes to be the current drive.

If no path name is given, QCL assumes that all source and object files given on the command line are in the current directory.

Examples

The command line

```
QCL A.C B.C C.OBJ D E.MOD
```

compiles the files `A.C` and `B.C`, creating object files named `A.OBJ` and `B.OBJ`. These object files are then linked with the object files `C.OBJ`, `D.OBJ`, and `E.MOD` to form an executable file named `A.EXE` (since the base name of the first file on the command line is `A`). Note that the extension `.OBJ` is assumed for `D` because no extension is given on the command line.

```
QCL TEAPOT.C \MSG\ERROR C:\GRAPHICS\GRAPHICS.LIB
```

This command line tells QCL to compile the file `TEAPOT.C` and to link the resulting object file with `\MSG\ERROR.OBJ` and the library `GRAPHICS.LIB`. QCL assumes the extension .OBJ for the file `\MSG\ERROR` because none was specified. It looks for the library in the `\GRAPHICS` directory on drive `C:`.

Controlling Compiling and Linking with QCL Options

The QCL command offers a variety of options that control the compiling and linking processes and modify the files created during each stage. For example, you can specify QCL options to rename output files, to control the operation of the QuickC preprocessor, to take advantage of an 80286 processor or a coprocessor, or to optimize your program for speed or size.

QCL options may begin with either a forward slash (/) or a dash (-). In this manual, the slash is used.

Important *Except as noted, QCL options are case sensitive, so you must use the exact combination of uppercase and lowercase letters shown in this manual.*

Some QCL options require arguments. For example, you may be required to give a number or a file name as part of a QCL option. For some options, you must put a space between the option and the argument; for others, you must place the argument immediately after the option. The description of each option gives its exact syntax.

The following sections list the most commonly used QCL options by type. See Chapter 4, "QCL Command Reference," for a complete list of QCL options or for more information about the effects of an option described in this chapter.

Help with QCL options

If you need fast help with QCL options, enter the following command:

```
QCL /HELP
```

This command displays a list of commonly used QCL options with a brief description of each option. Unlike other QCL options, /HELP is not case sensitive; you can type any combination of lowercase and uppercase letters.

Compiling without Linking

When you compile with the /c option, QCL compiles the source files you give on the command line, but ignores any object files or libraries that you give on the command line. Because QCL does not invoke the linker when you give this option, it does not create an executable file.

You might want to use this option in the following cases:

- To compile separate modules that you want to put in a library using the LIB utility (described in Chapter 2 of this manual)
- To link in a separate step as described later in this chapter (for example, in an NMAKE file)

Example

```
QCL /c SPELL.C THESRS.C
```

The example above compiles the C source files `SPELL.C` and `THESRS.C`, creating the object files `SPELL.OBJ` and `THESRS.OBJ`. No linking is performed, so no executable file is created.

Compiling Only Modified Functions

The /Gi option allows you to compile programs much faster than usual. It speeds compilation by telling QCL to compile only the parts of each C source file that have changed since the file was last compiled. This process is called "incremental compilation."

Information about the incremental compilation of each source file is maintained in an MDT (Module Description Table) file. One MDT file can contain this information for more than one source file.

If you give a file-name argument following the /Gi option, the compiler writes the change information for all the source files into that single MDT file. Do not put spaces between the /Gi option and the file name.

If you specify the /Gi option without a file name, the compiler creates an MDT file for each C source file that you give on the command line. Each MDT file has the base name of the source file and the .MDT extension.

Generally, when you compile with /Gi, only the changed functions in each C source file are recompiled. The entire file is recompiled only if a change affects the entire program.

See Section 4.3.15 in Part 2, "Reference to QuickC Tools," for details about incremental compilation and the /Gi option.

Example

```
QCL /GiASC.MDT alpha.c num.c
```

The example above compiles the changed portions of the files `alpha.c` and `num.c`. It creates a single .MDT file named `ASC.MDT` into which it writes change information for both source files.

Optimizing Programs

"Optimizing" a program is the process of making the program, or a part of the program, as fast or as small as possible. The following QCL options can help with this process:

Option	Effect
/O, /Ot	Tells the compiler to optimize the program for execution time over code size. The compiler makes the executable file faster, but it does not make the file size as small as possible.
/Ol	Tells the compiler to optimize loops in your program. This option makes the executable file run faster.
/Gs	Turns off stack-checking routines in your program. This option reduces the size of the executable file, but it may cause important stack-overflow errors to go undetected.
/Ox	Tells the compiler to perform all possible optimizations. This option combines the effects of the /Ot, /Ol, and /Gs options.
/Od	Tells the compiler not to optimize your program. This option speeds compilation, although it may result in a slightly slower executable file.

You may combine the /O options on the command line, specifying more than one letter following /O. For instance, /Olt optimizes loops and execution time. If the letters conflict, QCL uses the last one in the list.

Naming Output Files

Use the following options to name the object and executable files that QCL creates:

Option	Effect
/Fo*objfile*	Gives the name *objfile* to the object file. You may give more than one /Fo option; each option applies to the next C source-file name on the command line. For example, `QCL /FoOBJ1 SRC1.C SRC2.C` compiles `SRC1.C`, creating an object file named `OBJ1.OBJ`, then compiles `SRC2.C`, creating an object file named `SRC2.OBJ`.

If you give *objfile* without an extension, QCL automatically appends the .OBJ extension to the file name. If you give a complete path name with *objfile*, QCL creates the object file in that path. For example,

```
QCL /Fo\MODS\OBJ1.OBJ SRC1.C
```

compiles `SRC1.C`, creating an object file named `OBJ1.OBJ` in the `\MODS` directory. If you give only a drive or directory specification, the specification must end with a backslash (\) so that QCL can distinguish it from a file name.

/Feexefile Gives the name *exefile* to the executable file. If you give *exefile* without an extension, QCL automatically appends the .EXE extension to the file name. If you give a complete path name with *exefile*, QCL creates the executable file in that path. If you give a path specification without a file name, the path specification must end with a backslash (\) so that QCL can distinguish it from a file name.

If you don't tell it otherwise, QCL names output files as follows:

Type of File	Default
Object	Same base names as the original C source files with extensions of .OBJ. For example, if you compile a C source file named LEX.C, QCL creates an object file named LEX.OBJ.
Executable	Same base name as the first file name on the command line plus an extension of .EXE. For example, `QCL LEX.C GENCOD.OBJ OPTIMIZ` creates an executable file named `LEX.EXE` by compiling `LEX.C` (creating `LEX.OBJ`), then linking `LEX.OBJ`, `GENCOD.OBJ`, and `OPTIMIZ.OBJ`.

Turning Off Language Extensions

The /Za option tells the compiler to treat all Microsoft-specific keywords as ordinary identifiers and to display error messages if your programs use any other extended language features.

Compile with the /Za option if you plan to port your programs to environments that don't recognize Microsoft extensions to the C language, or if you want to ensure that your programs are strictly compatible with the American National Standards Institute (ANSI) definition of the C language. Microsoft extensions include the **near**, **far**, **huge**, **cdecl**, **fortran**, and **pascal** keywords, as well as several usages of standard C constructs that are not defined in the ANSI standard. (See Section 4.3.30, "/Ze, /Za," in Part 2, for more information about these extensions.)

Debugging and Syntax Checking

Several QCL options are useful when you want the compiler to check the syntax of your program, or when you want to track down logic errors using the debugger built into the QuickC environment (or other Microsoft debuggers). These options fall into three categories:

- Checking syntax
- Setting warning levels
- Compiling for a debugger

Checking Syntax

If you want to make sure that your program is free from syntax errors without compiling and linking the program, compile it with the /Zs option. This option tells the QCL command to display error messages if your program has syntax errors. QCL doesn't create object or executable files.

Setting Warning Levels

You may get warning messages during compilation if your program has problems that aren't serious enough to stop the compiling process. You can easily identify a warning message because it begins with the word "warning" and has "C4" as the first two characters in the error number.

The "warning level" options, /w and /W0 through /W3, allow you to suppress warning messages for certain classes of problems. In general, the lower the warning level, the less strict the compiler is about flagging possible errors in your program. You might want to use a lower warning level if you're intentionally using the flexibility of C in some operations and you want to suppress warnings about these operations.

The warning-level options are described below:

Option	Effect
/W0, /w	Turns off all warning messages.
/W1	Tells the compiler to display most warning messages. (This is the level of warnings you get by default.)
/W2	Tells the compiler to display all /W1 warnings plus warnings for problems such as functions without a declared return type; functions that have a return type other than **void** and don't have a **return** statement; and data conversions that cause loss of precision.

/W3	Tells the compiler to display all /W2 warnings plus warnings for any non-ANSI features or Microsoft-specific keywords. The /W3 option is similar to the /Za option, except that /W3 gives warnings for nonstandard features, while /Za gives error messages and aborts the compilation.

Appendix D lists all warning messages in order of error number. The description of each message indicates the warning level that must be set in order for the message to appear.

Compiling for a Debugger

You must compile your program with one or more of the following QCL options if you plan to debug it within the QuickC environment or with another Microsoft debugger:

Option	Effect
/Zi	Puts information needed for debugging into the program. Use /Zi if you plan to debug your program with the QuickC debugger or with the Microsoft® CodeView® window-oriented debugger provided with other Microsoft language products.
/Zd	Puts limited symbolic information in the object file. Use /Zd if you plan to debug your program with SYMDEB, the Microsoft Symbolic Debug Utility, shipped with earlier versions of Microsoft language products.
/Zr	Checks for null or out-of-range pointers in your program. Optional if you plan to debug with the QuickC debugger.

Controlling the Preprocessor

The QCL command provides several options that control the operation of the QuickC preprocessor. These options allow you to define macros and manifest (symbolic) constants from the command line, change the search path for include files, and stop compilation of a source file after the preprocessing stage to produce a preprocessed source-file listing.

Defining Constants

The C preprocessor directive **#define** defines a name for a constant or for C program text. Wherever the name appears in your program, the preprocessor substitutes the text you've defined for that name.

You can use the /D option to define constants from the QCL command line. This option has the form

/Didentifier=string

or

/Didentifier=number

The *identifier* is the name you're defining; *string* or *number* is the text or numeric value that is substituted for the name. The *string* must be in double quotation marks if it includes spaces.

You can leave off the equal sign and the string or number. If you do, the identifier is defined and its value is set to 1. This approach is useful when you need to define an identifier but do not care what its value is. For example, `/DCREATE` defines an identifier named `CREATE` and sets it equal to 1.

If you've defined a number for *identifier*, you can "turn off" the definition by using the following form of the /D option:

/Didentifier=

When you compile with this form, the *identifier* is no longer defined within your program and no value is substituted for it.

QCL allows you to define up to 15 constants using the /D option for each. You may be able to define as many as 20, depending on the other options you specify. (See the section in this chapter titled "Removing Predefined Identifiers" for more information about the number of constants you are allowed to define.)

Searching for Include Files

The QuickC preprocessor directive

#include *filename*

tells the QuickC preprocessor to insert the contents of *filename* in your source program, beginning at the line where the directive appears. Include files provided with the Microsoft QuickC Compiler contain prototypes of standard C library functions and the constants used by these functions. If *filename* is enclosed in angle brackets (<>), the preprocessor looks for the file in the directories given by the INCLUDE environment variable. If *filename* is enclosed in quotation marks (" "), the preprocessor looks for the file first in the current directory and then in the directories specified by the INCLUDE variable. (Enter the SET command at the DOS prompt to see the INCLUDE variable and the directories it specifies.)

Use the following options to override the usual search order without changing the value of the INCLUDE variable:

Option	Effect
/X	Tells the preprocessor not to search for include files in the directory given by the INCLUDE variable.
/I*directory*	Tells the compiler to search the given directory for include files before it searches the directories given by the INCLUDE environment variable. You can give more than one /I option, each specifying a directory. Directories are searched in the order in which they appear on the command line.

Creating Preprocessor Listings

If you want to see output from the QuickC preprocessor, give one or more of the following options on the QCL command line:

Option	Effect
/E	Writes preprocessor output to the standard output device (your screen, unless you redirect output to another device or to a file). The /E option also inserts **#line** directives in the output. The **#line** directives renumber the lines of the preprocessed file so that, if you recompile the preprocessed file, the errors generated during later stages of processing refer to the original source file rather than to the preprocessed file.
/P	Writes preprocessor output to a file and inserts **#line** directives in the output file. The preprocessor gives the file the base name of your C source file and an extension of .I.
/EP	Writes preprocessed output to the standard output device but does not insert **#line** directives.
/C	Leaves comments in the preprocessed output. Normally, the preprocessor strips comments from the source file. This option has an effect only if you also give the /E, /P, or /EP option.

Removing Predefined Identifiers

The QuickC compiler automatically defines certain identifiers, which represent conditions such as the current operating system or memory model. Your programs may use these identifiers along with the QuickC preprocessor directives **#if**, **#ifdef**, **#ifndef**, **#else**, **#elif**, and **#endif** to tell the compiler to "conditionally compile" portions of the program. For example, the **#ifdef** directive tells the compiler to compile subsequent code only if a given identifier is defined. Similarly, the **#ifndef** directive tells the compiler to compile subsequent code only if a given identifier is *not* defined.

The predefined identifiers are as follows: **_QC, MSDOS, M_I86, M_I86*m*M, M_I8086, M_I286, NO_EXT_KEYS,** and **_CHAR_UNSIGNED.** (For more information on how and when these identifiers are defined, see Table 4.4, "Predefined Names," in Section 4.3.27.) If you don't use these identifiers for conditional compilation, you might want to remove their definitions from the program. For each predefined identifier that you remove, you can define an additional identifier (over the default limit of 15) with the /D option on the QCL command line.

The following options turn off predefined identifiers:

Option	Effect
/U*identifier*	Turns off the definition of *identifier*
/u	Turns off the definition of all predefined identifiers

Compiling for Specific Hardware

QuickC creates executable programs that run on any processor in the 8086 family, including the 8086/8088, 80186, 80286, and 80386.

If your programs will always run on machines with 80286 or 80386 processors, or on machines with coprocessors, you can compile your programs with the following options to take advantage of the specific hardware configuration:

Option	Effect
/G2	Uses the 80286/80386 instruction set for your program. You cannot run the program on machines with 8088, 8086, or 80186 processors.
/FPi87	Handles math for floating-point types (**float** and **double**) by generating instructions for an 8087 or 80287 math coprocessor. This reduces the size of your program; however, the program must be run on a system with a coprocessor present.

The /G2 and /FPi87 options are the most commonly used options for hardware-specific compilation, but others are available. See Sections 4.3.12 and 4.3.13 for details.

Choosing Memory Models

The "memory model" your program uses determines how many 64K (kilobytes) segments the compiler allocates for its data and code. Ordinarily, you don't need to choose the memory model explicitly if your program's code can fit into one 64K segment and your program's data can fit into one 64K segment. This memory allocation, called the small memory model, is the default used by the QCL command.

If your program exceeds the default limit for code or data, you must use one of the other memory models. The following list summarizes the options for the memory model:

Option	Effect
/AS	Small model: provides one 64K segment for data and one 64K segment for code. No one data item can exceed 64K. This is the most efficient model for QuickC programs. QCL uses this option automatically if you don't give a memory-model option, so you never need to give this option explicitly.
/AM	Medium model: provides one 64K segment for data and multiple 64K segments for code. No one data item can exceed 64K. This is the most efficient model if your program exceeds the 64K default limit for code.
/AC	Compact model: provides multiple 64K segments for data and one 64K segment for code. No one data item can exceed 64K. This is the most efficient model if your program exceeds the 64K default limit for data.
/AL	Large model: provides multiple 64K segments for data and for code. No one data item can exceed 64K.
/AH	Huge model: same as large model, except that individual data items can be larger than 64K.

Although memory models give you additional flexibility in dealing with large programs, you must use them with care to avoid problems in your programs. See Section 4.3.1 or Appendix B if you need further information about the use of memory models.

Controlling the Linking Process

Several QCL options control the linking process rather than the compiling process. You've already encountered one of these options: the /Fe option, which renames the executable file. Here are the others:

Option	Effect

/Fm*mapfile*	Creates a map file showing program segments in order of appearance in the program. If you give *mapfile* without an extension, QCL automatically appends the .MAP extension to the file name. If you give a complete path name with *mapfile*, QCL creates the map file in that path. For example,
	`QCL /Fm\MODS\MAP1.MAP SRC1.C`
	compiles and links `SRC1.C`, creating a map file named `MAP1.MAP` in the `\MODS` directory. If the path specification lacks a file name, it must end with a backslash (\) to distinguish it from a file name.
	The *mapfile* is optional; if you don't specify a new name, the linker gives the map file the same base name as the executable file, with an extension of .MAP. For example,
	`QCL /Fm MOD1.C MOD2.C`
	creates an executable file named `MOD1.EXE` and a map file named `MOD1.MAP`.

/F *number*	Sets the stack size to the given number of bytes. The *number* may be in decimal, octal, or hexadecimal. (As in C programs, octal numbers start with the prefix 0 and hexadecimal numbers with the prefix 0x.) If you don't give this option, the executable file uses a 2K stack. Use this option if your program gets stack-overflow errors at run time.

See Sections 4.3.7, 4.3.9, and 4.3.10 for detailed information on these options and on map files.

The /link option

Another way of controlling the linking process is to use the /link option on the QCL command line. The /link option allows you to specify LINK command options—not QCL options—without invoking the linker separately. On the QCL command line, the /link option must follow the source and object files and all QCL options. QCL passes directly to the linker the options that follow /link. These options are described under "LINK Options" below and in Section 5.4.

Example

```
QCL /FPi87 /Fm SRC1.C SRC2 /link /INF
```

In the example, the /Fm and /FPi87 options apply to the QCL command and the /INF option applies only to the linker. As a result of this command line, QCL compiles `SRC1.C` to run on an 8087 or 80287 processor then passes `SRC1.OBJ` and `SRC2.OBJ` to the linker. The /Fm option to QCL causes the linker to create a map file named `SRC1.MAP`. The /INF option, which applies only to the linker and not to QCL, causes the linker to display information about the linking process.

Other QCL Options

The following QCL options are used for the specific purposes described:

Option	Effect
/Gc	"Calling-convention" option. Uses the FORTRAN/Pascal naming and calling conventions for functions in the program. Compile with this option if you want to call routines that use the Microsoft Pascal or Microsoft FORTRAN calling conventions or if you need to save space in the executable file. (See Section 4.3.14 for more information about the effects of this option.)
/G*tnumber*	"Threshold" option. Tells the compiler to allocate data items larger than *number* in a new data segment. If you give this option with no number, QCL allocates items larger than 256 bytes in their own segment. If you don't give this option, QCL allocates items larger than 32,767 bytes in their own segment. This option applies only if you compile with the compact (/AC), large (/AL), or huge (/AH) memory model. See Appendix B for more information about memory models and allocation.
/NT *textsegname*	"Name-text-segment" option. Assigns the given name to the text segment. The space is optional between /NT and *textsegname*. The text segment contains the program code for the entire program (if you compile with the /AS option, the /AC option, or no memory-model option) or for the module you are compiling (if you compile with the /AM, /AL, or /AH option).
/Zl	"Library" options. Tells the compiler not to put the name of the appropriate combined library in the object file. Use this option to compile modules that you want to put in a library with the LIB utility.

/Zpnumber	"Pack" option. Stores structure members after the first on *number*-byte boundaries. The *number* argument, if given, may be 1, 2, or 4; if it isn't given, QCL assumes a value of 2. This option may reduce the size of executable files, although it may also slow program execution.

The QCL options shown in this chapter are those most commonly used with QuickC programs. QCL supports a number of other options in addition to those described so far. See Section 4.3 for descriptions of all the QCL options.

Invoking the Linker Directly: the LINK Command

In some cases, you may choose to compile source files in one step, then link the resulting object files in a separate step. For example, in the first step, you would compile your C source files as shown below:

```
QCL /c SOURCE1.C SOURCE2.C
```

Then, in the second step, you would link the resulting object files, plus any additional object files or libraries, as shown below:

```
QCL SOURCE1 SOURCE2 GRAPHICS.LIB
```

As illustrated in the second step, if you give only object files or libraries on the QCL command line, the QCL command simply passes the object files and libraries to the linker.

Instead of using the QCL command to link, you can invoke the linker directly by entering the LINK command. The advantage to using LINK is that the linker prompts you for any input it needs; you don't need to give all file names and options on the command line, although you may do so.

The remainder of this section explains how to use the LINK command to link object files and libraries.

Giving Input to the LINK Command

To invoke the linker explicitly, simply enter

```
LINK
```

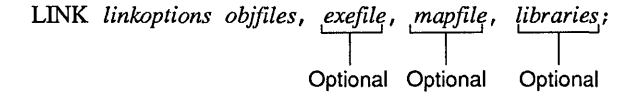
If you don't give any other information on the command line, LINK prompts you for input. The following list shows how to respond to each prompt:

At This Prompt:	Enter:
Object Modules:	The names of all object files that you want to link, separated by plus signs. If all the names do not fit on one line, type a plus sign as the last character on the line. LINK repeats the prompt on the next line, and you can type additional object-file names. Type a library name in response to this prompt if you want to include the entire library in the executable file. Make sure the library name has an extension of .LIB. (If you type the library name in response to the "Libraries:" prompt below, LINK places in the executable file only the library modules that are called in your source files.)
Run File:	The name of the executable file that you want to create. If you press ENTER without typing a name, LINK uses the base name of the first object file you gave plus the extension .EXE. This name is shown in brackets in the prompt.
List File:	The name of the map file, which shows segments in your program. If you press ENTER without typing a name, LINK doesn't create a map file. If you enter a name without an extension, LINK adds the .MAP extension utomatically.
Libraries:	The names of libraries other than the standard combined libraries that you want to link with the object files. If you enter a library name without an extension, LINK assumes the extension .LIB. If you enter more than one library name, put a plus sign between each library name and the next.

You can type LINK-command options as part of the response to any prompt. See the next section, "LINK Options," for a list of commonly used options.

Input on the command line

If you prefer, you can give all your input to LINK on the command line. The LINK command line has the form shown below:



Note: Semicolon may end command line wherever a comma appears.

Commas must appear as shown above to separate the names of the different files. You may type a semicolon to terminate the command line anywhere after the list of object files. The semicolon tells LINK to use defaults for the remaining files. LINK options may appear anywhere on the command line.

The prompts previously described correspond to the command line as follows: "Object Modules" is equivalent to *objfiles*, "Run File" to *exefile*, "List File" to *mapfile*, and "Libraries" to *libraries*.

Input in a response file

LINK allows you one other alternative for providing input. You can enter responses for all prompts in a file, then invoke LINK with the following command:

LINK @*responsefile*

Replace *responsefile* with the name of the file that contains your responses. The responses should look the same as if you were typing them in response to prompts. For example, type all object-file names on the first line, the executable-file name on the second line, and the map-file name on the third line. Use a plus sign at the end of a line to continue a response on the next line. Leave a blank line in the file if you want LINK to use the default for a prompt. Place LINK options at the end of any response or place them on one or more separate lines.

LINK Options

LINK options allow you to control the operation of the linker. If you're using the QCL command to link, give these options after the /link option on the command line. If you're using the LINK command to link, these options may appear anywhere on the command line.

Not all LINK options are applicable to QuickC programs. Some options are useful only for assembly-language programs. This section describes only the options that are useful for QuickC programs. See Chapter 5, "LINK," for a complete list of options.

Case sensitivity

LINK options are not case sensitive, so you can type any combination of uppercase and lowercase letters for each option.

Abbreviations

Because some LINK options have long names, LINK allows you to abbreviate each name. The abbreviation must include enough continuous letters to distinguish the option from others. Letters that you may leave off are surrounded by brackets in the following sections. In general, this manual refers to LINK options by their shortest possible abbreviations.

Numerical parameters

Some LINK options take numbers as parameters. You may specify the numbers in decimal, hexadecimal, or octal. As in C programs, hexadecimal numbers are identified by the prefix 0x and octal numbers by the prefix 0.

Help with LINK options

If you need quick help with LINK options, enter the following command:

```
LINK /help
```

Controlling the Linking Process with Options

Use the LINK options described below to control the linking process:

Option	Effect
/BA[TCH]	Tells the linker to continue processing if it can't find one of the files you've given, rather than stop processing and prompt you. Also prevents LINK from displaying its program banner and echoing the contents of response files on standard output. Use this option in batch files or NMAKE description files if you're building large executable files and don't want the linker to stop processing if it can't find a file it needs.
/INF[ORMATION]	Tells the linker to display information about the linking process, including the linking phase and the name of each object file being linked.
/NOD[EFAULTLIBRARYSEARCH]	Tells the linker not to search the standard C combined libraries to find C library functions. If you use this option, you should explicitly specify the name of a standard combined library.
/M[AP]	Includes a full public-symbol listing in the map file.
/PAU[SE]	Tells the linker to pause before it creates the executable file and to display a message. This allows you to insert a new disk to hold the executable file.

If you're running on a machine without a hard disk, you might want to create the executable file on a different removable disk. In this case, you would swap the current disk for the new disk before creating the executable file. If LINK displays the message

`Temporary file` ***tempfile*** `has been created.`

`Do not change diskette in drive,` ***letter.***

terminate your link session, copy the temporary file named *tempfile* to the disk where you want to create the executable file, and reenter the LINK command.

Optimizing the Executable File

The following LINK options make the executable file faster, smaller, or both:

Option	Effect
/E[XEPACK]	Compresses the executable file. This option reduces the program's size and load time. However, you cannot use the QuickC or CodeView debugger to debug the program.
/F[ARCALLTRANSLATION]	Reduces the size of the executable file and increases its speed by optimizing far calls to procedures in the same segment as the calling procedure.
/PAC[KCODE]	Given with the /F option, improves the efficiency of medium-, large-, and huge-model programs by grouping neighboring code segments.

Modifying the Executable File

You can use the following LINK options to modify the executable file (for example, to specify the maximum number of segments or set the stack size):

Option	Effect
/CP[ARMAXALLOC]:*number*	Sets the maximum number of 16-byte paragraphs needed for the program to *number*. The *number* may be any decimal, octal, or hexadecimal number in the range 1 – 65,535 decimal.
/SE[GMENTS]:*number*	Sets the maximum number of segments a program may have to *number*. The number may be any value in the range 1 – 3072 decimal. If you don't give this option, a program may have no more than 128 segments.
/ST[ACK]:*number*	Sets the stack size to *number* bytes. The *number* may be any decimal, octal, or hexadecimal number in the range 1 – 65,535 decimal. If you don't give this option, the stack is 2K.

Other LINK Options

The LINK options described in this chapter are those most typically used when linking QuickC programs. The linker supports additional options, including several that apply only to assembly-language programs. For complete information on all LINK options, see Chapter 5, "LINK," in Part 2, "Reference to QuickC Tools."

CHAPTER 2

2

Maintaining Software Libraries with LIB

The Microsoft Library Manager (LIB) lets you create and maintain object-code libraries. You can use the library manager for several tasks:

- To list the contents of a library
- To modify the contents of an existing library
- To copy object code from the library
- To create a new library

This chapter gives you an introduction to libraries then explains how to perform each of the tasks listed above.

Why Use a Library?

An "object-code library" is an organized collection of object code; that is, a library contains functions and data that are already assembled or compiled and are ready for linking. The structure of a library supports the mass storage of common procedures—procedures called by a variety of programs. Each library has an index of its components. These components, called "modules," can be added, deleted, changed, or copied. When you give the linker a library as input, the linker efficiently scans the library and uses only the modules needed by the program.

Object-code libraries are typically used for one of three purposes:

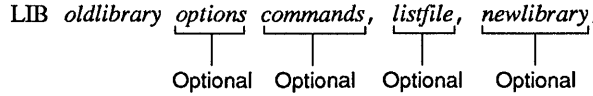
- To support high-level languages. Languages, including C, BASIC, and FORTRAN, perform input/output and floating-point operations by calling standard support routines. Because the support routines are available in a

library, the compiler never needs to regenerate code for these routines. Libraries that contain standard support routines are called "standard libraries."

- To perform complex and specialized activities, such as data-base management or advanced graphics. Libraries containing such routines are often sold by third-party software vendors or are provided by the makers of the compiler (in the case of graphics libraries for Microsoft QuickC).
- To support your own work. If you have created routines that you find useful for a variety of programs, you may want to put these routines in a library. That way, these routines do not need to be recoded or recompiled. You save development time by using work you have already done.

The LIB Command

The LIB command has the form shown below:



Note: Semicolon may end command line wherever a comma appears.

The *oldlibrary* field gives the name of a library. Object-code libraries typically have names that end with .LIB. You specify a library in this field whenever you use LIB.

The *options* field specifies one or more LIB options. For most tasks, you won't need to use any of these options. The options are described in Chapter 6, "LIB," in Part 2, "Reference to QuickC Tools."

The *commands* field gives the commands that modify the contents of the library. Commands are described below in "Modifying the Contents of a Library."

The *listfile* field specifies a file into which LIB puts a list of the library's contents. The next section tells how to list the contents of a library.

The *newlibrary* field specifies a name for the modified library if the commands you give change an existing library. See Section 6.1.1.5 for more information on this field.

Listing the Contents of a Library

You can use LIB to obtain a symbol listing for any object-code library. Listings are useful because they give the exact names of modules and public symbols. You may need a listing if you want to modify a library, as described in the next section.

To list the contents of a library, you need to use only the *oldlibrary* field and the *listfile* field. Use a semicolon (;) to terminate the command so that LIB does not prompt you for additional input.

In the *oldlibrary* field, give the name of the library that you want to examine. You can enter a full path name or a file name without a path. If you do not include a file extension, then LIB assumes the default .LIB extension. Typically, object-code libraries have a .LIB extension.

In the *listfile* field, give the name of the file in which you want the listing to be placed. If you enter the name of a file that does not yet exist, LIB creates the file. If you enter the name of a file that already exists, LIB replaces the current contents of the file with the new listing.

For example, the following command line directs LIB to place a listing of the contents of `MYLIB.LIB` into the file `LISTING.TXT`:

```
LIB MYLIB, LISTING.TXT;
```

The listing file summarizes the contents of the entire library. Each listing file contains two kinds of information in this order:

1. A list of public symbols with corresponding modules for each
2. A list of modules with corresponding symbols for each

Modules, which are basic to the operation of LIB, are discussed in the next section. For a more detailed description of listing files, see Section 6.1.1.4, "Cross-Reference-Listing File," in Part 2, "Reference to QuickC Tools."

Modifying the Contents of a Library

You can use LIB to alter the contents of any object-code library. There are a number of reasons why you might want to do so. For example, if you work with higher-level-language libraries, you may want to replace a standard routine with

your own version of the routine. Or you may want to add a new routine to the standard library, so that your routine is available along with the standard routines.

LIB operations involve these two important items, besides the library file itself:

Item	Description
Object file	An independent file containing object code corresponding to one source file. An object file normally has an .OBJ file extension.
Object module	A self-contained unit within a library, consisting of one or more routines. An object module in a library is in almost all respects identical to the corresponding object file. The object module, however, has no file extension or path because it is not a separate file. Object modules are simply called "modules" in this manual.

The sections that follow refer to both of these items extensively. Just remember: a module is stored in an object file when it is outside a library and becomes simply a "module" when it is loaded into a library.

Modifying the Library

To modify an object-code library, carry out the following steps:

1. To add or replace an object module, first compile or assemble the new code. If the procedure you want to add is part of a program, then copy the source code into its own file and compile or assemble it separately.
2. Add, delete, or replace the module with the command line

   LIB *oldlibrary commands*;

   in which *commands* consists of one or more LIB commands that use the syntax shown later in this section.

Note that in step 2 above, the command line does not use all the LIB fields. You may, however, include a *listfile* if you want a file listing. You may also use the *newlibrary* field to preserve old library contents. If you enter a *newlibrary*, LIB places the updated library contents in *newlibrary* and leaves the contents of *oldlibrary* unchanged. Otherwise, LIB updates the contents of *oldlibrary* and saves the old contents in the file *oldlibrary*.BAK.

You can use the library as input to the linker once the contents change. Any routines you have added or replaced become part of the library and can be called by your programs.

Adding a Module

To add an object file to a library, use the command

+file

in which *file* is the name of the object file you want to add as a module. You may specify a complete path name for *file* if the object file is not in the current directory. If the file-name extension is .OBJ, you can leave off the extension; LIB assumes the .OBJ extension by default. LIB adds the object module at the end of the library. The library contains only the base name of the module without the .OBJ extension.

For example, the following command line adds the module `PRINTOUT` to the library `MYLIB.LIB`, by copying the contents of the object file `\SOURCE\PRINTOUT.OBJ`:

```
LIB MYLIB +\SOURCE\PRINTOUT;
```

You can also add the entire contents of one library to another by specifying a library name for *file*. Remember to enter a complete file name (including extension) because LIB assumes that files in the *commands* field have the .OBJ extension. For example, the following command line adds the complete contents of the library `SMALL.LIB` to the library `SUPER.LIB`:

```
LIB SUPER +SMALL.LIB;
```

Deleting a Module

To delete an object module from a library, use the command

-module

in which *module* is the name of a module already stored in the library. For example, the following command deletes the module `DELETEME` from the library `BIGLIB.LIB`:

```
LIB BIGLIB -DELETEME;
```

Replacing a Module

To replace an object module within a library, use the command

-+module

in which *module* is the name of a module that is currently stored in the library. The old copy of *module* is deleted from the library. The current contents of *module*.OBJ are copied into the library. For example, to replace the QuickC

small-model library version of **printf()** with your own version, execute these steps:

1. Write your own version of **printf()**, and compile or assemble it.
2. Make sure that the resulting object file is named `PRINTF.OBJ` and that `PRINTF.OBJ` is located in the current directory. (If you look at a listing of the library, you will see that the public symbol for the **printf()** function is **_printf()**. The name of the module, however, is **printf()**. If you have any doubt about the exact name of an object module, get a listing of the library before trying to modify the library.)
3. Issue the following command line:

```
LIB SLIBCE -+PRINTF;
```

You can combine any number of operations in the *commands* field. Spaces between the commands are acceptable but not necessary. For example, the following command line adds a new module, `NEWFUN`, replaces a current module, `OLDFUN`, and deletes another current module, `BYENOW`:

```
LIB MYLIB +NEWFUN -+OLDFUN -BYENOW;
```

In the example above, the files `NEWFUN.OBJ` and `OLDFUN.OBJ` serve as input for the modules `NEWFUN` and `OLDFUN`, respectively.

Copying and Moving Object Modules From a Library

You can extract any object module from a library. The extracted object module is copied into an .OBJ file with the same name as the module. For example, if you extract a module named `OLDFUN`, LIB copies it into the object file `OLDFUN.OBJ`. If a file with that name already exists, its contents are overwritten.

To copy a module into an .OBJ file, use the command

**module*

in which *module* is the name of the module you wish to copy from the library. The module is placed in the file *module*.OBJ.

For example, the following command line copies the **printf()** module from the Microsoft QuickC small-model library, and places the contents of this module into the object file `PRINTF.OBJ`:

```
LIB SLIBCE *PRINTF;
```

You can move a module out of a library with the following command:

*–*module*

Moving a module is similar to copying a module, in that LIB copies the contents of the module into a file named *module*.OBJ. The move command (–*), however, causes LIB to delete the module from the library after copying it.

Creating a New Library

When you use LIB, creating a new object-code library is easy. You simply combine two techniques:

- In the *oldlibrary* field, enter the name of a file that does not yet exist.
- In the command field, use the add command (*+file*) to list entries for the new library. (Technically, this step is not required; however, if you do not use the add command, the library will be empty.)

For example, if the file `NEWLIB.LIB` does not yet exist, the following command line creates this file. The object files `MYPROC`, `MYFUN`, and `PRINTIT` provide the input for the new library.

```
LIB NEWLIB +MYPROC +MYFUN +PRINTIT;
```

Other Ways of Using LIB

This chapter has covered the basic operations of the LIB utility, so that you can quickly begin to create and maintain your own libraries. For a complete description of LIB, see Chapter 6, "LIB," in Part 2, "Reference to QuickC Tools." Some additional features described in that chapter include the following:

- How to make LIB case sensitive, so that it treats `Print` and `PRINT` as two different module names.
- How to specify alignment of modules within a library.
- How to let LIB prompt you for command fields, rather than requiring you to enter them all on a single command line.
- How to use a response file to give input to LIB. Response files are useful for giving unusually long command lines or for giving the same command line repeatedly.

CHAPTER 3

3

Maintaining Programs with NMAKE

NMAKE, the Microsoft Program-Maintenance Utility, helps to automate software development and maintenance. Following instructions that you supply, NMAKE determines whether a program is out of date and, if so, how to update it. Your instructions list all the sources, include files, and libraries the program depends on, and specify the commands to update the program.

NMAKE, however, is not limited to updating programs. It can also perform other actions, such as building distribution disks, cleaning up directories, and so forth. Any procedure that requires the latest version of several files is a good candidate for NMAKE. By using NMAKE for these operations instead of doing them by hand, you can avoid the headaches of invalid source modules, old libraries, and forgotten include files.

NMAKE is typically used in the following situations:

- In program development, to update an executable file whenever any of the source or object files has changed
- In library management, to rebuild a library whenever any of the modules in the library has changed
- In a networking environment, to update the local copy of a file that is stored on the network whenever the master copy has changed

This chapter describes what NMAKE does, defines the terms you need to understand, and tells you how to use NMAKE to manage your QuickC projects.

If you are unfamiliar with NMAKE, this chapter will introduce its most useful features. If you are an experienced MAKE user, you will find that this new utility is different; in fact, you may need to change your existing description files to make them compatible. (See Section 7.5 for a summary of changes.) This tutorial chapter also serves as an introduction to some of the new features.

For detailed information see Chapter 7, "NMAKE," in Part 2, "Reference to QuickC Tools."

How NMAKE Works

NMAKE relies on a "description file," sometimes called a "makefile," to determine which files to update, when to update them, and what operations to perform. The description file defines the dependencies among the files in the project and gives the commands NMAKE must execute to bring everything up to date. For a QuickC program comprising several object files, the description file lists the source and header files needed to build each object file, and all the object files needed to build the executable program. The description file also contains the QCL commands that must be executed to build the program.

Description files are made of several elements:

- Description blocks, which tell NMAKE how to build files
- Macros, similar to C macros, which provide a shorthand notation that allows you to change certain values when the file is processed
- Inference rules, which tell NMAKE what to do in the absence of explicit commands
- Directives, which provide conditionals and other structuring techniques

A simple description block

All of these elements need not be present in every description file. For many applications, a description file consisting of a single description block is adequate. The example below shows a description file with only one description block:

```
program.exe : program.obj  sub1.obj #update program
        QCL program.obj sub1.obj
```

The first line of the description block is called the "dependency line." It identifies the "target" to be updated (`program.exe`) and the "dependents" that make up the target (`program.obj, sub1.obj`). If any of the dependents has changed since the target was last modified, NMAKE rebuilds the target. When NMAKE executes this description, it checks the date when each of the object files was last modified. If either has been modified since the executable program was created, NMAKE executes the second line called the "command line." The QCL command in the example relinks the program.

What about the C source files?

Note that the target is an executable file (.EXE) and its dependents are object files (.OBJ). You might wonder why the C source files `program.c` and `sub1.c` do not appear in the description block. The reason is that NMAKE assumes that .OBJ files depend on C source files and knows that it must compile `program.c` and `sub1.c` to create `program.obj` and `sub1.obj`. How and why NMAKE works this way are advanced topics covered later in the section titled "Using Inference Rules." You don't need to understand inference rules to create description files and use NMAKE.

Of course, if you prefer, you can make your target-executable files depend on the C source files and give the QCL command to compile and link the sources. It is, however, generally preferable to list the object files as dependents.

The next section in this chapter, "Building a Simple Description File," shows how to construct description files, such as the one above, that consist of a single block.

Building a Simple Description File

Before you invoke NMAKE, you need to create a description file. A description file is simply a text file, so you can use any text editor (including the one in the QuickC environment) to create one. NMAKE places no restrictions on the name of the description file, but always looks for a file named MAKEFILE in the current directory unless you tell it otherwise. The section below titled "Invoking NMAKE" gives more information on how NMAKE identifies the description file.

Depending on the size of the project you are maintaining, your description file may contain one or more description blocks. This section describes the components of a description block and shows you how to build description files that consist only of description blocks.

Description Blocks

Description blocks are the basic elements of description files. A description block tells NMAKE how to update a target from a group of dependents. Every description block consists of a dependency line, any number of command lines, and optional comments. Figure 3.1 shows the components of a description block.

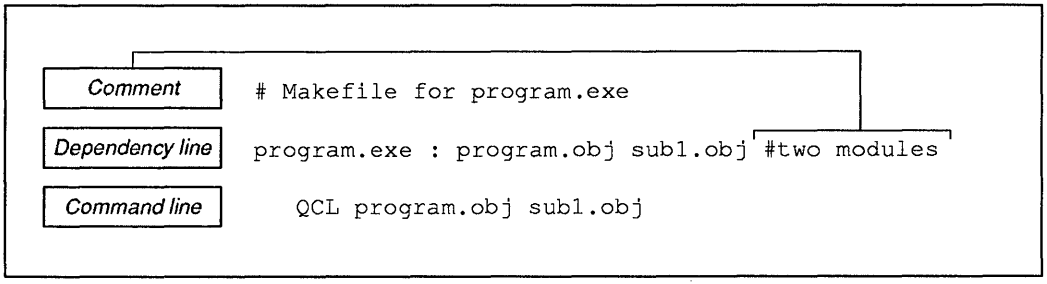


Figure 3.1 Components of a Description Block

Dependency Lines

A dependency line lists a target and one or more of its dependents. A colon (:) separates the target from the dependents.

Targets The name of the target goes at the beginning of the line with no tabs or spaces preceding it. NMAKE creates the target in the current directory unless you include a drive and path specification in its name. A dependency line may contain more than one target, but at least one space must separate each pair of names. Below are some example target names:

```
test1.exe :

c:\cprogs\test1.exe :

test1.exe test2.exe :
```

The first example specifies the target `test1.exe` in the current directory. In the second, the target is built in the directory `c:\cprogs`. The last lists two targets to be built from the same set of dependents.

Pseudotargets All the targets shown above are executable files. A target, however, need not be an executable file; in fact, it need not be a file at all. NMAKE allows you to specify the following:

```
UPDATE :
```

In this case, UPDATE is considered a "pseudotarget" because it is not a file but simply a label for a set of dependents and commands. Pseudotargets are useful for updating directories and copying groups of files. NMAKE always considers pseudotargets out of date.

Specifying dependents

List the names of the dependent files on the same line as the target, but after the colon. Separate the dependent names by one or more spaces. A target can have any number of dependents. If the names of all the dependents do not fit on one line, use a backslash (\) to end the line and continue the list on the next line. This is NMAKE's standard method of line continuation.

Dependent names, like target names, may contain drive and path specifications. If you do not include a drive or path specification, NMAKE looks for the dependents in the current directory. For example:

```
mycprog.exe : mycprog.obj  \public\src\graphics.obj

UPDATE : *.c \inc\*.h
```

The first example shows two dependents for `mycprog.exe`. One of them is `mycprog.obj`, for which NMAKE searches the current directory. The other is `graphics.obj`, for which NMAKE searches the `\public\src` directory. The second example shows that the pseudotarget `UPDATE` depends on all the .C files in the current directory and all the header files in the directory `\inc`.

Search paths for dependents

You can direct NMAKE to search for dependents in a sequence of other directories by adding a search path enclosed in braces. NMAKE uses the search path if it cannot find the file in the current directory. Separate each pair of directories in the path with a semicolon. The backslash at the end of the path is optional. Consider the following:

```
program.exe : program.obj {\me\cwork; q:\src\}tables.obj
```

This line lists two dependents for `program.exe`. The first, `program.obj`, is assumed to be in the current directory. For `tables.obj`, a search path is specified. The search path causes NMAKE to look first in the current directory, then in `\me\cwork`, and then in `q:\src` until it finds the file. If it cannot find the file, all is not lost; it relies on its inference rules to build the file. (For more information on inference rules, see "Using Inference Rules" in this chapter; for a more detailed description, see Section 7.3.3.)

Command Lines

The command lines in a description block give the commands to be carried out if a target is out of date with respect to any of its dependents. Commands can be the names of programs, batch files, or any DOS commands—in short, any command that can be issued on the DOS command line.

Rules for specifying commands

Typically, NMAKE users put their commands on separate lines from the target and its dependents, one command per line. Each line must start with one or more spaces or tab characters. If you forget the space or tab, NMAKE assumes you are specifying a dependency line (or a macro) and gives an error. You may find it helpful to use a tab to indent the line, making it easy to identify the commands that apply to each target. (This manual uses that convention.) For example:

```
program.exe : program.obj sub.obj
        qcl program.obj sub.obj
```

The command line in the example invokes QCL to link the two dependent files into a single executable image.

If you prefer, however, you can put your commands on the same line as the target and dependents. In that case, a semicolon must precede each command to separate it from the previous item on the line, whether a dependent or another command. The following has the same effect as the previous example:

```
program.exe : program.obj sub.obj ; qcl program.obj sub.obj
```

If a command is too long to fit on one line, you can split it across two or more lines with a backslash (\), in the same way that you split a long dependency list. For example:

```
program.exe : program.obj sub1.obj sub2.obj sub3.obj
        qcl program.obj sub1.obj sub2.obj  \
                sub3.obj
```

Be sure that every line that is part of a command begins with a space or tab.

Comments

You may put comments in your makefiles just as you do in your C programs. Every comment starts with a number sign (#) and extends to the end of the current line. NMAKE ignores all the text between the number sign and the next new-line character. Comments may appear anywhere in an NMAKE description file except on a command line. You may place comment lines among the command lines, but the number sign that starts the comment must be the first character on the line with no spaces or tabs preceding it. The following example shows the use of comments:

```
#makefile for program.exe
program.exe : program.obj sub1.obj sub2.obj
        qcl program.obj sub1.obj sub2.obj
#       program
```

The first comment documents the purpose of the file. The second causes NMAKE to treat the line `program` as a comment. When NMAKE executes this description, it will rebuild `program.exe` but will not run it.

Escape Character

Some characters, such as the number sign (#), have a special meaning when they appear in an NMAKE description file. If you want NMAKE to interpret these characters literally, and not with their special NMAKE meanings, you must use the caret (^) as an "escape" character.

For example, the number sign (#) denotes the start of a comment. To use it in a file name, you must precede it with a caret to "escape" its special meaning, as follows:

```
winning^#.txt
```

NMAKE interprets the example as the file name `winning#.txt`.

The following characters have special significance to NMAKE, so you must precede them with a caret whenever you want NMAKE to interpret them literally:

( ) $ ^ \ { } ! @ -

NMAKE ignores a caret that precedes any other character. In addition, carets that appear within quotation marks are not treated as escape characters.

Description-File Examples

Assume you are developing a program named `handle`. Your directories are organized so that all your source files and object files are stored under the current directory and your include files are in the `\inc` directory. Consider the following makefile:

```
handle.exe : main.obj comm.obj \inc\comm.h
        QCL /Fehandle.exe main.obj comm.obj
        handle
```

The dependency line says that `handle.exe` should be updated if any of three files change. Two of these files are object files; the third is an include file. If NMAKE determines that it must create a new version of the target, it executes the QCL command. QCL's /Fe option specifies the name `handle.exe` for the executable program. NMAKE executes the new version of `handle.exe` after creating it.

If the current directory contains only the files for `handle.exe`, and none for any other programs, the description file could be rewritten as follows:

```
handle.exe : *.obj \inc\comm.h
        QCL /Fehandle.exe *.obj
        handle
```

NMAKE expands the wild cards in the dependent names when it starts to build the target.

The CC Macro

The sample description files presented so far have contained only description blocks—no macros, directives, or inference rules. For the most part, you can get along fine without any of these features.

Before you use NMAKE with QuickC, however, you need to know about one particular macro, CC. The predefined macro CC tells NMAKE which C compiler to use when it tries to create .OBJ files from .C files. As you may be aware, NMAKE is provided with both QuickC and the Microsoft C Optimizing Compiler. For that reason, CC is predefined to invoke the C Optimizing Compiler, CL. You must redefine CC to invoke QCL, the Microsoft QuickC Compiler.

Redefining CC in a makefile

To redefine the CC macro, add the following at the top of your description file:

```
CC = qcl
```

No spaces or tabs may precede CC; it must be the first item on the line. The spaces around the equal sign are optional.

Continuing with the example in the previous section, the description file would look like the following:

```
CC = qcl
handle.exe : *.obj \inc\comm.h
        QCL /Fehandle.exe *.obj
        handle
```

This description has the same effect as before but ensures that NMAKE will use the QuickC compiler to generate the .OBJ files, if necessary, from any .C files in the current directory. The QCL command in the example invokes QuickC to link the object files into an executable file.

Redefining CC in the TOOLS.INI file

As an alternative, you can redefine CC in TOOLS.INI, the tools-initialization file. The TOOLS.INI file contains environment variables and initial (default) settings for various utility programs. You may already have a TOOLS.INI file; if not, you can create one with any text editor.

Items that apply to NMAKE appear in the file following the tag [nmake]. To change the definition of the CC macro, add a line that reads CC=qcl, as follows:

```
[nmake]
CC=qcl
```

Whenever you invoke NMAKE, the utility looks for TOOLS.INI first in the current directory and then in the directory specified by the INIT environment variable. To see what INIT is set to, type the SET command at DOS command level.

Invoking NMAKE

You can invoke NMAKE in either of two ways:

- By giving the NMAKE command and all options, macros, and targets on the DOS command line
- By giving the NMAKE command and the name of a response file that contains all the options, macros, and targets

This section describes both methods.

Invoking NMAKE from the DOS Command Line

Under most circumstances, you'll probably just issue the NMAKE command from the DOS command line. The command has the following format:

NMAKE *options macrodefinitions targets filename*

Optional

All the arguments are optional.

The *options* modify the action of the NMAKE command. The most commonly used NMAKE options are described under "NMAKE Options" below; the complete set is covered in Chapter 7.

The *macrodefinitions* give text to replace macro names in the description file. "Using Macros," later in this chapter, introduces macros and explains how and when to use them. See Section 7.3.2 for details.

The *targets* field lists one or more targets for NMAKE to build. If you do not specify a target, NMAKE builds the first one in the file. You can find more information on targets under "Description Blocks," above, and in Section 7.3.1.

Finally, the *filename* field gives the name of the description file that tells NMAKE what to do. Like the others, this field is optional. Thus, the simplest form of the NMAKE command is just

```
NMAKE
```

The default file MAKEFILE

When you invoke NMAKE with the preceding command, it looks in the current directory for a file named MAKEFILE to use as the description file. If no such file exists, it displays an error message.

When you give a file name, as in

```
NMAKE history.mak
```

NMAKE looks first for a file named MAKEFILE in the current directory, as in the previous example. If no such file exists, NMAKE treats the first argument on the command line that is not an option, a target, or a macro definition as the name of a description file. In the example above, NMAKE uses the description file `history.mak` from the current directory.

A third way is to use the /F option, described below in the section "Controlling Input."

Invoking NMAKE with a Response File

For more complicated updates, and whenever the NMAKE command line exceeds the DOS limit of 128 characters, you will need to create a response file. The response file contains the options, targets, and macros you would type on the DOS command line. It is *not* the same as the NMAKE description file; instead, it is comparable to a LINK or LIB response file.

To invoke NMAKE with a response file, issue the following command:

NMAKE @*responsefile*

For *responsefile*, use the name of the file that contains the options, targets, and macros you would otherwise type on the NMAKE command line.

NMAKE Options

NMAKE provides a rich set of options that control the descriptions it reads as input, the details of its execution, and the messages it displays on output. The sections below describe some of the most useful NMAKE options. Chapter 7 covers all the options in detail.

Options immediately follow the NMAKE command on the DOS command line and precede the name of the description file, if you supply one. NMAKE accepts options in uppercase or lowercase letters, with either a slash (/) or a dash (-) to introduce each option. For example, -F, /F, -f, and /f all represent the same option. In options that take file-name arguments, for example, /F and /X, the file name and the option must be separated by a space.

Controlling Input

The /F option specifies the name of the description file. This option has the following form:

/F filename

If you specify the /F option, NMAKE uses *filename* as the name of the description file. The space separating /F and *filename* is required. In place of a file name, you can enter a dash (-) to tell NMAKE to read the description from standard input, typically your keyboard.

If you omit the /F option, NMAKE looks for a file named MAKEFILE in the current directory. If none exists, NMAKE will take a file name from the command line as shown in the previous section.

NOTE *Unless you use the /F option, NMAKE always searches for the file MAKEFILE in the current directory. Therefore, you should explicitly specify /F to avoid unintentionally using MAKEFILE.*

The following is an example of the /F option:

```
NMAKE /F hello.mak
```

This command invokes the NMAKE utility and specifies `hello.mak`, in the current directory, as the description file.

Controlling Execution

The following options change the way NMAKE interprets the description file:

Option	Effect
/A	Builds all of the targets requested, even if they are not out of date.
/I	Ignores exit codes returned by commands executed within a description file. NMAKE continues processing the description file despite the errors.
/N	Displays the commands from the description file, but does not execute them. This option is useful for determining which targets are out of date without rebuilding them. It is also helpful in debugging description files.
/T	"Touches" any target files that are outdated. "Touching" a file causes its date of modification to be changed to the current date. It has no effect on the contents of the file.

Controlling Output

As NMAKE runs, it displays on standard output each command that it executes. It issues a diagnostic message if it cannot find a file or command needed to complete a description block or if any command returns an error. You can change the type and number of messages that NMAKE returns by using the options below:

Option	Effect
/C	Suppresses the Microsoft copyright message and all non-fatal or warning messages.
/D	Displays the modification date of each target or dependent file when it checks the date.
/P	Prints all macro definitions and target descriptions.
/S	Executes "silently"; does not display commands as they are executed.
/X *filename*	Sends all error output to *filename*. A space must separate /X from *filename*. Specifying a dash (-) instead of a file name sends error output to the standard output device.

Options Examples

The following command invokes NMAKE with `physics.mak` as the description file:

```
NMAKE /F physics.mak /N
```

The /N option tells NMAKE to read, but not to execute, any of the commands within the file `physics.mak`. NMAKE checks the modification dates on the files and displays the commands it would execute if the /N option were not present. This option is useful for finding out ahead of time what files are out of date so you can estimate how long a build might take. It can also be used in debugging description files.

After using the /N option to check what NMAKE would do, you might invoke it with the command line below:

```
NMAKE /F physics.mak /C /S
```

The /C option suppresses the NMAKE copyright message and any warning messages. The /S option suppresses the display of commands. You will, however, still see the copyright messages for any commands that NMAKE invokes and the output those commands generate.

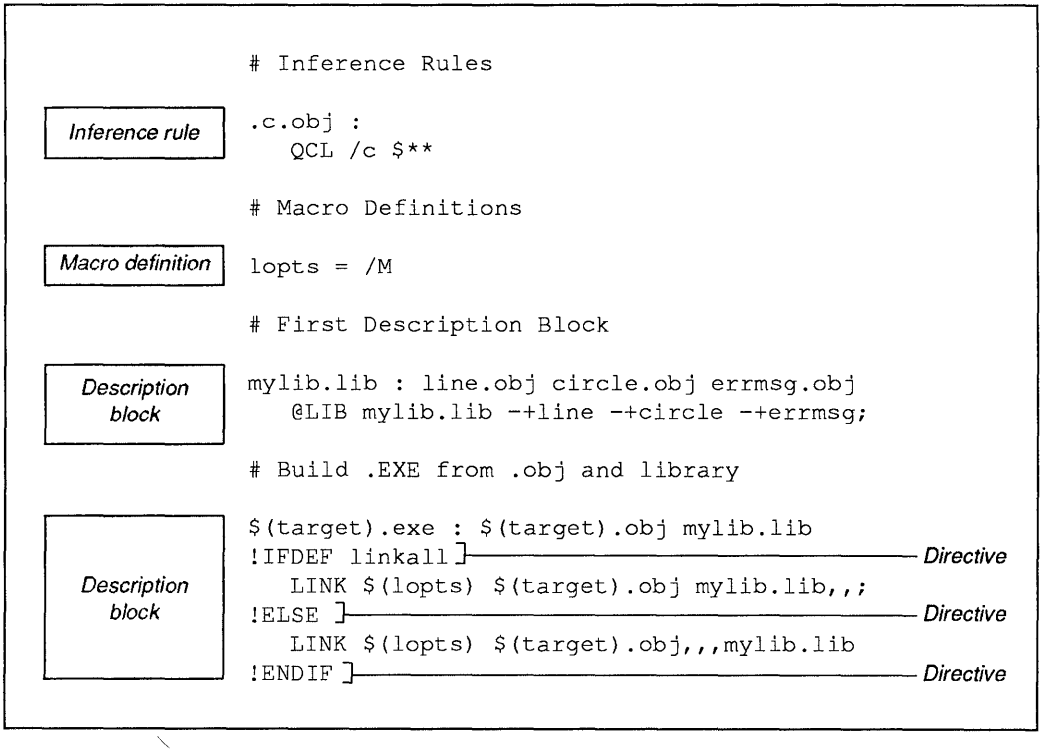
Building Complex Description Files

Most software projects can be maintained using the features already described. Description files for large projects, however, may become complicated and cumbersome, especially if each module is dependent on many source and include files. Using NMAKE's advanced features, you can shorten your description files and make them more powerful at the same time.

This section covers several of NMAKE's advanced features:

- Special characters on command lines
- Macros
- Inference rules
- Directives

Figure 3.2 shows a more complicated description file than those presented so far.




```
# Inference Rules

.c.obj :
   QCL /c $**

# Macro Definitions

lopts = /M

# First Description Block

mylib.lib : line.obj circle.obj errmsg.obj
   @LIB mylib.lib -+line -+circle -+errmsg;

# Build .EXE from .obj and library

$(target).exe : $(target).obj mylib.lib
!IFDEF linkall
   LINK $(lopts) $(target).obj mylib.lib,,;
!ELSE
   LINK $(lopts) $(target).obj,,,mylib.lib
!ENDIF
```

Figure 3.2 A More Complex Description File

Using Special Characters to Modify Commands

NMAKE recognizes three special characters that modify its treatment of commands. These characters give you additional control over how the individual commands are executed, whereas NMAKE's options apply to all the commands in the description file.

The characters go in front of the command name and may be separated from the name by one or more spaces, though they need not be. At least one space or tab must precede the character on the line. To use two or three special characters with a single command, put them one after the other on the command line. The special characters are as follows:

Character	Action
Dash (-)	Turns off error checking for the command it precedes so that NMAKE continues executing if an error occurs. A dash followed by a number suspends error checking for error levels at the number and below.
At sign (@)	Suppresses display of the command when it is executed.
Exclamation point (!)	Causes the command to be executed iteratively, once for each dependent file, if it uses one of the macros for dependent names. (The macros are described in the next section.)

Note that the dash (-) has the same effect as the /I option. Also, the "at" sign (@) is similar to the /S option.

Examples

```
beatles.exe : john.asm paul.c george.c ringo.c
        -QCL /c paul.c george.c ringo.c
        MASM john
        LINK john paul george ringo, beatles.exe;
```

In the example above, the dash preceding the QCL command means that NMAKE will attempt to execute the MASM and LINK commands even if errors occur during compilation.

```
beatles.exe : john.asm paul.c george.c ringo.c
        -@QCL /c paul.c george.c ringo.c
        MASM john
        @LINK john paul george ringo, beatles.exe;
```

The description in this example has the same effect as that in the previous example, except that neither the QCL nor the LINK command will be displayed when it is executed.

Using Macros

One way to streamline your description files is to use macros. A macro is a name that replaces other text in the description file in the same way that a macro defined in a QuickC **#define** directive replaces other text in a program. Wherever the name appears in the description file, NMAKE substitutes the text associated with it. To change the meaning of the name, you simply change the text assigned to it in the macro definition.

Macros are most useful in two situations:

- To replace all or part of a file name so that a single NMAKE description file can be used to update more than one program.
- To supply options for commands within the NMAKE description file. For example, you might define a macro to represent your usual debug options for the QCL command. Then, to compile with a different set of options, you need not edit the description file. You merely change the macro definition.

NMAKE provides two types of macros: predefined macros and macros you define. This section shows how to use them.

Defining Your Own Macros

A "macro definition" tells NMAKE what text to substitute for a macro. You can put macro definitions in the description file, on the NMAKE command line, or in your TOOLS.INI file. In the description file, each macro definition must be on a separate line. On the command line, macro definitions follow any NMAKE options and precede any targets. In the TOOLS.INI file, macro definitions appear in a section following the tag [nmake], each on a separate line, as described previously in the section "The CC Macro."

No matter where you put them, macro definitions take the following form:

macroname = string

The *macroname* is the name you use in the description file. A macro name may consist of any alphanumeric characters and the underscore (_) character. The *string* is the text that replaces the macro name when the description file is processed.

Macros on the command line

On the NMAKE command line, white space may not appear on either side of the equal sign because it causes DOS to treat the macro name and its definition as separate tokens. In addition, if *string* contains any embedded white space, you must enclose it in double quotation marks, as follows:

```
my_macro="this string"
```

Alternatively, you may enclose the entire macro definition—*macroname* and *string*—in double quotation marks. In that case, spaces may surround the equal sign because DOS treats all the characters within quotation marks as a single token. Thus, the following is also acceptable:

```
"my_macro = this string"
```

Macros in the description file

In a description file, define each macro on a new line. The definition must start at the beginning of the line with no preceding white space. NMAKE ignores any white space surrounding the equal sign. Quotation marks are unnecessary as well; if you use them, they will become part of the string.

This example defines a macro named `pname` and another named `t`:

```
pname = mycprog.exe

t = this
```

To use a macro within a command or dependency line, specify its name in parentheses preceded by a dollar sign ($), as follows:

$(*macroname*)

If you need to use a literal dollar sign in a description file, type it twice ($$) or use the caret (^) escape character.

The lines below show how to refer to the macros defined in the previous example. Note that if the name of a macro is only one character long, you can omit the parentheses.

```
$(pname)

$t
```

Once a macro is defined, the only way to remove its definition is to use the !UNDEF directive. See "Using Directives," later in this chapter, for more information.

Example

A common use of macros is to specify the options for a command. For example, the following description block uses the macro `copts` to represent QCL options.

```
picture.exe : picture.c graphics.c fileio.c
        qcl $(copts)  picture.c graphics.c fileio.c
```

Assuming the description file is named `picture.mak`, the command line might be the following:

```
NMAKE /F picture.mak copts="/C /P"
```

At execution time, NMAKE substitutes `/C /P` wherever `$(copts)` appears in the description file. The result is the same as if the following description were used:

```
picture.exe : picture.c graphics.c fileio.c
        qcl /C /P  picture.c graphics.c fileio.c
```

Note that the /P option causes QuickC to create a preprocessor listing, and the /C option retains the comments from the source files in the preprocessor listing.

Predefined Macros

Some macros are predefined by NMAKE. You have already seen one of these, CC. Some of the other predefined macros are described below. For a complete list, see Section 7.3.2.3.

Macros for Program Names (CC, AS, MAKE) The CC macro, already introduced, represents the C compiler command that NMAKE executes to create object files from C source files. The AS macro is similar. It stands for the name of the assembler that NMAKE executes when it needs to create object files from .ASM sources. Both of these macros are predefined by NMAKE. You can change their definitions in the description file, in the TOOLS.INI file, or on the NMAKE command line. Their default definitions are the following:

CC = cl
AS = masm

These two macros are primarily used in inference rules. (See "Using Inference Rules" in this chapter, or Section 7.3.3, for information on inference rules.)

The MAKE macro is defined as the command you used to invoke NMAKE. Use this macro, rather than the NMAKE command itself, to invoke NMAKE recursively within a description file. Recursion is typically used in building large software projects, such as compilers, and frequently involves the use of conditional directives. An example of the recursive use of NMAKE appears later in this chapter in the section titled "Conditional Directives."

Macros for Target Names ($@, $*) The $@ macro represents the full name of the target and the $* macro represents the base name of the target, that is, the full name with the extension deleted. These two macros are typically used in inference rules but, for the sake of discussion, this section will show their use in description files.

Consider the following description block:

```
$(target) : picture.obj graphics.obj fileio.obj
        LINK picture.obj graphics.obj fileio.obj, $@;
        $*
```

Assume the file is invoked with the command line that follows:

```
NMAKE target=trees.exe
```

The command line supplies text for the macro `target`, which sets the full name of the target to `trees.exe`. At execution time, NMAKE substitutes the text for the macro as explained in the previous section. However, this file goes one step further. Instead of repeating the user-defined `$(target)` macro as the output of the LINK command, it uses the predefined macro $@. This macro stands for the full name of the target and therefore has the same meaning as `$(target)`. Thus, the LINK command links the object files into `trees.exe`. In the last line of the file, the macro $* stands for the base name of the target. This line causes `trees.exe` to be executed as a program.

NMAKE automatically substitutes for these macros. It picks up the target name from its position on the dependency line in the description file. You cannot assign a value to a predefined macro on the command line.

NMAKE provides additional predefined macros that you can use to specify target names. See Section 7.3.2.3 for details.

Macros for Dependent Names ($**, $?) These macros signify the names of one or more dependents. The $** macro represents the complete list of dependent files for the target. The $? macro represents only the dependents that are out of date relative to the target. These two macros are commonly used with the special characters that modify commands to prevent NMAKE from doing any more work than necessary.

The example below shows the description file from the previous section using macros for the dependent names:

```
$(target) : picture.obj graphics.obj fileio.obj
        LINK $**, $@;
        $*
```

The first line of the example defines all the dependents for the target. On the next line, the LINK command links all the dependents, represented by $**, into a single executable image. Finally, the target is run as a program.

NMAKE provides additional predefined macros that you can use to specify dependent names. See Section 7.3.2.3 for details.

Precedence of Macro Definitions

Because macros can be defined in so many places, it is quite possible to give a macro more than one definition. Sometimes this is desirable. For instance, you may wish to override a macro definition for a single execution of the makefile.

NMAKE's precedence rules determine which macro definition it uses. Precedence depends on where the macro is defined. The following lists the order of precedence from highest to lowest priority:

1. Macros defined on the NMAKE command line
2. Macros defined in the description file and in files included in the description file with the !INCLUDE directive (see "Using Directives" below)
3. Macros defined by environment variables
4. Macros defined in the TOOLS.INI file
5. Macros defined by NMAKE, such as CC and AS

Figure 3.3 shows how macros defined on the command line take priority over those in the description file.

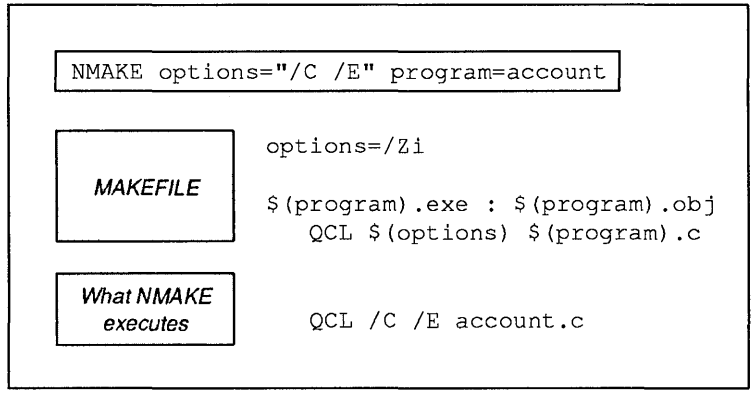


Figure 3.3 Precedence of Macro Definitions

In addition, you can force environment variables to override assignments in the description file. See Sections 7.2 and 7.3.2.4 for details.

Using Inference Rules

Most of the description blocks shown so far in this chapter contain commands to update the target from its dependents. Under certain conditions, however, NMAKE will follow a set of rules, called "inference rules," to create the target. Like macros, several inference rules are predefined, and NMAKE allows you to define your own.

If you supply a description block that does not contain any commands, or if the dependents of your target do not exist, NMAKE relies on inference rules. Whether predefined or user defined, inference rules are based on the file-name extensions of the target and dependent files. In short, they tell NMAKE how to create a file with a particular extension from a file with the same base name and a different extension.

Below is a simple inference rule:

```
.c.exe :
        QCL $*.c
```

This rule defines how to make a file with the .EXE extension from a file with the same base name and the .C extension. The first line says that the rule tells how to go *from* a file with the .C extension *to* a file with the .EXE extension. The second line gives the command that creates the .EXE file—in this case, the QCL command. The macro $* represents the base name of the target with the extension deleted.

Note that an inference rule looks very similar to a description block, with two exceptions:

1. An inference rule lists two file-name extensions instead of target names.
2. Inference rules do not list dependents.

If the preceding rule were in effect, NMAKE would use it for the following description block:

```
zfile.exe : zfile.c
```

NMAKE applies the inference rule for three reasons: first, the description block does not contain any commands; second, the file-name extensions of the target file and its dependent match those in the rule; and third, the base name of the target and dependent are the same. The combination of the inference rule and the description above has the same effect as the following description block:

```
zfile.exe : zfile.c
        QCL zfile.c
```

Predefined Inference Rules

NMAKE has three predefined inference rules. The predefined rules make use of the default macros CC and AS and several of the predefined macros that have already been presented.

From .C to .OBJ One predefined rule builds .OBJ files from .C files, as follows:

```
.c.obj:
        $(CC) $(CFLAGS) /c $*.c
```

When NMAKE applies this rule, it substitutes the current values of the macros CC and CFLAGS for $(CC) and $(CFLAGS). (The CFLAGS macro lists options for the C compiler.) It then looks for a C source file with the same name as the target and compiles the source file without linking. This is the rule NMAKE uses for the examples in this chapter that list .OBJ files—not C source files—as dependents.

With the description block below, NMAKE would use this inference rule if it needed to create or update one or more of the .OBJ files listed in the dependency list:

```
menu.exe : menu.obj funcs.obj draw.obj
        LINK menu funcs draw;
```

If the current directory contains .C source files with the same base names as the .OBJ files in the example, NMAKE compiles them according to the inference rule.

From .C to .EXE Another predefined rule, shown below, builds .EXE files from .C files:

```
.c.exe:
        $(CC) $(CFLAGS) $*.c
```

This rule causes NMAKE to use the same files as the previous rule but to link the output into an executable image. Continuing with the example, NMAKE would use this rule if the description file contained the following:

```
menu.exe : menu.c
```

Note that the files `funcs.c` and `draw.c` are not shown here. NMAKE would not create .EXE files for them because their base names are different from that of the .EXE file that NMAKE is trying to create.

From .ASM to .OBJ The third predefined rule builds .OBJ files from .ASM files:

```
.asm.obj:
        $(AS) $(AFLAGS) $*;
```

This rule tells NMAKE to look for an assembly-language source file with the same name as the target file and to invoke the Macro Assembler to create an object file. (The AFLAGS macro lists options for the assembler command.) NMAKE would use this inference rule under the same conditions as the first rule. For example:

```
menu.exe : menu.obj funcs.obj draw.obj
        LINK menu funcs draw;
```

If the current directory contains .ASM files with the same base names as any of the .OBJ files, NMAKE uses this final inference rule.

Defining Inference Rules

The predefined inference rules are adequate for most situations. Nevertheless, NMAKE allows you to define your own inference rules in the description file or in your TOOLS.INI file. You can also define them in a separate file that is included in your description file. (See the next section for information on the !INCLUDE directive.) Inference rules cannot be defined on the NMAKE command line.

To define an inference rule, use a statement in the following form :

.fromext.toext:
    command
    .
    .
    .

The first line defines the types of files to which the rule applies. The extension of the "from" file is given first followed by the extension of the "to" file. The second and subsequent lines give the commands that NMAKE must execute to create a file with the "to" file extension from a file that has the same base name and the "from" file extension. You can specify one or more commands, just as in a description block.

Specifying a path for .toext or .fromext

Sometimes you may want to associate a directory with each type of file. For instance, some programmers organize all their source files in one directory and their object files in another. NMAKE allows you to precede each of the extensions with a path, as follows:

{frompath}.fromext{topath}.toext

The example below shows a rule that starts with source files in one directory and creates object files in a different directory:

```
{\usr\graphics\source}.c{\usr\graphics\obj}.obj
```

You may specify only one path for each extension. If you need to pull source files from several different directories and place all the object files in one directory, you must define a separate inference rule for each source directory.

Precedence of Inference Rules

Like macros, inference rules can be defined in several places, so where an inference rule is defined establishes its precedence. NMAKE applies inference rules in the following order from highest to lowest priority:

1. Inference rules defined in the description file or in files included in the description file by the !INCLUDE directive (described below under "Using Directives")

2. Inference rules defined in the TOOLS.INI file
3. Predefined inference rules

Using Directives

Directives provide additional control over the execution of commands beyond what you can do with macros and inference rules. Using directives, you can

- Include the contents of another file in your description file
- Conditionally execute a command or group of commands
- Issue error messages from within a description file

In effect, directives let you build description files that act like DOS batch files. Such description files are especially useful for large software projects where the work is divided among several people. A description file can compile each source file, build any necessary libraries, and link the entire program. If errors occur anywhere in the process, the description file can issue diagnostic messages, possibly take corrective action, or terminate execution.

Each directive begins on a new line in the description file. A directive starts with an exclamation point (!) as the first character on the line. NMAKE allows, but does not require, spaces between the name of the directive and the exclamation point.

The sections that follow describe several of the NMAKE directives. For information on all the directives, see Section 7.3.4.

The !INCLUDE Directive

The !INCLUDE directive is similar to the **#include** preprocessor directive in QuickC. When NMAKE comes across !INCLUDE, it reads the contents of another description file before continuing with the current description file. The !INCLUDE directive is useful for including a standard set of inference rules or macros in your description files. For example:

```
!INCLUDE rules.mak
```

The !INCLUDE directive in this example tells NMAKE to begin reading from the file `rules.mak` in the current directory and to evaluate the contents of `rules.mak` as part of the current description file.

If you enclose the file name in angle brackets (<>), NMAKE searches for the file in the directories specified by the INCLUDE environment variable.

Conditional Directives (!IF, !ELSE, !ENDIF)

The conditional directives allow you to specify blocks of commands to be executed depending on the value of a constant expression. A conditional block has the following form:

!IF *expression*
statements
!ELSE
statements
!ENDIF

If the value of *expression* is nonzero (true), NMAKE executes the statements between the !IF directive and the !ELSE directive. If the value of the constant expression is zero (false), NMAKE executes the statements between the !ELSE directive and the !ENDIF directive.

Expressions The *expression* may consist of integer constants, string constants, or program invocations that return constants. Integer constants can use the C unary operators for numerical negation (–), logical negation (!), and one's complement arithmetic (~); or the C binary operators, including arithmetic operators, bitwise operators, and logical operators. (See Section 7.3.4 for a complete list.) For string constants, only the equality (==) and inequality (!=) operators are valid. You can use parentheses to group expressions wherever necessary. Program invocations, when used in conditionals, must be enclosed in square brackets.

Recursion Conditional directives are commonly used to test whether a program executed successfully. The program can be a DOS command, a program you have written, or even NMAKE itself. In the following description block, note the use of the macro $(MAKE) to invoke the program recursively:

```
$(target) : picture.obj fileio.obj error.obj
# Try to build pix.lib
!IF ![$(MAKE) /f pix.mak]
        LINK $**,$(target),,pix.lib;
        COPY pix.lib \mylibs
!ELSE
#Build didn't work, so link with old version
        LINK $**,$(target),,\mylibs\pix.lib;
!ENDIF
```

In this case, the expression is the value returned by another invocation of NMAKE. NMAKE, like many programs, returns the value 0 if it executes successfully and a nonzero errorlevel code otherwise. This is the opposite of the usual conditional test, which considers zero to be true and nonzero to be false. Therefore, the !IF directive must test the logical negation of the expression; that is, it uses the exclamation-point operator outside the square brackets.

If the library `pix.lib` is built successfully, NMAKE executes the LINK and COPY commands on the two lines immediately following the !IF directive.

If the library cannot be built successfully, NMAKE executes the command following the !ELSE directive. This command links all the dependents (named by the special macro $**) with an old version of the library.

Testing for Macro Definitions (!IFDEF, !IFNDEF, !UNDEF)

The !IFDEF and !IFNDEF directives test whether a macro is defined and execute a block of statements depending on the result. You use these two directives with the !ELSE and !ENDIF directives to construct conditional blocks, as described in the previous section.

The description block below shows the use of !IFDEF and !IFNDEF directives:

```
$(target) : picture.obj fileio.obj error.obj
# Macro $(newlib) is defined to use new pix.lib
!IFDEF newlib
        LINK $**,$(target),,pix.lib;
!ELSE
#       Just link with existing version
        LINK $**,$(target),,\mylibs\pix.lib;
!ENDIF
```

When NMAKE encounters the !IFDEF directive, it checks whether `newlib` has been defined. If so, it executes the LINK command on the next line. If not, it executes the LINK command following the !ELSE directive.

NMAKE considers a macro to be defined if its name appears to the left of an equal sign anywhere in the description file or on the NMAKE command line. So, if the file MAKEFILE contains the above description, both of the commands below would result in execution of the statements following the !IFDEF directive:

```
NMAKE newlib=true target=eliot.exe

NMAKE newlib= target=eliot.exe
```

Even though the second command line sets `newlib` to the null string, `newlib` is still considered defined because its name appears to the left of the equal sign.

The !IFNDEF directive acts in exactly the same way as !IFDEF, except that the statements following it are executed only if the macro is not defined.

Once you have defined a macro, the only way to remove its definition is to use the !UNDEF directive. You might want to remove a macro definition before including another file, as in the following example:

```
!UNDEF opts
!INCLUDE newlib.mak
```

The !UNDEF directive ensures that the macro `opts` is not defined when the file `newlib.mak` is processed.

The !ERROR Directive

The !ERROR directive causes NMAKE to print some text, then quit processing the makefile. This directive is commonly used in conditionals to terminate execution when fatal errors occur. For example, when NMAKE comes across the conditional

```
!IF "$(continue)" == "n"
!ERROR  Could not continue because of errors.
!ELSE
        LINK $**, $@;
!ENDIF
```

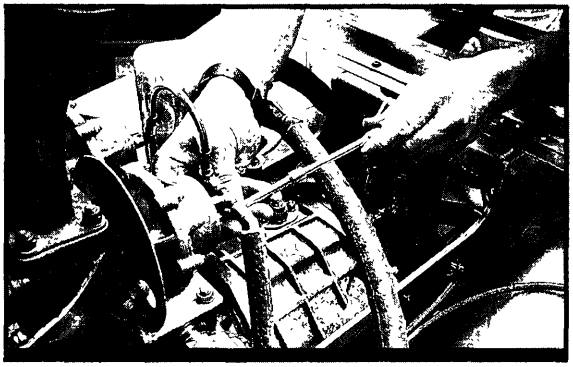
it tests the value of the macro `continue`. If `continue` holds the string `"n"`, NMAKE displays the text that follows the !ERROR directive then stops execution. If `continue` holds any other value, NMAKE executes the LINK command that follows the !ELSE directive.

Summary

This chapter has covered a subset of NMAKE intended to get you started but not to turn you into an overnight expert. In addition to the features described in this chapter, the NMAKE utility lets you

- Specify additional command-line options
- Specify more than one set of dependents for a target
- Create description files that build more than one target, and specify the target to build at invocation
- Use additional predefined macros
- Substitute text within macros
- Use additional directives
- Generate response files for use with other programs
- Use predefined "pseudotargets," which provide special rules and information

As you become more familiar with NMAKE, and as your software projects grow, you will probably need to use some of these features. See Chapter 7 for more information.



PART 2

Reference to QuickC Tools

PART 2

Reference to QuickC Tools

Part 2 of the *QuickC Tool Kit* is a reference to the tools. Here you find [illegible] information on all the options of the utilities. [illegible] have a specific question about one of the tools or [illegible] gained experience with the utilities and need [illegible] more about them.

[illegible] in great [illegible] in particular, those who program in other languages [illegible] (such as Pascal, FORTRAN, and Assembler) or with special hardware.

Part 2 also includes a chapter on the HELPMAKE program. HELPMAKE is a special tool that allows you to replace or customize help files supplied by Microsoft, including those for the

CHAPTERS

CHAPTER 4

4 QCL Command Reference

This chapter describes in detail the QCL command, which is used to compile and link QuickC programs. It explains the rules for giving input on the QCL command line, describes the options to QCL in alphabetical order, and shows how to change the stack space allocated to a program.

The chapter provides reference material for programmers who are familiar with the Microsoft QuickC Compiler in general and the QCL command in particular. If you are new to QuickC, see Chapter 1, "Creating Executable Programs," in Part 1, "QuickC Tools Tutorial" for an introductory approach.

4.1 The QCL Command Line

The QCL command line has the following format:

QCL [[*option...*]] *file...* [[*option*|*file*]]... [[/link [[*lib... link-opt...*]]]]

The following list describes input to the QCL command:

Entry	Meaning
option	One or more QCL options; see Section 4.3, "QCL Options," for descriptions.
file	The names of one or more source files, object files, or libraries. At least one file name is required. QCL compiles source files and passes object files and libraries to the linker.

lib	One or more library names. QCL passes these libraries to the linker for processing.
link-opt	One or more of the linker options described in Chapter 5, "LINK." The QCL command passes these options to the linker for processing.

Maximum command-line length

You may specify any number of options, file names, and library names, as long as the length of the command line does not exceed 128 characters.

Specifying file names

In file names, any combination of uppercase and lowercase letters is legal. Any file name can include a full or partial path name; if so, QCL assumes the file to be in that path. A full path name includes a drive name and one or more directory names. A partial path name omits the drive name, which QCL assumes to be the current drive. If no path name is given, QCL assumes the file is in the current directory.

QCL determines how to process each file depending on its file-name extension, as follows:

Extension	Processing
.C	QCL assumes the file is a C source file and compiles it.
.OBJ	QCL assumes the file is an object file and passes it to the linker.
.LIB	QCL assumes the file is a library and passes it to the linker. The linker links this library with the object files QCL created from source files and the object files given on the command line.
Any other extension or no extension	QCL assumes the file is an object file and passes it to the linker.

4.2 How the QCL Command Works

The QCL command follows the procedure described below to create an executable file from one or more C source files:

1. QCL compiles each source file, creating an object file for each. In each object file, QCL places the name of the appropriate standard combined library. The memory model and floating-point-math package used to compile the program determine this library name.

2. QCL invokes the linker, passing the object files it has created plus any object files or libraries given on the command line. The linker is invoked with the options listed in the LINK environment variable. If you use /link to specify linker options on the QCL command line, these options apply as well. If conflicts occur, options that follow /link override those in the LINK environment variable.

3. The linker links the object files and libraries passed by QCL to create a single executable file.

   Before it creates the executable file, the linker resolves "external references" in the object files. An external reference is a function call in one object file that refers to a function defined in another object file or in a library. To resolve an external reference, the linker searches the following locations in the order shown for the called function:

   a. The object files passed by QCL

   b. The libraries given on the QCL command line, if any

   c. The libraries named in the object files

Example

Assume that you are compiling three C source files: `MAIN.C`, `MOD1.C`, and `MOD2.C`. Each file includes a call to a function defined in a different file:

- `MAIN.C` calls the function named `mod1()` in `MOD1.C` and the function named `mod2()` in `MOD2.C`.
- `MOD1.C` calls the standard-library functions **printf** and **scanf**.
- `MOD2.C` calls graphics functions named `myline()` and `mycircle()`, which are defined in a library named `MYGRAPH.LIB`.

First, compile with a command line of the following form:

```
QCL MAIN.C MOD1.C MOD2.C /link MYGRAPH.LIB
```

In step 1, QCL compiles the C source files and creates the object files `MAIN.OBJ`, `MOD1.OBJ`, and `MOD2.OBJ`. It places the name of the standard library SLIBCE.LIB in each object file.

In step 2, QCL passes these source files to the linker. In step 3, the linker resolves the external references as follows:

1. `MAIN.OBJ` : resolves the reference to the `mod1()` function using the definition in `MOD1.OBJ` and resolves the reference to the `mod2()` function using the definition in `MOD2.OBJ`.
2. `MOD1.OBJ` : resolves the references to **printf** and **scanf** using the definitions in SLIBCE.LIB. The linker knows that this is the appropriate library because it finds the library name within `MOD1.OBJ`.
3. `MOD2.OBJ` : resolves the references to `myline` and `mycircle` using the definitions in `MYGRAPH.LIB`.

4.3 QCL Options

Options to the QCL command consist of either a forward slash (/) or a dash (-) followed by one or more letters. Certain QCL options take arguments; in some of these options, a space is required between the option and the argument, and in others, no space may appear. The spacing rules for each option are given in its description.

Important *QCL options (except for the /HELP option) are case sensitive. For example, /C and /c are two different options.*

Command-line order Options can appear anywhere on the QCL command line. With a few exceptions (/c, /Fe), each QCL option applies to the files that follow it on the command line and does not affect files preceding it on the command line. You can also define QCL options in the CL environment variable; these options are used every time QCL is invoked. (See Section 4.3.36, "Giving Options with the CL Environment Variable.")

The remainder of this section describes all the QCL options in alphabetical order. See Chapter 1, "Creating Executable Programs," for descriptions of the various categories of QCL options and the more commonly used options belonging to each category. If an option can take one or more arguments, its format is shown under an "Option" heading before its description.

4.3.1 /A Options (Memory Models)

A program's memory model defines the rules that the compiler uses to set up the program's code and data segments in memory. QCL offers the memory-model options described in Table 4.1.

Table 4.1 Memory Models

QCL Option	Memory Model	Data Segments	Code Segments
/AS	Small	One	One
/AM	Medium	One	One code segment per module
/AC	Compact	Multiple data segments; data items must be smaller than 64K	One
/AL	Large	Multiple data segments; data items must be smaller than 64K	One code segment per module
/AH	Huge	Multiple data segments; data items may be larger than 64K	One code segment per module

Default memory model

By default, the Microsoft QuickC Compiler uses the small memory model.

Uses of memory models

Generally, memory models with multiple code segments can accommodate larger programs than can memory models with one code segment, and memory models with multiple data segments can accommodate more data-intensive programs than can memory models with one data segment. Programs with multiple code or data segments, however, are usually slower than programs with a single code or data segment. It is often more efficient to compile with the smallest possible memory model and use the **near**, **far**, and **huge** keywords to override the default addressing conventions for any data items or functions that can't be accommodated in that model. (See Appendix B for more information about these keywords and their interactions with standard memory models.)

Memory models and default libraries

The memory-model and math options used to compile the program determine the library that the linker searches to resolve external references. The library name is *m*LIBC*f*.LIB, where the memory-model option determines *m*: S for small (default) model, M for medium model, C for compact model, or L for large or huge model. The math option (see Section 4.3.12, "/FP Options") determines *f*: E for emulator (default) or 7 for 8087/80287 option.

4.3.2 /c (Compile Without Linking)

The /c option tells the QCL command to compile all C source files given on the command line, creating object files, but not to link the object files. No executable file is produced. Regardless of its position on the command line, this option applies to all source files on the command line.

Example

```
QCL FIRST.C SECOND.C /c THIRD.OBJ
```

This example compiles `FIRST.C`, creating the object file `FIRST.OBJ`, and `SECOND.C`, creating the object file `SECOND.OBJ`. No processing is performed with `THIRD.OBJ` because QCL skips the linking step.

4.3.3 /C (Preserve Comments During Preprocessing)

The /C (for "comment") option preserves comments during preprocessing. If this option is not given, the preprocessor strips comments from a source file because they do not serve any purpose in later stages of compiling.

This option is valid only if the /E, /P, or /EP option is also used.

Example

```
QCL /P /C SAMPLE.C
```

This example produces a listing named `SAMPLE.I`. The listing file contains the original source file, including comments, with all preprocessor directives expanded or replaced.

4.3.4 /D (Define Constants and Macros)

Option

/D*identifier*[[={*string*|*number*}]]

Use the /D option to define constants or macros for your source file.

The *identifier* is the name of the constant or macro. It may be defined as a string or as a number. The *string* must be enclosed in quotes if it includes spaces.

If you leave out both the equal sign and the string or number, the identifier is assumed to be defined, and its value is set to 1. For example, `/DSET` is sufficient to define a macro named `SET` with a value of 1.

The /D option is especially useful with the **#if** directive to conditionally compile source files.

Removing definitions

If you have defined a numeric constant, giving the equal sign with no number removes the definition of that constant from the source file. For example, to remove all occurrences of `RELEASE`, use the following option:

```
/DRELEASE=
```

Note that the *identifier* argument is case sensitive. For example, the /D option above would have no effect on a constant named `release` that is defined in the source file.

Defining macros and constants with the /D option has the same effect as using a **#define** preprocessor directive at the beginning of your source file. The identifier is defined until either an **#undef** directive in the source file removes the definition or the compiler reaches the end of the file.

Duplicate definitions

If an identifier defined in a /D option is also defined within the source file, QCL uses the definition on the command line until it encounters the redefinition of the identifier in the source file, as illustrated in Figure 4.1.

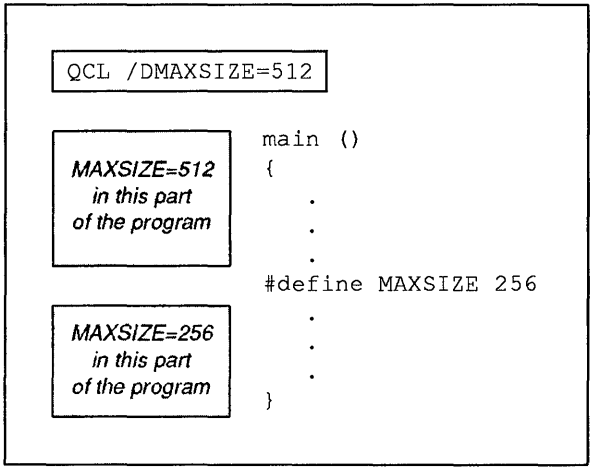


Figure 4.1 Duplicate Definitions with the /D Option

The /D option has the same effect as the Define text box in the QuickC environment.

Example

```
#if !defined(RELEASE)
        _nheapchk();
#endif
```

This code fragment calls a function to check the near heap unless the constant `RELEASE` is defined. While the program is under development, you can leave

RELEASE undefined and perform heap checking to find bugs. Assuming the program name is BIG.C, you would compile with the following command:

```
QCL BIG.C
```

After you have found all of the bugs in the program, you can define RELEASE in a /D option so that the program will run faster, as follows:

```
QCL /DRELEASE BIG.C
```

4.3.5 /E (Copy Preprocessor Output to Standard Output)

The /E option copies output from the preprocessor to the standard output (usually your terminal). This output is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. The /E option is normally used with the /C option (see Section 4.3.3), which preserves comments in the preprocessed output. DOS redirection can be used to save the output in a disk file.

The /E option also places a **#line** directive at the beginning and end of each included file and around lines removed by preprocessor directives that specify conditional compilation.

This option is useful when you want to resubmit the preprocessed listing for compilation. The **#line** directives renumber the lines of the preprocessed file so that errors generated during later stages of processing refer to the original source file rather than to the preprocessed file.

The /E option suppresses compilation. No object file or map file is produced, even if you specify the /Fo or /Fm option on the QCL command line.

Example

```
QCL /E /C ADD.C > PREADD.C
```

The command above creates a preprocessed file with inserted **#line** directives from the source file ADD.C. The output is redirected to the file PREADD.C.

4.3.6 /EP (Copy Preprocessor Output to Standard Output)

The /EP option is similar to the /E option: it preprocesses the C source file and copies preprocessor output to the standard output. Unlike the /E option, however, the /EP option does not add **#line** directives to the output.

Preprocessed output is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. The /EP option is normally used with the /C option (see Section 4.3.3), which preserves comments in the preprocessed output.

The /EP option suppresses compilation; no object file or map file is produced, even if you specify the /Fo or /Fm option on the QCL command line.

Examples

```
QCL /EP /C ADD.C
```

The command above creates a preprocessed file from the source file `ADD.C`. It preserves comments but does not insert **#line** directives. The output appears on the screen.

4.3.7 /F (Set Stack Size)

Option

/F *number*

The /F option sets the program stack size to *number* bytes, where *number* is a hexadecimal number in the range 0001 to FFFF. If this option is not given, a stack size of 2K is used.

You may want to increase the stack size if your program gets stack-overflow diagnostic messages. Conversely, if your program uses the stack very little, you may want to decrease the size of your program by reducing the stack size. In general, if you modify the stack size, you should not use the /Gs option to suppress stack checking.

4.3.8 /Fb (Bind a Program)

Option

/Fb[[*boundexe*]]

If you have installed the Microsoft C Optimizing Compiler, Version 5.1 or later, and have created protected-mode libraries, you can use the /Fb option to bind a program after compiling and linking. The *boundexe* argument specifies a name for the bound executable program. The name must follow the option immediately with no intervening spaces.

Binding permits the same executable file to run in both OS/2 protected mode and DOS 3.x (real mode). See the *Version 5.1 Update* document for the Microsoft C Optimizing Compiler for more information.

4.3.9 /Fe (Rename Executable File)

Option

/Fe*exefile*

By default, QCL names the executable file with the base name of the first file (source or object) on the command line plus the extension .EXE. The /Fe option lets you give the executable file a different name or create it in a different directory.

Because QCL creates only one executable file, you can give the /Fe option anywhere on the command line. If more than one /Fe option appears, QCL gives the executable file the name specified in the last /Fe option on the command line.

The /Fe option applies only in the linking stage. If you specify the /c option to suppress linking, /Fe has no effect.

Path names and extensions

The *exefile* argument must appear immediately after the option with no intervening spaces. The *exefile* argument can be a file specification, a drive name, or a path specification. If *exefile* is a drive name or path specification, the QCL command creates the executable file in the given location, using the default name (base name of the first file plus .EXE). A path specification must end with a backslash (\) so that QCL can distinguish it from an ordinary file name.

You are free to supply any name and any extension you like for *exefile*. If you give a file name without an extension, QCL automatically appends the .EXE extension.

Examples

```
QCL /FeC:\BIN\PROCESS *.C
```

The example above compiles and links all source files with the extension .C in the current directory. The resulting executable file is named `PROCESS.EXE` and is created in the directory `C:\BIN`.

```
QCL /FeC:\BIN\ *.C
```

The example above is similar to the first example except that the executable file, instead of being named `PROCESS.EXE`, is given the same base name as the first file compiled. The executable file is created in the directory `C:\BIN`.

4.3.10 /Fm (Create Map File)

Option

/Fm[[*mapfile*]]

The /Fm option produces a map file. The map file contains a list of segments in order of their appearance within the load module.

Path names and extensions

The *mapfile* argument must follow the /Fm option immediately with no intervening spaces. The *mapfile* can be a file specification, a drive name, or a path specification. It can also be omitted.

If you give just a path specification as the *mapfile* argument, the path specification must end with a backslash (\) so that QCL can distinguish it from an ordinary file name. For example, to create a map file in the path `C:\LIST`, the appropriate /Fm option is `/FmC:\LIST\`.

If you do not specify a name for the map file or if you supply only a drive name or path, QCL uses the base name of the first source or object file on the command line plus the extension .MAP.

A fragment of a sample map file is shown below:

```
Start  Stop   Length Name                   Class
00000H 01E9FH 01EA0H _TEXT                  CODE
01EA0H 01EA0H 00000H C_ETEXT                ENDCODE
.
.
.
```

Segment information

The information in the `Start` and `Stop` columns shows the 20-bit address (in hexadecimal) of each segment, relative to the beginning of the load module. The load module begins at location zero. The `Length` column gives the length of the segment in bytes. The `Name` column gives the name of the segment, and the `Class` column gives information about the segment type.

Group information

The starting address and name of each group appear after the list of segments. A sample group listing is shown below:

```
Origin   Group
01EA:0   DGROUP
```

In this example, DGROUP is the name of the data group. DGROUP is used for all near data (that is, all data not explicitly or implicitly placed in their own data segment) in Microsoft QuickC programs.

Global symbols

The map file shown below contains two lists of global symbols: the first list is sorted in ASCII-character order by symbol name and the second is sorted by symbol address. The notation `Abs` appears next to the names of absolute symbols (symbols containing 16-bit constant values that are not associated with program addresses).

Global symbols in a map file usually have one or more leading underscores because the QuickC compiler adds an underscore to the beginning of variable names. Many of the global symbols that appear in the map file are symbols used internally by the Microsoft QuickC Compiler and the standard libraries.

```
Address         Publics by Name

01EA:0096       STKHQQ
0000:1D86       _brkctl
01EA:04B0       _edata
01EA:0910       _end
.
.
.
01EA:00EC       __abrkp
01EA:009C       __abrktb
01EA:00EC       __abrktbe
0000:9876  Abs  __acrtmsg
0000:9876  Abs  __acrtused
.
.
.
01EA:0240       ___argc
01EA:0242       ___argv

Address         Publics by Value

0000:0010       _main
0000:0047       _htoi
0000:00DA       _exp16
0000:0113       __chkstk
0000:0129       __astart
0000:01C5       __cintDIV
.
.
.
```

The addresses of the external symbols show the location of the symbol relative to zero (the beginning of the load module).

Program entry point

Following the lists of symbols, the map file gives the program entry point, as shown in the following example:

```
Program entry point at 0000:0129
```

NOTE *If you use the /Fm option with the /Gi option (for incremental compilation), QCL produces a segmented-executable map file rather than a DOS executable map file. The segment addresses in the file are different from those in DOS map files, and the file itself has a different format.*

4.3.11 /Fo (Rename Object File)

Option

/Fo*objfile*

By default, QCL gives each object file it creates the base name of the corresponding source file plus the extension .OBJ. The /Fo option lets you give different names to object files or create them in a different directory. If you are compiling more than one source file, you can use the /Fo option with each source file to rename the corresponding object file.

Keep the following rules in mind when using this option:

- The *objfile* argument must appear immediately after the option, with no intervening spaces.
- Each /Fo option applies only to the next source file on the command line.

Path names and extensions

You are free to supply any name and any extension you like for the *objfile*. However, it is recommended that you use the conventional .OBJ extension because the linker and the LIB library manager use .OBJ as the default extension when processing object files.

If you do not give a complete object-file name with the /Fo option (that is, if you do not give an object-file name with a base and an extension), QCL names the object file according to the following rules:

- If you give an object-file name without an extension (such as `TEST`), QCL automatically appends the .OBJ extension.
- If you give an object-file name with a blank extension (such as `TEST.`), QCL leaves the extension blank.
- If you give only a drive or directory specification following the /Fo option, QCL creates the object file on that drive or directory and uses the default file name (the base name of the source file plus .OBJ).

You can use this option to create the object file in another directory or on another disk. When you give only a directory specification, the directory specification must end with a backslash (\) so that QCL can distinguish between a directory specification and a file name.

Examples

```
QCL /FoB:\OBJECT\ THIS.C
```

In the example above, the source file `THIS.C` is compiled; the resulting object file is named `THIS.OBJ` (by default). The directory specification `B:\OBJECT\` tells QCL to create `THIS.OBJ` in the directory named `\OBJECT` on drive B.

```
QCL /Fo\OBJECT\ THIS.C THAT.C /Fo\SRC\NEWTHOSE.OBJ THOSE.C
```

In the example above, the first /Fo option tells the compiler to create the object file `THIS.OBJ` (the result of compiling `THIS.C`) in the `\OBJECT` directory. The second /Fo option tells the compiler to create the object file named `NEWTHOSE.OBJ` (the result of compiling `THOSE.C`) in the `\SRC` directory. The compiler also creates the object file `THAT.OBJ` (the result of compiling `THAT.C`) in the current directory.

4.3.12 /FP Options (Select Floating-Point-Math Package)

The /FPi and /FPi87 options specify how your program handles floating-point-math operations.

4.3.12.1 /FPi (Emulator)

The /FPi option is useful if you do not know whether an 8087 or 80287 coprocessor will be available at run time. Programs compiled with /FPi work as follows:

- If a coprocessor is present at run time, the program uses the coprocessor.
- If no coprocessor is present or if the NO87 environment variable has been set, the program uses the emulator.

The /FPi option generates in-line instructions for an 8087 or 80287 coprocessor and places the name of the emulator library (*m*LIBCE.LIB) in the object file. At link time, you can specify an 8087/80287 library (*m*LIBC7.LIB) instead. If you do not choose a floating-point option, QCL uses the /FPi option by default.

Interrupt fix-ups

This option works whether or not a coprocessor is present because the Microsoft QuickC Compiler does not generate "true" in-line 8087/80287 instructions. Instead, it generates software interrupts to library code. The library code, in turn,

fixes up the interrupts to use either the emulator or the coprocessor depending on whether a coprocessor is present. The fix-ups can be removed by linking the file RMFIXUP.OBJ with the C program.

Linking this file with QuickC programs can save execution time (the time required to fix up all the interrupts the first time). However, a C program linked with RMFIXUP.OBJ will run only if a coprocessor is present.

4.3.12.2 /FPi87 (Coprocessor)

The /FPi87 option includes the name of an 8087/80287 library (*m*LIBC7.LIB) in the object file. At link time, you can override this option and specify an emulator library (*m*LIBCE.LIB) instead so that the program will run on computers without coprocessors.

If you use the /FPi87 option and link with *m*LIBC7.LIB, an 8087 or 80287 coprocessor *must* be present at run time; otherwise, the program fails and the following error message is displayed:

```
run-time error R6002
- floating point not loaded
```

If you compile with /FPi87 and link with *m*LIBCE.LIB, you can set the NO87 environment variable to suppress the use of the coprocessor (see Section 4.3.12.5).

Compiling with the /FPi87 option results in the smallest, fastest programs possible for handling floating-point arithmetic.

4.3.12.3 Library Considerations for Floating-Point Options

You may want to use libraries in addition to the default library for the floating-point option you have chosen on the QCL command line. For example, you may want to create your own libraries or object files, then link them at a later time with object files that you have compiled using different QCL options.

You must be sure that you use only one standard combined C library when you link. You can control which library is used in one of two ways:

1. Make sure the first object file passed to the linker has the name of the desired library. For example, if you want to use an 8087/80287 library, give the /FPi87 option before the first source-file name on the QCL command line; or give the name of an object file compiled with /FPi87 as the first file name on the command line. All floating-point calls in this object file refer to the 8087/80287 library.

2. Give the /NOD (no default-library search) option after the /link option on the QCL command line. Then specify the name of the library you want to use on the QCL command line. The /NOD option overrides the library names embedded in the object files. Because the linker searches libraries given on the command line before it searches libraries named in object files, all floating-point calls will refer to the libraries you specify.

Removing library names

Another complication might arise if you create your own libraries: normally, each module in the library you create will contain a standard-library name, and the linker will try to search the standard libraries named in the modules when it links with your library.

The safest course, especially when you are distributing libraries to others, is to use the /Zl option when you compile the object files that make up your libraries. The /Zl option tells the compiler not to put library names in the object files. Later, when you link other object files with your library, the standard library used for linking will depend only on the floating-point and memory-model options used to compile those object files.

Examples

```
QCL CALC.C ANOTHER SUM
```

In the example above, no floating-point option is given, so QCL compiles the source file `CALC.C` with the default floating-point option, /FPi. The /FPi option generates in-line instructions and selects the small-model-emulator combined library (SLIBCE.LIB), which is the default.

```
QCL /FPi87 CALC.C ANOTHER.OBJ SUM.OBJ /link SLIBCE.LIB /NOD
```

In the example above, `CALC.C` is compiled with the /FPi87 option, which selects the SLIBC7.LIB library. The /link option, however, overrides the default library specification: the /NOD option suppresses the search for the default library, and the alternate math library (SLIBCE.LIB) is specified. The resulting executable file, `CALC.EXE`, is linked with SLIBCE.LIB.

4.3.12.4 Compatibility between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link two or more source files to produce an executable program file, you are responsible for ensuring that floating-point operations are handled in a consistent way.

Examples

```
QCL /AM CALC.C ANOTHER SUM /link MLIBC7.LIB /NOD
```

The example above compiles the program `CALC.C` with the medium-model option (`/AM`). Because no floating-point option is specified, the default (/FPi) is used. The /FPi option generates in-line 8087/80287 instructions and specifies the emulator library MLIBCE.LIB in the object file. The /link field specifies the /NOD option and the name of the medium-model 8087/80287 library, MLIBC7.LIB. Specifying the 8087/80287 library forces the program to use an 8087 coprocessor; the program fails if a coprocessor is not present.

4.3.12.5 The NO87 Environment Variable

Programs compiled with the /FPi option automatically use an 8087 or 80287 coprocessor at run time if one is installed. You can override this and force the use of the emulator instead by setting an environment variable named NO87.

Coprocessor-suppression message

If NO87 is set to any value when the program is executed, the program will use the emulator even if a coprocessor is present. When this occurs, the NO87 setting is displayed on the standard output as a message. The message is displayed only if a coprocessor is present and its use is suppressed; if no coprocessor is present, no message appears. If you want to force use of the emulator, but don't want a message to appear, set NO87 equal to one or more spaces. The variable is still considered to be defined.

Note that the presence or absence of the NO87 definition determines whether use of the coprocessor is suppressed. The actual value of the NO87 setting is used only for the message.

The NO87 variable takes effect with any program linked with an emulator library (*m*LIBCE.LIB). It has no effect on programs linked with 8087/80287 libraries (*m*LIBC7.LIB).

Examples

```
SET NO87=Use of coprocessor suppressed
```

The example above causes the message `Use of coprocessor suppressed` to appear when a program that would use an 8087 or 80287 coprocessor is executed on a computer that has such a coprocessor.

```
SET NO87=space
```

The example above sets the NO87 variable to the space character. Use of the coprocessor is still suppressed, but no message is displayed.

4.3.12.6 Standard Combined Libraries

Table 4.2 shows each combination of memory-model and floating-point options and the corresponding library name that QCL embeds in the object file.

Table 4.2 QCL Options and Default Libraries

Floating-Point Option	Memory-Model Option	Default Library
/FPi87	/AS	SLIBC7.LIB
	/AM	MLIBC7.LIB
	/AC	CLIBC7.LIB
	/AL or /AH	LLIBC7.LIB
/FPi	/AS	SLIBCE.LIB
	/AM	MLIBCE.LIB
	/AC	CLIBCE.LIB
	/AL or /AH	LLIBCE.LIB

4.3.13 /G0, /G2 Options (Generate Instructions for 8086 or 80286 Processor)

If you have an 80286 processor, you can use the /G2 option to enable the instruction set for your processor. When you use the /G2 option, the compiler automatically defines the identifier **M_I286.**

Although it is usually advantageous to enable the appropriate instruction set, you are not required to do so. If you have an 80286 processor, for example, but you want your code to be able to run on an 8086, do not compile with the /G2 option.

The /G0 option enables the instruction set for the 8086/8088 processor. You do not have to specify this option explicitly because the 8086/8088 instruction set is used by default. Programs compiled this way will also run on the machines with the 80286 processor but will not take advantage of its processor-specific instructions. When you compile programs for the 8086/8088 processor, the compiler automatically defines the identifier **M_I8086.**

The /G0 and /G2 options also enable the assembling of instructions with in-line assembler. If your program includes in-line assembler code that uses a mnemonic instruction supported only on 80286 processors or 80287 coprocessors, you must compile with the /G2 option; compiling with /G0 results in an error. Note that it is illegal to use 80286 mnemonics as labels regardless of the processor option you choose.

These options apply to all file names that follow on the command line until another /G0 or /G2 option appears.

Note that you may also specify /G1, which allows 80186 instructions in in-line assembly. The code generated with the /G1 option, however, is restricted to the 8086 instruction set. This option is of limited usefulness.

4.3.14 /Gc (Use FORTRAN/Pascal Calling Convention)

The **fortran**, **pascal**, and **cdecl** keywords and the /Gc option allow you to control the function-calling and naming conventions so that your QuickC programs can call and be called by functions that are written in FORTRAN or Pascal.

Parameter-passing conventions

Because functions in Microsoft QuickC programs can take a variable number of arguments, QuickC must handle function calls differently from languages such as Pascal and FORTRAN. Pascal and FORTRAN normally push actual parameters to a function in left-to-right order so that the last argument in the list is the last one pushed on the stack. In contrast, because QuickC functions do not always know the number of actual parameters, they must push their arguments from right to left, so that the first argument in the list is the last one pushed.

Stack-cleanup conventions

Another difference between QuickC programs and FORTRAN or Pascal programs is that in QuickC programs, the *calling* function must remove the arguments from the stack. In Pascal and FORTRAN programs, the *called* function must remove the arguments. If the code for removing arguments is in the called function (as in Pascal and FORTRAN), it appears only once; if it is in the calling function (as in QuickC), it appears every time there is a function call. Because a typical program has more function calls than functions, the Pascal/FORTRAN method results in slightly smaller, more efficient programs.

The pascal and fortran keywords

The Microsoft QuickC Compiler can generate the Pascal/FORTRAN calling convention in one of several ways. The first is through the use of the **pascal** and **fortran** keywords. When these keywords are applied to functions, or to pointers to functions, they indicate a corresponding Pascal or FORTRAN function (or a function that uses the Pascal/FORTRAN calling convention). Therefore, the correct calling convention must be used. In the following example, `sort` is declared as a function using the alternative calling convention:

```
short pascal sort(char *, char *);
```

The **pascal** and **fortran** keywords can be used interchangeably. Use them when you want to use the left-to-right calling sequence for selected functions only.

The /Gc option

The second method for generating the Pascal/FORTRAN calling convention is to use the /Gc option. If you use the /Gc option, the entire module is compiled using the alternative calling convention. You might use this method to make it

possible to call all the functions in a QuickC module from another language or to gain the performance and size improvement provided by this calling convention.

When you use /Gc to compile a module, the compiler assumes that all functions called from that module use the Pascal/FORTRAN calling convention, even if the functions are defined outside that module. Therefore, using /Gc would normally mean that you cannot call or define functions that take variable numbers of parameters and that you cannot call functions such as the QuickC library functions that use the QuickC calling sequence. In addition, if you compile with the /Gc option, either you must declare the **main** function in the source program with the **cdecl** keyword, or you must change the start-up routine so that it uses the correct naming and calling conventions when calling **main**.

The cdecl keyword

The **cdecl** keyword in Microsoft QuickC is the "inverse" of the **fortran** and **pascal** keywords. When applied to a function or function pointer, it indicates that the associated function is to be called using the normal QuickC calling convention. This allows you to write QuickC programs that take advantage of the more efficient Pascal/FORTRAN calling convention while still having access to the entire QuickC library, other QuickC objects, and even user-defined functions that accept variable-length argument lists. The **cdecl** keyword takes precedence over the /Gc option.

For convenience, the **cdecl** keyword has already been applied to run-time-library function declarations in the include files distributed with the QuickC compiler. Therefore, your QuickC programs can call the library functions freely, no matter which calling conventions you compile with. Just make sure to use the appropriate include file for each library function the program calls.

Naming conventions

Use of the **pascal** and **fortran** keywords, or the /Gc option, also affects the naming convention for the associated item (or, in the case of /Gc, all items): the name is converted to uppercase letters, and the leading underscore that QuickC normally prefixes is not added. The **pascal** and **fortran** keywords can be applied to data items and pointers, as well as to functions; when applied to data items or pointers, these keywords force the naming convention described above for that item or pointer.

The **pascal**, **fortran**, and **cdecl** keywords, like the **near**, **far**, and **huge** keywords, are disabled by use of the /Za option. If this option is given, these names are treated as ordinary identifiers, rather than keywords.

Examples

```
int cdecl var_print(char*,...);
```

In the example above, `var_print` is declared with a variable number of arguments using the normal right-to-left QuickC function-calling convention and

naming conventions. The **cdecl** keyword overrides the left-to-right calling sequence set by the /Gc option if the option is used to compile the source file in which this declaration appears. If this file is compiled without the /Gc option, **cdecl** has no effect since it is the same as the default QuickC convention.

```
float *pascal nroot(number, root);
```

The example above declares `nroot` to be a function returning a pointer to a value of type **float**. The function `nroot` uses the default calling sequence (left-to-right) and naming conventions for Microsoft FORTRAN and Pascal programs.

4.3.15 /Gi (Use Incremental Compilation)

Option

/Gi[*mdtname*]

When the /Gi option is given, QCL compiles only those functions in each C source file that have changed since the last time the source file was compiled. The process of compiling only the changed functions in a source file is known as "incremental compilation." Because the compiler does not need to handle the entire source file, incremental compilation is considerably faster than regular compilation. The object files created and the code generated when you compile incrementally, however, may be larger.

If you specify any of the optimization (/O*string*) options on the same line with /Gi, the compiler ignores the /Gi option.

Module-description table (MDT)

The compiler tracks changes for incremental compilation in a file known as a "module-description table," or MDT. A single MDT can contain change information for multiple source files. If *mdtname* is given, the compiler saves change information for all source files in a single MDT named *mdtname*. If you do not specify *mdtname*, the compiler creates one MDT for each source file named on the command line. Each MDT has the base name of the source file and the .MDT extension.

The types of changes made to a source file determine whether the compiler can incrementally compile a source file and whether the compiler creates or updates the corresponding MDT.

Incrementally compile, update MDT

Except as noted below, if changes are confined to function bodies, the QCL command compiles only those changed functions and the "global regions" of the source file. Global regions are the parts of the source file between the closing

curly brace (}) of one function and the opening curly brace ({) of the next function (see Figure 4.2). The compiler also updates the MDT to reflect the changes to the source file.

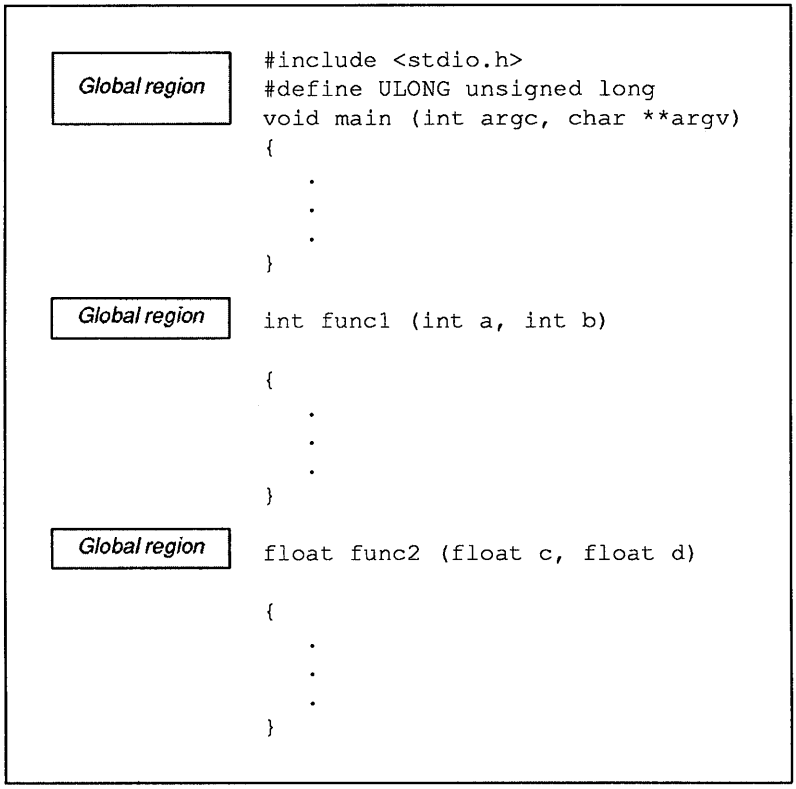


Figure 4.2 Global Regions for Incremental Compilation

If a global region of the source file has changed, QCL recompiles from the point where the change occurred. A change in a global region means any change in the storage-class specifier, type specifier, function declarator, or formal-parameter declarations of a function. Similarly, if a file specified in an **#include** directive has a more recent modification date than the including object module, the source file is recompiled from the point where the **#include** directive appears. In addition, if a function is defined within an include file, the source file is recompiled from the start of the function.

Compile whole program, don't update MDT

The compiler must recompile an entire source file, but does not update its MDT, in any of these cases:

- A function definition appears within an include file.
- The compiler does not have enough memory to create the MDT.

Using function prototypes

For fastest compilation with /Gi, use a prototype for each function in your program. A function prototype lists the name and type of the function and the name and type of each of its parameters. (See "Declaring a Function's Parameters," in Chapter 2 of *C for Yourself* for more information.) The C include files that Microsoft supplies contain prototypes for all the functions in the C run-time library. The information in the prototypes lets the compiler check the number and type of arguments to the function.

If you use the /Gi option and your program contains functions without corresponding prototypes, the compiler issues the following level 3 warning message:

```
no function prototype given
```

Compilation errors

When the /Gi option is given and errors occur during compilation, the compiler still creates a partial object file; that is, it generates object code up to the point where the error occurs. It places a record in each object file indicating that the object file is invalid. If you try to link one of these object files, the linker issues the following error message:

```
invalid object due to aborted incremental compile
```

Incremental linking

When the compiler can perform incremental compilation, it invokes a special form of the linker that performs "incremental linking." Like incremental compiling, incremental linking links only the object files that changed from the previous link. No library searches are performed; the assumption is that the libraries are exactly the same as in the previous link. Incremental linking is considerably faster than regular linking. If any of the changes to the program prevent QuickC from performing an incremental link, it automatically performs a full link.

NOTE *If you use the /Gi option with the /Fm option (which produces a map file), the map file is a segmented-executable map file rather than a DOS executable map file. The segment addresses in the file are different from those in DOS map files, and the file itself has a different format.*

Examples

Assume three C source files named `MOD1.C`, `MOD2.C`, and `MOD3.C` for the following examples.

```
QCL /Gi MOD1.C MOD2.C MOD3.C
```

The example above incrementally compiles and links the three C source files. Three MDTs are created or updated: `MOD1.MDT`, `MOD2.MDT`, and `MOD3.MDT`.

```
QCL /GiMYMDT.MDT MOD1.C MOD2.C MOD3.C
```

The example above has the same effect as the preceding example, except that the compiler creates or updates only one MDT named `MYMDT.MDT`. This MDT includes all change-control information for the three C source files.

4.3.16 /Gs (Turn Off Stack Checking)

You can reduce the size of a program and speed up execution slightly by removing stack probes. You can do this with either the /Gs option or the **check_stack** pragma. Note that the /Gs option and the **check_stack** pragma have no effect on standard C library routines because no stack checking is performed for these routines.

Stack probes

A "stack probe" is a short routine called on entry to a function to verify that the program stack has enough room to allocate local variables required by the function. The stack-probe routine is called at every function-entry point. Ordinarily, the stack-probe routine generates a stack-overflow message if the required stack space is not available. When stack checking is turned off, the stack-probe routine is not called, and stack overflow can occur without being diagnosed (that is, no stack-overflow message is printed).

When to use the /Gs option

Use the /Gs option when you want to turn off stack checking for an entire module if you know that the program does not exceed the available stack space. For example, stack probes may not be needed for programs that make very few function calls or that have only modest local-variable requirements. In the absence of the /Gs option, stack checking is on. The /Gs option should be used with great care. Although it can make programs smaller and faster, it may mean that the program will not be able to detect certain execution errors.

When to use the check_stack pragma

Use the **check_stack** pragma when you want to turn stack checking on or off only for selected routines, leaving the default (as determined by the presence or absence of the /Gs option) for the rest. When you want to turn off stack checking, put the following line before the definition of the function you don't want to check:

```
#pragma check_stack (off)
```

Note that the preceding line disables stack checking for all routines that follow it in the source file, not just the routines on the same line. To reinstate stack checking, insert the following line:

```
#pragma check_stack (on)
```

If no argument is given for the **check_stack** pragma, stack checking reverts to the behavior specified on the command line: disabled if the /Gs option is given, or enabled if it is not. The interaction of the **check_stack** pragma with the /Gs option is summarized in Table 4.3.

Table 4.3 Using the check_stack Pragma

Syntax	Compiled with /Gs Option?	Action
#pragma check_stack()	Yes	Turns off stack checking for routines that follow
#pragma check_stack()	No	Turns on stack checking for routines that follow
#pragma check_stack(on)	Yes or no	Turns on stack checking for routines that follow
#pragma check_stack(off)	Yes or no	Turns off stack checking for routines that follow

NOTE *For versions of Microsoft QuickC prior to 2.0, the* ***check_stack*** *pragma had a different format:* ***check_stack(+)*** *to enable stack checking and* ***check_stack(–)*** *to disable stack checking. Although the Microsoft QuickC Compiler still accepts this format, its use is discouraged because it may not be supported in future versions.*

Example

```
QCL /Gs FILE.C
```

This example optimizes the file `FILE.C` by removing stack probes with the /Gs option. If you want stack checking for only a few functions in `FILE.C`, you can use the **check_stack** pragma before and after the definitions of functions you want to check.

4.3.17 /Gt (Set Data Threshold)

Option

/Gt[[*number*]]

The /Gt option causes all data items whose size is greater than or equal to *number* bytes to be allocated in a new data segment.

When *number* is specified, it must follow the /Gt option immediately with no intervening spaces. When /Gt is specified without a number, the default threshold value is 256. When the /Gt option is omitted, the default threshold value is 32,767.

The option is particularly useful with programs that have more than 64K of initialized static and global data in small data items.

By default, the compiler allocates all static and global data items within the default data segment in the small and medium memory models. In compact-, large-, and huge-model programs, only *initialized* static and global data items are assigned to the default data segment.

NOTE *You can use the /Gt option only if you are creating a compact-, large-, or huge-model program because small- and medium-model programs have only one data segment.*

4.3.18 /HELP (List the Compiler Options)

Option

/HELP
/help

This option displays a list of the most commonly used compiler options. QCL processes all information on the line containing the /help option and displays the command list.

Unlike all the other QCL options, /HELP is not case sensitive. Any combination of uppercase and lowercase letters is acceptable. For example, /hELp is a valid form of this option. The option has no abbreviation.

4.3.19 /I (Search Directory for Include Files)

Option

/I*directory*

You can add to the list of directories searched for include files by using the /I (for "include") option. This option causes the compiler to search the directory you specify before searching the directories given by the INCLUDE environment variable. That way, you can give a particular file special handling without changing the compiler environment you normally use.

The space between /I and *directory* is optional. To search more than one directory, you can give additional /I options on the QCL command line. The directories are searched in order of their appearance on the command line.

The directories are searched only until the specified include file is found. If the file is not found in the given directories or the standard places, the compiler prints an error message and stops processing. When this occurs, you must restart compilation with a corrected directory specification.

Examples

```
QCL /I \INCLUDE /I\MY\INCLUDE MAIN.C
```

In the example above, QCL looks for the include files requested by `MAIN.C` in the following order: first in the directory `\INCLUDE`, then in the directory `\MY\INCLUDE`, and finally in the directory or directories assigned to the INCLUDE environment variable.

```
QCL /X /I \ALT\INCLUDE MAIN.C
```

In the example above, the compiler looks for include files only in the directory `\ALT\INCLUDE`. First the /X option tells QCL to consider the list of standard places empty; then the /I option specifies one directory to be searched.

4.3.20 /J (Change Default char Type)

In Microsoft QuickC, the **char** type is signed by default, so if a **char** value is widened to **int** type, the result is sign-extended.

You can change this default to **unsigned** with the /J option, causing the **char** type to be zero-extended when widened to **int** type. If a **char** value is explicitly declared **signed**, however, the /J option does not affect it, and the value is sign-extended when widened to **int** type. This option is useful when working with character data that eventually will be translated into a language other than English.

When you specify /J, the compiler automatically defines the identifier **_CHAR_UNSIGNED**, which is used with **#ifndef** in the LIMITS.H include file to define the range of the default **char** type.

4.3.21 /Lc, Lr (Compile for Real Mode), /Lp (Compile for Protected Mode)

If you have installed the Microsoft C Optimizing Compiler, Version 5.1 or later, and have created protected-mode libraries, you can use the /Lp option to compile programs for the OS/2 protected-mode environment.

If you compile with /Lp, you must make sure the linker uses the appropriate protected-mode library. You can use either of the following methods:

- Rename the protected-mode library so that it has the same name as the appropriate standard combined library. For example, if you use the small memory model and emulator math package (the defaults) for compilation, the linker searches for a library named SLIBCE.LIB. You could change the name of the protected-mode library SLIBCEP.LIB to SLIBCE.LIB, and the linker would link with the protected-mode library automatically.
- Give the /NOD option to the linker so that it does not look for the standard combined library and specify the protected-mode library explicitly. Using the same example, you would compile and link using the following command line:

  ```
  QCL PROTMOD.C /link /NOD SLIBCEP.LIB
  ```

  This command line tells the linker not to link with SLIBCE.LIB and to link with SLIBCEP.LIB instead.

The /Lc and /Lr options are synonymous. Both options compile the program for OS/2 real mode or for the DOS environment. As with the /Lp option, if you compile with /Lc or /Lr, you must make sure the linker uses the appropriate real-mode library; either use /NOD to tell the linker not to search for the default library, or rename the appropriate real-mode library so that it has the default name.

4.3.22 /Li (Link Incrementally)

The /Li option specifies incremental linking. When you link incrementally, the linker by default pads all **near** functions to a 40-byte boundary. Note that the incremental linker is automatically invoked whenever you use the /Gi option for incremental compilation.

4.3.23 /NT (Name the Text Segment)

Option

/NT *textsegment*

The /NT option renames a text segment in a QuickC program. The *textsegment* argument can be any combination of letters and digits. The space between /NT and *textsegment* is optional.

A "segment" is a contiguous block of binary information (code or data) produced by the Microsoft QuickC Compiler. Every module (that is, every object file produced by the compiler) has at least two segments: a text segment containing the program instructions and a data segment containing the program data. The program's memory model determines how many text segments and how many data segments the program has (see Sections B.2.1-B.2.5).

Every segment in every module has a name. The linker uses this name to define the order in which the segments of the program appear in memory when loaded for execution. (Note that the segments in the group named **DGROUP** are an exception.)

The QuickC compiler normally creates text- and data-segment names. In small- and compact-memory models, which have a single text segment, the text segment is named _TEXT. In medium, large, and huge models, which have multiple text segments, the code for each module is placed in a separate segment named *module*_TEXT where *module* is the base name of the module. In general, you should not use the /NT option with the small and compact memory models. Doing so may cause fixupp-overflow errors at link time.

The /NT option overrides the default text-segment name used by the QuickC compiler (thus overriding the default loading order). This option gives the text segment a new name in each module being compiled.

4.3.24 /O Options (Optimize Program)

Option

/O *string*

The /O options give you control over the optimizing procedures that the compiler performs. The *string* consists of one or more of the letters "d," "l," "t," and "x." The list below shows how each of these affects optimization:

Letter	Optimizing Procedure
/Od	Turns off all optimization
/Ol	Enables loop optimization
/O, /Ot	Favors execution speed during optimization (the default)
/Ox	Maximizes optimization

The letters can appear in any order. More than one /O option can be given; the compiler uses the last one on the command line if any conflict arises. Each option applies to all source files that follow on the command line.

4.3.24.1 /Od (Turn Off Optimization)

The /Od (for "debug") option tells the compiler to turn off all optimizations in the program. This option speeds compilation because the compiler does not take time to perform optimizations.

The /Od option is recommended when you compile with the /Zi option (described in Section 4.3.31) to include debugging information. This is because the /Od option does not reorganize code, thus making it easier to debug.

4.3.24.2 /Ol (Optimize Loops)

The /Ol option tells the compiler to perform loop optimizations, which store frequently used loop variables in registers. The /Ox option implicitly turns on the /Ol option.

4.3.24.3 /O and /Ot (Minimize Execution Time)

When you do not use any of the /O options, the QCL command automatically favors program execution speed in the optimization. The /O and /Ot options have the same effect as this default.

Wherever the compiler has a choice between producing smaller (but perhaps slower) and larger (but perhaps faster) code, the compiler generates faster code. For example, when the /Ot option is in effect, the compiler generates intrinsic functions to perform shift operations on long operands.

4.3.24.4 /Ox (Use Maximum Optimization)

The /Ox option is a shorthand way to combine optimizing options to produce the fastest possible program. Its effect is the same as using the following options on the same command line:

```
/Olt /Gs
```

That is, the /Ox option performs loop optimizations, favors execution time over code size, and removes stack probes.

Example

```
QCL /Ol FILE.C
```

This command tells the compiler to perform loop optimizations when it compiles `FILE.C`. The compiler favors program speed over program size because the /Ot option is also specified by default.

4.3.25 /P (Create Preprocessor-Output File)

The /P writes preprocessor output to a file with the same base name as the source file but with the .I extension. The preprocessed listing file is identical to the original source file except that all preprocessor directives are carried out and macro expansions are performed. The /P option is normally used with the /C option (discussed in Section 4.3.3), which preserves comments in the preprocessed output.

The /P option suppresses compilation; no object file or listing is produced, even if you specify the /Fo or /Fm option on the QCL command line.

Example

```
QCL /P MAIN.C
```

The example above creates the preprocessed file `MAIN.I` from the source file `MAIN.C`.

4.3.26 /Tc (Specify C Source File)

Option

/Tc *filename*

The /Tc option tells the QCL command that the given file is a C source file. The space between /Tc and *filename* is optional.

If this option does not appear, QCL assumes that files with the extension .C are C source files, files with the extension .LIB are libraries, and files with any other extension or with no extension are object files. If you use the /Tc option, QCL treats the given file as a C source file regardless of its extension, if any.

If you need to specify more than one source file with an extension other than .C, you must specify each source file in a separate /Tc option.

Example

```
QCL MAIN.C /Tc TEST.PRG /Tc COLLATE.PRG PRINT.PRG
```

In the example above, the QCL command compiles the three source files `MAIN.C`, `TEST.PRG`, and `COLLATE.PRG`. Because the file `PRINT.PRG` is given without a /Tc option, QCL treats it as an object file. Therefore, after compiling the three source files, QCL links the object files `MAIN.OBJ`, `TEST.OBJ`, `COLLATE.OBJ`, and `PRINT.PRG`.

4.3.27 /U, /u (Remove Predefined Names)

Options

/Uname
/u

The /U (for "undefine") option turns off the definition of one of the names that the QuickC compiler predefines. The /u option turns off the definitions of all predefined names except for the name of the memory model. These options do not apply to user-defined names.

These names are useful in writing portable programs. For instance, they can be used with compiler directives to conditionally compile parts of a program, depending on the processor and operating system being used. The predefined identifiers and their meanings are listed in Table 4.4.

Table 4.4 Predefined Names

Syntax	Purpose	When Defined
_QC	Identifies compiler as Microsoft QuickC.	Always
MSDOS	Identifies target operating system as MS-DOS.	Always
M_I8086	Identifies target machine as an 8086.	When the /G0 option is given and by default
M_I286	Identifies target machine as an 80286.	When the /G2 option is given
M_I86	Identifies target machine as a member of the Intel® family.	Always
M_I86mM	Identifies memory model, where *m* is either S (small model), C (compact model), M (medium model), L (large model), or H (huge model). If huge model is used, both M_I86LM and M_I86HM are defined.	Always
NO_EXT_KEYS	Indicates that Microsoft-specific language extensions and extended keywords are disabled.	When the /Za option is given
_CHAR_UNSIGNED	Indicates that the **char** type is unsigned by default.	When the /J option is given

One or more spaces may separate /U and *name*. You may specify more than one /U option on the same command line.

The /u option turns off the definitions of all predefined names except **M_I86*m*M**, which identifies the memory model. You can use the /U option to remove the definition of **M_I86*m*M**. If you do, however, you must explicitly define the **NULL** constant in your program because the definition of **NULL** in the **STDIO.H** and **STDDEF.H** files depends on the memory model in use.

Limits on command-line definitions

The /U and /u options are useful if you need to give more than the maximum number of definitions (15 if the /Za or /J option is used, 14 if both options are given, or 16 otherwise) on the command line or if you have other uses for the predefined names. For each predefined name you remove, you can substitute a definition of your own on the command line. When the definitions of all six predefined names are removed, you can specify up to 23 command-line definitions. Because MS-DOS limits the number of characters you can type on a command line, however, the number of definitions you can specify in practice is probably fewer than 23.

Example

```
QCL /UMSDOS /UM_I86 WORK.C
```

This example removes the definitions of two predefined names. Note that the /U option must be given twice to do this.

4.3.28 /W, /w (Set Warning Level)

Options

/W{0|1|2|3}
/w

You can suppress certain classes of warning messages produced by the compiler by using the /w, /W0, /W1, /W2, or /W3 option. Compiler warning messages are any messages beginning with `C4`; see Appendix D, "Error-Message Reference," for a complete list of these messages.

Warnings indicate potential problems (rather than actual errors) with statements that may not be compiled as you intend.

The /W options affect only source files given on the command line; they do not apply to object files.

/W0, /w

The /W0 option turns off all warning messages. This option is useful when you compile programs that deliberately include questionable statements. The /W0 option applies to the remainder of the command line or until the next occurrence of a /W option on the command line. The /w option has the same effect as the /W0 option.

/W1 The /W1 option (the default) causes the compiler to display most warning messages.

/W2 The /W2 option causes the compiler to display an intermediate level of warning messages. Level-2 warnings may or may not indicate serious problems. They include warnings such as the following:

- Use of functions with no declared return type
- Failure to put **return** statements in functions with non-**void** return types
- Data conversions that would cause loss of data or precision

/W3 The /W3 option displays the highest level of warning messages, including warnings about the use of non-ANSI features and extended keywords and about function calls that precede their function prototypes in the program.

Note that the descriptions of the warning messages in Appendix D indicate the warning level that must be set (that is, the number for the appropriate /W option) for the message to appear.

Example

```
QCL /W3 CRUNCH.C PRINT.C
```

This example enables all possible warning messages when the source files `CRUNCH.C` and `PRINT.C` are compiled.

4.3.29 /X (Ignore Standard Include Directory)

You can prevent the QuickC compiler from searching the standard places for include files by using the /X (for "exclude") option. When QCL sees the /X option, it does not search the current directory or any directories specified in the INCLUDE environment variable.

This option is often used with the /I option to define the location of include files that have the same names as include files found in other directories but that contain different definitions. See Section 4.3.19 for an example of /X used with /I.

4.3.30 /Ze, /Za (Enable or Disable Language Extensions)

The Microsoft QuickC Compiler supports the latest draft of the ANSI C standard. In addition, it offers a number of features beyond those specified in the ANSI C standard. These features are enabled when the /Ze (default) option is in effect and disabled when the /Za option is in effect. They include the following:

- The **cdecl**, **far**, **fortran**, **huge**, **near**, and **pascal** keywords
- Use of casts to produce lvalues, as in the following example:

```
int *p;
((long *)p)++;
```

The preceding example could be rewritten to conform with the ANSI C standard as shown below:

```
p = (int *)((char *)p + 1);
```

- Redefinitions of **extern** items as **static**, as in the example below:

```
extern int foo();
static int foo()
{}
```

- Use of trailing commas (,) rather than an ellipsis (,...) in function declarations to indicate variable-length argument lists, as in the following example:

```
int printf(char *,);
```

- Benign **typedef** redefinitions within the same scope, as in the following example:

```
typedef int INT;
typedef int INT;
```

- Use of mixed character and string constants in an initializer, as in the following example:

```
char arr[5] = {'a', 'b', "cde"};
```

- Use of bit fields with base types other than **unsigned int** or **signed int**
- The use of single-line comments, which are introduced with two slash characters, as in the following example:

```
// This is a single-line comment.
```

Use the /Za option if you plan to port your program to other environments. The /Za option tells the compiler to treat extended keywords as simple identifiers and disable the other extensions listed above.

When you specify /Za, the compiler automatically defines the identifier **NO_EXT_KEYS**. In the include files provided with the Microsoft QuickC Compiler run-time library, this identifier is used with **#ifndef** to control use of the **cdecl** keyword on library function prototypes. For an example of this conditional compilation, see the file STDIO.H.

4.3.31 /Zi, /Zd (Compile For Debugging)

The /Zi option produces an object file containing full symbolic-debugging information for use with the QuickC integrated debugger and the CodeView symbolic debugger. This object file includes full symbol-table information and line numbers.

The /Zd option produces an object file containing line-number records corresponding to the line numbers of the source file. Use /Zd if you plan to debug with the SYMDEB debugger. This option is also useful in cases where you want to reduce the size of an executable file that you will be debugging with the CodeView debugger, and you do not need to use the expression evaluator during debugging.

Example

```
QCL /c /Zi TEST.C
```

This command produces an object file named `TEST.OBJ` that contains line numbers corresponding to the lines in `TEST.C`.

4.3.32 /Zl (Remove Default-Library Name from Object File)

Ordinarily QCL places the name of the default library, SLIBCE.LIB, in the object file so that the linker can automatically find the correct library to link with the object file.

The /Zl option tells the compiler not to place the default-library name in the object file. As a result, the object file is slightly smaller.

The /Zl option is useful when you are using the LIB utility (described in Chapters 2 and 6) to build a library. You can use /Zl to compile the object files you plan to put in your library, thereby omitting the default-library names from your object modules. Although the /Zl option saves only a small amount of space for a single object file, the total amount of space saved is significant in a library containing many object modules.

Example

```
QCL ONE.C /Zl TWO.C
```

The example above creates the following two object files:

- An object file named `ONE.OBJ` that contains the name of the C library SLIBCE.LIB
- An object file named `TWO.OBJ` that contains no default-library information

When ONE.OBJ and TWO.OBJ are linked, the default-library information in ONE.OBJ causes the default library to be searched for any unresolved references in either ONE.OBJ or TWO.OBJ.

4.3.33 /Zp (Pack Structure Members)

Option

/Zp[[{1|2|4}]]

When storage is allocated for structures, structure members are ordinarily stored as follows:

- Items of type **char** or **unsigned char**, or arrays containing items of these types, are byte aligned.
- Structures are word aligned; structures of odd size are padded to an even number of bytes.
- All other types of structure members are word aligned.

To conserve space or to conform to existing data structures, you may want to store structures more or less compactly. The /Zp option and the **pack** pragma control how structure data are "packed" into memory.

Use the /Zp option when you want to specify the same packing for all structures in a module. When you give the /Zp*n* option, where *n* is 1, 2, or 4, each structure member after the first is stored on *n*-byte boundaries depending on the option you choose. If you use the /Zp option without an argument, structure members are packed on two-byte boundaries.

On some processors, the /Zp option may result in slower program execution because of the time required to unpack structure members when they are accessed. For example, on an 8086 processor this option can reduce efficiency if members with **int** or **long** type are packed in such a way that they begin on odd-byte boundaries.

Use the **pack** pragma in your source code to pack particular structures on different boundaries from the packing specified on the command line. Give the **pack**(*n*) pragma, where *n* is 1, 2, or 4, before structures that you want to pack differently. To reinstate the packing given on the command line, give the **pack**() pragma with no arguments.

Table 4.5 shows the interaction of the /Zp option with the **pack** pragma.

Table 4.5 Using the pack Pragma

Syntax	Compiled with /Zp Option?	Action
#pragma pack()	Yes	Reverts to packing specified on the command line for structures that follow
#pragma pack()	No	Reverts to default packing for structures that follow
#pragma pack(*n*)	Yes or no	Packs the following structures to the given byte boundary until changed or disabled

Example

```
QCL /Zp PROG.C
```

This command causes all structures in the program `PROG.C` to be stored without extra space for alignment of members on **int** boundaries.

4.3.34 /Zr (Check Pointers)

The /Zr option checks for null or out-of-range pointers in your program. A run-time error occurs if you try to run a program with such pointers.

#pragma check_pointer

If you compile with the /Zr option, you can use the **check_pointer** pragma within your source file to turn checking on or off only for selected pointers leaving the default (see below) for the remaining pointers in the program. When you want to turn on pointer checking, put the following line before the declaration of the pointer you want to check:

#pragma check_pointer (on)

This line turns on pointer checking for all pointers that follow it in the source file, not just the pointers on the following line. To turn off pointer checking, insert the following line:

#pragma check_pointer (off)

If no argument is given for the **check_pointer** pragma, pointer checking reverts to the behavior specified on the command line: turned on if the /Zr option is given or turned off otherwise.

Example

```
QCL /Zr prog.c
```

This command causes QCL to check for null or out-of-range pointers in the file `prog.c`. All pointers in the file are checked except those to which a **check_pointer(off)** pragma applies.

4.3.35 /Zs (Check Syntax Only)

The /Zs option tells the compiler only to check the syntax of the source files that follow the option on the command line. This option provides a quick way to find and correct syntax errors before you try to compile and link a source file.

When you give the /Zs option, the compiler does not generate code or produce object files, object listings, or executable files. The compiler, however, does display error messages if the source file has syntax errors.

Example

```
QCL /Zs TEST*.C
```

This command causes the compiler to perform a syntax check on all source files in the current working directory that begin with `TEST` and end with the .C extension. The compiler displays messages for any errors found.

4.3.36 Giving Options with the CL Environment Variable

Use the CL environment variable to specify files and options without giving them on the command line. This variable has the following format:

SET CL=[[[[*option*]] ... [[*file*]] ...]] [[/link [[*link-libinfo*]]]]

This variable is useful if you usually give a large number of files and options when you compile. Ordinarily, the command line is limited to 128 characters. The files and options that you define with the CL environment variable, however, do not count toward this limit. Therefore, you can define the files and options you use most often with the CL variable and then give only the files and options you need for specific purposes on the command line.

The information defined in the CL variable is treated as though it appeared *before* the information given on the CL command line, as illustrated in Figure 4.3.

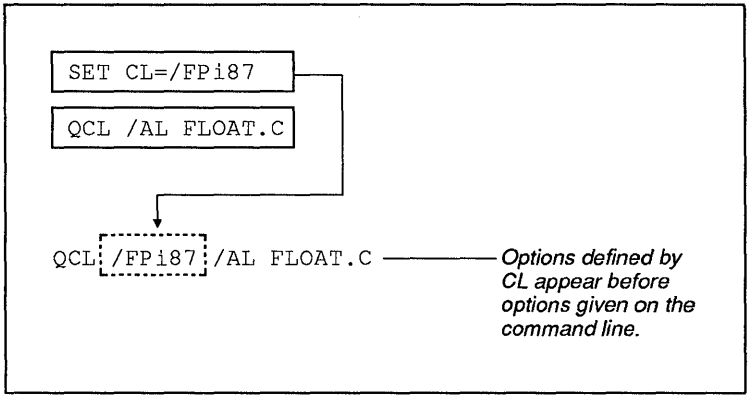


Figure 4.3 Effect of the CL Environment Variable

Note that if you have given an option in the CL environment variable, you generally cannot turn off or change the option from the command line. You must reset the CL environment variable and omit the file or option that you do not want to use.

Also note that you cannot use CL to set options that use an equal sign (for example, the /D*identifier*= *string* option), and you cannot use wild-card characters in file names to specify multiple files to CL.

Examples

```
SET CL=/Zp /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ
QCL INPUT.C
```

In the example above, the CL environment variable tells the QCL command to use the /Zp, /Ox, and /I options during compilation and then to link with the object file `\LIB\BINMODE.OBJ`. With CL defined as shown, the QCL command above has the same effect as the command line

```
QCL /Zp /Ox /I\INCLUDE\MYINCLS \LIB\BINMODE.OBJ INPUT.C
```

That is, both would specify structure packing on two-byte boundaries; perform maximum optimizations; search for include files in the `\INCLUDE\MYINCLS` directory; and would suppress translation of carriage-return–line-feed character combinations for the source file `INPUT.C`.

```
SET CL=FILE1.C FILE2.C
QCL FILE3.OBJ
```

In the example above, the CL environment variable tells the QCL command to compile and link the source files `FILE1.C` and `FILE2.C`. The QCL command

```
QCL FILE1.C FILE2.C FILE3.OBJ
```

would then have the same effect as the previous command line.

```
SET CL=/Za
QCL FILE1.C /Ze FILE2.C
```

The example above illustrates how to turn off the effects of a QCL option defined in the environment. In this example, the CL environment variable is set to the /Za option, which tells the compiler not to recognize Microsoft extensions to the C language. This option causes Microsoft-specific keywords to be treated as ordinary identifiers rather than reserved words. The QCL command specifies the inverse option, /Ze, which tells the compiler to treat language extensions as reserved words. Since the effect is the same as compiling with the command line

```
QCL /Za FILE1.C /Ze FILE2.C
```

`FILE1.C` is compiled with language extensions turned off and `FILE2.C` is compiled with language extensions enabled.

4.4 Controlling Stack and Heap Allocation

The "stack" and the "heap" are two important memory areas that are allocated for QuickC programs. The stack is used for all local data (that is, data that are defined within a function); the heap is used for all dynamically allocated data (that is, data allocated by one of the **alloc** family of functions).

Programs compiled and linked under the Microsoft QuickC Compiler run with a fixed stack size (the default size is 2048 bytes). The stack resides above static data, and the heap uses whatever space is left above the stack. For some programs, however, a fixed-stack model may not be ideal; a model where the stack and heap compete for space is more appropriate.

Linking with the *m*VARSTCK.OBJ object files gives you such a model: when the heap runs out of memory, it tries to use available stack space until it runs into the top of the stack. When the allocated space in the heap is freed, it is once again made available to the stack. Note that the stack cannot grow beyond the last-allocated heap item in the stack or, if there are no heap items in the stack, beyond the size it was given at link time. Furthermore, while the heap can employ unused stack space, the reverse is not true: the stack cannot employ unused heap space.

You can change the model used to allocate heap space by linking your program with one of the *m*VARSTCK.OBJ object files (where *m* is the first letter of the library you choose). These files are the small-, medium-, compact-, and large-model versions of a routine that allows the memory-allocation functions (**malloc**, **calloc**, **_expand**, **_fmalloc**, **_nmalloc**, and **realloc**) to allocate items in unused stack space if they run out of other memory. (If you use the huge memory model, link with the large-model object file **LVARSTCK.OBJ**.)

When you link your program with one of the *m*VARSTCK.OBJ files, do not suppress stack checking with the **#check_stack** pragma, or with the /Gs or /Ox option. Stack overflow can occur more easily in programs that link with the variable-stack object files, possibly causing errors that would be difficult to detect.

Example

```
QCL TEST.C SVARSTCK
```

This command line compiles `TEST.C` and then links the resulting object module with **SVARSTCK.OBJ**, the variable-stack object file for small-model programs.

CHAPTER 5 LINK

This chapter describes in detail the operation of the Microsoft Overlay Linker (LINK) and includes an alphabetical reference to the LINK options.

5.1 Overview

The Microsoft Overlay Linker (LINK) combines object files into a single executable file. It can be used with object files compiled or assembled for 8086/8088, 80286, or 80386 machines. The format of input to the linker is the Microsoft Relocatable Object-Module Format (OMF), which is based on the Intel® 8086 OMF.

The output file from LINK (that is, the executable file) is not bound to specific memory addresses. Thus, the operating system can load and execute this file at any convenient address. LINK can produce executable files containing up to one megabyte of code and data.

5.2 Invoking LINK

Instead of using the QCL command to invoke the linker, you can use the LINK command to invoke LINK directly. You can specify the input required for this command in one of three ways:

1. By placing it on the command line.
2. By responding to prompts.
3. By specifying a file containing responses to prompts. This type of file is known as a "response file."

Regardless of how you invoke LINK, you may press CTRL+C at any time to terminate a LINK operation and exit to DOS.

You can use any combination of uppercase and lowercase letters for the file names you specify on the LINK command line or give in response to the LINK command prompts.

If you specify file names without extensions, LINK uses the following default file-name extensions:

File Type	Default Extension
Object	.OBJ
Executable	.EXE
Map (or "Listing")	.MAP
Library	.LIB

You can override the default extension for a particular command-line field or prompt by specifying a different extension. To enter a file name that has *no* extension, type the name followed by a period.

5.2.1 Command Line

Use the following form of the LINK command to specify input on the command line:

LINK [[*linkoptions*]] *objfiles* [[, [[*exefile*]] [[, [[*mapfile*]] [[, [[*libraries*]]]]]] [[;]]

A comma must separate each command-line field from the next. You may omit the text from any field (except the required *objfiles*), but you must include the comma. A semicolon may end the command line after any field causing LINK to use defaults for the remaining fields. For details of LINK defaults, see Section 5.2.1.6.

The command-line fields are explained below.

5.2.1.1 LINK Options

You may specify command-line options after any field, but before the comma that terminates the field. You do not have to give any options when you run the linker. Linker options are described in Section 5.4.

5.2.1.2 Object Files

The *objfiles* field allows you to specify the names of the object files you are linking. At least one object-file name is required. A space or plus sign (+) must separate each pair of object-file names. LINK automatically supplies the .OBJ extension when you give a file name without an extension. If your object file has a different extension or if it appears in another directory or on another disk, you must give the full name—including the extension and path name—for the file to be found. If LINK cannot find a given object file and the drive associated with the object file is a removable-disk (floppy) drive, then LINK displays a message and waits for you to change disks.

You may also specify one or more libraries in the *objfiles* field. To enter a library in this field, make sure that you include the .LIB extension; otherwise, LINK assumes the .OBJ extension. Libraries entered in this field are called "load libraries" as opposed to "regular libraries." LINK automatically links in every object module in a load library; it does not search for unresolved external references first. The effect of entering a load library is exactly the same as if you had entered the names of all the library's object modules in the *objfiles* field. This feature is useful if you are developing software using many modules and wish to avoid typing the name of each module on the LINK command line.

5.2.1.3 Executable File

The *exefile* field allows you to specify the name of the executable file. If the file name you give does not have an extension, LINK automatically adds .EXE as the extension. You can give any file name you like; however, if you are specifying an extension, you should always use .EXE because DOS expects executable files to have either this extension or the .COM extension.

5.2.1.4 Map File

The *mapfile* field allows you to specify the name of the map file if you are creating one. To include public symbols and their addresses in the map file, specify the /MAP option on the LINK command line.

If you specify a map-file name without an extension, LINK automatically adds an extension of .MAP. LINK creates the map file in the current working directory unless you specify a path name for the map file.

5.2.1.5 Libraries

The *libraries* field allows you to specify the name of one or more libraries that you want linked with the object file(s). When LINK finds the name of a library in this field, it treats the library as a "regular library" and links in only those object modules needed to resolve external references.

Each time you compile a source file for a high-level language, the compiler places the name of one or more libraries in the object file that it creates; the linker automatically searches for a library with this name (see Section 5.2.4). Because of this, you do not need to give library names on the LINK command line unless you want to search libraries other than the default libraries or search for libraries in different locations.

When you link your program with a library, the linker pulls into your executable file any library modules that your program references. If the library modules have external references to other library modules, your program is linked with those other library modules as well.

5.2.1.6 Choosing Defaults

If you include a comma (to indicate where a field would be) but do not put a file name before the comma, then LINK selects the default for that field. However, if you use a comma to include the *mapfile* field (but do not include a name), then LINK creates a map file. This file has the same base name as the executable file. Use NUL for the map-file name if you do not want to produce a map file.

You can also select default responses by using a semicolon (;). The semicolon tells LINK to use the defaults for all remaining fields. Anything after the semicolon is ignored. If you do not give all file names on the command line or if you do not end the command line with a semicolon, the linker prompts you for the files you omitted. See Section 5.2.2 for a description of these prompts.

The list below summarizes the linker's defaults for each field:

Field	Default
exefile	Creates a file with the base name of the first object file and a .EXE extension.C
mapfile	Does not create a map file unless you include the *mapfile* field. The field may be empty, as in the following command line: `LINK myfile yourfile, ourfile, ;` If you include the field but not a file name, LINK creates a map file with the base name of the executable file and the .MAP extension. Thus the example creates a map file named `ourfile.map`.
libraries	Searches only the default libraries specified in the object files.

If you do not specify a drive or directory for a file, the linker assumes that the file is on the current drive and directory. If you want the linker to create files in a location other than the current drive and directory, you must specify the new drive and directory for each such file on the command line.

Examples

```
LINK SPELL+TEXT+DICT+THES, ,SPELLIST, XLIB.LIB
```

The command line above causes LINK to load and link the object modules SPELL.OBJ, TEXT.OBJ, DICT.OBJ, and THES.OBJ, and to search for unresolved references in the library file XLIB.LIB and the default libraries. By default, the executable file produced by LINK is named SPELL.EXE. LINK also produces a map file, SPELLIST.MAP. Note that no semicolon is required because a library is specified.

```
LINK SPELL,,;
```

The LINK command line shown above produces a map file named SPELL.MAP because a comma appears as a placeholder for the *mapfile* specification on the command line.

```
LINK SPELL,;
```

```
LINK SPELL;
```

These two command lines do not produce a map file because commas do not appear as placeholders for the *mapfile* specification.

```
LINK MAIN+GETDATA+PRINTIT, , MAIN;
```

The command above causes LINK to link the three files MAIN.OBJ, GETDATA.OBJ, and PRINTIT.OBJ into an executable file. A map file named MAIN.MAP is also produced.

5.2.2 Prompts

If you want the linker to prompt you for input, start LINK by entering

```
LINK
```

at the DOS prompt. LINK also displays prompts if you type an incomplete command line that does not end with a semicolon or if a response file (see Section 5.2.3) is missing any required responses.

LINK prompts you for the input it needs by displaying the following lines, one at a time. The items in square brackets are the defaults LINK applies if you press ENTER in response to the prompt. (You must supply at least one object-file name for the "Object Modules" prompt.) LINK waits for you to respond to each prompt before it displays the next one.

```
Object Modules [.OBJ]:
Run File [basename.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
```

Note that the default for the `Run File` prompt is the base name of the first object file with the .EXE extension.

The responses you give to the LINK command prompts correspond to the fields on the LINK command line as follows:

Prompt	Command-Line Field
"Object Modules"	*objfiles*
"Run File"	*exefile*
"List File"	*mapfile*
"Libraries"	*libraries*

Continuation character (+)

If you type a plus sign (+) as the last character on a response line, the same prompt appears on the next line, and you can continue typing responses. The plus sign must appear at the end of a complete file or library name, path name, or drive name.

Choosing defaults: current prompt

To select the default response to the current prompt, press ENTER without giving a file name. The next prompt appears.

Choosing defaults: all remaining prompts

To select default responses to the current prompt and all remaining prompts, type a semicolon (;) and press ENTER. After you type a semicolon, you cannot respond to any of the remaining prompts for that link session. This option saves time when you want the default responses. Note, however, that you cannot enter only a semicolon in response to the "Object Modules" prompt because there is no default response for that prompt; the linker requires the name of at least one object file.

Defaults for prompts

The following list shows the defaults for the other linker prompts:

Prompt	Default
"Run File"	The name of the first object file submitted for the "Object Modules" prompt with the .EXE extension replacing the .OBJ extension
"List File"	The special file name NUL.MAP, which tells LINK *not* to create a map file
"Libraries"	The default libraries encoded in the object files (see Section 5.2.4)

5.2.3 Response File

A response file contains responses to the LINK prompts. The responses must be in the same order as the LINK prompts discussed in the previous section. Each new response must appear on a new line or must begin with a comma; however, you can extend long responses across more than one line by typing a plus sign (+) as the last character of each incomplete line. You may give options at the end of any response or place them on one or more separate lines.

LINK treats the input from the response file just as if you had entered it in response to prompts or on a command line. It treats any new-line character in the response file as if you had pressed ENTER in response to a prompt or included a comma in a command line. (This mechanism is illustrated in Figure 5.1.) For compatibility with OS/2 versions of the linker, it is recommended that all linker response files end with a semicolon after the last line.

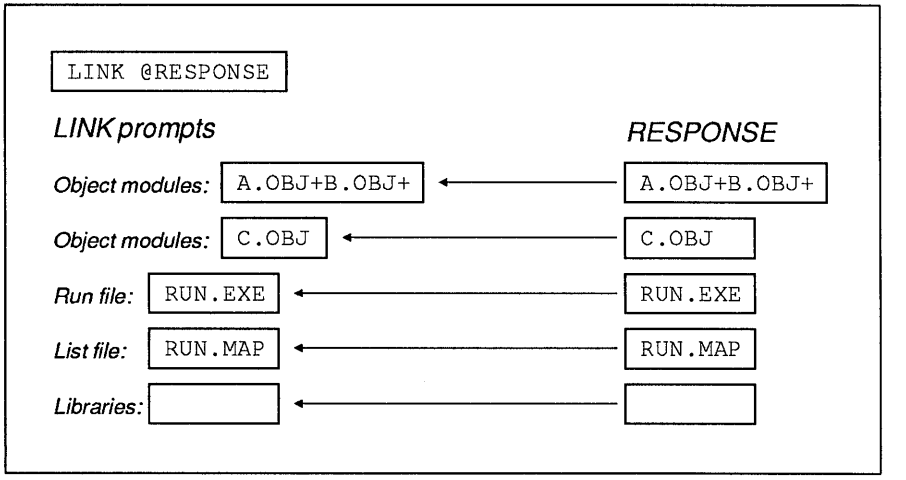


Figure 5.1 LINK Response File

To use the linker with a response file, create the response file, then type the following command:

LINK @*responsefile*

Here *responsefile* specifies the name or path name of the response file for the linker. You can also enter the name of a response file, preceded by an "at" sign (@), after any LINK command prompt or at any position in the LINK command line; in this case, the response file completes the remaining input.

Options and command characters

You can use options and command characters in the response file in the same way as you would use them in responses you type at the keyboard. For example, if you type a semicolon on the line of the response file corresponding to the "Run File" prompt, LINK uses the default responses for the executable file and for the remaining prompts.

Prompts

When you enter the LINK command with a response file, each LINK prompt is displayed on your screen with the corresponding response from your response file. If the response file does not include a line with a file name, semicolon, or carriage return for each prompt, LINK displays the appropriate prompt and waits for you to enter a response. When you type an acceptable response, LINK continues.

Example

Assume that the following response file is named `SPELL.LNK`:

```
SPELL+TEXT+DICT+THES
/PAUSE /MAP
SPELLIST
XLIB.LIB;
```

You can type the following command to run LINK and tell it to use the responses in `SPELL.LNK`:

```
LINK @SPELL.LNK
```

The response file tells LINK to load the four object files `SPELL`, `TEXT`, `DICT`, and `THES`. LINK produces an executable file named `SPELL.EXE` and a map file named `SPELLIST.MAP`. The /PAUSE option tells LINK to pause before it produces the executable file so that you can swap disks, if necessary. The /MAP option tells LINK to include public symbols and addresses in the map file. LINK also links any needed routines from the library file `XLIB.LIB`. The semicolon is included after the library name for compatibility with the OS/2 version of the linker.

5.2.4 How LINK Searches for Libraries

LINK searches for libraries that are specified in either of the following ways:

- In the *libraries* field on the command line or in response to the "Libraries" prompt.
- By an object module. The QuickC compiler writes the name of a default combined library in each object module it creates.

NOTE *The material in this section does not apply to libraries that LINK finds in the objfiles field, either on the command line or in response to the "Object Modules" prompt. Those libraries are treated simply as a series of object files, and LINK does not conduct extensive searches in such cases.*

Library name with path specification

If the library name includes a path specification, LINK searches only that directory for the library. Libraries specified by object modules (that is, default libraries) normally do not include a path specification.

Library name without path specification

If the library name does not include a path specification, LINK searches the following locations, in the order shown, to find the library file:

1. The current directory
2. Any path specifications or drive names that you give on the command line or type in response to the "Libraries" prompt in the order in which they appear
3. The locations given by the LIB environment variable

Because object files created by the Microsoft QuickC Compiler contain the names of all the standard libraries you need, you are not required to specify a library on the LINK command line or in response to the LINK "Libraries" prompt unless you want to do one of the following:

- Add the names of additional libraries to be searched
- Search for libraries in different locations
- Override the use of one or more default libraries

For example, if you have developed your own customized libraries, you might want to include one or more of them as additional libraries at linking time.

5.2.4.1 Searching Additional Libraries

You can tell LINK to search additional libraries by specifying one or more library files on the command line or in response to the "Libraries" prompt. LINK searches these libraries in the order you specify *before* it searches default libraries.

LINK automatically supplies the .LIB extension if you omit it from a library-file name. If you want to link a library file that has a different extension, be sure to specify the extension.

5.2.4.2 Searching Different Locations for Libraries

You can tell LINK to search additional locations for libraries by giving a drive name or path specification in the *libraries* field on the command line or in response to the "Libraries" prompt. You can specify up to 32 additional paths. If you give more than 32 paths, LINK ignores the additional paths without displaying an error message.

5.2.4.3 Overriding Libraries Named in Object Files

If you do not want to link with the library whose name is included in the object file, you can give the name of a different library instead. You might need to specify a different library name in the following cases:

- You assigned a "custom" name to a standard library when you set up your libraries.
- You want to link with a library that supports a different math package than the math package you gave on the compiler command line (or the default).

If you specify a new library name on the LINK command line, the linker searches the new library to resolve external references before it searches the library specified in the object file.

If you want the linker to ignore the library whose name is included in the object file, you must use the /NOD option. This option tells LINK to ignore the default-library information that is encoded in the object files created by high-level-language compilers. Use this option with caution; for more information, see Section 5.4.14, "Ignoring Default Libraries."

Example

```
LINK

Microsoft® QuickC Linker  Version 4.00.
Copyright© Microsoft Corp 1988.  All rights reserved.
```

```
Object Modules [.OBJ]: SPELL TEXT DICT THES
Run File [SPELL.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]: C:\TESTLIB\ NEWLIBV3
```

This example links four object modules to create an executable file named `SPELL.EXE`. LINK searches `NEWLIBV3.LIB` before searching the default libraries to resolve references. To locate `NEWLIBV3.LIB` and the default libraries, the linker searches the current working directory, then the `C:\TESTLIB\` directory, and finally the locations given by the LIB environment variable.

5.3 LINK Memory Requirements

LINK uses available memory for the link session. If the files to be linked create an output file that exceeds available memory, LINK creates a temporary disk file to serve as memory. This temporary file is handled in one of the following ways, depending on the DOS version:

- For the purpose of creating a temporary file, the linker uses the directory specified by the TMP environment variable. If the TMP variable is set to `C:\TEMPDIR`, for example, then LINK puts the temporary file in `C:\TEMPDIR`.

  If there is no TMP environment variable or if the directory specified by TMP does not exist, then LINK puts the temporary file in the current directory.

- If the linker is running on DOS Version 3.0 or later, it uses a DOS system call to create a temporary file with a unique name in the temporary-file directory.

- If the linker is running on a version of DOS prior to 3.0, it creates a temporary file named VM.TMP.

When the linker creates a temporary disk file, you see the message

```
Temporary file tempfile has been created.
Do not change diskette in drive, letter.
```

In the message displayed above, *tempfile* is ".\" followed by either `VM.TMP` or a name generated by DOS, and *letter* is the drive containing the temporary file.

If you are running on a removable-disk system, the "Do not change diskette" message appears. After this message appears, do not remove the disk from the

specified drive until the link session ends. If you remove the disk, the operation of LINK is unpredictable, and you may see the following message:

```
unexpected end-of-file on scratch file
```

If this happens, rerun the link session. The temporary file created by LINK is a working file only. LINK deletes it at the end of the link session.

NOTE *Do not give any of your own files the name VM.TMP. The linker displays an error message if it encounters an existing file with this name.*

5.4 LINK Options

This section explains how to use linker options to specify and control the tasks performed by LINK.

When you use the LINK command line to invoke LINK, you may put options at the end of the line or after individual fields on the line. Options, however, must immediately precede the comma that separates each field from the next.

If you respond to the individual prompts for the LINK command, you may specify linker options at the end of any response. When you use more than one option, you can either group the options at the end of a single response or distribute the options among several responses. Every option must begin with the slash character (/) or a dash (-), even if other options precede it on the same line.

In a response file, options may appear on a line by themselves or after individual response lines.

Abbreviations

Because linker options are named according to their functions, some of their names are quite long. You can abbreviate the options to save space and effort. Be sure that your abbreviation is unique so that the linker can determine which option you want. The minimum legal abbreviation for each option is indicated in the syntax description of the option.

Abbreviations must begin with the first letter of the name and must be continuous through the last letter typed. No gaps or transpositions are allowed. Options may be entered in uppercase or lowercase letters.

Numeric arguments

Some linker options take numeric arguments. A numeric argument can be any of the following:

- A decimal number from 0 to 65,535.
- An octal number from 00 to 0177777. A number is interpreted as octal if it starts with 0. For example, the number `10` is interpreted as a decimal number, but the number `010` is interpreted as an octal number, equivalent to 8 in decimal.

- A hexadecimal number from 0X0 to 0XFFFF. A number is interpreted as hexadecimal if it starts with 0X. For example, `0X10` is a hexadecimal number, equivalent to 16 in decimal.

LINK environment variable

You can use the LINK environment variable to cause certain options to be used each time you link. The linker checks the environment variable for options if the variable exists.

The linker expects to find options listed in the variable exactly as you would type them on the command line. It does not accept any other arguments; for instance, including file names in the environment variable causes the error message `unrecognized option name`.

Each time you link, you can specify other options in addition to those in the LINK environment variable. If you enter the same option both on the command line and in the environment variable, the linker ignores the redundant option. If the options conflict, however, the command-line option overrides the effect of the environment variable option. For example, the command-line option `/SE:512` cancels the effect of the environment-variable option `/SE:256`.

NOTE *The only way to prevent an option in the environment variable from being used is to reset the environment variable itself.*

Example

```
>SET LINK=/NOI /SE:256 /CO

>LINK TEST;

>LINK /NOD /CO PROG;
```

In the example above, the file `TEST.OBJ` is linked with the options /NOI, /SE:256, and /CO. The file `PROG.OBJ` is then linked with the option /NOD, in addition to /NOI, /SE:256, and /CO. Note that the second /CO option is ignored.

5.4.1 Running in Batch Mode (/BA)

Option

/BA[[TCH]]

By default, the linker prompts you for a new path name whenever it cannot find a library that it has been directed to use. It also prompts you if it cannot find an object file that it expects to find on a removable disk. If you use the /BA option, however, the linker does not prompt you for any libraries or object files that it cannot find. Instead, the linker generates an error or warning message, if appropriate. In addition, when you use /BA, the linker does not display its copyright banner, nor does it echo commands from response files. This option does *not*

prevent the linker from prompting for command-line arguments. You can prevent such prompting only by using a semicolon on the command line or in a response file.

Using this option may result in unresolved external references. It is intended primarily for use with batch or NMAKE files that link many executable files with a single command and to prevent linker operation from halting.

NOTE *In earlier versions of LINK, the /BATCH option was abbreviated to /B.*

5.4.2 Creating a .COM File (/BI)

Option

/BI[[NARY]]

The /BI option is used to generate a .COM file instead of a .EXE file as the output from the linker. The result is the same as if you had linked a .EXE file, then used the EXE2BIN command to convert it to a .COM file. (See *The MS-DOS Programmer's Reference Manual* for more information on .COM files and the EXE2BIN command.)

When you use the /BI option, the linker by default produces an output file with the .COM extension instead of .EXE. If you specify a file name with a different extension, the linker applies the extension you specify. Note that the "Run File" prompt shows the .EXE extension if you have not yet given the /BI option. After you give the option, the linker issues a warning message that the extension of the output file is .COM.

Files with the .COM extension may not perform load-time relocations and therefore may not include far-segment references. The linker issues an error if it detects such references.

NOTE *This option applies only to assembly-language programs.*

5.4.3 Preparing for Debugging (/CO)

Option

/CO[[DEVIEW]]

The /CO option is used to prepare for debugging with the integrated QuickC debugger or the Microsoft CodeView window-oriented debugger. This option tells the linker to prepare a special executable file containing symbolic data and line-number information.

Object files linked with the /CO option must first be compiled with the /Zi option, which is described in Section 4.3.31.

You can run this executable file outside the CodeView debugger; the extra data in the file are ignored. To keep file size to a minimum, however, use the special-format-executable file only for debugging; then you can link a separate version without the /CO option after the program is debugged.

5.4.4 Setting the Maximum Allocation Space (/CP)

Option

/CP[ARMAXALLOC]:*number*

The /CP option sets the maximum number of 16-byte paragraphs needed by the program when it is loaded into memory. The operating system uses this value when allocating space for the program before loading it. The option is useful when you want to execute another program from within your program and you need to reserve space for that other program.

LINK normally requests the operating system to set the maximum number of paragraphs to 65,535. Since this represents more memory than could be available under DOS, the operating system always denies the request and allocates the largest contiguous block of memory it can find. If the /CP option is used, the operating system allocates no more space than the option specifies. This means any additional space in memory is free for other programs.

The *number* can be any integer value in the range 1 – 65,535. If *number* is less than the minimum number of paragraphs needed by the program, LINK ignores your request and sets the maximum value equal to whatever the minimum value happens to be. The minimum number of paragraphs needed by a program is never less than the number of paragraphs of code and data in the program. To free more memory for programs compiled in the medium- and large-memory models, link with /CP:1; this leaves no space for the near heap.

5.4.5 Ordering Segments (/DO)

Option

/DO[SSEG]

The /DO option forces a special ordering on segments. This option is automatically enabled by a special object-module record in Microsoft QuickC libraries. If you are linking to one of these libraries, then you do not need to specify this option.

This option is also enabled by assembly modules that use the Microsoft Macro Assembler directive .DOSSEG.

The /DO option forces segments to be ordered as follows:

1. All segments with a class name ending in CODE
2. All other segments outside DGROUP
3. DGROUP segments, in the following order:
   a. Any segments of class BEGDATA (this class name reserved for Microsoft use)
   b. Any segments not of class BEGDATA, BSS, or STACK
   c. Segments of class BSS
   d. Segments of class STACK

When the /DO option is in effect the linker initializes two special variables as follows:

```
_edata = DGROUP : BSS
_end   = DGROUP : STACK
```

The variables _edata and _end have special meanings for the Microsoft C and FORTRAN compilers, so it is not wise to give these names to your own program variables. Assembly modules can reference these variables but should not change them.

5.4.6 Controlling Data Loading (/DS)

Option

/DS[[ALLOCATE]]

By default, LINK loads all data starting at the low end of the data segment. At run time, the DS (data segment) register is set to the lowest possible address to allow the entire data segment to be used.

Use the /DS option to tell LINK to load all data starting at the high end of the data segment instead of at the low end. In this case, the DS register is set at run time to the lowest data-segment address that contains program data.

The /DS option is typically used with the /HI option (see Section 5.4.10) to take advantage of unused memory within the data segment.

WARNING *This option should be used only with assembly-language programs.*

5.4.7 Packing Executable Files (/E)

Option

/E[[XEPACK]]

The /E option directs LINK to remove sequences of repeated bytes (typically null characters) and to optimize the load-time-relocation table before creating the executable file. (The load-time-relocation table is a table of references, relative to the start of the program. Each reference changes when the executable image is loaded into memory and an actual address for the entry point is assigned.)

Executable files linked with this option may be smaller, and thus load faster, than files linked without this option. Programs with many load-time relocations (about 500 or more) and long streams of repeated characters are usually shorter if packed. The /E option, however, does not always save a significant amount of disk space and sometimes may increase file size. LINK notifies you if the packed file is larger than the unpacked file.

Note that you cannot use the QuickC debugger, the Symbolic Debug Utility (SYMDEB), or the CodeView® window-oriented debugger to debug packed files. The /EXEPACK option strips symbolic information needed by the debuggers from the input file and issues a warning message to notify you.

5.4.8 Optimizing Far Calls (/F)

Option

/F[[ARCALLTRANSLATION]]

The /F option directs the linker to optimize far calls to procedures that lie in the same segment as the caller. Using the /F option may result in slightly faster code and smaller executable-file size. It should be used with the /PAC option (see Section 5.4.21) for significant results. By default, the /F option is off. Furthermore, once you have enabled it, you can disable it for one or more object files by using the /NOF option (see Section 5.4.16).

For example, a medium- or large-model program may include a machine instruction that makes a far call to a procedure in the same segment. Because both the instruction and the procedure it calls have the same segment address, only a near call is truly necessary. A near-call instruction does not require an entry in the relocation table as a far-call instruction does. In this situation, use of /F (together with /PAC) would result in a smaller executable file because the relocation table is smaller. Such files load faster.

When /F has been specified the linker optimizes code by removing the following instruction:

```
call FAR label
```

and substituting the sequence

```
push    cs
call    NEAR label
nop
```

Upon execution, the called procedure still returns with a far-return instruction. Because both the code segment and the near address are on the stack, however, the far return is executed correctly. The `nop` (no-op) instruction appears so that exactly five bytes replace the five-byte far-call instruction; the linker may in some cases place `nop` at the beginning of the sequence.

The /F option has no effect on programs that make only near calls. Of the high-level Microsoft languages, only small- and compact-model C programs use near calls.

NOTE *There is a small risk involved with the /F option: the linker may mistakenly translate a byte in a code segment that happens to have the far-call opcode (9A hexadecimal). If a program linked with /F inexplicably fails, then you may want to try linking with this option off. Object modules produced by Microsoft high-level languages, however, should be safe from this problem because relatively little immediate data is stored in code segments.*

In general, assembly-language programs are also relatively safe for use with the /F option, as long as they do not involve advanced system-level code, such as might be found in operating systems or interrupt handlers.

5.4.9 Viewing the Options List (/HE)

Option

/HE[LP]

The /HELP option causes LINK to display a list of its options on the screen. This gives you a convenient reminder of the options.

When you use this option, LINK ignores any other input you give and does not create an executable file.

5.4.10 Controlling Executable-File Loading (/HI)

Option

/HI[[GH]]

The /Hi option allows you to control where the executable file is placed in memory. The executable file can be placed either as low or as high in memory as possible. The /HI option tells LINK to place the executable file as high as possible in memory. Without the /HI option, LINK places the executable file as low as possible.

WARNING *This option should be used only with assembly-language programs.*

5.4.11 Displaying Linker-Process Information (/INF)

Option

/INF[[ORMATION]]

The /INF option tells the linker to display information about the linking process, including the phase of linking and the names of the object files being linked. This option is useful if you want to determine the locations of the object files being linked and the order in which they are linked.

Output from this option is sent to the standard error output.

Example

The following is a sample of the linker output when the /INF option is specified on the LINK command line:

```
**** PASS ONE ****
HSTGM.OBJ(hstgm.c)
**** LIBRARY SEARCH ****
\qc\lib\SLIBCE.LIB(CRT0)
\qc\lib\SLIBCE.LIB(CRT0DAT)
\qc\lib\SLIBCE.LIB(CRT0MSG)
\qc\lib\SLIBCE.LIB(CRT0FP)
\qc\lib\SLIBCE.LIB(CHKSTK)
\qc\lib\SLIBCE.LIB(CHKSUM)
.
.
.
```

```
**** ASSIGN ADDRESSES ****
**** PASS TWO ****
HSTGM.OBJ(hstgm.c)
\qc\lib\SLIBCE.LIB(CRT0)
\qc\lib\SLIBCE.LIB(CRT0DAT)
\qc\lib\SLIBCE.LIB(CRT0MSG)
\qc\lib\SLIBCE.LIB(CRT0FP)
\qc\lib\SLIBCE.LIB(CHKSTK)
\qc\lib\SLIBCE.LIB(CHKSUM)
**** WRITING EXECUTABLE ****

Segments              31
Groups                1
Bytes in symbol table  32784
```

5.4.12 Including Line Numbers in the Map File (/LI)

Option

/LI[[NENUMBERS]]

You can include the line numbers and associated addresses of your source program in the map file by using the /LI option. This option is primarily useful if you will be debugging with the SYMDEB debugger included with earlier releases of Microsoft language products.

Ordinarily the map file does not contain line numbers. To produce a map file with line numbers, you must give LINK an object file (or files) with line-number information. The /Zd option of the QCL command (see Section 4.3.31) directs the compiler to include line numbers in the object file. If you give LINK an object file without line-number information, the /LI option has no effect.

The /LI option forces LINK to create a map file even if you did not explicitly tell the linker to create a map file. By default, the file is given the same base name as the executable file plus the extension .MAP. You can override the default name by specifying a new map file on the LINK command line or in response to the "List File" prompt.

5.4.13 Listing Public Symbols (/M)

Option

/M[[AP]]

You can list all public (global) symbols defined in the object file(s) by using the /M option. When you invoke LINK with the /M option, the map file contains a list of all the symbols sorted by name and a list of all the symbols sorted by address. If you do not use this option, the map file contains only a list of segments.

When you use this option, the default for the *mapfile* field or "List File" prompt response is no longer NUL. Instead, the default is a name that combines the base name of the executable file with a .MAP extension. You may still specify NUL in the *mapfile* field (which indicates that no map file is to be generated); if you do, the /M option has no effect.

NOTE *In earlier versions of LINK, number specified the maximum number of public symbols that LINK could sort; the current version of LINK sorts the maximum number of symbols that can be sorted in available memory.*

5.4.14 Ignoring Default Libraries (/NOD)

Option

/NOD[EFAULTLIBRARYSEARCH] [:*filename*]

The /NOD option tells LINK *not* to search any library specified in the object file to resolve external references. If you specify *filename*, then LINK searches all libraries specified in the object file except for *filename*.

In general, higher-level-language programs do not work correctly without a standard library. Therefore, if you use the /NOD option, you should explicitly specify the name of a standard library in the *libraries* field.

5.4.15 Ignoring Extended Dictionary (/NOE)

Option

/NOE[XTDICTIONARY]

The /NOE option prevents the linker from searching the extended dictionary, which is an internal list of symbol locations that the linker maintains. Normally, the linker consults this list to speed up library searches. The effect of the /NOE option is to slow down the linker. You often need this option when a library symbol is redefined. Use /NOE if the linker issues the following error message:

```
symbol name multiply defined
```

5.4.16 Disabling Far-Call Optimization (/NOF)

Option

/NOF[ARCALLTRANSLATION]

This option is normally not necessary because far-call optimization (translation) is turned off by default. However, if an environment variable such as LINK

(or CL) turns on far-call translation automatically, you can use /NOF to turn far-call translation off again.

5.4.17 Preserving Compatibility (/NOG)

Option

/NOG[ROUPASSOCIATION]

The /NOG option causes the linker to ignore group associations when assigning addresses to data and code items. It is provided primarily for compatibility with previous versions of the linker (Versions 2.02 and earlier) and early versions of Microsoft language compilers.

WARNING *This option should be used only with assembly-language programs.*

5.4.18 Preserving Case Sensitivity (/NOI)

Option

/NOI[GNORECASE]

By default, LINK treats uppercase letters and lowercase letters as equivalent. Thus `ABC`, `abc`, and `Abc` are considered the same name. When you use the /NOI option, the linker distinguishes between uppercase letters and lowercase letters, and considers `ABC`, `abc`, and `Abc` to be three separate names. Because names in some high-level languages are not case sensitive, this option can have minimal importance. In Microsoft QuickC, however, case is significant. If you plan to link your files from other high-level languages with Microsoft QuickC routines, you may need to use this option.

5.4.19 Disabling Segment Packing (/NOP)

Option

/NOP[ACKCODE]

This option is normally not necessary because code-segment packing is turned off by default. However, if an environment variable such as LINK (or CL) turns on code-segment packing automatically, you can use /NOP to turn segment packing off again.

5.4.20 Setting the Overlay Interrupt (/O)

Option

/O[[VERLAYINTERRUPT]]:*number*

By default, the interrupt number used for passing control to overlays is 63 (3F hexadecimal). The /O option allows you to select a different interrupt number.

The *number* can be a decimal number from 0 to 255, an octal number from octal 0 to octal 0377, or a hexadecimal number from hexadecimal 0 to hexadecimal FF. Numbers that conflict with DOS interrupts can be used; however, their use is not advised.

In general, you should not use /O with programs. The exception to this guideline would be a program that uses overlays and spawns another program that also uses overlays. In this case, each program should use a separate overlay-interrupt number, meaning that at least one of the programs should be compiled with /O.

5.4.21 Packing Contiguous Segments (/PAC)

Option

/PAC[[KCODE]][[:*number*]]

The /PAC option affects code segments only in medium- and large-model programs. It is intended to be used with the /F option. It is not necessary to understand the details of the /PAC option in order to use it. You only need to know that this option, used in conjunction with /F, produces slightly faster and more compact code. The packing of code segments provides more opportunities for far-call optimization, which is enabled with /F. The /PAC option is off by default and can always be turned off with the /NOP option.

The /PAC option directs the linker to group neighboring code segments. Segments in the same group are assigned the same segment address; offset addresses are adjusted upward accordingly. In other words, all items have the correct physical address whether the /PAC option is used or not. However, /PAC changes segment and offset addresses so that all items in a group share the same segment address.

The *number* field specifies the maximum size of groups formed by /PAC. The linker stops adding segments to a group as soon as it cannot add another segment without exceeding *number*. At that point, the linker starts forming a new group. The default for *number* is 65,530.

You can safely use /PAC with programs developed with the Microsoft QuickC Compiler. The /PAC option, however, should not be used with assembly programs that make assumptions about the relative order of code segments. For example, the following assembly code attempts to calculate the distance between CSEG1 and CSEG2. This code would produce incorrect results when used with /PAC because /PAC causes the two segments to share the same segment address. Therefore, the procedure would always return 0.

```
CSEG1     SEGMENT PARA PUBLIC 'CODE'
  .
  .
  .
CSEG1     ENDS

CSEG2     SEGMENT PARA PUBLIC 'CODE'
          ASSUME  cs:CSEG2

; Return the length of CSEG1 in AX.

codsize   PROC    NEAR
          mov     ax,CSEG2      ; Load para address of CSEG1
          sub     ax,CSEG1      ; Load para address of CSEG2
          mov     cx,4          ; Load count, and convert
          shl     ax,cl         ;   distance from paragraphs
                                ;   to bytes
codsize   ENDP

CSEG2     ENDS
```

5.4.22 Pausing during Linking (/PAU)

Option

/PAU[[SE]]

The /PAU option tells LINK to pause before it writes the executable (.EXE) file to disk. This option is useful on machines without hard disks, where you might want to create the executable file on a new removable (floppy) disk. Without the /PAU option, LINK performs the linking session from beginning to end without stopping.

If you specify the /PAU option, LINK displays the following message before it creates the file:

```
About to generate .EXE file
Change diskette in drive letter and press <ENTER>
```

The *letter* corresponds to the current drive. LINK resumes processing when you press ENTER.

NOTE *Do not remove the disk that will receive the listing file or the disk used for the temporary file.*

Depending on how much memory is available, LINK may create a temporary disk file during processing, as described in Section 5.3, and display the following message:

```
Temporary file tempfile has been created.
Do not change diskette in drive, letter
```

If the file is created on the disk you plan to swap, press CTRL+C to terminate the LINK session. Rearrange your files so that the temporary file and the executable file can be written to the same disk, then try linking again.

5.4.23 Setting Maximum Number of Segments (/SE)

Option

/SE[[GMENTS]]:*number*

The /SE option controls the number of segments that the linker allows a program to have. The default is 128, but you can set *number* to any value (decimal, octal, or hexadecimal) in the range 1–3072 (decimal).

For each segment, the linker must allocate some space to keep track of segment information. By using a relatively low segment limit as a default (128), the linker is able to link faster and allocate less storage space.

When you set the segment limit higher than 128, the linker allocates additional space for segment information. This option allows you to raise the segment limit for programs with a large number of segments. For programs with fewer than 128 segments, you can keep the storage requirements of the linker at the lowest level possible by setting *number* to reflect the actual number of segments in the program. If the number of segments allocated is too high for the amount of memory available to the linker, LINK issues the following error message:

```
segment limit set too high
```

If this occurs, relink the object files, specifying a lower segment limit.

5.4.24 Controlling Stack Size (/ST)

Option

/ST[[ACK]]:*number*

The /ST option allows you to specify the size of the stack for your program. The *number* is any positive value (decimal, octal, or hexadecimal) up to 65,535

(decimal). It represents the size, in bytes, of the stack. If you do not use this option, the stack size is 2K.

If your program returns a stack-overflow message, you may need to increase the size of the stack. In contrast, if your program uses the stack very little, you may save some space by decreasing the stack size.

5.5 Linker Operation

LINK performs the following steps to combine object modules and produce an executable file:

1. Reads the object modules submitted
2. Searches the given libraries, if necessary, to resolve external references
3. Assigns addresses to segments
4. Assigns addresses to public symbols
5. Reads code and data in the segments
6. Reads all relocation references in object modules
7. Performs fixups
8. Outputs an executable file (executable image and relocation information)

Steps 5, 6, and 7 are performed concurrently: in other words, LINK moves back and forth between these steps before it progresses to step 8.

The "executable image" contains the code and data that constitute the executable file. The "relocation information" is a list of references, relative to the start of the program. The references change when the executable image is loaded into memory and an actual address for the entry point is assigned.

The following sections explain the process LINK uses to concatenate segments and resolve references to items in memory.

5.5.1 Alignment of Segments

LINK uses a segment's alignment type to set the starting address for the segment. The alignment types are BYTE, WORD, PARA, and PAGE. These correspond to starting addresses at byte, word, paragraph, and page boundaries, representing addresses that are multiples of 1, 2, 16, and 256, respectively. The default alignment is PARA.

When LINK encounters a segment, it checks the alignment type before copying the segment to the executable file. If the alignment is WORD, PARA, or PAGE, LINK checks the executable image to see if the last byte copied ends on the appropriate boundary. If not, LINK pads the image with null bytes.

5.5.2 Frame Number

LINK computes a starting address for each segment in the program. The starting address is based on the segment's alignment and the sizes of the segments already copied to the executable file (as described in the previous section). The starting address consists of an offset and a "canonical frame number." The canonical frame number specifies the address of the first paragraph in memory that contains one or more bytes of the segment. (A paragraph is 16 bytes of memory; therefore, to compute a physical location in memory, multiply the frame number by 16 and add the offset.) The offset is the number of bytes from the start of the paragraph to the first byte in the segment. For BYTE and WORD alignments, the offset may be nonzero. The offset is always zero for PARA and PAGE alignments. (An offset of zero means that the physical location is an exact multiple of 16.)

You can find the frame number for each segment in the map file created by LINK. The first four digits of the segment's start address give the frame number in hexadecimal. For example, a start address of `0C0A6` indicates the frame number `0C0A`.

5.5.3 Order of Segments

LINK copies segments to the executable file in the same order that it encounters them in the object files. This order is maintained throughout the program unless LINK encounters two or more segments that have the same class name. Segments having identical segment names are copied as a contiguous block to the executable file.

The /DOSSEG option may change the way in which segments are ordered. (See Section 5.4.5.)

5.5.4 Combined Segments

LINK uses combine types to determine whether two or more segments that share the same segment name should be combined into one large segment. The valid combine types are PUBLIC, STACK, COMMON, and PRIVATE.

If a segment has combine type PUBLIC, LINK automatically combines it with any other segments that have the same name and belong to the same class. When LINK combines segments, it ensures that the segments are contiguous and that

all addresses in the segments can be accessed using an offset from the same frame address. The result is the same as if the segment were defined as a whole in the source file.

LINK preserves each individual segment's alignment type. This means that even though the segments belong to a single, large segment, the code and data in the segments do not lose their original alignment. If the combined segments exceed 64K, LINK displays an error message.

If a segment has combine type STACK, LINK carries out the same combine operation as for PUBLIC segments. The only exception is that STACK segments cause LINK to copy an initial stack-pointer value to the executable file. This stack-pointer value is the offset to the end of the first stack segment (or combined stack segment) encountered.

If a segment has combine type COMMON, LINK automatically combines it with any other segments that have the same name and belong to the same class. When LINK combines COMMON segments, however, it places the start of each segment at the same address, creating a series of overlapping segments. The result is a single segment no larger than the largest segment combined.

A segment has combine type PRIVATE only if no explicit combine type is defined for it in the source file. LINK does not combine private segments.

5.5.5 Groups

Groups allow segments to be addressed relative to the same frame address. When LINK encounters a group, it adjusts all memory references to items in the group so that they are relative to the same frame address.

Segments in a group do not have to be contiguous, belong to the same class, or have the same combine type. The only requirement is that all segments in the group fit within 64K.

Groups do not affect the order in which the segments are loaded. Unless you use class names and enter object files in the right order, there is no guarantee that the segments will be contiguous. In fact, LINK may place segments that do not belong to the group in the same 64K of memory. LINK does not explicitly check whether all the segments in a group fit within 64K of memory; however, LINK is likely to encounter a fixup-overflow error if they do not.

5.5.6 Fixups

Once the linker knows the starting address of each segment in the program and has established all segment combinations and groups, LINK can "fix up" any unresolved references to labels and variables. To fix up unresolved references, LINK computes the appropriate offset and segment address and replaces the temporary values generated by the assembler with the new values.

LINK carries out fixups for the types of references shown in Table 5.1.

Table 5.1 LINK Fixups

Type	Location of Reference	LINK Action
Short	In JMP instructions that attempt to pass control to labeled instructions in the same segment or group. The target instruction must be no more than 128 bytes from the point of reference.	Computes a signed, eight-bit number for the reference, and displays an error message if the target instruction belongs to a different segment or group (has a different frame address), or if the target is more than 128 bytes away in either direction.
Near self-relative	In instructions that access data relative to the same segment or group.	Computes a 16-bit offset for the reference and displays an error if the data are not in the same segment or group.
Near segment-relative	In instructions that attempt to access data in a specified segment or group, or relative to a specified segment register.	Computes a 16-bit offset for the reference, and displays an error message if the offset of the target within the specified frame is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.
Long	In CALL instructions that attempt to access an instruction in another segment or group.	Computes a 16-bit frame address and 16-bit offset for this reference, and displays an error message if the computed offset is greater than 64K or less than 0, or if the beginning of the canonical frame of the target is not addressable.

The size of the value to be computed depends on the type of reference. If LINK discovers an error in the anticipated size of a reference, it displays a fixup-overflow message. This can happen, for example, if a program attempts to use a 16-bit offset to reach an instruction which is more than 64K away. It can also occur if all segments in a group do not fit within a single 64K block of memory.

5.6 Using Overlays

You can direct LINK to create an overlaid version of a program. In an overlaid version of a program, specified parts of the program (known as "overlays") are loaded only if and when they are needed. These parts share the same space in memory. Only code is overlaid; data are never overlaid. Programs that use overlays usually require less memory, but they run more slowly because of the time needed to read and reread the code from disk into memory.

You specify overlays by enclosing them in parentheses in the list of object files that you submit to the linker. Each module in parentheses represents one overlay. For example, you could give the following object-file list in the *objfiles* field of the LINK command line:

```
a + (b+c) + (e+f) + g + (i)
```

In this example, the modules `(b+c)`, `(e+f)`, and `(i)` are overlays. The remaining modules, and any drawn from the run-time libraries, constitute the resident part (or root) of your program. Overlays are loaded into the same region of memory, so only one can be resident at a time. Duplicate names in different overlays are not supported, so each module can appear only once in a program.

The linker replaces calls from the root to an overlay, and calls from an overlay to another overlay, with an interrupt (followed by the module identifier and offset). By default, the interrupt number is 63 (3F hexadecimal). You can use the /OVERLAYINTERRUPT option of the LINK command to change the interrupt number.

The CodeView debugger is compatible with overlaid modules. In fact, in the case of large programs, you may need to use overlays to leave sufficient room for the debugger to operate.

5.6.1 Restrictions on Overlays

You can overlay only modules to which control is transferred and returned by a standard 8086 long (32-bit) call/return instruction. Therefore, because calls to subroutines modified with the **near** attribute are short (16-bit) calls, you cannot overlay modules containing **near** subroutines if other modules call those subroutines. You cannot use long jumps with the **longjmp** library function. Also, the linker does not produce overlay modules that can be called indirectly through function pointers. When a function is called through a pointer, the called function must be in the same overlay or root.

5.6.2 Overlay-Manager Prompts

The overlay manager is part of the language's run-time library. If you specify overlays during linking, the code for the overlay manager is automatically linked with the other modules of your program. Even with overlays, the linker produces only one .EXE file. At run time, the overlay manager opens the .EXE file each time it needs to extract new overlay modules. The overlay manager first searches for the file in the current directory; then, if it does not find the file, the manager searches the directories listed in the PATH environment variable. When it finds the file, the overlay manager extracts the overlay modules specified by the root program. If the overlay manager cannot find an overlay file when needed, it prompts you for the file name.

For example, assume that an executable program called `PAYROLL.EXE` uses overlays and does not exist in either the current directory or the directories specified by PATH. If you run `PAYROLL.EXE` (by entering a complete path specification), the overlay manager displays the following message when it attempts to load overlay files:

```
Cannot find PAYROLL.EXE
Please enter new program spec:
```

You can then enter the drive or directory, or both, where `PAYROLL.EXE` is located. For example, if the file is located in directory `\EMPLOYEE\DATA\` on drive B, you could enter `B:\EMPLOYEE\DATA\` or simply `\EMPLOYEE\DATA\` if the current drive is B.

If you later remove the disk in drive B and the overlay manager needs to access the overlay again, it does not find `PAYROLL.EXE` and displays the following message:

```
Please insert diskette containing
B:\EMPLOYEE\DATA\PAYROLL.EXE
in drive B: and strike any key when ready.
```

After reading the overlay file from the disk, the overlay manager displays the following message:

```
Please restore the original diskette.
Strike any key when ready.
```

Execution of the program then continues.

CHAPTER 6 LIB

The Microsoft Library Manager (LIB) helps you create and maintain object-code libraries. An "object-code library" is a collection of separately compiled or assembled object files combined into a single file. Object-code libraries provide a convenient source of commonly used routines. A program that calls library routines is linked with the library to produce the executable file. Only the necessary routines, not all library routines, are linked into the executable file.

Library files are usually identified by their .LIB extension, although other extensions are allowed. In addition to accepting DOS object files and library files, LIB can read the contents of 286 XENIX® archives and Intel-style libraries and combine their contents with DOS libraries.

You can use the LIB utility for the following tasks:

- Create a new library file
- Add object files or the contents of a library to an existing library
- Delete library modules
- Replace library modules
- Copy library modules to object files

6.1 Invoking LIB

To invoke the Library Manager (LIB), type the LIB command on the DOS command line. You can specify the input required in one of three ways:

1. Type it on the command line.
2. Respond to prompts.
3. Specify a file containing responses to prompts ("response file").

The three sections below present the three methods of invoking LIB. Section 6.1.1 describes the input fields in detail and is relevant to all three methods.

To terminate the library session at any time and return to DOS, press CTRL+C.

6.1.1 Command Line

You can start LIB and specify all the necessary input from the command line. In this case, the LIB command line has the following form:

LIB *oldlibrary* [[*options*]] [[*commands*]][[,[[*listfile*]][[,[[*newlibrary*]]]]][[;]]

The individual components of the command line are discussed in the sections that follow.

Type a semicolon (;) after any field except the *oldlibrary* field to tell LIB to use the default responses for the remaining fields. The semicolon should be the last character on the command line.

Typing a semicolon after the *oldlibrary* field causes LIB to perform a consistency check on the library—no other action is performed. LIB displays any consistency errors it finds and returns to the operating-system level.

Examples

```
LIB GRAPHIC;
```

The example above causes LIB to perform a consistency check of the library file `GRAPHIC.LIB`.

```
LIB GRAPHIC ,SYMBOLS.LST;
```

This example tells LIB to perform a consistency check of the library file `GRAPHIC.LIB` and to create `SYMBOLS.LST`, a cross-reference-listing file.

```
LIB GRAPHIC +STAR;
```

The example above uses the add-command symbol (+) to instruct LIB to add the file STAR to the library GRAPHIC.LIB. The semicolon at the end of the command line causes LIB to use the default responses for the remaining fields. As a result, no listing file is created and the original library file is renamed GRAPHIC.BAK. The modified library is GRAPHIC.LIB.

```
LIB GRAPHIC -*JUNK *STAR, ,SHOW
```

This last example instructs LIB to move the module JUNK from the library GRAPHIC.LIB to an object file called JUNK.OBJ. The module JUNK is removed from the library in the process. The module STAR is copied from the library to an object file called STAR.OBJ; the module remains in the library. No cross-reference-listing file is produced. The revised library is called SHOW.LIB. It contains all the modules in GRAPHIC.LIB except JUNK, which was removed by using the move-command symbol (-*). The original library, GRAPHIC.LIB, remains unchanged.

6.1.1.1 Library File

Use the *oldlibrary* field to specify the name of the library to be modified. The LIB utility assumes that the file-name extension is .LIB, because this is usually the case. If your library file has the .LIB extension, you can omit it. Otherwise, include the extension. You must give LIB the path name of a library file if it is in another directory or on another disk.

There is no default for the *oldlibrary* field. This field is required and LIB issues an error message if you do not give a file name.

Consistency check

If you type a library name and follow it immediately with a semicolon (;), LIB only performs a consistency check on the given library. A consistency check tells you whether all the modules in the library are in usable form. No changes are made to the library. It usually is not necessary to perform consistency checks because LIB automatically checks object files for consistency before adding them to the library. LIB prints a message if it finds an invalid object module; no message appears if all modules are intact.

6.1.1.2 LIB Options

The Library Manager has four options. Specify options on the command line following the required library-file name and preceding any commands.

Option

/I[[GNORECASE]]

The /I option tells LIB to ignore case when comparing symbols, which is the default. Use this option when you are combining a library that is case sensitive (was created with the /NOI option) with others that are not case sensitive. The resulting library will not be case sensitive. The /NOI option is described later in this section.

Option

/NOE[[XTDICTIONARY]]

The /NOE option tells LIB not to generate an extended dictionary. The extended dictionary is an extra part of the library that helps the linker process libraries faster.

Use the /NOE option if you get either the error message `insufficient memory` or `no more virtual memory`, or if the extended dictionary causes problems with the linker. For more information on how the linker uses the extended dictionary, see Section 5.4.15.

Option

/NOI[[GNORECASE]]

The /NOI option tells LIB not to ignore case when comparing symbols; that is, /NOI makes LIB case sensitive. By default, LIB ignores case. Using this option allows symbols that are the same except for case, such as `Spline` and `SPLINE`, to be put in the same library.

Note that when you create a library with the /NOI option, LIB "marks" the library internally to indicate that /NOI is in effect. Earlier versions of LIB did not mark libraries in this way. If you combine multiple libraries and any one of them is marked /NOI, then /NOI is assumed to be in effect for the output library.

Option

/PA[[GESIZE]]:*number*

The /PA option specifies the library-page size of a new library or changes the library-page size of an existing library. The *number* specifies the new page size. It must be an integer value representing a power of 2 between the values 16 and 32,768.

A library's page size affects the alignment of modules stored in the library. Modules in the library are always aligned to start at a position that is a multiple of the

page size (in bytes) from the beginning of the file. The default page size for a new library is 16 bytes; for an existing library, the default is its current page size. Because of the indexing technique used by the LIB utility, a library with a large page size can hold more modules than a library with a smaller page size. For each module in the library, however, an average of *number*/2 bytes of storage space is wasted. In most cases, a small page size is advantageous; you should use a small page size unless you need to put a very large number of modules in a library.

Another consequence of the indexing technique is that the page size determines the maximum possible size of the library file. Specifically, this limit is *number* * 65,536. For example, `/PA:16` means that the library file must be smaller than 1 megabyte (16 * 65,536 bytes).

6.1.1.3 Commands

The *commands* field allows you to specify the command symbols for manipulating modules. In this field, type a command symbol followed immediately by a module name or an object-file name. The command symbols are the following:

Symbol	Action
+	Adds an object file or library to the library
–	Deletes a module from the library
–+	Replaces a module in the library
*	Copies a module from the library to a object file
–*	Moves a module (copies the module and then deletes it)

Note that LIB does not process commands in left-to-right order; it uses its own precedence rules for processing, as described in Section 6.2. You can specify more than one operation in the *commands* field, in any order. LIB makes no changes to *oldlibrary* if you leave this field blank.

6.1.1.4 Cross-Reference-Listing File

The *listfile* field allows you to specify a file name for a cross-reference-listing file. You can give the listing file any name and any extension. To create it outside your current directory, supply a path specification. Note that the LIB utility does not assume any defaults for this field on the command line. If you do not specify a name for the file, the file is not created.

A cross-reference-listing file contains the following two lists:

1. An alphabetical list of all public symbols in the library.

   Each symbol name is followed by the name of the module in which it is defined. The example output below shows that the public symbol `ADD` is contained in the module `junk` and the public symbols `CALC`, `MAKE`, and `ROLL` are contained in the module `dice`:

```
ADD.............junk
CALC............dice
MAKE............dice
ROLL............dice
```

2. A list of the modules in the library.

   Under each module name is an alphabetical listing of the public symbols defined in that module. The example output below shows that the module `dice` contains the public symbols `CALC`, `MAKE`, and `ROLL` and the module `junk` contains the public symbol `ADD`:

```
dice              Offset: 00000010H  Code and data size: 621H
  CALC               MAKE                ROLL

junk              Offset: 00000bc0H  Code and data size: 118H
  ADD
```

6.1.1.5 Output Library

If you specify a name in the *newlibrary* field, LIB gives this name to the modified library it creates. You need not give a name unless you specify commands to change the library.

If you leave this field blank, the original library is renamed with a .BAK extension and the modified library receives the original name.

6.1.2 Prompts

If you want to respond to individual prompts to give input to LIB, start the LIB utility at the DOS command level by typing LIB. The library manager prompts you for the input it needs by displaying the following four messages, one at a time:

```
Library name:
Operations:
List file:
Output library:
```

LIB waits for you to respond to each prompt before printing the next prompt. If you notice that you have entered an incorrect response to a previous prompt, press CTRL+C to exit LIB and begin again.

The responses to the LIB command prompts correspond to fields on the LIB command line (Section 6.1.1), as follows:

Prompt	Command-Line Field
"Library name"	The *oldlibrary* field and the options. To perform a consistency check on the library, type a semicolon (;) immediately after the library name. If the library you name does not exist, LIB displays the following prompt: `Library does not exist. Create? (y/n)` Type `y` to create the library file, or `n` to terminate the session. This message does not appear if a command, a comma, or a semicolon immediately follows the library name.
"Operations"	The *commands* field.
"List file"	The *listfile* field.
"Output library"	The *newlibrary* field. This prompt appears only if you specify at least one operation at the "Operations" prompt.

Extending lines

If you have many operations to perform during a library session, use the ampersand symbol (&) to extend the operations line. Type the ampersand symbol after the name of an object module or object file; do not put the ampersand between a command symbol and a name.

The ampersand causes LIB to display the "Operations" prompt again, allowing you to specify more operations.

Default responses

Press ENTER to choose the default response for the current prompt. Type a semicolon (;) and press ENTER after any response except "Library name" to select default responses for all remaining prompts.

The following list shows the defaults for LIB prompts:

Prompt	Default
"Operations"	No operation; no change to library file
"List file"	NUL; tells LIB not to create a listing file
"Output library"	The current library name

6.1.3 Response File

Using a response file lets you conduct the library session without typing responses to prompts at the keyboard. To run LIB with a response file, you must first create the response file. Then type the following at the DOS command line:

LIB @*responsefile*

The *responsefile* is the name of a response file. Specify a path name if the response file is not in the current directory.

You can also enter @*responsefile* at any position on a command line or after any of the prompts. The input from the response file is treated exactly as if it had been entered on a command line or after the prompts. A new-line character in the response file is treated the same as pressing the ENTER key in response to a prompt.

A response file uses one text line for each prompt. Responses must appear in the same order as the command prompts appear. Use command symbols in the response file the same way you would use responses typed on the keyboard. You can type an ampersand (&) at the end of the response to the "Operations" prompt, for instance, and continue typing operations on the next line. This mechanism is illustrated in Figure 6.1.

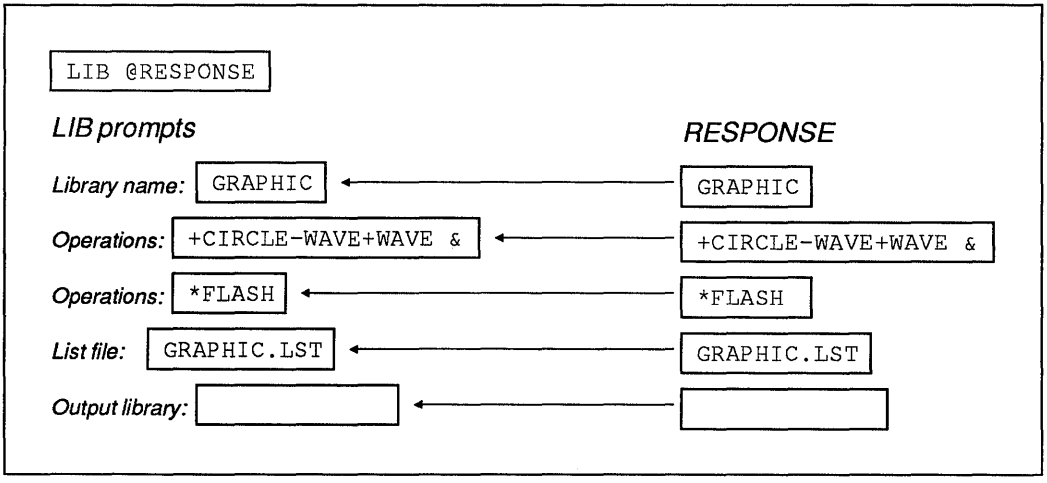


Figure 6.1 LIB Response File

When you run LIB with a response file, the prompts are displayed with the responses from the response file. If the response file does not contain responses for all the prompts, LIB uses the default responses.

Example

```
GRAPHIC
+CIRCLE+WAVE-WAVE*FLASH
GRAPHIC.LST
```

Assume that a response file named `response` in the directory `b:\proj` contains the above lines and you invoke LIB with the command shown below:

```
LIB @b:\proj\response
```

LIB deletes the module `WAVE` from the library `GRAPHIC.LIB`, copies the module `FLASH` into an object file named `FLASH.OBJ`, and appends the object files `CIRCLE.OBJ` and `WAVE.OBJ` as the last two modules in the library. LIB also creates a cross-reference-listing file named `GRAPHIC.LST`.

6.2 LIB Commands

The LIB utility can perform a number of library-management functions, including creating a library file, adding an object file as a module to a library, deleting modules from a library, replacing a module in the library file, copying a module to a separate object file, and moving a module out of a library and into an object file.

For each library session, LIB reads and interprets commands in the order listed below. It determines whether a new library is being created or an existing library is being examined or modified.

1. LIB processes any deletion and move commands.

   LIB does not actually delete modules from the existing file. Instead, it marks the selected modules for deletion, creates a new library file, and copies only the modules *not* marked for deletion into the new library file.

2. LIB processes any addition commands.

   Like deletions, additions are not performed on the original library file. Instead, the additional modules are appended to the new library file. (If there were no deletion or move commands, a new library file would be created in the addition stage by copying the original library file.)

As the LIB utility carries out these commands, it reads the object modules in the library, checks them for validity, and gathers the information necessary to build a library index and a listing file. When you link a library with other object files, the linker uses the library index to search the library.

LIB never makes changes to the original library; it copies the library and makes changes to the copy. Therefore, if you press CTRL+C to terminate the session, you do not lose your original library. Therefore, when you run LIB, make sure your disk has enough space for both the original library file and the copy.

Once an object file is incorporated into a library, it becomes an "object module." The LIB utility makes a distinction between object files and object modules: an object file exists as an independent file while an object module is part of a library file. An object file has a full path name, including a drive designation, directory path name, and file-name extension (usually .OBJ). Object modules have only a name. For example, `B:\RUN\SORT.OBJ` is an object-file name, while `SORT` is an object-module name.

6.2.1 Creating a Library File

To create a new library file, give the name of the library file you want to create in the *oldlibrary* field of the command line or at the "Library name" prompt. LIB supplies the .LIB extension.

If the name of the new library file is the same as the name of an existing library file, LIB assumes that you want to change the existing file. If the name of the new library file is the same as the name of a file that is not a library, LIB issues an error message.

When you give the name of a file that does not currently exist, LIB displays the following prompt:

```
Library does not exist. Create? (y/n)
```

Type `y` to create the file, or `n` to terminate the library session. This message does not appear if the name is followed immediately by a command, a comma, or a semicolon.

You can specify a page size for the library by specifying the /PAGESIZE option when you create the library (see Section 6.1.1.2). The default page size is 16 bytes.

Once you have given the name of the new library file, you can insert object modules into the library by using the add-command symbol (+).

6.2.2 Add Command (+)

Use the add-command symbol (+) to add an object module to a library. Give the name of the object file to be added, without the .OBJ extension, immediately following the plus sign.

LIB uses the base name of the object file as the name of the object module in the library. For example, if the object file `B:\CURSOR.OBJ` is added to a library file, the name of the corresponding object module is `CURSOR`.

Object modules are always added to the end of a library file.

Combining libraries

You can also use the plus sign to combine two libraries. When you give a library name following the plus sign, a copy of the contents of that library is added to the library file being modified. You must include the .LIB extension when you give a library-file name. Otherwise, LIB uses the default .OBJ extension when it looks for the file. If both libraries contain a module with the same name, LIB ignores the second module of that name. For information on replacing modules, see Section 6.2.4.

LIB adds the modules of the library to the end of the library being changed. Note that the added library still exists as an independent library because LIB copies the modules without deleting them.

In addition to allowing DOS libraries as input, LIB also accepts 286 XENIX archives and Intel-format libraries. Therefore, you can use LIB to convert libraries from either of these formats to the DOS format.

Examples

```
LIB mainlib +flash;
```

This command adds the file `flash.obj` to the library `mainlib.lib`.

```
LIB math +trig.lib;
```

The command above adds the contents of the library `trig.lib` to the library `math.lib`. The library `trig.lib` is unchanged after this command is executed.

6.2.3 Delete Command (–)

Use the delete-command symbol (–) to delete an object module from a library. After the minus sign, give the name of the module to be deleted. Module names do not have path names or extensions.

Example

```
LIB mainlib -flash;
```

The command shown above deletes the module `flash` from the library `mainlib.lib`.

6.2.4 Replace Command (–+)

Use the replace-command symbol (–+) to replace a module in a library. Following the symbol, give the name of the module to be replaced. Module names do not have path names or extensions.

To replace a module, LIB first deletes the existing module, then appends an object file that has the same name as the module. The object file is assumed to have the .OBJ extension and to reside in the current directory; if not, give the object-file name with an explicit extension or path.

Example

```
LIB mainlib -+flash;
```

This command replaces the module `flash` in the `mainlib.lib` library with the contents of `flash.obj` from the current directory. Upon completion of this command, the file `flash.obj` still exists and the `flash` module is updated in `mainlib.lib`.

6.2.5 Copy Command (*)

Use the copy-command symbol (*) followed by a module name to copy a module from the library into an object file of the same name. The module remains in the library. When LIB copies the module to an object file, it adds the .OBJ extension to the module name and places the file in the current directory.

Example

```
LIB mainlib *flash;
```

This command copies the module `flash` from the `mainlib.lib` library to a file called `flash.obj` in the current directory. Upon completion of this command, `mainlib.lib` still contains the module `flash`.

6.2.6 Move Command (–*)

Use the move-command symbol (–*) to move an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, then deleting the module from the library.

Example

```
LIB mainlib -*flash;
```

This command moves the module `flash` from the `mainlib.lib` library to a file called `flash.obj` in the current directory. Upon completion of this command, `mainlib.lib` no longer contains the module `flash`.

CHAPTER 7
NMAKE

The Microsoft Program-Maintenance Utility (NMAKE) can save you time by automating the process of updating project files. NMAKE compares the modification dates for one set of files, the target files, to those of another set of files, the dependent files. If any of the dependent files have changed more recently than the target files, NMAKE executes a specified series of commands.

NMAKE is typically used by specifying a project's executable files as target files and the project's source files as the dependent files. If any of the source files have changed since the executable file was created, NMAKE can issue a command to assemble or compile the changed source files and link them into the executable file.

NMAKE reads the target- and dependent-file specifications from a "description file," also called a "makefile." The description file comprises any number of description blocks. Each description block lists one or more targets and the dependent files related to those targets. The block also gives the commands that NMAKE must execute to bring the targets up to date. The description file may also contain macros, inference rules, and directives.

7.1 Invoking NMAKE

Two methods for invoking NMAKE are available:

1. Specify options, macro definitions, and the names of targets to be built on the DOS command line.
2. Specify options, macro definitions, and the names of targets to be built in a response file, and give the file name on the DOS command line.

7.1.1 Using a Command Line to Invoke NMAKE

The syntax for invoking NMAKE from the command line is as follows:

NMAKE [[*options*]] [[*macrodefinitions*]] [[*target...*]] [[*filename*]]

The *options* field specifies options that modify the action of NMAKE. (Options are not required.) They are described in Section 7.2.

The optional *macrodefinitions* field lists macro definitions for NMAKE to use. Macros provide a convenient method for replacing a string of text in the description file. Macro definitions that contain spaces must be enclosed by quotation marks. Macros are discussed in Section 7.3.2.

The optional *target...* field specifies the name of one or more targets to build. If you do not list any targets, NMAKE builds the first target in the description file.

The optional *filename* field gives the name of the description file from which NMAKE reads target- and dependent-file specifications and commands. A better way of designating the description file is to use the /F option (described in Section 7.2). By default, NMAKE looks for a file named MAKEFILE in the current directory. If MAKEFILE does not exist, NMAKE uses the *filename* field: it interprets the first string on the command line that is not an option or macro definition as the name of the description file, provided its file-name extension isn't listed in the .SUFFIXES list. (See Section 7.3.5 for more information on the .SUFFIXES list.)

NOTE *Unless you use the /F option, NMAKE always searches for a file named MAKEFILE in the current directory.*

Example

```
NMAKE /S "program = flash" sort.exe search.exe
```

This example invokes NMAKE with the /S option, a macro assigning `flash` to `program`, and two targets, `sort.exe` and `search.exe`. By default, NMAKE uses the file named MAKEFILE as the description file.

7.1.2 Using a Response File to Invoke NMAKE

To invoke NMAKE with a response file, first create the response file, then issue a command with the following syntax:

NMAKE *@responsefile*

Here *commandfile* is the name of a file containing the same information that would be specified on the command line: options, macro definitions, and targets. The response file is *not* the same as the description file.

A response file is useful for invoking NMAKE with a long string of command-line arguments, such as macro definitions, that might exceed the DOS limit of 128 characters. NMAKE treats line breaks that occur between arguments as spaces. Macro definitions can span multiple lines by ending each line except the last with a backslash (\). Macro definitions that contain spaces must be enclosed by quotation marks, just as if they were entered directly on the command line.

Example

```
/S "program \
= flash" sort.exe search.exe
```

Assume a file named `update` contains the text above. The command below invokes NMAKE with the description file MAKEFILE, the /S option, the macro definition `program = flash`, and the targets `sort.exe` and `search.exe`. Note that the backslash ending the line allows the macro definition to span two lines.

```
NMAKE @update
```

7.2 NMAKE Options

NMAKE accepts a number of command-line options, which are listed below. You may specify options in uppercase or lowercase and use either a slash or dash. For example, –B, /B, –b, and /b all represent the same option.

Option	Action
/A	Executes commands to build all the targets requested even if they are not out of date.
/C	Suppresses the NMAKE copyright message and prevents nonfatal error or warning messages from being displayed.
/D	Displays the modification date of each file when the date is checked.
/E	Causes environment variables to override macro definitions within description files.
/F *filename*	Specifies *filename* as the name of the description file to use. If a dash (-) is entered instead of a file name, NMAKE accepts input from the standard input device instead of using a description file. If /F is not specified, NMAKE uses the file named MAKEFILE as the description file. If MAKEFILE does not exist, NMAKE uses the first string on the command line that is not an option or macro definition as the name of the file, provided the extension is not listed in the .SUFFIXES list (see Section 7.3.5).

/I	Ignores exit codes (also called return or "errorlevel" codes) returned by programs called from the NMAKE description file. NMAKE continues executing the rest of the description file despite the errors.
/N	Displays the commands from the description file that NMAKE would execute but does not execute these commands. This option is useful for checking which targets are out of date and for debugging description files.
/P	Prints all macro definitions and target descriptions.
/Q	Returns a zero status code if the target is up to date and a nonzero status code if it is not. This option is useful when invoking NMAKE from within a batch file.
/R	Ignores inference rules and macros contained in the TOOLS.INI file.
/S	Does not display commands as they are executed.
/T	Changes the modification dates for out-of-date target files to the current date. The file contents are *not* modified.
/X *filename*	Sends all error output to *filename*, which can be either a file or a device. If a dash (-) is entered instead of a file name, the error output is sent to the standard output device.

Examples

```
NMAKE /f quick /c f1 f2
```

The example above causes NMAKE to execute the commands in the description file `quick` to update the targets `f1` and `f2`. The /c option prevents NMAKE from displaying nonfatal error messages and warnings.

```
NMAKE  /D /N f1 f1.mak
```

In the example above, NMAKE updates the target `f1`. If the current directory does not contain a file named MAKEFILE, NMAKE reads the file `f1.mak` as the description file. The /D option displays the modification date of each file and the /N option displays the commands without executing them.

7.3 Description Files

NMAKE reads a description file to determine what to do. The description file may contain any number of description blocks, along with macros, inference rules, and directives. These can be in any order.

When NMAKE runs, it builds the first target in the description file by default. You can override this default by specifying on the command line the names of

the targets to build. The sections that follow describe the elements of a description file.

7.3.1 Description Blocks

An NMAKE description file contains one or more description blocks. Each has the following form:

target... : [[*dependent...*]] [[;*command*]][[#*comment*]]
    [[*command*]]
[[#*comment*]]
[[#*comment*]] | [[*command*]]
    .
    .
    .

The file to be updated is *target*; *dependent* is a file upon which *target* depends; *command* is a command used to update *target*; and *comment* documents what is happening. The line containing *target* and *dependent* is called the dependency line because *target* depends on *dependent*.

Each component of a description block is discussed below.

The target... field The *target* field specifies the name of one or more files to update. If you specify more than one file, separate the file names by a space. The first target name must start in the first column of the line; it may not be preceded by any tabs or spaces. Note that the target need not be a file; it may be a pseudotarget, as described in Section 7.3.5.

The dependent... field The *dependent* field lists one or more files on which the target depends. If you specify more than one file, separate the file names by a space. You can specify directories for NMAKE to search for the dependent files by using the following form:

target : {*directory1*;*directory2...*}*dependent*

NMAKE searches the current directory first, then *directory1*, *directory2*, and so on. If *dependent* cannot be found in any of these directories, NMAKE looks for an inference rule to create the dependent in the current directory. See Section 7.3.3 for more information on inference rules.

In the following example, NMAKE first searches the current directory for `pass.obj`, then the `\src\alpha` directory, and finally the `d:\proj` directory:

```
forward.exe : {\src\alpha;d:\proj}pass.obj
```

The command field

The *command* is used to update the target. This can be any command that can be issued on the DOS command line. A semicolon must precede the command if it is given on the same line as the target and dependent files. Commands may be placed on separate lines following the dependency line, but each line must start with at least one space or tab character. Blank lines may be intermixed with commands. A long command may span several lines if each line ends with a backslash (\). If no commands are specified, NMAKE looks for an inference rule to build the target.

The comment field

NMAKE considers any text between a number sign (#) and a new-line character to be a comment and ignores it. You may place a comment on a line by itself or at the end of any line except a command line. In the command section of the description file, comments must start in the first column.

Wild-card characters

You can use the DOS wild-card characters (* and ?) when specifying target- and dependent-file names. NMAKE expands wild cards in target names when it reads the description file. It expands wild cards in the dependent names when it builds the target. For example, the following description block compiles all source files with the .C extension:

```
astro.exe : *.c
        QCL *.c
```

Escape character

You can use a caret (^) to escape any DOS or OS/2 file-name character in a description file, so that the character takes on its literal meaning and does not have any special significance to NMAKE. The following characters must be preceded by an escape character for NMAKE to interpret them literally.

( ) $ ^ \ { } ! @ -

For example, NMAKE interprets the specification

```
big^#.c
```

as the file name

```
big#.c
```

Using the caret, you can include a literal new-line character in a description file. This capability is primarily useful in macro definitions, as in the following example:

```
XYZ=abc^
def
```

NMAKE interprets this example as if you had assigned to the XYZ macro the C-style string `abc\ndef`. Note that this effect differs from the use of the backslash (\) to continue a line. A new-line character that follows a backslash is replaced with a space.

NMAKE ignores a caret that is not followed by any of the characters mentioned above, as in the following:

```
mno ^: def
```

In this case, NMAKE ignores the caret and treats the line as

```
mno : def
```

Carets that appear within quotation marks are not treated as escape characters.

7.3.1.1 Modifying Commands

Three different characters may be placed in front of a command to modify the command's effect. The character must be preceded by at least one space, and spaces may separate the character from the command. You may use more than one character to modify a single command. The characters are listed below:

Character	Action

Dash (-)	Turns off error checking for the command. If the dash is followed by a number, NMAKE halts only if the error level returned by the command is greater than the number. In the following example, if the program `flash` returned an error code NMAKE would not halt, but would continue to execute commands:
	`light.lst:light.txt`
	`-flash light.txt`

At sign(@)	Prevents NMAKE from displaying the command as it executes. In the example below, NMAKE does not display the ECHO command line:
	`sort.exe:sort.obj`
	`@ECHO sorting`
	The output of the ECHO command, however, appears as usual.

Exclamation point (!)	Causes the command to be executed for each dependent file if the command uses one of the special macros $? or $**. The $? macro refers to all dependent files that are out of date with respect to the target, while $** refers to all dependent files in the description block. (See Section 7.3.2 for more information on macros.) For example,
	`print: hop.asm skip.bas jump.c`
	`!print $** lpt1:`
	causes the following three commands to be generated:
	`print hop.asm lpt1:`
	`print skip.bas lpt1:`
	`print jump.c lpt1:`

7.3.1.2 Specifying a Target in Multiple Description Blocks

You can specify more than one description block for the same target by using two colons (::) as the separator instead of one. For example:

```
target.lib :: a.asm b.asm c.asm
   ML a.asm b.asm c.asm
   LIB target -+a.obj -+b.obj -+c.obj;
target.lib :: d.c e.c
   QCL /c d.c e.c
   LIB target -+d.obj -+e.obj;
```

These two description blocks both update the library named `target.lib`. If any of the assembly-language files have changed more recently than the library file, NMAKE executes the commands in the first block to assemble the source files and update the library. Similarly, if any of the C-language files have changed, NMAKE executes the second group of commands, which compile the C files and then update the library.

If you use a single colon in the above example, NMAKE issues an error message. It is legal, however, to use single colons if commands are listed in only one block. In this case, dependency lines are cumulative. For example,

```
target: jump.bas
target: up.c
   commands
```

is equivalent to

```
target: jump.bas up.c
   commands
```

7.3.2 Macros

Macros provide a convenient way to replace a string in the description file with another string. The text is automatically replaced each time NMAKE is invoked. This feature makes it easy to change text used throughout the description file without having to edit every line that uses the text.

Macros can be used in a variety of situations, including the following:

- To create a standard description file for several projects. The macro represents the file names used in commands. These file names are then defined when you run NMAKE. When you switch to a different project, changing the macro changes the file names NMAKE uses throughout the description file.
- To control the options that NMAKE passes to the compiler, assembler, or linker. When you use a macro to specify the options, you can quickly change the options used throughout the description file in one easy step.

7.3.2.1 Macro Definitions

A macro definition uses the following form:

macroname = *string*

The *macroname* may be any combination of alphanumeric characters and the underscore (_) character. The *string* may be any valid string.

You can define macros on the NMAKE command line or in the description file. Because of the way DOS parses command lines, the rules for the two methods are slightly different.

Defining macros in description files

In NMAKE description files, define each macro on a separate line. The first character of the macro name must be the first character on the line. NMAKE ignores spaces following *macroname* or preceding *string*. The *string* may be a null string and may contain embedded spaces. Do not enclose *string* in quotation marks; NMAKE will consider them part of the string.

Defining macros on the NMAKE command line

On the command line, no spaces may surround the equal sign. Spaces cause DOS to treat *macroname* and *string* as separate tokens. Strings that contain embedded spaces must be enclosed in double quotation marks. Alternatively, you can enclose the entire macro definition—*macroname* and *string*—in quotation marks. The *string* may be a null string.

After you have defined a macro, use the following to include it in a dependency line or command:

$(*macroname*)

The parentheses are not required if *macroname* is only one character long. The *macroname* is converted to uppercase letters. If you want to use a dollar sign ($) in the file but do not want to invoke a macro, enter two dollar signs ($$), or use the caret (^) as an escape character preceding the dollar sign.

When NMAKE runs, it replaces all occurrences of $(*macroname*) with *string*. If the macro is undefined, that is, if its name does not appear to the left of an equal sign in the file or on the NMAKE command line, NMAKE treats it as a null string. Once a macro is defined, the only way to cancel its definition is to use the !UNDEF directive (see Section 7.3.4).

Example

Assume the following text is in a file named MAKEFILE:

```
program = flash
c = LINK
options =

$(program).exe : $(program).obj
   $c  $(options)  $(program).obj;
```

When you invoke NMAKE, it interprets the description block as the following:

```
flash.exe : flash.obj
   LINK    flash.obj;
```

7.3.2.2 Macro Substitutions

Just as macros allow you to substitute text in a description file, you can also substitute text within a macro itself. Use the following form:

$(*macroname*:*string1* = *string2*)

Every occurrence of *string1* is replaced by *string2* in the macro *macroname*. Spaces between the colon and *string1* are considered part of *string1*. Any spaces following *string1* or preceding *string2* are ignored. If *string2* is a null string, all occurrences of *string1* are deleted from the *macroname* macro.

Example

```
SRCS = prog.c sub1.c sub2.c
prog.exe : $(SRCS:.c=.obj)
        LINK  $**;

DUP : $(SRCS)
        !COPY $** c:\backup
```

Note that the special macro $** stands for the names of all the dependent files (see Section 7.3.2.3). If the description file above is invoked with a command line that specifies both targets, NMAKE will execute the following commands:

```
LINK prog.obj sub1.obj sub2.obj;

COPY prog.c c:\backup
COPY sub1.c c:\backup
COPY sub2.c c:\backup
```

The macro substitution does not alter the definition of the macro `SRCS`, but simply replaces the listed characters. When NMAKE builds the target `prog.exe`, it picks up the definition for the special macro $** (that is, the list of dependents) from the dependency line, which specifies the macro substitution in `SRCS`. The same is true for the second target, `DUP`. In this case, however, no macro substitution is requested, so `SRCS` retains its original value, and $** represents the names of the C source files.

7.3.2.3 Special Macros

Several macros have special meaning. These macros are listed below with their values:

Macro	Value
$*	The target name with the extension deleted.
$@	The full name of the current target.
$**	The complete list of dependent files.
$<	The dependent file that is out of date with respect to the target (evaluated only for inference rules).
$?	The list of dependents that are out of date with respect to the target.
$$@	The target NMAKE is currently evaluating. This is a dynamic dependency parameter that can be used only in dependency lines. See "Examples," below, for a typical use of this macro.
$(CC)	The command to invoke the C compiler. By default, NMAKE predefines this macro as `CC = cl`, which invokes the Microsoft C Optimizing Compiler. To redefine the macro to invoke the QuickC compiler, use the following: `CC = qcl` You might want to place the above definition in your TOOLS.INI file to avoid having to redefine it for each description file.
$(AS)	The command to invoke the Microsoft Macro Assembler. NMAKE predefines this macro as `AS = masm`.
$(MAKE)	The name with which the NMAKE utility was invoked. This macro is used to invoke NMAKE recursively. It causes the line on which it appears to be executed even if the /N option is on. You may redefine this macro if you want to execute another program; however, NMAKE returns a warning message.
$(MAKEFLAGS)	The NMAKE options currently in effect. If you invoke NMAKE recursively, you should use the command: `$(MAKE) $(MAKEFLAGS)`. You cannot redefine this macro.

Characters that modify special macros

You can append characters to any of the first six macros in the above list to modify its meaning. Appending a D specifies the directory part of the file name only, an F specifies the file name, a B specifies just the base name, and an R specifies the complete file name without the extension. If you add one of these characters, you must enclose the macro name in parentheses. (The special macros $$@ and $** are the only exceptions to the rule that macro names more than one character long must be enclosed in parentheses.)

For example, assume that $@ has the value `C:\SOURCE\PROG\SORT.OBJ`. The list below shows the effect the special characters have when combined with $@:

Macro	Value
$(@D)	`C:\SOURCE\PROG`
$(@F)	`SORT.OBJ`
$(@B)	`SORT`
$(@R)	`C:\SOURCE\PROG\SORT`

Examples

```
trig.lib : sin.obj cos.obj arctan.obj
        !LIB trig.lib -+$?;
```

In the example above, the macro $? represents the names of all dependents that are more recent than the target. The exclamation point causes NMAKE to execute the LIB command once for each dependent in the list. As a result of this description, the LIB command is executed up to three times, each time replacing a module with a newer version.

```
# Include files depend on versions in current directory
DIR=c:\include
$(DIR)\globals.h : globals.h
        COPY globals.h $@
$(DIR)\types.h : types.h
        COPY types.h $@
$(DIR)\macros.h : macros.h
        COPY macros.h $@
```

This example shows the use of NMAKE to update a group of include files. In the description file above, each of the files `globals.h`, `types.h`, and `macros.h` in the directory `c:\include` depends on its counterpart in the current directory. If one of the include files is out of date, NMAKE replaces it with the file of the same name from the current directory.

The following description file, which uses the special macro $$@, is equivalent:

```
# Include files depend on versions in current directory
DIR=c:\include
$(DIR)\globals.h $(DIR)\types.h $(DIR)\macros.h: $$(@F)
        !COPY $? $@
```

In this example, the special macro $$(@F) signifies the file name (without the directory) of the current target.

When NMAKE executes the description, it evaluates the three targets, one at a time, with respect to their dependents. Thus, NMAKE first checks whether `c:\include\globals.h` is out of date compared with `globals.h` in the current directory. If so, it executes the command to copy the dependent file `globals.h` to the target. NMAKE repeats the procedure for the other two targets. Note that in the command line, the macro $? refers to the dependent for this target. The macro $@ means the full name of the target.

7.3.2.4 Precedence of Macro Definitions

If the same macro is defined in more than one place, the rule with the highest priority is used. The priority from highest to lowest is as follows:

1. Definitions on the command line
2. Definitions in the description file or in an include file
3. Definitions by an environment variable
4. Definitions in the TOOLS.INI file
5. Predefined macros such as CC and AS

If NMAKE is invoked with the /E option, which causes environment variables to override macro definitions, macros defined by environment variables take precedence over those defined in a description file.

7.3.3 Inference Rules

Inference rules are templates that NMAKE uses to generate files with a given extension. When NMAKE encounters a description block with no commands, it looks for an inference rule that specifies how to create the target from the dependent files, given the two file extensions. Similarly, if a dependent file does not exist, NMAKE looks for an inference rule that specifies how to create the dependent from another file with the same base name.

The use of inference rules eliminates the need to put the same commands in several description blocks. For example, you can use inference rules to specify a single QCL command that changes any C source file (which has an extension of .C) to an object file (which has an extension of .OBJ).

Inference rules have the following form:

.fromext.toext:
    command
    [[*command*]]
    .
    .
    .

In this format, *command* specifies one of the commands involved in converting a file with the extension *fromext* to a file with the extension *toext*. Using the earlier example of converting C source files to object files, the inference rule looks as follows:

```
.C.OBJ:
   QCL -c $<;
```

The special macro $< represents the name of a dependent that is out of date relative to the target.

Path specifications

You can specify a single path for each of the extensions, using the following form:

{frompath}.fromext{topath}.toext
    commands

NMAKE takes the files with the *fromext* extension it finds in the directory specified by *frompath* and uses *commands* to create files with the *toext* extension in the directory specified by *topath.*

If NMAKE finds a description block without commands, it looks for an inference rule that matches both extensions. NMAKE searches for inference rules in the following order:

1. In the current description file.
2. In the tools-initialization file, TOOLS.INI. NMAKE first looks for the TOOLS.INI file in the current working directory and then in the directory indicated by the INIT environment variable. If it finds the file, NMAKE looks for the inference rules following the line that begins with the tag [nmake].

NOTE *NMAKE applies an inference rule only if the base name of the file it is trying to create matches the base name of a file that already exists.*

In effect, this means that inference rules are useful only when there is a one-to-one correspondence between the files with the "from" extension and the files with the "to" extension. You cannot, for example, define an inference rule that inserts a number of modules into a library.

Predefined inference rules

NMAKE uses three predefined inference rules, summarized in Table 7.1. Note that these rules use the macro CC, which invokes the Microsoft C Optimizing Compiler by default. If you plan to rely on inference rules to build your targets, you should redefine CC to invoke the QuickC compiler, as shown in Section 7.3.2.3.

Table 7.1 Predefined Inference Rules

Inference Rule	Command	Default Action
.c.obj	$(CC) $(CFLAGS) /c $*.c	CL /c $*.c
.c.exe	$(CC) $(CFLAGS) $*.c	CL $*.c
.asm.obj	$(AS) $(AFLAGS) $*;	masm $*;

Example

```
.OBJ.EXE:
   LINK $<;

EXAMPLE1.EXE: EXAMPLE1.OBJ

EXAMPLE2.EXE: EXAMPLE2.OBJ
   LINK /CO EXAMPLE2,,,LIBV3.LIB
```

In the sample description file above, the first line defines an inference rule that executes the LINK command on the second line to create an executable file whenever a change is made in the corresponding object file. The file name in the inference rule is specified with the special macro $< so that the rule applies to any .OBJ file that has an out-of-date executable file.

When NMAKE does not find any commands in the first description block, it checks for a rule that may apply and finds the rule defined on the first two lines of the description file. NMAKE applies the rule, replacing the $< macro with `EXAMPLE1.OBJ` when it executes the command, so that the LINK command becomes

```
LINK EXAMPLE1.OBJ;
```

NMAKE does not search for an inference rule when examining the second description block because a command is explicitly given.

7.3.4 Directives

Using directives, you can construct description files that are similar to batch files. NMAKE provides directives that specify conditional execution of commands, display error messages, include the contents of other files, and turn on or off some of NMAKE's options.

Each directive begins with an exclamation point (!) in the first column of the description file. Spaces can be placed between the exclamation point and the directive keyword. The list below describes the directives:

Directive	Description
!IF *constantexpression*	Executes the statements between the !IF keyword and the next !ELSE or !ENDIF directive if *constantexpression* evaluates to a nonzero value.
!ELSE	Executes the statements between the !ELSE and !ENDIF directives if the statements preceding the !ELSE directive were not executed.
!ENDIF	Marks the end of the !IF, !IFDEF, or !IFNDEF block of statements.
!IFDEF *macroname*	Executes the statements between the !IFDEF keyword and the next !ELSE or !ENDIF directive if *macroname* is defined in the description file. NMAKE considers a macro with a null value to be defined.
!IFNDEF *macroname*	Executes the statements between the !IFNDEF keyword and the next !ELSE or !ENDIF directive if *macroname* is not defined in the description file.
!UNDEF *macroname*	Marks *macroname* as being undefined in NMAKE's symbol table.
!ERROR *text*	Causes *text* to be printed and then stops execution.
!INCLUDE *filename*	Reads and evaluates the file *filename* before continuing with the current description file. If *filename* is enclosed by angle brackets (<>), NMAKE searches for the file in the directories specified by the INCLUDE macro; otherwise it looks in the current directory only. The INCLUDE macro is initially set to the value of the INCLUDE environment variable.

!CMDSWITCHES: {+|–}*opt*...	Turns on or off one of four NMAKE options: /D, /I, /N, and /S. If no options are specified, the options are reset to the way they were when NMAKE was started. Turn an option on by preceding it with a plus sign (+), or turn it off by preceding it with a minus sign (–). Using this directive updates the MAKEFLAGS macro.

The *constantexpression* used with the **!IF** directive may consist of integer constants, string constants, or program invocations. Integer constants can use the C unary operators for numerical negation (–), one's complement (~), and logical negation (!). They may also use any of the C binary operators listed below:

Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
%	Modulus
&	Bitwise AND
|	Bitwise OR
^	Bitwise XOR
&&	Logical AND
||	Logical OR
<<	Left shift
>>	Right shift
==	Equality
!=	Inequality
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

You can use parentheses to group expressions. Values are assumed to be decimal values unless specified with a leading 0 (octal) or leading 0x (hexadecimal). Use the equality (==) operator to compare two strings for equality or the inequality (!=) operator to compare for inequality. Strings are enclosed by quotes. Program invocations must be in square brackets ([ ]).

Example

```
!INCLUDE <infrules.txt>
!CMDSWITCHES +D
winner.exe:winner.obj
!IFDEF debug
!  IF "$(debug)"=="y"
     LINK /CO winner.obj;
!  ELSE
     LINK winner.obj;
!  ENDIF
!ELSE
!  ERROR Macro named debug is not defined.
!ENDIF
```

The !INCLUDE directive causes the file `INFRULES.TXT` to be read and evaluated as if it were a part of the description file. The !CMDSWITCHES directive turns on the /D option, which displays the dates of the files as they are checked. If `winner.exe` is out of date with respect to `winner.obj`, the !IFDEF directive checks to see if the macro `debug` is defined. If it is defined, the !IF directive checks to see if it is set to `y`. If it is, then the linker is invoked with the /CO option; otherwise it is invoked without. If the `debug` macro is not defined, the !ERROR directive prints the message and NMAKE stops executing.

7.3.5 Pseudotargets

A "pseudotarget" is a target that is not a file but instead is a name that serves as a "handle" for building a group of files or executing a group of commands. In the following example, `UPDATE` is a pseudotarget:

```
UPDATE: *.*
        !copy $** a:\product
```

When NMAKE evaluates a pseudotarget, it always considers the dependents out of date. In the description above, NMAKE copies each of the dependent files to the specified drive and directory.

The NMAKE utility includes four predefined pseudotargets that provide special rules within a description file. The list below describes these pseudotargets:

Pseudotarget	Action
.SILENT:	Does not display lines as they are executed. Same effect as invoking NMAKE with the /S option.
.IGNORE:	Ignores exit codes returned by programs called from the description file. Same effect as invoking NMAKE with the /I option.

.SUFFIXES:*list*

Lists file suffixes for NMAKE to try if it needs to build a target file for which no dependents are specified. NMAKE searches the current directory for a file with the same name as the target file and a suffix from the list. If NMAKE finds such a file, and if an inference rule applies to the file, then NMAKE treats the file as a dependent of the target. The order of the suffixes in the list defines the order in which NMAKE searches for the files. The list is predefined as follows:

```
.SUFFIXES: .obj .exe .c .asm
```

To add suffixes to the list, specify .SUFFIXES: followed by the new suffixes. To clear the list, specify the following:

```
.SUFFIXES:
```

.PRECIOUS: *target...*

Tells NMAKE not to delete *target* if the commands that build it are quit or interrupted. Using this pseudotarget overrides the NMAKE default. By default, NMAKE deletes the target if it cannot be sure the target was built successfully. For example:

```
.PRECIOUS: tools.lib
tools.lib : a2z.obj z2a.obj
    .
    .
    .
```

If the commands (not shown here) to build `tools.lib` are interrupted, leaving an incomplete file, NMAKE does not delete the partially built `tools.lib` because it is listed with .PRECIOUS.

Note, however, that .PRECIOUS is useful only in limited circumstances. Most professional development tools, including those provided by Microsoft, have their own interrupt handlers and "clean up" when errors occur.

7.4 Response-File Generation

At times, you may need to issue a command in the description file that has a list of arguments that exceeds the DOS limit of 128 characters. Just as NMAKE supports the use of response files, it can also generate response files for use with other programs.

The syntax for creating a response file is

target : *dependents*
  command @<< [[*filename*]]
response-file-text
<<

All of the text between the two sets of angle brackets (<<) is placed in a response file. The second pair of angle brackets must be at the beginning of the line, with no preceding white-space characters. Note that the at sign (@) is not part of the NMAKE syntax but is the typical response-file character for utilities such as LINK and LIB.

To name the response file, specify a *filename* immediately after the first pair of angle brackets, with no intervening spaces. If you do not specify a file name, NMAKE gives the response file a unique name in the directory specified by the TMP environment variable if the variable is defined; if the TMP variable is not defined, NMAKE creates the response file in the current directory.

Example

```
math.lib : add.obj sub.obj mul.obj div.obj
  LIB @<<
math.lib
-+add.obj-+sub.obj-+mul.obj-+div.obj
listing

<<
```

The above example creates a response file and uses it to invoke the Microsoft Library Manager (LIB). The response file specifies which library to use, the commands to execute, and the listing file to produce. The response file contains the following:

```
math.lib
-+add.obj-+sub.obj-+mul.obj-+div.obj
listing
```

7.5 Differences between NMAKE and MAKE

NMAKE differs from MAKE in the following ways:

- It accepts command-line arguments from a file.
- It provides more command-line options.
- It does not evaluate targets sequentially, as MAKE does. Instead, it updates the targets specified on the command line, regardless of where they appear in the description file. If no targets are specified, NMAKE updates the first target in the file.
- It provides more special macros.
- It permits substitutions within macros.
- It supports directives placed in the description file.
- It allows you to specify include files in the description file.

MAKE assumed that all targets in the description file would be built. Because NMAKE builds the first target in the file unless you specify otherwise, you may need to change your old description files to work with the new utility.

Description files written for use with MAKE typically list a series of subordinate targets followed by a higher-level target that depends on the subordinates. As MAKE executed, it would build the targets sequentially, creating the highest-level target at the end.

The easiest way to convert these description files is to create a new description block at the top of the file. Give this block a pseudotarget named ALL and set its dependents to all of the other targets in the file. When NMAKE executes the description, it will assume you want to build the target ALL and consequently will build all targets in the file.

Alternatively, if your description file already contains a block that builds a single, top-level target, you can simply make that block the first in the file.

Example

```
one.obj: one.c

two.obj: two.c

three.obj: three.c

prog1.exe: one.obj two.obj three.obj
        link one two three, prog1;

x.obj: x.c

y.obj: y.c

z.obj: z.c

xyz.exe: x.obj y.obj z.obj
        link x y z, xyz;
```

Assume the above is an old MAKE description file named MAKEFILE. Note that it builds two top-level targets, `prog1.exe` and `xyz.exe`. To use this file with the new NMAKE, insert the following as the first line in the file:

```
ALL : prog1.exe xyz.exe
```

With the addition of this line, `ALL` becomes the first target in the file. Since NMAKE, by default, builds the first target, you can invoke NMAKE with

```
NMAKE
```

and it will build both `prog1.exe` and `xyz.exe`.

7.6 Interchanging NMAKE and QuickC .MAK Files

If you create a program list for a program within the QuickC environment, QuickC creates a description file for the program. The description file has the same base name as the program, with the extension .MAK. The file is used within the QuickC environment to rebuild the program. The QuickC environment supports a subset of the NMAKE features for use in the .MAK files. Generally speaking, NMAKE can execute any QuickC .MAK file, but QuickC cannot execute all NMAKE description files. This section summarizes the differences between the two.

7.6.1 Syntax Rules

The QuickC environment limits lines to 100 characters. It does not allow the backslash (\) as a line-continuation character.

7.6.2 Order of Targets

QuickC does not use inference rules in the same way as NMAKE. The only inference rule it applies is one that creates .EXE files from .OBJ files. It does not infer the creation of .OBJ files from .C or .ASM sources as NMAKE does.

7.6.3 Macro Definitions

You can change macro definitions in QuickC .MAK files, but not macro names. For instance, you might need to change a list of options; modify the CFLAGS, AFLAGS, or LFLAGS macro to build a target for debugging; or edit the list of object files or libraries used in the LINK step.

7.6.4 Dependency Lines

NMAKE applies three predefined inference rules for creating targets and dependents, and allows you to define others, as described in Section 7.3.3. The QuickC environment, however, assumes that all files exist, except for .EXE files it can create from .OBJ files. To use a NMAKE description file with QuickC, you must explicitly create all .OBJ files from their .C and .ASM sources.

CHAPTER 8

HELPMAKE

The Microsoft Help-File-Creation Utility (HELPMAKE) allows you to make your own help files for use with Microsoft products. HELPMAKE also allows you to customize the help files supplied with Microsoft language products.

HELPMAKE translates help text files into a help data base accessible from within the following:

- Microsoft Editor
- QuickC
- Microsoft® QuickBASIC
- OS/2 Programmer's Toolkit QuickHelp Utility

To use HELPMAKE, you specify the name of a help text file formatted in one of several simple styles and the amount by which to compress the file. HELPMAKE can also decompress a help data base to its original text format.

8.1 Structure and Contents of a Help Data Base

HELPMAKE creates a help data base from one or more input files that contain information specially formatted for the help system. This section defines some of the terms involved in formatting and outlines the types of files HELPMAKE can take as input.

8.1.1 What's in a Help File?

Contexts and topic text

If you have used the QuickC Advisor, you probably have a good idea what a help file looks like. As you might expect, each file starts with a subject and some information about the subject, then lists another subject and some information about it, then another, and so on. In HELPMAKE terminology, the subjects are called "contexts" and the information is called "topic text." Whenever someone asks for help on the **open** function, the Advisor looks for the context "open" and displays its topic text. (The name of every function in the C run-time library is a context throughout the QuickC Advisor.)

Whether a context is one or several words depends on the application. QuickC, for example, considers spaces to be delimiters, so contexts in QuickC help files are limited to a single word. Other applications, such as the Microsoft Editor, can handle contexts that span several words. Either way, the application simply hands the context to an internal "help engine," which searches the data base for information. All Microsoft products that provide on-line help use the same help engine.

Often, especially with library routines, the same information applies to more than one subject. For example, the string-to-number functions **strtod**, **strtol**, and **stroul** share the same help text. The help file lists all three function names as contexts for one block of topic text. The converse, however, is not true. You cannot specify different blocks of topic text, in different places in the help file, to describe a single subject.

Cross-references

To make it easier for users to navigate through a help data base, you can put cross-references in your help text. Cross-references bring up information on related topics, including header files and code examples. The help for the **open** function, for example, references the **access** function and the ASCII header file FCNTL.H. Cross-references can point to other contexts in the same help data base, to contexts in other help data bases, or to ASCII files outside the data base.

Help files can have two kinds of cross-references:

- Implicit cross-references
- Explicit cross-references, or "hyperlinks"

Implicit cross-references

The word "open" is an "implicit cross-reference" throughout QuickC help because it is the name of a function. If a user selects the word "open" anywhere in QuickC help, the help system displays information on the **open** function. Cross-references like this are called "implicit cross-references" because they are implicit in the help file and require no special coding. Anywhere a context appears, the help system makes an implicit cross-reference to its topic text.

Hyperlinks Explicit cross-references, also called "hyperlinks," are tied to a word or phrase at a specific location in the help file. You set up explicit cross-references when you write the help text. For example, to cause one instance of the word "formatting" to bring up help on the **printf** function, you would create an explicit cross-reference from the word "formatting" to the context "printf." Anywhere else in the file, "formatting" would have no special significance, but at that one position, it would reference the help for **printf**.

Formatting flags Help text can also include formatting flags to control the appearance of the text on the screen. Using these flags, you can make certain words appear in boldface, others underlined, and so forth, depending on the graphics capabilities of the user's computer. In QC.HLP, the help data base Microsoft supplies for QuickC, cross-references appear underlined when displayed on a monochrome monitor. On a color monitor, they appear highlighted instead. Other applications may represent cross-references differently; for example, in italics or in color.

8.1.2 Help File Formats

You can create help files in any of three formats:

- QuickHelp format
- Rich Text Format (RTF)
- Minimally formatted ASCII

In addition, you can reference unformatted ASCII files, such as include files, from within a help data base.

QuickHelp QuickHelp format is the default and is the format in which HELPMAKE writes files it decodes from existing help data bases. Use any text editor to create a QuickHelp-format help text file. QuickHelp format also lends itself to a relatively easy automated translation from other document formats.

QuickHelp files can contain all the various cross-references and formatting attributes. Typically, you would use QuickHelp format for any changes you want to make to the standard help data base. Most of the examples in this chapter are in QuickHelp format.

RTF Rich Text Format (RTF) is a Microsoft word-processing format that many other word processors also support. You can create RTF help text with any word processor capable of generating RTF output. You may also use any utility program that takes word-processor output and produces an RTF file.

Use RTF when you want to transfer help files from one application to another while retaining formatting information. You can format RTF files directly with

the word processing program and need not edit them to insert any special commands or tags. Like QuickHelp files, RTF files can contain formatting attributes and cross-references.

Minimally formatted ASCII Minimally formatted ASCII files simply define contexts and their topic text. These files cannot contain cross-references or screen-formatting commands.

Unformatted ASCII Unformatted ASCII files are exactly what their name implies: regular ASCII files with no special formatting commands, context definitions, or special information whatsoever. An unformatted ASCII file does not become part of the help data base. Instead, only its name is used as the object of a cross-reference. The standard C header (include) files are unformatted ASCII files used for cross-references by the help system for the C run-time library. Unformatted ASCII files are also useful for program examples.

8.2 Invoking HELPMAKE

HELPMAKE is a general support program to encode or decode help files. Encoding is the process of converting a text file into a compressed help data base. Decoding reverses the process: it converts a help data base into a text file. The utility can decode any Microsoft help data base file to a QuickHelp formatted text file for editing. It can also encode an RTF (Rich Text Format), QuickHelp, or minimally formatted ASCII text file into help-data-base format.

HELPMAKE is required to create and modify Microsoft-compatible help data bases. It is not required, however, merely to access data bases supplied with Microsoft language products.

The HELPMAKE command-line syntax is as follows:

HELPMAKE [[*options*]] { /E*n* | /D } { *sourcefiles* }

The *options* modify the action of HELPMAKE. They are described in Section 8.3.

Either the /E (encode) or the /D (decode) option must be supplied. When encoding (/E) to create a help data base, you must use the /O option to specify the file name of the data base.

The *sourcefile* field is required. It specifies the input file for HELPMAKE. If you use the /D (decode) option, *sourcefile* may be one or more help-data-base files (such as QC.HLP). HELPMAKE decodes the data-base files into a single text file. If you use the /E (encode) option, *sourcefile* may be one or more help text files (such as QC.SRC). Separate file names with a space. Standard wildcard characters may also be used.

Example

```
HELPMAKE /V /E /Omy.hlp my.txt > my.log
```

This example invokes HELPMAKE with the /V, /E, and /O options (see Section 8.3.1). HELPMAKE reads input from the text file `my.txt` and writes the compressed help data base in the file `my.hlp`. The /E option causes maximum compression. Note that the DOS redirection symbol (>) sends a log of HELPMAKE activity to the file `my.log`. You may find it helpful to redirect the log file because, in its more verbose modes (given by /V), HELPMAKE may generate a lengthy log.

```
HELPMAKE /V /D /Omy.src my.hlp > my.log
```

This example invokes HELPMAKE to decode the help data base `my.hlp` into the text file `my.src`, given with the /O option. Once again, the /V option results in verbose output, and the output is directed to the log file `my.log`. Section 8.3.2 describes additional options for decoding.

8.3 HELPMAKE Options

HELPMAKE accepts a number of command-line options, which are listed in the sections that follow. You may specify options in uppercase or lowercase letters, and precede them with either a forward slash (/) or a dash (–). For example, -L, /L, -l, and /l all represent the same option.

Most options apply only to encoding; others apply only to decoding; and a few apply to both. Section 8.3.1 describes all the options that apply to encoding, and Section 8.3.2 describes all the options that apply to decoding.

8.3.1 Options for Encoding

When you encode a file—that is, when you build a help data base—you must specify the /E option. In addition, you may supply various other options that control the way HELPMAKE encodes the data base. All the options that apply when encoding are listed below:

Option	Action
/A*c*	Specifies *c* as an application-specific control character for the help-data-base file. The character marks a line that contains special information for internal use by the application. For example, QuickC uses the colon (:).
/C	Indicates that the context strings for this help file are case sensitive. At run time, all searches for help topics are case sensitive if the help data base was built with the /C option in effect.

/E[[*n*]]

Creates ("encodes") a help data base from a specified text file. The optional *n* indicates the amount of compression to take place. If *n* is omitted, HELPMAKE compresses the file as much as possible, thereby reducing the size of the file by about 50 percent. The more compression requested, the longer HELPMAKE takes to create a data-base file. The value of *n* is a number in the range 0 – 15. It is the sum of successive powers of 2 representing various compression techniques, as listed below:

Value	Technique
0	No compression
1	Run-length compression
2	Keyword compression
4	Extended-keyword compression
8	Huffman compression

Add values to combine compression techniques. For example, use /E3 to get run-length and keyword compression. This is useful in the testing stages of help-data-base creation where you need to create the data base quickly and are not too concerned with size.

/H

Displays a summary of HELPMAKE syntax and exits without encoding or decoding any files.

/L

"Locks" the generated file so that it cannot be decoded by HELPMAKE at a later time.

/O*destfile*

Specifies *destfile* as the name of the help data base.

/S*n*

Specifies the type of input file, according to the following *n* values:

Option	File Type
/S1	RTF
/S2	QuickHelp (default)
/S3	Minimally formatted ASCII

/V[[*n*]]

Indicates the "verbosity" of diagnostic and informational output, depending on the value of *n*. Increasing the value adds more information to the output. If you omit this option or specify only /V, HELPMAKE gives you its most verbose output. The possible values of *n* are listed below:

Option	Effect
/V	Maximum diagnostic output
/V0	No diagnostic output and no banner
/V1	Prints only HELPMAKE banner
/V2	Prints pass names
/V3	Prints contexts on first pass
/V4	Prints contexts on each pass

	/V5	Prints any intermediate steps within each pass
	/V6	Prints statistics on help file and compression
/W*width*	Indicates the fixed width of the resulting help text in number of characters. The values of *width* may range from 11 to 255. If the /W option is omitted, the default is 76. When encoding RTF source (/S1), HELPMAKE automatically formats the text to *width*. When encoding QuickHelp (/S2) or minimally formatted ASCII (/S3) files, HELPMAKE truncates lines to this width.	

8.3.2 Options for Decoding

To decode a help data base into QuickHelp files, you must use the /D option. In addition, HELPMAKE accepts other options to control the decoding process. The list below shows all the options that are valid when decoding:

Option	Action	
/D[[*letter*]]	Decodes the input file into its component parts. If a destination file is not specified with the /O option, the help file is decoded to **stdout**. HELPMAKE decodes the file differently depending on the letter specified, as shown below:	
	Letter	**Effect**
	/D	Fully decodes the help data base, leaving all cross-references and formatting information intact.
	/DS	"Decode split." Splits the concatenated, compressed help data base into its components using their original names. If the data base was created without concatenation (the default), HELPMAKE simply copies it to a file with its original name. No decompression occurs.
	/DU	"Decode unformatted." Decompresses the data base and removes all screen formatting and cross-references. The output can still be used later for input and recompression, but all screen formatting and cross-references are lost.
/H	Displays a summary of HELPMAKE syntax and exits without encoding or decoding any files.	

/O[*destfile*] Specifies *destfile* for the decoded output from HELPMAKE. If *destfile* is omitted, the help data base is decoded to **stdout.** HELPMAKE always decodes help-data-base files into QuickHelp format.

/V[*n*] Indicates the "verbosity" of diagnostic and informational output depending on the value of *n*. The possible values are listed below. If you omit this option or specify only /V, HELPMAKE gives you its most verbose output.

Option	Effect
/V	Maximum diagnostic output
/V0	No diagnostic output and no banner
/V1	Prints only the HELPMAKE banner
/V2	Prints pass names
/V3	Prints contexts on first pass

8.4 Creating a Help Data Base

You can create a Microsoft-compatible help data base by either of two methods.

The first method is to decompress an existing help data base, modify the resulting help text file, and recompress the help text file to form a new data base. Note that, if you decompress the Microsoft help-data-base file QC.HLP, the resulting text file occupies about 800K on disk.

The second and simpler method is to append a new help data base onto an existing help data base. This method involves the following steps:

1. Create a help text file in QuickHelp format, RTF, or minimally formatted ASCII. For your convenience in experimenting with HELPMAKE, the file SAMPLE.TXT (distributed with QuickC) contains a short help text file in QuickHelp format.
2. Use HELPMAKE to create a help-data-base file. The example below invokes HELPMAKE, using SAMPLE.TXT as the input file and producing a help-data-base file named `sample.hlp`:

```
HELPMAKE /V /E /Osample.hlp sample.txt > sample.log
```

3. Make a back-up copy of the existing data-base file (for safety's sake).

4. Append the new help-data-base file onto the existing help data base. The example below concatenates the new data base `sample.hlp` onto the end of the QC.HLP data base:

   ```
   COPY qc.hlp /b + sample.hlp /b
   ```

5. Test the data base. The `sample.hlp` data base contains the context `sample`. If you type the word "sample" in the QuickC environment and request help on it, the help window will display the text associated with the context `sample`.

8.5 Help Text Conventions

Using common structure and conventions ensures that help files for one application will make sense when viewed using another. This section outlines organizational conventions used in help data bases supplied by Microsoft. You should follow the same conventions to create Microsoft-compatible help files.

8.5.1 The Help Text File

The help-retrieval facility that is built into Microsoft products is simply a data-retrieval tool. It imposes no restrictions on the content and format of the help text. The HELPMAKE utility and the display routines built into Microsoft language environments, however, make certain assumptions about the format of help text. This section provides some guidelines for creating help text files that are compatible with those assumptions.

In all three help text formats, the help text source file is a sequence of topics, each preceded by one or more unique context definitions.

Contexts in QuickHelp

In QuickHelp format, each topic begins with one or more context definitions that define the context strings that map to the topic text. Subsequent lines up to the next context definition constitute the topic text, as shown below:

```
.context strtod
.context strtol
.context stroul
     .
     .  topic text describing the string-to-number functions
     .
.context strtok
     .
     .  topic text describing strtok function
     .
```

Contexts in RTF

In RTF, each context definition must be in a paragraph of its own, beginning with the help delimiter (>>). Subsequent paragraphs up to the next context definition constitute the topic text, as shown below:

```
{rtf0
>>strtod \par
>>strtol \par
>>stroul \par
      .
      .  topic text describing the string-to-number functions
      .
>>strtok \par
      .
      .  topic text describing strtok function
      .
}
```

Note that RTF uses curly braces ({}) for nesting.

Contexts in minimally formatted ASCII

In minimally formatted ASCII, each context definition must be on a separate line, and each must begin with the help delimiter (>>). As in RTF and QuickHelp files, subsequent lines up to the next context definition constitute the topic text. The following is the same help file as the previous two, but in minimally formatted ASCII:

```
>>strtod
>>strtol
>>stroul
      .
      .  topic text describing the string-to-number functions
      .
>> strtok
      .
      .  topic text describing strtok function
      .
```

8.5.2 Context Conventions

Certain contexts are defined by convention across the help data bases for all Microsoft languages. If you decompress any of the help data base files that Microsoft supplies, you will see these contexts in the text output.

The contexts listed below are required and are present in all Microsoft help files. If you modify or replace the standard files, be sure to retain these definitions.

Context	Description
h.default	The default help screen, typically displayed when the user presses F1 at the "top level" in most applications. The contents are generally devoted to information on using help.
h.notfound	The help text that is displayed when the help system cannot find information on the requested context. The text could be an index of contexts, a topical list, or general information on using help.
h.pg1	The help text that is logically first in the file. This is used by some applications in response to a "go to the beginning" request made within the help window.
h.pg$	The help text that is logically last in the file. This is used by some applications in response to a "go to the end" request made within the help window.

Note that each of the contexts above begins with **h.** Microsoft help systems consider context strings beginning with *x.*, where *x* is a specified character prefix, as "internal" or "constructed" help contexts. Except for the contexts listed above, these apply to menu items, error numbers, and so forth; in general, you do not need to insert these in your help files. The following character prefixes denote internal help contexts:

Character	Description
h.	Help item. Prefixes miscellaneous help contexts that may be constructed or otherwise hidden from the user. For example, the Contents menu item under the HELP menu item is a cross-reference to the context **h.contents.**
m.	Menu item. Contexts that relate to product menu items are defined by their accelerator keys. For example, the Exit selection on the FILE menu item is accessed by ALT+F+X, and is referenced in help by **m.fx**.
e.	Error number. If a product supports the uniform error numbering scheme used by Microsoft languages, it references the help for each error by prefixing the error number with **e.**. For example, the context **e.c1234** refers to the C compiler error message number C1234.

8.5.3 Hyperlinks and Cross-References

Explicit cross-references, or hyperlinks, in the help text file are marked with invisible text. A hyperlink comprises a word or phrase followed by invisible text that gives the context to which the hyperlink refers.

The keystroke that activates the hyperlink depends on the application. Consult the documentation for each product to find the specific keystroke needed.

When the user activates the hyperlink, the help system displays the topic named by the invisible text.

Examples

```
\bSee also:\p   \uExample\p\vopen.ex\v
```

In this example, the word `Example` is a hyperlink. It cross-references to `open.ex`. A mouse click or other form of selection with the cursor on any of the letters of `Example` brings up the help topic whose context is `open.ex`. On the user's screen, this line appears as follows:

```
See also:   Example
```

On a monochrome monitor, `See also:` is in boldface and `Example` is underlined. On a color monitor, they appear in different colors, depending on the user's default color selection.

```
\bSee also:\p   \uExample\p\vprintf.ex\v, fprintf, scanf,
sprintf, vfprintf, vprintf, vsprintf
            \aformatting table\vprintf.table\v
```

When a hyperlink needs to cross-reference more than one word, you must use an anchor, as in the example above. Anchored hyperlinks must fit on a single line. In this case, the hyperlink consists of the phrase `formatting table`, which references the context `printf.table`. The \v flag makes the name `printf.table` invisible; it does not appear on the screen when the help is displayed.

8.5.4 Formatting Cross-Reference Text

The invisible cross-reference text is formatted as one of the following:

Cross-Reference Text	Action
context_string	Causes the help topic associated with *context_string* to be displayed. For example, `exe_format` results in the display of the help topic associated with the context `exe_format`.
filename!	Causes the entire file *filename* to be treated as a single topic to be displayed. For example, `$INCLUDE:stdio.h!` would search the INCLUDE environment variable for the file STDIO.H and display it as a single help topic.

filename!context_string	Works same way as *context_string* above, except that only the help file *filename* is searched for the context. If the file is not already open, the help system finds it (by searching either the current path or an explicit environment variable), and opens it. For example, `$BIN:readme.doc!patches` would search for `readme.doc` in the BIN environment variable, and bring up the topic associated with `patches`.

8.5.4.1 Local Contexts

Context strings that begin with an "at" sign (@) are defined as "local" and have no implicit cross-references. They are used in cross-references instead of the context string that would otherwise be generated.

When you use a local context, HELPMAKE does not generate a context string that can be used from elsewhere in the help file. Instead, it embeds a cross-reference that has meaning only within the current context. An example of this usage is shown below:

```
.context normal
This is a normal topic, accessible by the context string
"normal."
[button\v@local\v] is a cross-reference to the following
topic.

.context @local
This topic can be reached only if the user browses
sequentially through the file or uses the  cross-reference
in the previous topic.
```

In the example, the text `[button\v@local\v]` defines `local` as a local context. If the user selects the text `[button]`, or scrolls through the file, the help system displays the topic text that follows the context definition for `local`. Because `local` is defined with the "at" sign, it can be accessed only by a hyperlink within the help file or by sequentially browsing through the file.

8.5.4.2 Application-Specific Control Characters

The help data base supports application-specific characters that have special meaning for Microsoft language products. The application-specific character

may appear at the beginning of any line of help text. This special character is interpreted by the application. If the application does not support this character, it is ignored.

Within the data bases and applications provided with Microsoft languages, a colon is used as the control character, and the following colon commands are supported:

Command	Action
:l*n*	Indicates the default initial window size, in *n* lines, of the topic about to be displayed. Always the first line in the topic if present.
:n *text*	Defines *text* as the name (or title) to be displayed in place of the context string if the application help displays a title. Always the first line in the context unless :l is used, in which case :n appears on the line following the :l command.
:p	Indicates a screen break for environment help. The lines following :p are accessible only by using the PgDn command within the environment-help dialog box.

Example

```
.context open
:113
\bInclude:\p     <fcntl.h>, <io.h>, <sys\\types.h>,
<sys\stat.h>

\bPrototype:\p  int open(char *path, int flag[, int mode]);
       flag:  O_APPEND O_BINARY O_CREAT O_EXCL O_RDONLY
              O_RDWR    O_TEXT    O_TRUNC  O_WRONLY
              (may be joined by |)
       mode:  S_IWRITE  S_IREAD   S_IREAD | S_IWRITE

\bReturns:\p     a handle if successful, or -1 if not.
            errno:  EACCES, EEXIST, EMFILE, ENOENT

\bSee also:\p   \uExample\p\vopen.ex\v,
\uTemplate\p\vopen.tp\v, access, chmod, close,
             creat, dup, dup2, fopen, sopen, umask
```

This example shows the data-base entry from the C run-time library for the **open** routine. The `:l13` command on the second line of the file defines the default size of the initial window for the help text as 13 lines.

8.6 Formatting a Help Data Base

The text format of the data base may be any of three types. The list below briefly describes these types. The sections that follow describe each formatting type in detail.

An entire help system (such as the one supplied with Microsoft C, QuickC, or QuickBASIC) may use any combination of files formatted with different format types. With C, for example, the README.DOC information file is encoded as minimally formatted ASCII; and the help files for the C language and run-time library are encoded in the QuickHelp format. The data base also cross-references the header (include) files, which are unformatted ASCII files stored outside the data base.

The list below summarizes the three formats and their characteristics:

Type	Characteristics
QuickHelp	Uses dot commands and embedded formatting characters (the default formatting type expected by HELPMAKE); supports highlighting, color, and cross-references. This format must be compressed before using.
Minimally formatted ASCII	Uses a help delimiter (>>) to define help contexts; does not support highlighting, color, or cross-references. This format may be compressed, but compression is not required.
RTF	Uses a subset of standard RTF; supports highlighting, color, and cross-references. This format must be compressed before using.

8.6.1 QuickHelp Format

The QuickHelp format uses a dot command and embedded formatting flags to convey information to HELPMAKE.

8.6.1.1 The QuickHelp Context Command

QuickHelp supports a single dot command, the **.context** command. Additional dot commands may be added in a future release.

One or more **.context** commands precedes each topic in a QuickHelp file. Each **.context** command defines a context string for the topic text. You may define more than one context for a single topic, as long as you do not place any topic text between them.

Typical context commands are shown below. The first defines a context for the **#include** C preprocessor directive. The second set illustrates multiple contexts

for one block of topic text. In this case, the same topic text explains all of the string-to-number conversion routines in C.

```
.context #include
        .
        . description of #include goes here
        .
.context strtod
.context strtol
.context strtoul
        .
        . description of string-to-number functions goes here
        .
```

8.6.1.2 QuickHelp Formatting Flags

The QuickHelp format supports a number of formatting flags that are used to highlight parts of the help data base and to mark hyperlinks in the help text.

Each formatting flag consists of a backslash (\) followed by a character. The table below lists the formatting flags:

Formatting Flag	Action
\a	Anchors text for cross-references
\b, \B	Turns boldface on or off
\i, \I	Turns italics on or off
\p, \P	Turns off all attributes
\u, \U	Turns underlining on or off
\v, \V	Turns invisibility on or off (hides cross-references in text)
\\	Inserts a single backslash in text

On monochrome monitors, text labeled with the bold, italic, and underlining attributes appears in boldface, italics, or underlined, respectively. On color monitors, these attributes are translated by the application into suitable colors, depending on the user's default color selections.

The \b, \i, \u, and \v options are toggles, turning on and off their respective attributes. You may use several of these on the same text. Use the \p attribute to turn off all attributes. Use the \v attribute to hide cross-references and hyperlinks in the text.

HELPMAKE truncates the lines in QuickHelp files to the width specified with the /W option. (See Section 8.3.1 for a description of this option.) The formatting flags do not count toward the character-width limit. Lines that begin with an application-specific control character are truncated to 255 characters regardless of the width specification. See Section 8.5.4.2 for details on application-specific control characters.

Examples

```
\bReturns:\p    a handle if successful, or -1 if not.
           errno:  EACCES, EEXIST, EMFILE, ENOENT
```

In this example, the \b flag initiates boldface text for the word `Returns:` and the \p flag that follows the word reverts to plain text for the remainder of the line.

```
\bSee also:\p   \uExample\p\vopen.ex\v
```

In this example, the \b and \p flags surrounding `See also:` work in the same way as those surrounding `Returns:` in the previous example. The \u flag that precedes `Example` causes that word to be underlined on monitors that support underlining and highlighted on monitors that do not. The \p flag that follows `Example` turns off underlining for the text that follows. The \v flag causes the text `open.ex` to be invisible and defines a cross-reference, as described in the following section.

8.6.1.3 QuickHelp Cross-References

Help data bases contain two types of cross-references, as described in Section 8.1.1: implicit cross-references and explicit cross-references.

An implicit cross-reference is any word that appears both in the topic text and as a context in the help file. For example, any time you request help on the word "close," the help window will display help on the **close** function. You need not code implicit cross-references in your help text files.

Explicit cross-references ("hyperlinks") are words or phrases on the screen that are associated with a context. For example, the word "Example" in the initial help-screen area for any C function is an explicit cross-reference to the C program example for that function. You must insert formatting flags in your help text files to mark explicit cross-references.

If the hyperlink consists of a single word, you can use invisible text to flag it in the source file. The \v formatting flag creates invisible text, as follows:

hyperlink\v*context*\v

Specify the first \v flag immediately following the word you want to use as the hyperlink. Following the flag, insert the context that the hyperlink cross-references. The second \v flag marks the end of the context, that is, the end of

the invisible text. HELPMAKE generates a cross-reference whose context is the invisible text, and whose hyperlink is the entire word.

If the hyperlink consists of a phrase, rather than a single word, you must use anchored text to create explicit cross-references. Use the \a and \v flags to create anchored text as follows:

\a*hyperlink-words*\v*context*\v

The \a flag marks an "anchor" for the cross-reference. The text that follows the \a flag is the hyperlink. The hyperlink must fit entirely on one line. The first \v flag marks both the end of the hyperlink and the beginning of the invisible text that contains the cross-reference context. The second \v flag marks the end of the invisible text.

Examples

```
See also: abs, cabs, fabs
```

The example above contains three implicit cross-references to the C routines **abs**, **cabs**, and **fabs**.

```
See also: Example\vopen.ex\v, Template\vopen.tm\v, close
```

The example above shows the encoding for an explicit cross-reference to an example program and a function template from the help data base for the C run-time library. The hyperlinks are `Example` and `Template`, which reference the contexts `open.ex` and `open.tm`. The example also contains an implicit cross-reference to the **close** function.

```
See also: \ais... functions\vis_functions\v, atoi
```

The example above shows the encoding for an explicit cross-reference to an entire family of functions. This cross-reference uses anchored text to associate a phrase, rather than just a word, with a context. In this example, the hyperlink is the anchored phrase `is... functions`, and it cross-references the context `is_functions`. In addition, the example contains an implicit cross-reference to the **atoi** routine.

```
.context open
:113
\bInclude:\p    <fcntl.h>, <io.h>, <sys\\types.h>,
<sys\\stat.h>

\bPrototype:\p  int open(char *path, int flag[, int mode]);
          flag:  O_APPEND O_BINARY O_CREAT O_EXCL O_RDONLY
                 O_RDWR    O_TEXT    O_TRUNC  O_WRONLY
                 (may be joined by |)
          mode:  S_IWRITE  S_IREAD   S_IREAD | S_IWRITE
```

```
\bReturns:\p    a handle if successful, or -1 if not.
            errno:  EACCES, EEXIST, EMFILE, ENOENT

\bSee also:\p  \uExample\p\vopen.ex\v,
\uTemplate\p\vopen.tp\v,
              access, chmod, close, creat, dup, dup2, fopen,
sopen, umask
```

The code above is an example of a help-data-base file in QuickHelp format that contains a single entry using QuickHelp format. The `:l` sequence is the QuickC-specific character used in the help display. The number that follows `l` specifies the size of the initial window for the help text. In this case, the initial window displays 13 lines.

The manifest constants (such as **O_WRONLY** and **EEXIST**), the C keywords (such as **int** and **char**), and the other functions (such as **sopen** and **access**) are all implicit cross-references. The words `Example` and `Template` are explicit cross-references to the example `open.ex` and to the **open** template `open.tp`, respectively. Note the use of double backslashes in the include file names.

8.6.2 Minimally Formatted ASCII

You can use uncompressed, minimally formatted ASCII help files instead of compressed QuickHelp format files, although they are larger and slower to search. Unformatted ASCII files are of fixed width, and they may not contain highlighting (or other nondefault attributes) or cross-references.

A minimally formatted ASCII text file comprises a sequence of topics, each preceded by one or more unique context definitions. Each context definition must be on a separate line beginning with a help delimiter (>>). Subsequent lines up to the next context definition constitute the topic text.

Example

```
>>open

Include:    <fcntl.h>, <io.h>, <sys\\types.h>, <sys\\stat.h>

Prototype:  int open(char *path, int flag[, int mode]);
        flag:  O_APPEND  O_BINARY  O_CREAT  O_EXCL  O_RDONLY
               O_RDWR    O_TEXT    O_TRUNC  O_WRONLY
               (may be joined by |)
        mode:  S_IWRITE  S_IREAD   S_IREAD | S_IWRITE

Returns:    a handle if successful, or -1 if not.
            errno:  EACCES, EEXIST, EMFILE, ENOENT

See also:  access, chmod, close, creat, dup, dup2, fopen,
sopen, umask
```

The preceding example, coded in minimally formatted ASCII, shows the same text as the previous example. The first line of the example defines `open` as a context string; therefore the topic text that follows will be displayed when the user requests help on that topic. No formatting flags or cross-references are included because minimally formatted ASCII does not support them. Note, however, the double backslashes in the file names `sys\\types` and `sys\\stat.h`. The double backslashes ensure that HELPMAKE interprets the characters as backslashes and not as the start of a formatting flag.

The minimally formatted ASCII help file *must* begin with the help delimiter (>>), so that HELPMAKE can verify that the file is indeed an ASCII help file.

8.6.3 Rich Text Format (RTF)

RTF is a Microsoft word-processing format supported by many other word processors. It allows documents to be transferred from one application to another with losing any formatting information. The HELPMAKE utility recognizes a subset of the full RTF syntax. If your file contains any RTF code that is not part of the subset, HELPMAKE ignores the code and strips it out of the file.

In general, word-processing and file-conversion programs generate the RTF code automatically as output. You need not worry about inserting RTF codes yourself; you can simply format your help files directly with a word-processor that generates RTF, using the attributes supported by the subset. The only items you need to insert are the help delimiter (>>) and context string that start each entry.

HELPMAKE recognizes the subset of RTF listed below:

RTF Code	Action
\plain	Default attributes. On most screens this is nonblinking normal intensity.
\b	Boldface. This is displayed as intensified text.
\i	Italic. This is displayed as reverse video.
\v	Hidden text. Hidden text is used for cross-reference information and for some application-specific communications; it is not displayed.
\ul	Underline. This is represented as blue text on adapters that do not support underlining.
\par	End of paragraph.
\pard	Default paragraph formatting.

\fi	Paragraph first-line indent.
\li	Paragraph indent from left margin.
\line	New line (not new paragraph).
\tab	Tab character.

Using the word-processing program, you can break the topic text into paragraphs. When HELPMAKE compresses the file, it formats the text to the width given with the /W option, ignoring the paragraph formats.

As with the other text formats, each entry in the data base source consists of one or more context strings, followed by topic text. The help delimiter (>>) at the beginning of any paragraph denotes the beginning of a new help entry. The text that follows on the same line is defined as a context for the topic. If the next paragraph also begins with the help delimiter, it also defines a context string for the same topic text. You may define any number of contexts for a block of topic text. The topic text comprises all subsequent paragraphs up to the next paragraph that begins with the help delimiter.

Example

```
{rtf0
>> open \par
{\b Include:}   <fcntl.h>, <io.h>, <sys\\types.h>,
<sys\\stat.h>

{\b Prototype:}  int open(char *path, int flag[, int mode]);
         flag: O_APPEND  O_BINARY  O_CREAT  O_EXCL  O_RDONLY
               O_RDWR    O_TEXT    O_TRUNC  O_WRONLY
               (may be joined by |)
         mode: S_IWRITE  S_IREAD   S_IREAD | S_IWRITE

{\b Returns:}    a handle if successful, or -1 if not.
            errno:  EACCES, EEXIST, EMFILE, ENOENT

{\b See also:}   {\u Example}{\v open.ex},
{\u Template}{\v open.tp}, access, chmod, close,
 creat, dup, dup2, fopen, sopen, umask

}
```

The code above is an example of a help data base that contains a single entry using subset RTF text. Note that RTF uses curly braces ({}) for nesting. Thus, the entire file is enclosed in curly braces, as is each specially-formatted text item.

Appendixes

Appendix A

Exit Codes

Most of the utilities return an exit code (sometimes called an "errorlevel" code) that can be used by DOS batch files or other programs such as NMAKE. If the program finishes without errors, it returns exit code 0. The code returned is non-zero if the program encounters an error. This appendix discusses several uses for exit codes and lists the exit codes that can be returned by each utility.

A.1 Exit Codes with NMAKE

The Microsoft Program-Maintenance Utility (NMAKE) automatically stops execution if a program executed by one of the commands in the NMAKE description file encounters an error. (Invoke NMAKE with the /I option to disable this behavior for the entire description file; or place a minus sign (–) in front of a command to disable it for only that command.) The exit code returned by the program is displayed as part of the error message.

For example, assume the NMAKE description file `TEST` contains the following lines:

```
TEST.OBJ :      TEST.C
      QCL /c TEST.C
```

If the source code in `TEST.C` contains a program error (but not if it contains a warning error), you would see the following message the first time you use NMAKE with the NMAKE description file `TEST`:

```
"nmake: fatal error U1077: return code 2"
```

This error message indicates that the command `QCL /c TEST.C` in the NMAKE description file returned exit code 2.

You can also test exit codes in NMAKE description files with the !IF directive.

A.2 Exit Codes with DOS Batch Files

If you prefer to use DOS batch files instead of NMAKE description files, you can test the code returned with the **IF** command. The following sample batch file, called `COMPILE.BAT`, illustrates how to do this:

```
QCL /c %1.C
IF NOT ERRORLEVEL 1 LINK %1;
IF NOT ERRORLEVEL 1 %1
```

You can execute this sample batch file with the following command:

```
COMPILE TEST
```

DOS then executes the first line of the batch file, substituting `TEST` for the parameter `%1`, as in the following command line:

```
QCL /c TEST.C
```

It returns exit code 0 if the compilation is successful or a higher code if the compiler encounters an error. In the second line, DOS tests to see if the code returned by the previous line is 1 or higher. If it is not (that is, if the code is 0), DOS executes the following command:

```
LINK TEST;
```

LINK also returns a code, which is tested by the third line. If this code is 0, the `TEST` program is executed.

The compiler returns the following exit codes:

Code	Meaning
0	No error
Nonzero number	Program or system-level error

A.3 Exit Codes for Programs

An exit code 0 always indicates execution of the program with no fatal errors. Warning errors also return exit code 0. NMAKE can return several codes indicating different kinds of errors, while other programs return only one code to indicate that an error occurred.

A.3.1 LINK Exit Codes

The linker returns the following exit codes:

Code	Meaning
0	No error.
2	Program error. Commands or files given as input to the linker produced the error.

4	System error. The linker encountered one of the following problems: 1) ran out of space on output files; 2) was unable to reopen the temporary file; 3) experienced an internal error; 4) was interrupted by the user.

A.3.2 LIB Exit Codes

The Microsoft Library Manager (LIB) returns the following exit codes:

Code	Meaning
0	No error.
2	Program error. Commands or files given as input to the utility produced the error.
4	System error. The library manager encountered one of the following problems: 1) ran out of memory; 2) experienced an internal error; 3) was interrupted by the user.

A.3.3 NMAKE Exit Codes

The Microsoft Program-Maintenance Utility (NMAKE) returns the following exit codes:

Code	Meaning
0	No error
2	Program error
4	System error—out of memory

If a program called by a command in the NMAKE description file produces an error, the exit code is displayed in the NMAKE error message.

Appendix B

Working with QuickC Memory Models

You can gain greater control over how your program uses memory by specifying the memory model for the program. You do not need to specify a memory model except in the following cases:

- Your program has more than 64K of code or more than 64K of static data.
- Your program contains individual arrays that need to be larger than 64K.

In these cases, you have the following options:

1. If you are compiling with the QCL command, you can specify one of the other standard memory models (medium, compact, large, or huge) using one of the /A options.
2. You can create a mixed-model program using the **near**, **far**, and **huge** keywords.
3. You can combine method 2 with method 1.

B.1 Near, Far, and Huge Addressing

The terms "near," "far," and "huge" are crucial to understanding the concept of memory models. These terms indicate how data can be accessed in the segmented architecture of the 8086 family of microprocessors (8086, 80186, and 80286).

Segments

DOS loads the code and data allocated by your program into "segments" in physical memory. Each segment is up to 64K long. Because separate segments are always allocated for the program code and data, the minimum number of segments allocated for a program is two. These two segments, required for every program, are called the default segments. The small memory model uses only the two default segments. The other memory models discussed in this chapter allow more than one code segment per program, or more than one data segment per program, or both.

Near addresses

In the 8086 family of microprocessors, all memory addresses consist of two parts:

1. A 16-bit number that represents the base address of a memory segment
2. Another 16-bit number that gives an offset within that segment

The architecture of the 8086 microprocessor is such that code can be accessed within the default code or data segment by using just the 16-bit offset value. This is possible because the segment addresses for the default segments are always known. This 16-bit offset value is called a "near address"; it can be accessed with a "near pointer." Because only 16-bit arithmetic is required to access any near item, near references to code or data are smaller and more efficient.

Far addresses When data or code lie outside the default segments, the address must use both the segment and offset values. Such addresses are called "far addresses"; they can be accessed by using "far pointers" in a C program. Accessing far data or code items is more expensive in terms of program speed and size, but using them enables your programs to address all memory, rather than just the standard 64K code and data segments.

Huge addresses There is a third type of address in Microsoft QuickC: the "huge" address. A huge address is similar to a far address in that both consist of a segment value and an offset value; but the two differ in the way address arithmetic is performed on pointers. Because items (both code and data) referenced by far pointers are still assumed to lie completely within the segment in which they start, pointer arithmetic is done only on the offset portion of the address. This gain in pointer arithmetic efficiency is achieved, however, by limiting the size of any single item to 64K. With data items, huge pointers overcome this size limitation: pointer arithmetic is performed on all 32 bits of the data item's address, thus allowing data items referenced by huge pointers to span more than one segment, provided they conform to the rules outlined in Section B.2.5, "Creating Huge-Model Programs."

The rest of this chapter deals with the various methods you can use to control whether your program makes near or far calls to access code or data.

B.2 Using the Standard Memory Models

The libraries created by the SETUP program support five standard memory models. Using the standard memory models is the simplest way to control how your program accesses code and data in memory.

When you use the standard memory models, the compiler handles library support for you. The library corresponding to the memory model you specify is used automatically. Each memory model, except the huge model, has its own library. The huge model uses the same library as the large model.

The advantage of using standard models for your programs is simplicity. In the standard models, memory management is specified by compiler options; since the standard models do not require the use of extended keywords, they are the best way to write code that can be ported to other systems (particularly systems that do not use segmented architectures).

The disadvantage of using standard memory models exclusively is that they may not produce the most efficient code. For example, if you have an otherwise small-model program containing a large array that pushes the total data size for your program over the 64K limit for small model, it may be to your advantage to declare the one array with the **far** keyword, while keeping the rest of the program small model, as opposed to using the standard compact memory model for the entire program. For maximum flexibility and control over how your program uses memory, you can combine the standard-memory-model method with the **near**, **far**, and **huge** keywords, described in Section B.3.

The /A options for QCL are used to specify one of the five standard memory models (small, medium, compact, large, or huge) at compile time. These memory-model options are discussed in the next five sections.

NOTE *In the following sections, which describe the different memory-model addressing conventions, it is important to keep in mind two common features of all five models:*

1. *No* single *source module can generate 64K or more of code.*
2. *No* single *data item can exceed 64K, unless it appears in a huge-model program or it has been declared with the huge keyword.*

B.2.1 Creating Small-Model Programs

The /AS option tells the compiler to create a program that occupies the two default segments—one for code and one for data.

Small-model programs are typically QuickC programs that are short or have a limited purpose. Because code and data for these programs are each limited to 64K, the total size of a small-model program can never exceed 128K. Most programs fit easily into this model.

Because programs compiled within the QuickC environment use the small memory model by default, you should give the /AS option in cases where you use the QCL command to compile a module for use within the QuickC environment.

By default, both code and data items in small-model programs are accessed with near addresses. You can override the defaults by using the **far** or **huge** keyword for data or by using the **far** keyword for code.

The compiler in the QuickC environment and the QCL command create small-model programs automatically if you do not specify a memory model. The /AS option is provided for completeness; you never need to give it explicitly.

Figure B.1 illustrates how memory is set up for the small memory model.

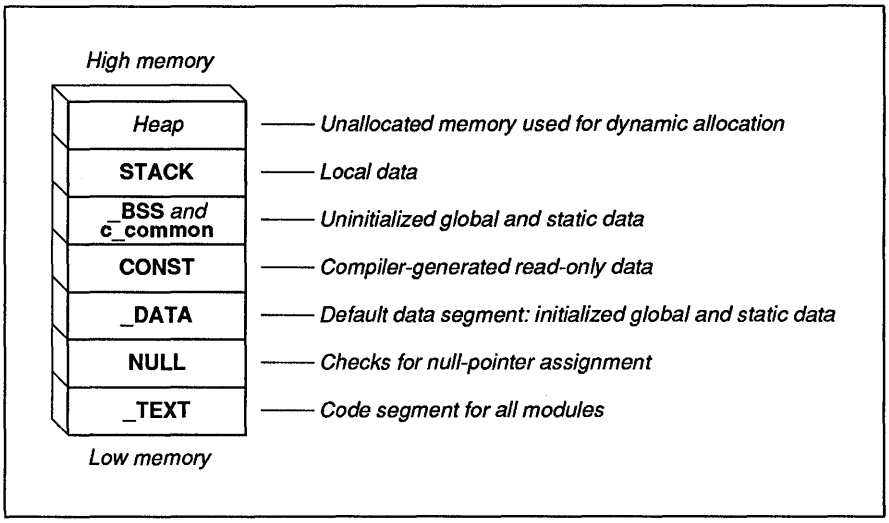


Figure B.1 Memory Map for Small Memory Model

B.2.2 Creating Medium-Model Programs

The /AM option provides a single segment for program data and multiple segments for program code. Each source module is given its own code segment.

Medium-model programs are typically QuickC programs that have a large number of program statements (more than 64K of code), but a relatively small amount of data (less than 64K). Program code can occupy any amount of space and is given as many segments as needed; total program data cannot be greater than 64K.

By default, code items in medium-model programs are accessed with far addresses, and data items are accessed with near addresses. You can override the default by using the **far** or **huge** keyword for data and the **far** keyword for code.

The medium model provides a useful trade-off between speed and space, since most programs refer more frequently to data items than to code. Figure B.2 illustrates how memory is set up for the medium memory model.

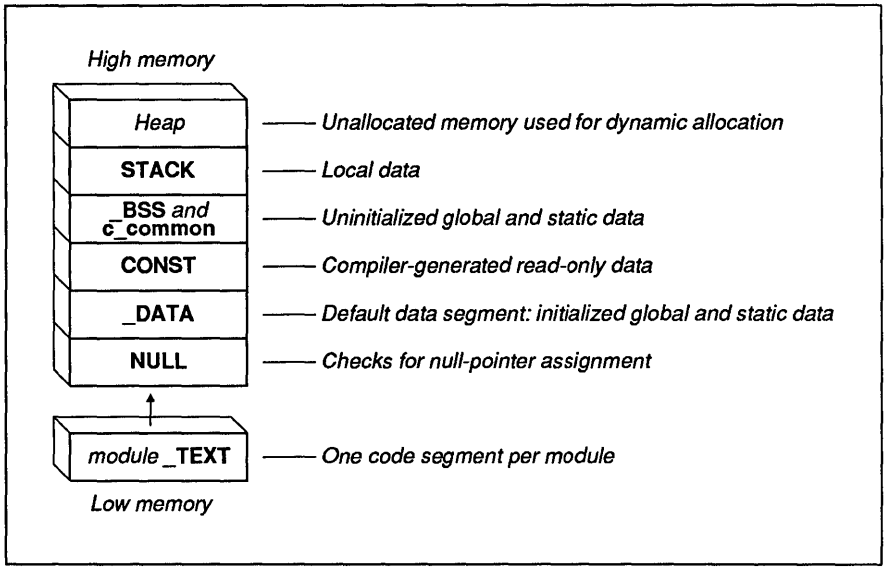


Figure B.2 Memory Map for Medium Memory Model

B.2.3 Creating Compact-Model Programs

The /AC option directs the compiler to allow multiple segments for program data but only one segment for the program code.

Compact-model programs are typically QuickC programs that have a large amount of data but a relatively small number of program statements. Program data can occupy any amount of space and are given as many segments as needed.

By default, code items in compact-model programs are accessed with near addresses, and data items are accessed with far addresses. You can override the defaults by using the **near** or **huge** keyword for data or by using the **far** keyword for code.

In the medium and compact models, **NULL** must be used carefully in certain situations. **NULL** actually represents a null data pointer. In the small, large, and huge memory models, where code and data pointers are the same size, it can be used with either. This is not the case, however, in medium and compact memory models, where code and data pointers are different sizes. Consider the following example:

```
void func1(char *dp)
{
.
.
.
}
```

```
void func2(char (*fp)(void))
{
.
.
.
}

main()
{
func1(NULL);
func2(NULL);
}
```

This example passes a 16-bit pointer to both `func1` and `func2` if compiled using the medium model, and a 32-bit pointer to both `func1` and `func2` if compiled using the compact model. To override this behavior, add prototypes to the beginning of the program to indicate the types, or use an explicit cast on the argument to `func1` (compact model) or `func2` (medium model).

Figure B.3 illustrates how memory is set up for the compact memory model.

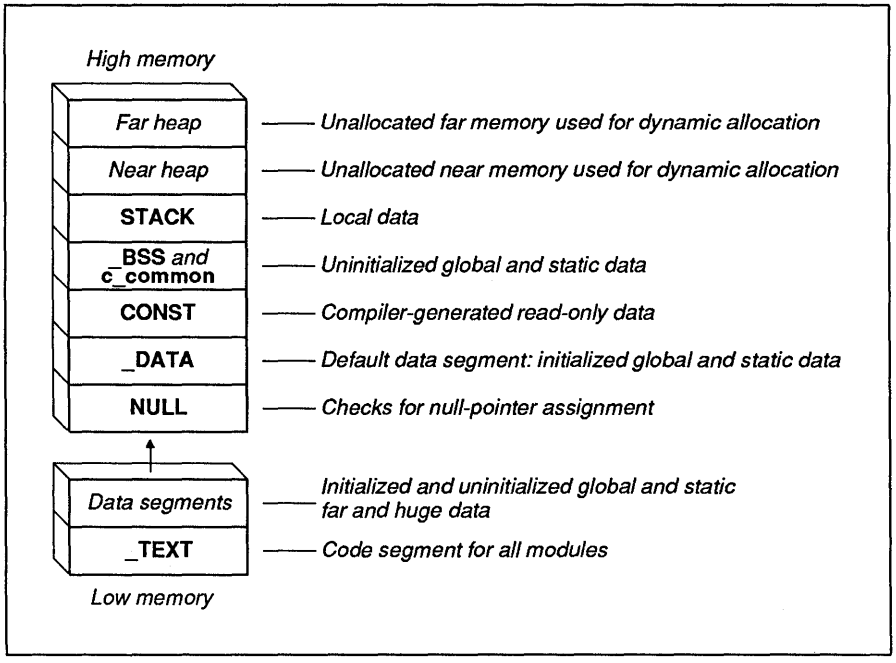


Figure B.3 Memory Map for Compact Memory Model

B.2.4 Creating Large-Model Programs

The /AL option allows the compiler to create multiple segments, as needed, for both code and data. No one data item, however, may exceed 64K.

Large-model programs are typically very large C programs that use a large amount of data storage during normal processing.

By default, both code and data items in large-model programs are accessed with far addresses. You can override the defaults by using the **near** or **huge** keyword for data or by using the **near** keyword for code.

Figure B.4 illustrates how memory is set up for the large and huge memory models.

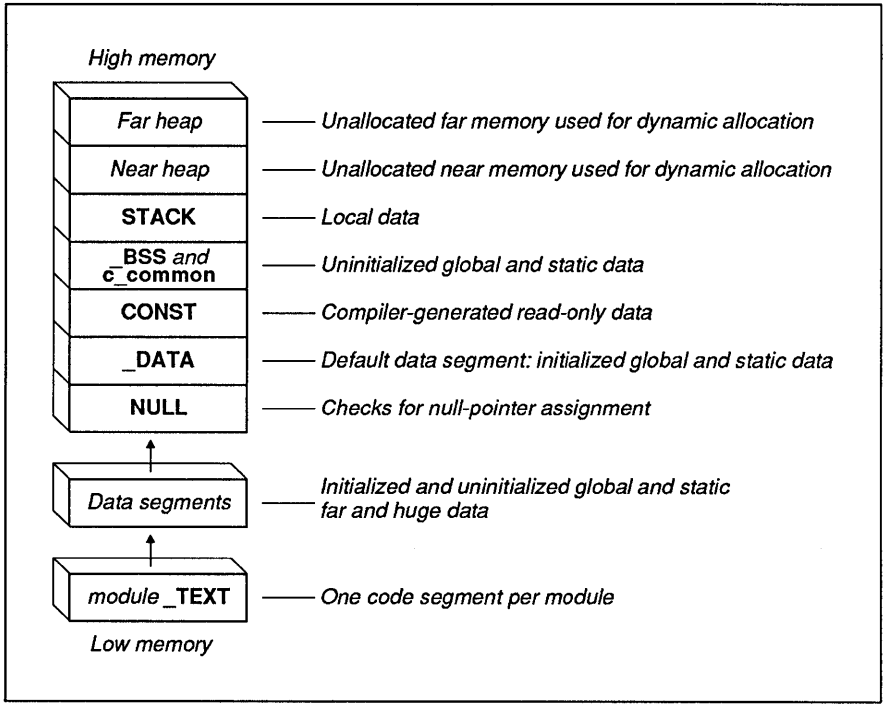


Figure B.4 Memory Map for Large and Huge Memory Models

B.2.5 Creating Huge-Model Programs

The /AH option is similar to the /AL option, except that the restriction on the size of individual data items is removed for arrays.

Restrictions Some size restrictions do apply to elements of huge arrays where the array is larger than 64K. To provide efficient addressing, array elements are not permitted to cross segment boundaries. This has the following implications:

1. No array element can be larger than 64K. For instance, this might occur when an array has elements that are structures or arrays.
2. For any array larger than 128K, all elements must have a size in bytes equal to a power of 2 (that is, 2 bytes, 4 bytes, 8 bytes, 16 bytes, and so on). If the array is 128K or smaller, however, its elements may be any size, up to and including 64K.

Pointer subtraction In huge-model programs, care must be taken when using the **sizeof** operator or when subtracting pointers. The C language defines the value returned by the **sizeof** operator to be an **unsigned int** value, but the size in bytes of a huge array is an **unsigned long** value. To solve this discrepancy, the Microsoft QuickC Compiler produces the correct size of a huge array when a type cast like the following is used:

```
(unsigned long)sizeof(huge_item)
```

Similarly, the C language defines the result of subtracting two pointers as an **int** value. When subtracting two huge pointers, however, the result may be a **long int** value. The Microsoft QuickC Compiler gives the correct result when a type cast like the following is used:

```
(long)(huge_ptr1 - huge_ptr2)
```

B.3 Using the near, far, and huge Keywords

One limitation of the predefined memory-model structure is that, when you change memory models, all data and code address sizes are subject to change. The Microsoft QuickC Compiler, however, lets you override the default addressing convention for a given memory model and access items with a near, far, or huge pointer. This is done with the **near**, **far**, and **huge** keywords. These special type modifiers can be used with a standard memory model to overcome addressing limitations for particular data or code items, or to optimize access to these items without changing the addressing conventions for the program as a whole. Table B.1 explains how the use of these keywords affects the addressing of code or data, or pointers to code or data.

Table B.1 Addressing of Code and Data Declared with near and far

Keyword	Data	Function	Pointer Arithmetic
near	Resides in default data segment; referenced with 16-bit addresses (pointers to data are 16 bits)	Assumed to be in current code segment; referenced with 16-bit addresses (pointers to functions are 16 bits)	16 bits
far	May be anywhere in memory—not assumed to reside in current data segment; referenced with 32-bit addresses (pointers to data are 32 bits)	Not assumed to be in current code segment; referenced with 32-bit address (pointers to functions are 32 bits)	16 bits
huge	May be anywhere in memory—not assumed to reside in current data segment; individual data items (arrays) can exceed 64K in size; referenced with 32-bit addresses (pointers to data are 32 bits)	Not applicable to code	32 bits

NOTE *The* ***near, far,*** *and* ***huge*** *keywords are not a standard part of the C language; they are meaningful only for systems that use a segmented architecture similar to that of the 8086 microprocessors. Keep this in mind if you want your code to be ported to other systems.*

In the Microsoft QuickC Compiler, the words **near**, **far**, and **huge** are C keywords by default. To treat these keywords as ordinary identifiers, you must do one of the following:

- For programs compiled within the QuickC environment, compile with the Language Extensions option turned off.
- For programs compiled with the QCL command, give the /Za option at compile time.

These options are useful if you are compiling programs with compilers in which **near**, **far**, and **huge** are not keywords—for instance, if you are porting a program in which one of these words is used as a label.

B.3.1 Library Support for near, far, and huge

When using the **near**, **far**, and **huge** keywords to modify addressing conventions for particular items, you can usually use one of the standard libraries (small, compact, medium, large, or huge) with your program. The large-model libraries are also appropriate for use with huge-model programs. However, you must use care when calling library routines. In general, you cannot pass far pointers, or the addresses of far data items, to a small-model library routine. (Some exceptions to this statement are the library routines **halloc** and **hfree**, and the **printf** family of functions.) Of course, you can always pass the *value* of a far item to a small-model library routine, as shown in the following example:

```
long far time_val;

time(&time_val);                    /* Illegal */
printf("%ld\n", time_val);          /* Legal */
```

If you use the **near**, **far**, or **huge** keyword, it is strongly recommended that you use function prototypes with argument-type lists to ensure that all pointer arguments are passed to functions correctly (see Section B.3.4).

B.3.2 Declaring Data with near, far, and huge

The **near**, **far**, and **huge** keywords modify either objects or pointers to objects. When using them to declare data or code (or pointers to data or code), keep the following rules in mind:

- The keyword always modifies the object or pointer immediately to its right. In complex declarators, think of the **far** keyword and the item immediately to its right as being a single unit. For example, in the declarator

  ```
  char far* *p;
  ```

  `p` is a pointer (whose size depends on the memory model specified) to a far pointer to **char**.

- If the item immediately to the right of the keyword is an identifier, the keyword determines whether the item will be allocated in the default data segment (**near**) or a separate data segment (**far** or **huge**). For example,

  ```
  char near a;
  ```

  allocates `a` as an item of type **char** with a near address.

- If the item immediately to the right of the keyword is a pointer, the keyword determines whether the pointer will hold a near address (16 bits), a far address (32 bits) or a huge address (also 32 bits). For example,

  ```
  char far *p;
  ```

  allocates `p` as a far pointer (32 bits) to an item of type **char**.

Examples

The examples in this section show data declarations using the **near**, **far**, and **huge** keywords.

```
char a[3000];              /* small-model program */
char far b[30000];
```

The first declaration in the example above allocates the array `a` in the default data segment. By contrast, the array `b` in the second declaration may be allocated in any far data segment. Since these declarations appear in a small-model program, array `a` probably represents frequently used data that were deliberately placed in the default segment for fast access. Array `b` probably represents seldom used data that might make the default data segment exceed 64K and force the programmer to use a larger memory model if the array were not declared with the **far** keyword. The second declaration uses a large array because it is more likely that a programmer would want to specify the address-allocation size for items of substantial size.

```
char a[3000];              /* large-model program */
char near b[3000];
```

In the example above, access speed would probably not be critical for array `a`. Even though it may or may not be allocated within the default data segment, it is always referenced with a 32-bit address. Array `b` is explicitly allocated near to improve speed of access in this memory model (large).

```
char huge *pa;     /* small-model program */
```

In the small-model program above, `pa` is declared as a pointer to huge data. Any pointer arithmetic for `pa` (such as `pa++`) would be performed using 32-bit arithmetic. QuickC does not support static huge data; thus the array to which `pa` might point must be allocated with the huge data allocation function **halloc**.

```
char *pa;                  /* small-model program */
char far *pb;
```

The pointer `pa` is declared as a near pointer to an item of type **char** in the example above. The pointer is near by default since the example appears in a small-model program. By contrast, `pb` is allocated as a far pointer to an item of type **char**; `pb` could be used to point to, and step through, an array of characters stored in a segment other than the default data segment. For example, `pa` might

be used to point to array `a` in the first example, while `pb` might be used to point to array `b`.

```
char far * *pa;          /* small-model program */
char far * *pa;          /* large-model program */
```

The pointer declarations in the example above illustrate the interaction between the memory model chosen and the **near**, **far**, and **huge** keywords. Although the declarations for `pa` are identical, in a small-model program `pa` is declared as a near pointer to an array of far pointers to type **char**, while in a large-model program, `pa` is declared as a far pointer to an array of far pointers to type **char**.

```
char far * near *pb;      /* any model */
char far * far *pb;
```

In the first declaration in this sixth and final example, `pb` is declared as a near pointer to an array of far pointers to type **char**; in the second declaration, `pb` is declared as a far pointer to an array of far pointers to type **char**. Note that, in this example, the **far** and **near** keywords override the model-specific addressing conventions shown in the fifth example; the declarations for `pb` would have the same effect, regardless of the memory model.

B.3.3 Declaring Functions with the near and far Keywords

The rules for using the **near**, **far**, and **huge** keywords for functions are similar to those for using them with data, as listed below:

- The keyword always modifies the function or pointer immediately to its right. See Chapter 2, "Functions," of *C for Yourself*, for more information about rules for evaluating complex declarations.
- If the item immediately to the right of the keyword is a function, then the keyword determines whether the function will be allocated as near or far. For example,

  ```
  char far fun( );
  ```

  defines `fun` as a function called with a 32-bit address and returning type **char**.
- If the item immediately to the right of the keyword is a pointer to a function, then the keyword determines whether the function will be called using a near (16-bit) or far (32-bit) address. For example,

  ```
  char (far * pfun)( );
  ```

  defines `pfun` as a far pointer (32 bits) to a function returning type **char**.
- Function declarations must match function definitions.
- The **huge** keyword cannot be applied to functions.

Examples

```
void char far fun(void);                /* small model */
void char far fun(void)
        {
        .
        .
        .
        }
```

In the example above, `fun` is declared as a function returning type **char**. The **far** keyword in the declaration means that `fun` must be called with a 32-bit call.

```
static char far * near fun( );          /* large model */
static char far * near fun( )
        {
        .
        .
        .
        }
```

In the large-model example above, `fun` is declared as a near function that returns a far pointer to type **char**. Such a function might be seen in a large-model program as a helper routine that is used frequently, but only by the routines in its own module. Because all routines in a given module share the same code segment, the function could always be accessed with a near call. However, you could not pass a pointer to `fun` as an argument to another function outside the module in which `fun` was declared.

```
void far *fun(void);                    /* small model */
void (far * pfun) ( ) = fun;
```

The small-model example above declares `pfun` as a far pointer to a function that has return type **void**, and then assigns the address of `fun` to `pfun`. In fact, `pfun` could be used to point to any function accessed with a far call. Note that if the function pointed to by `pfun` has not been declared with the **far** keyword, or if it is not far by default, then calling that function through `pfun` would cause the program to fail.

```
double far * (far fun)( );              /* compact model */
double far * (far *pfun)( ) = fun;
```

The final example above declares `pfun` as a far pointer to a function that returns a far pointer to type **double**, and then assigns the address of `fun` to `pfun`. This might be used in a compact-model program for a function that is not used frequently and thus does not need to be in the default code segment. Both the function and the pointer to the function must be declared with the **far** keyword.

B.3.4 Pointer Conversions

Passing pointers as arguments to functions may cause automatic conversions in the size of the pointer argument because passing a pointer to a function forces the pointer size to the larger of the following two sizes:

- The default pointer size for that type, as defined by the memory model used during compilation.

  For example, in medium-model programs, data pointer arguments are near by default and code pointer arguments are far by default.

- The type of the argument.

If a function prototype with argument types is given, the compiler performs type checking and enforces the conversion of actual arguments to the declared type of the corresponding formal argument. However, if no declaration is present or the argument-type list is empty, the compiler will convert pointer arguments automatically to either the default type or the type of the argument, whichever is largest. To avoid mismatched arguments, you should always use a prototype with the argument types.

Examples

```
/* This program produces unexpected results in compact-,
** large-, or huge-model programs.
*/

main( )

        {
        int near *x;
        char far *y;
        int z = 1;

        test_fun(x, y, z);    /*x coerced to far pointer*/

        }

int test_fun(ptr1, ptr2, a)
        int near *ptr1;
        char far *ptr2;
        int a;

        {
        printf("Value of a = %d\n", a);
        }
```

If the preceding example is compiled as a small-model program (default for QCL) or medium-model program (with the /AM option on the QCL command line), the size of pointer argument `x` is 16 bits, the size of pointer argument `y` is 32 bits, and the value printed for `a` is 1. However, if the preceding example is compiled with the /AC, /AL, or /AH option, both `x` and `y` are automatically converted to far pointers when they are passed to `test_fun`. Because `ptr1`, the first parameter of `test_fun`, is defined as a near-pointer argument, it takes only 16 bits of the 32 bits passed to it. The next parameter, `ptr2`, takes the remaining 16 bits passed to `ptr1`, plus 16 bits of the 32 bits passed to it. Finally, the third parameter, `a`, takes the leftover 16 bits from `ptr2`, instead of the value of `z` in the `main` function. This shifting process does not generate an error message, because both the function call and the function definition are legal, but in this case the program does not work as intended because the value assigned to `a` is not the value intended.

To pass `ptr1` as a near pointer, you should include a forward declaration that specifically declares this argument for `test_fun` as a near pointer, as shown below:

```
/* First, declare test_fun so the compiler knows in advance
** about the near pointer argument:
*/
int test_fun(int near*, char far *, int);

main( )

        {
        int near *x;
        char far *y;
        int z = 1;

        test_fun(x, y, z);    /* now, x will not be coerced
                              ** to a far pointer; it will be
                              ** passed as a near pointer,
                              ** no matter what memory
                              ** model is used
                              */
        }

int test_fun(ptr1, ptr2, a)
        int near *ptr1;
        char far *ptr2;
        int a;

        {
        printf("Value of a = %d\n", a);
        }
```

Note that it would not be sufficient to reverse the definition order for `test_fun` and `main` in the first example to avoid pointer coercions; the pointer arguments must be declared in a forward declaration, as in the second example.

Appendix C

Hardware-Specific Utilities

This appendix describes three utility programs provided with Microsoft QuickC. These utilities support special hardware that some QuickC users may have. The utilities are the following:

- The FIXSHIFT utility fixes a bug associated with some COMPAQ® and compatible keyboards.
- The MSHERC driver supports the Hercules® display adapter.
- The MOUSE driver supports the Microsoft mouse.

C.1 Fixing Keyboard Problems with FIXSHIFT

On the keyboards of some COMPAQ and compatible computers, the arrow keys are not part of the numeric keypad. Because of a bug in the ROM BIOS, the QuickC editor (and other software not supplied by Microsoft) may not operate correctly on these machines. The FIXSHIFT utility fixes this bug.

To correct the problem, copy FIXSHIFT.COM to the directory that contains the QuickC program files, and type the following command:

```
fixshift
```

Any combination of uppercase and lowercase letters is acceptable for this command. When FIXSHIFT executes, it first prompts you to press the DOWN key to ascertain whether the BIOS has the bug. If not, FIXSHIFT displays a message telling you so, then exits. You need not run FIXSHIFT again. If your machine's BIOS has the bug, FIXSHIFT displays additional prompts that guide you through the appropriate actions.

FIXSHIFT requires approximately 450 bytes of memory. It fixes only the BIOS bug and has no effect on other programs that you run. You can include the FIXSHIFT command in your AUTOEXEC.BAT file to correct the problem each time you start the computer.

C.2 Using Hercules® Graphics

This section briefly summarizes the support that Microsoft QuickC provides for the Hercules® display adapter. For more information, see your Hercules documentation. Note that the graphics demonstration program GRDEMO.C supports Hercules graphics.

C.2.1 Support for Cards and Display Characteristics

QuickC supports the Hercules Graphics Card, Graphics Card Plus, InColor Card, and other cards that are 100 per cent compatible.

Only monochrome (two-color) text and graphics are supported. In monochrome, the screen resolution is 720 x 348 pixels. The character box is 9 x 14 pixels. Text dimensions are 80 columns by 25 rows, but the bottom two scan lines of the 25th row are not visible.

C.2.2 The MSHERC Driver

MSHERC.COM is the driver for Hercules graphics. You must load the driver before running programs that use Hercules graphics. To load the driver, type the following command:

```
MSHERC
```

To load the driver when you start the machine, put the MSHERC command in your AUTOEXEC.BAT file.

If you have both a Hercules monochrome card and a color video card, you should invoke MSHERC.COM with the /H (/HALF) option, as follows:

```
MSHERC /H
```

The /H option causes the driver to use one instead of two graphics pages. This prevents the two video cards from trying to use the same area of memory. You need not use the /H option if you have only a Hercules card.

If you are developing a commercial application that uses graphics, you should include MSHERC.COM with your product; you are free to include this file without an explicit licensing agreement.

NOTE *MSHERC.COM is identical to QBHERC.COM, the Hercules driver shipped with Microsoft QuickBASIC, Version 4.0, and the Microsoft BASIC Compiler, Version 6.0.*

C.2.3 Using a Mouse

To use a mouse with the Hercules adapter, follow the special instructions for Hercules cards in the *Microsoft Mouse Programmer's Reference Guide*. (This manual must be ordered separately; it is not supplied with either Microsoft QuickC or the Microsoft Mouse package.)

C.2.4 Setting Hercules Graphics Mode

The graphics include file GRAPH.H sets manifest constants needed for Hercules graphics operation. In GRAPH.H, the constant **_HERCMONO** sets the video mode to 720 x 348 pixels in monochrome. GRAPH.H also includes the constant **_HGC** in the section labeled "videoconfig adapter values."

C.3 The Mouse Driver

The Microsoft Mouse is optional software and is not required for QuickC. If you use the mouse, however, you must have Version 6.10 or later of the MOUSE.COM driver for QuickC to operate correctly. If you have an earlier release, you need to use the MOUSE.COM driver provided with QuickC. See your Microsoft Mouse manual for installation instructions. If you update the driver, be sure to delete any outdated MOUSE.SYS drivers from your CONFIG.SYS file.

Appendix D

Error-Message Reference

This appendix lists error messages you may encounter as you develop a program and gives a brief description of actions you can take to correct the errors. The following list tells where to find error messages for the various components of the Microsoft QuickC Compiler:

Component	Section
The Microsoft QuickC Compiler	"Compiler Errors"
The command line used to invoke the Microsoft QuickC Compiler	"Command-Line Errors"
The Microsoft C run-time libraries and other run-time situations	"Run-Time Errors"
The Microsoft Overlay Linker, LINK	"LINK Error Messages"
The Microsoft Library Manager, LIB	"LIB Error Messages"
The Microsoft Program-Maintenance Utility, NMAKE	"NMAKE Error Messages"
The Microsoft Help-File Creation Utility, HELPMAKE	"HELPMAKE Error Messages"

Note that the compiler, command-line, and run-time error messages are listed alphabetically by category in this appendix.

See "Compiler Limits" in the "Compiler Errors" section for information about compiler limits; see "Run-Time Limits" in the "Run-Time Errors" section for information about run-time limits.

D.1 Compiler Errors

The error messages produced by the C compiler fall into three categories:

1. Fatal-error messages
2. Compilation-error messages
3. Warning messages

The messages for each category are listed below in numerical order, with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number. Each message that is generated within the QuickC environment appears in the error window; QuickC moves the cursor to the line that caused the error. Each message that is generated by compiling with the QCL command gives the file name and line number where the error occurs.

Fatal-Error Messages

Fatal-error messages indicate a severe problem, one that prevents the compiler from processing your program any further. These messages have the following format:

filename (*line*) : `fatal error C1`*xxx*: *messagetext*

After the compiler displays a fatal-error message, it terminates without producing an object file or checking for further errors.

Compilation-Error Messages

Compilation-error messages identify actual program errors. These messages appear in the following format:

filename (*line*) : `error C2`*xxx*: *messagetext*

The compiler does not produce an object file for a source file that has compilation errors in the program. When the compiler encounters such errors, it attempts to recover from the error. If possible, it continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Warning Messages

Warning messages are informational only; they do not prevent compilation or linking. These messages appear in the following format:

filename (*line*) : `warning C4`*xxx* :*messagetext*

You can use the /W option to control the level of warnings that the compiler generates.

D.1.1 Fatal-Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it terminates after displaying the error message.

Number	Fatal-Error Message
C1000	**UNKNOWN FATAL ERROR** **Contact Microsoft Technical Support** An unknown error condition was detected by the compiler. Please report this condition to Microsoft Corporation, using the Product Assistance Request form at the back of this manual.
C1001	**Internal Compiler Error** **(compiler file '*filename*', line *n*)** **Contact Microsoft Technical Support** The compiler detected an internal inconsistency. Please report this condition using the Product Assistance Request form at the back of this manual. Please include the file name and line number where the error occurred in this report; note that the file name refers to an internal compiler file, *not* your source file.
C1002	**out of heap space** The compiler ran out of dynamic memory space. This usually means that your program has too many symbols and/or complex expressions. To correct the problem, divide the file into several smaller source files, or break expressions into smaller subexpressions.
C1003	**error count exceeds *n*; stopping compilation** Errors in the program were too numerous or too severe to allow recovery, and the compiler must terminate.
C1004	**unexpected EOF** This message appears when you have insufficient space on the default disk drive for the compiler to create the temporary files it needs. The space required is approximately two times the size of the source file. This message can also occur when a comment does not have a closing delimiter (*/), or when the **#if** directive occurs without a corresponding closing **#endif** directive.

Number	Fatal-Error Message
C1007	**unrecognized flag '*string*' in '*option*'**
	The *string* in the command-line *option* was not a valid option.
C1008	**no input file specified**
	The compiler was given no file to compile.
C1009	**compiler limit : macros too deeply nested**
	The expansion of a macro exceeds the available space.
	Check to see if the macro is recursively defined or if the expanded text is too large.
C1010	**compiler limit : macro expansion too big**
	The expansion of a macro exceeds the available space.
C1012	**bad parenthesis nesting - missing '*character*'**
	The parentheses in a preprocessor directive were not matched; *character* is either a left or right parenthesis.
C1013	**cannot open source file '*filename*'**
	The given file either did not exist, could not be opened, or was not found.
	Make sure your environment settings are valid and that you have given the correct path name for the file. If this error appears without an error message, the compiler has run out of file handles. In that case, increase the value of the FILES= variable in your CONFIG.SYS file and reboot.
C1014	**too many include files**
	Nesting of **#include** directives exceeds the 10-level limit.
C1015	**cannot open include file '*filename*'**
	The given file either did not exist, could not be opened, or was not found. Make sure your environment settings are valid and that you have given the correct path name for the file. If these are correct, the problem may be that the compiler has run out of far heap space. See the description of error C1060 for alternative solutions. If this error appears without an error message, the compiler has run out of file handles. In that case, increase the value of the FILES= variable in your CONFIG.SYS file and reboot.

Number	Fatal-Error Message

C1016 **#if[n]def expected an identifier**

You must specify an identifier with the **#ifdef** and **#ifndef** directives.

C1017 **invalid integer constant expression**

The expression in an **#if** directive must evaluate to a constant.

C1018 **unexpected '#elif'**

The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** construct.

C1019 **unexpected '#else'**

The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** construct.

C1020 **unexpected '#endif'**

An **#endif** directive appears without a matching **#if**, **#ifdef**, or **#ifndef** directive.

C1021 **bad preprocessor command '***string***'**

The characters following the number sign (#) do not form a valid preprocessor directive.

C1022 **expected '#endif'**

An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.

C1028 ***segment*** **segment allocation exceeds 64K**

More than 64K of far data were allocated to the given segment. A single module can have only 64K of far data.

To solve this problem, either break declarations up into separate modules, reduce the amount of data your program uses, or compile your program with the Microsoft C Optimizing Compiler.

C1031 **compiler limit : function calls too deeply nested**

The program exceeded the compiler limit on nested function calls.

Number	Fatal-Error Message

C1032 **cannot open object listing file '*filename*'**

One of the following statements about the file name or path name given by filename is true:

- The given name is not valid.
- The file with the given name cannot be opened for lack of space.
- A read-only file with the given name already exists.

C1035 **expression too complex, please simplify**

The compiler was unable to generate code for a complex expression.

To solve this problem, break the expression into simpler subexpressions and recompile.

C1037 **cannot open object file '*filename*'**

One of the conditions listed under error message C1032 prevents the given file from being opened.

C1041 **cannot open compiler intermediate file – no more files**

The compiler could not create intermediate files used in the compilation process because no more file handles were available.

This error can usually be corrected by changing the `FILES=`*number* line in CONFIG.SYS to allow a larger number of open files (20 is the recommended setting).

C1045 **floating point overflow**

The compiler generated a floating-point exception while doing constant arithmetic on floating-point items at compile time, as in the following example:

```
float fp_val = 1.0e100;
```

In this example, the double-precision constant `1.0e100` exceeds the maximum allowable value for a floating-point data item.

C1047 **too many *option* flags, '*string*'**

The *option* appeared too many times. The *string* contains the occurrence of the option that caused the error.

Number	Fatal-Error Message
C1048	**unknown option '*character*' in '*optionstring*'**

The *character* was not a valid letter for *optionstring*.

C1049 **invalid numerical argument '*string*'**

A numerical argument was expected instead of *string*.

C1052 **too many #if/#ifdef's**

The program exceeded the maximum nesting level for **#if/#ifdef** directives.

C1053 **compiler limit : struct/union nesting**

Structure and union definitions were nested to more than 10 levels.

C1054 **compiler limit : initializers too deeply nested**

The compiler limit on nesting of initializers was exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized.

To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.

C1055 **compiler limit : out of keys**

The file being compiled contained too many symbols. Try to separate it into two files that can be compiled independently.

C1056 **compiler limit : out of macro expansion space**

The compiler overflowed an internal buffer during the expansion of a macro. Simplify the macro and/or shorten its text.

C1059 **out of near heap space**

The compiler ran out of storage for items that it stores in the "near" (default data segment) heap.

C1060 **out of far heap space**

The compiler ran out of storage for items that it stores in the far heap.

Usually this error occurs in programs compiled within the QuickC environment because the symbol table contains too many symbols. To fix this error, try compiling with the Debug option turned off, or try including fewer include files. If these solutions do not work, try compiling the program using the QCL command. Finally, it may help to free additional memory on your system by removing TSRs (processes that terminate and stay resident).

Number	Fatal-Error Message
C1061	**compiler limit : blocks too deeply nested**

Nested blocks in the program exceeded the nesting limit allowed by the compiler.

To correct this problem, rewrite the program so that fewer blocks are nested within other blocks.

C1062 **error writing to preprocessor output file**

You compiled with the /P, /E, or /EP option to produce a preprocessor output file, but not enough room was available to hold the file.

C1063 **compiler limit : compiler stack overflow**

Your program was too complex and caused the compiler stack to overflow. Simplify your program by making it more modular and recompile.

C1065 **compiler limit : '*identifier*' : macro definition too big**

The macro definition was longer than allowed.

Try to split the definition into two shorter definitions.

C1067 **compiler limit : identifier overflowed internal buffer**

The compiler read an identifier that is longer than the internal buffer used for identifier names. Shorten the name and recompile.

C1068 **cannot open file '*filename*'**

The compiler cannot open the given file. A number of conditions may cause this error, including (but not limited to) the following: the directory does not exist; the file is needed for output but is marked read only; the file is needed for input but does not exist; the FILES= line in CONFIG.SYS does not allow enough files.

C1069 **write error on file '*filename*'**

An error occurred while the compiler tried to write to the file. One possible cause of this error is insufficient disk space.

C1070 **mismatched #if/#endif pair in file '*filename*'**

The preprocessor found the **#if**, **#ifdef**, or **#ifndef** directive, but did not find a corresponding **#endif** directive in the same source file.

C1126 ***identifier*: automatic allocation exceeds '*size*'**

The space allocated for the local variables of a function exceeded the given limit.

D.1.2 Compilation-Error Messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues parsing the program (if possible) and outputs additional error messages. However, no object file is produced.

Number	Compilation-Error Message
C2000	**UNKNOWN ERROR** **Contact Microsoft Technical Support** The compiler detected an unknown error condition. Please report this condition to the Microsoft Corporation, using the Product Assistance Request form at the back of this manual.
C2001	**newline in constant** A new-line character in a character or string constant was not in the correct escape-sequence format (\n).
C2003	**expected 'defined id'** The identifier to be checked in an **#if** directive was not found.
C2004	**expected 'defined(id)'** An **#if** directive caused a syntax error.
C2005	**#line expected a line number, found '***string***'** A **#line** directive lacked the required line-number specification.
C2006	**#include expected a file name, found '***string***'** An **#include** directive lacked the required file-name specification.
C2007	**#define syntax** A **#define** directive caused a syntax error.
C2008	**'***character***' : unexpected in macro definition** The given character was used incorrectly in a macro definition.

Number	Compilation-Error Message
C2009	**reuse of macro formal '*identifier*'**
	The given identifier was used more than once in the formal-parameter list of a macro definition.
C2010	**'*character*' : unexpected in formal list**
	The given character was used incorrectly in the formal-parameter list of a macro definition.
C2012	**missing name following '<'**
	An **#include** directive lacked the required file-name specification.
C2013	**missing '>'**
	The closing angle bracket (>) was missing from an **#include** directive.
C2014	**preprocessor command must start as first non-whitespace**
	Non-white-space characters appeared before the number sign (#) of a preprocessor directive on the same line.
C2015	**too many chars in constant**
	A character constant containing more than one character or escape sequence was used.
C2016	**no closing single quote**
	A character constant was not enclosed in single quotation marks.
C2017	**illegal escape sequence**
	The character or characters after the escape character (\) did not form a valid escape sequence.
C2018	**unknown character '0x*character*'**
	The given hexadecimal number did not correspond to a character.
C2019	**expected preprocessor command, found '*character*'**
	The given character followed a number sign (#), but it was not the first letter of a preprocessor directive.

Number	Compilation-Error Message

C2020 **illegal digit '*character*' for base '*number*'**

The given character was not a legal digit for the base used.

C2021 **expected exponent value, not '*character*'**

The given character was used as the exponent of a floating-point constant but was not a valid number.

C2022 **'*number*' : too big for char**

The *number* was too large to be represented as a character.

C2023 **divide by 0**

The second operand in a division operation (/) evaluated to 0, giving undefined results.

C2024 **mod by 0**

The second operand in a remainder operation (%) evaluated to 0, giving undefined results.

C2025 **'*identifier*' : enum/struct/union type redefinition**

The given identifier had already been used for an enumeration, structure, or union tag.

C2026 **'*identifier*' : member of enum redefinition**

The given identifier has already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.

C2028 **struct/union member needs to be inside a struct/union**

Structure and union members must be declared within the structure or union.

This error may be caused by an enumeration declaration that contains a declaration of a structure member, as in the following example:

```
enum a {
       january,
       february,
       int march;     /* structure declaration:
                      ** illegal
                      */
       };
```

C2029 **'*identifier*' : bit-fields allowed only in structs**

Only structure types may contain bit fields.

Number	Compilation-Error Message
C2030	**'*identifier*' : struct/union member redefinition**

The *identifier* was used for more than one member of the same structure or union.

C2031 **'*identifier*' : function cannot be struct/union member**

The given function was declared to be a member of a structure or union.

To correct this error, use a pointer to the function instead.

C2032 **'*identifier*' : base type with near/far/huge not allowed**

The given structure or union member was declared with the **near**, **far**, or **huge** keyword.

C2033 **'*identifier*' : bit-field cannot have indirection**

The given bit field was declared as a pointer (*), which is not allowed.

C2034 **'*identifier*' : bit-field type too small for number of bits**

The number of bits specified in the bit-field declaration exceeded the number of bits in the given base type.

C2035 **struct/union '*identifier*' : unknown size**

The given structure or union had an undefined size.

C2036 **left of '*member*' must have struct/union type**

The expression before the member-selection operator (–>) was not a pointer to a structure or union type, or the expression before the member-selection operator (.) did not evaluate to a structure or union. In this message, *member* is a member designator in one of the following forms:

–>identifier
. identifier

C2037 **left of '*operator*' specifies undefined struct/union '*identifier*'**

The expression before the member-selection operator (–> or .) identified a structure or union type that was not defined.

C2038 **'*identifier*' : not struct/union member**

The given identifier was used in a context that required a structure or union member.

Number	Compilation-Error Message
C2039	**'–>' requires struct/union pointer**

The expression before the member-selection operator (–>) was a structure or union name, not a pointer to a structure or union as expected.

C2040 **'.' requires struct/union name**

The expression before the member-selection operator (.) was a pointer to a structure or union, not a structure or union name as expected.

C2042 **signed/unsigned keywords mutually exclusive**

Both the **signed** and the **unsigned** keywords were used in a single declaration, as in the following example:

```
unsigned signed int i;
```

C2043 **illegal break**

A **break** statement is legal only within a **do**, **for**, **while**, or **switch** statement.

C2044 **illegal continue**

A **continue** statement is legal only within a **do**, **for**, or **while** statement.

C2045 *'identifier'* **: label redefined**

The label appeared before more than one statement in the same function.

C2046 **illegal case**

The **case** keyword may appear only within a **switch** statement.

C2047 **illegal default**

The **default** keyword may appear only within a **switch** statement.

C2048 **more than one default**

A **switch** statement contained more than one **default** label.

C2050 **non-integral switch expression**

A switch expression was not integral.

C2051 **case expression not constant**

Case expressions must be integral constants.

C2052 **case expression not integral**

Case expressions must be integral constants.

Number	Compilation-Error Message
C2053	**case value *number* already used**
	The case value was already used in this **switch** statement.
C2054	**expected '(' to follow '*identifier*'**
	The context requires parentheses after the function *identifier*.
C2055	**expected formal parameter list, not a type list**
	An argument-type list appeared in a function definition instead of a formal parameter list.
C2056	**illegal expression**
	An expression was illegal because of a previous error. (The previous error may not have produced an error message.)
C2057	**expected constant expression**
	The context requires a constant expression.
C2058	**constant expression is not integral**
	The context requires an integral constant expression.
C2059	**syntax error : '*token*'**
	The token caused a syntax error.
C2060	**syntax error : EOF**
	The end of the file was encountered unexpectedly, causing a syntax error. This error can be caused by a missing closing brace (}) at the end of your program.
C2061	**syntax error : identifier '*identifier*'**
	The identifier caused a syntax error.
C2062	**type '*type*' unexpected**
	The type was misused.
C2063	**'*identifier*' : not a function**
	The identifier was not declared as a function, but an attempt was made to use it as a function.

Number	Compilation-Error Message
C2064	**term does not evaluate to a function**

An attempt was made to call a function through an expression that did not evaluate to a function pointer.

C2065	*'identifier'* **: undefined**

The identifier was not defined.

C2066	**cast to function returning . . . is illegal**

An object was cast to a function type.

C2067	**cast to array type is illegal**

An object was cast to an array type.

C2068	**illegal cast**

A type used in a cast operation was not a legal type.

C2069	**cast of 'void' term to non-void**

The **void** type was cast to a different type.

C2070	**illegal sizeof operand**

The operand of a **sizeof** expression was not an identifier or a type name.

C2071	*'identifier'* **: bad storage class**

The given storage class cannot be used in this context.

C2072	*'identifier'* **: initialization of a function**

An attempt was made to initialize a function.

C2073	*'function'* **: storage class must be extern**

A function declaration appears within a block, but the function is not declared **extern**. This causes an error if the /Za option is in effect, that is, when language extensions are not enabled. The following example would cause this error:

```
main ()
{
static int foo ();   /* This causes error if /Za is on. */
}
```

C2075	*'identifier'* **: array initialization needs curly braces**

The braces ({}) around the given array initializer were missing.

Number	Compilation-Error Message

C2076 **'*identifier*' : struct/union initialization needs curly braces**

The braces ({}) around the given structure or union initializer were missing.

C2077 **non-scalar field initializer '*identifier*'**

An attempt was made to initialize a bit-field member of a structure with a non-scalar value.

C2078 **too many initializers**

The number of initializers exceeded the number of objects to be initialized.

C2079 **'*identifier*' uses undefined struct/union '*name*'**

The identifier was declared as structure or union type *name*, which has not been defined.

C2082 **redefinition of formal parameter '*identifier*'**

A formal parameter to a function was redeclared within the function body.

C2084 **function '*identifier*' already has a body**

The function had already been defined.

C2085 **'*identifier*' : not in formal parameter list**

The identifier was declared in a function definition but not in the formal parameter list.

C2086 **'*identifier*' : redefinition**

The given identifier was defined more than once.

C2087 **'*identifier*' : missing subscript**

The definition of an array with multiple subscripts was missing a subscript value for a dimension other than the first dimension, as in the following example:

```
int func(a)
     char a[10][];          /* Illegal */
     {
     .
     .
     .
     }
```

Number Compilation-Error Message

```
int func(a)
     char a[][5];          /* Legal */
     {
     .
     .
     .
     }
```

C2088 **use of undefined enum/struct/union '*identifier*'**

The given identifier referred to a structure or union type that was not defined.

C2090 **function returns array**

A function cannot return an array. (It can return a pointer to an array.)

C2091 **function returns function**

A function cannot return a function. (It can return a pointer to a function.)

C2092 **array element type cannot be function**

Arrays of functions are not allowed; however, arrays of pointers to functions are allowed.

C2093 **cannot initialize a static or struct with address of automatic vars**

The program tried to use the address of an automatic variable in the initializer of a static item, as in the following example:

```
func()
{
     int i;
     static int *ip=&i;
     .
     .
     .
}
```

C2094 **label '*identifier*' was undefined**

The function did not contain a statement labeled with the given identifier.

C2095 *function* **: actual has type void : parameter** *number*

Formal parameters and arguments to functions cannot have type **void**; they can, however, have type **void** * (pointer to void).

Number	Compilation-Error Message
C2096	**struct/union comparison illegal**
	You cannot compare two structures or unions. (You can, however, compare individual members of structures and unions.)
C2097	**illegal initialization**
	An attempt was made to initialize a variable using a nonconstant value.
C2098	**non-address expression**
	An attempt was made to initialize an item that was not an lvalue.
C2099	**non-constant offset**
	An initializer used a nonconstant offset.
C2100	**illegal indirection**
	The indirection operator (*) was applied to a nonpointer value.
C2101	**'&' on constant**
	The address-of operator (&) did not have an lvalue as its operand.
C2102	**'&' requires lvalue**
	The address-of operator must be applied to an lvalue expression.
C2103	**'&' on register variable**
	An attempt was made to take the address of a register variable.
C2104	**'&' on bit-field ignored**
	An attempt was made to take the address of a bit field.
C2105	'*operator*' **needs lvalue**
	The given operator did not have an lvalue operand.
C2106	'*operator*' **: left operand must be lvalue**
	The left operand of the given operator was not an lvalue.
C2107	**illegal index, indirection not allowed**
	A subscript was applied to an expression that did not evaluate to a pointer.
C2108	**non-integral index**
	A nonintegral expression was used in an array subscript.

Number	Compilation-Error Message
C2109	**subscript on non-array** A subscript was used on a variable that was not an array.
C2110	**'+' : 2 pointers** An attempt was made to add one pointer to another.
C2111	**pointer + non-integral value** An attempt was made to add a nonintegral value to a pointer.
C2112	**illegal pointer subtraction** An attempt was made to subtract pointers that did not point to the same type.
C2113	**'–' : right operand pointer** The right operand in a subtraction operation (–) was a pointer, but the left operand was not.
C2114	**'*operator*' : pointer on left; needs integral right** The left operand of the given operator was a pointer; the right operand must be an integral value.
C2115	**'*identifier*' : incompatible types** An expression contained incompatible types.
C2116	**'*operator*' : bad '*direction*' operand** The left or right operand of the given operator was illegal for that operator.
C2117	**'*operator*' : illegal for struct/union** Structure and union type values are not allowed with the given operator.
C2118	**negative subscript** A value defining an array size was negative.

C2119	**'typedefs' both define indirection**
	
	Two **typedef** types were used to declare an item and both **typedef** types had indirection. For example, the declaration of `p` in the following example is illegal:
	
	`typedef int *P_INT;`
	`typedef short *P_SHORT;`
	`/* this declaration is illegal */`
	`P_SHORT P_INT p;`

Number	Compilation-Error Message

C2120 **'void' illegal with all types**

The **void** type was used in a declaration with another type.

C2125 **'*identifier*': allocation exceeds 64K**

The given item exceeds the size limit of 64K.

C2127 **parameter allocation exceeds 32K**

The storage space required for the parameters to a function exceeded the limit of 32K.

C2130 **#line expected a string containing the file name, found '*string*'**

A file name was missing from a **#line** directive.

C2131 **attributes specify more than one near/far/huge**

More than one of the keywords **near**, **far**, or **huge** were applied to an item, as in the following example:

```
typedef int near NINT;
NINT far a;          /* Illegal */
```

C2132 **syntax error : unexpected identifier**

An identifier appeared in a syntactically illegal context.

C2133 **'*identifier*' : unknown size**

An attempt was made to declare an unsized array as a local variable, as in the following example:

```
int mat_add(array1)
     int array1[];       /* Legal */
     {
     int array2[];       /* Illegal */
     .
     .
     .
     }
```

C2134 ***identifier* : struct/union too large**

The size of a structure or union exceeded the compiler limit (2^{32} bytes).

Number	Compilation-Error Message

C2137 **empty character constant**

The illegal empty-character constant (' ') was used.

C2138 **unmatched close comment '*/'**

The compiler detected an open-comment delimiter (/*) without a matching close-comment delimiter (*/). This error usually indicates an attempt to use illegal nested comments.

C2139 **type following '*type*' is illegal**

An illegal type combination, such as the following, was used:

```
long char a;
```

This message also appears if more than one **enum**, **struct**, or **union** type is used in the same declaration.

C2140 **argument type cannot be function returning ...**

A function was declared as a formal parameter of another function, as in the following example:

```
int func1(a)
     int a( );      /* Illegal */
```

C2141 **value out of range for enum constant**

An enumeration constant had a value outside the range of values allowed for type **int**.

C2142 **ellipsis requires three periods**

The compiler detected a token consisting of two periods (..) and assumed that an ellipsis (...) was intended.

C2143 **syntax error : missing '*token1*' before '*token2*'**

The compiler expected *token1* to appear before *token2*. This message may appear if a required closing brace (}), right parenthesis ()), or semicolon (;) is missing.

C2144 **syntax error : missing '*token*' before type '*type*'**

The compiler expected the given token to appear before the given type name.

This message may appear if a required closing brace (}), right parenthesis ()), or semicolon (;) is missing.

Number	Compilation-Error Message
C2145	**syntax error : missing '*token*' before identifier**

The compiler expected the given token to appear before an identifier. This message may appear if a semicolon (;) does not appear after the last declaration of a block.

C2146 **syntax error : missing '*token*' before identifier '*identifier*'**

The compiler expected the given token to appear before the given identifier.

C2147 **unknown size**

An attempt was made to increment an index or pointer to an array whose base type has not yet been declared.

C2148 **array too large**

An array exceeded the maximum legal size (2^{32} bytes).

C2149 *identifier*: **named bit-field cannot have 0 width**

The given named bit field had a zero width. Only unnamed bit fields are allowed to have zero width.

C2150 *identifier*: **bit-field must have type int, signed int, or unsigned int**

The ANSI C standard requires bit fields to have types of **int**, **signed int**, or **unsigned int**. This message appears only if you compiled your program with the /Za option.

C2151 **more than one cdecl/fortran/pascal attribute specified**

More than one keyword specifying a function-calling convention was given.

C2152 *identifier* : **pointers to functions with different attributes**

An attempt was made to assign a pointer to a function declared with one calling convention (**cdecl**, **fortran**, or **pascal**) to a pointer to a function declared with a different calling convention.

C2153 **hex constants must have at least 1 hex digit**

The hexadecimal constants **0x** and **0X** are illegal. At least one hexadecimal digit must follow the "x" or "X."

C2156 **pragma must be at outer level**

Certain pragmas must be specified at a global level, outside a function body, and one of these pragmas occurred within a function.

Number	Compilation-Error Message

C2159 **more than one storage class specified**

More than one storage class was given in a declaration, as in the following example:

```
extern static int i;
```

C2160 **## cannot occur at the beginning of a macro definition**

A macro definition began with a token-pasting operator (##), as in the following example:

```
#define mac(a,b) ##a...
```

C2161 **## cannot occur at the end of a macro definition**

A macro definition ended with a token-pasting operator (##).

C2162 **expected macro formal parameter**

The token following a stringizing operator (#) was not a formal parameter name, as in the following example:

```
#define print(a) printf(#b)
```

C2165 **'*keyword*' : cannot modify pointers to data**

The **fortran**, **pascal**, or **cdecl** keyword was used illegally to modify a pointer to data, as in the following example:

```
char pascal *p;
```

C2166 **lval specifies 'const' object**

An attempt was made to assign a value to an item declared with **const** storage class.

C2171 **'*operator*' : bad operand**

The given unary operator was used with an illegal operand type, as in the following example:

```
int (*fp)();
double d,d1;
.
.
.
fp++;
d = ~d1
```

Number	Compilation-Error Message
C2172	*function* **: actual is not a pointer : parameter** *number*

An attempt was made to pass a non-pointer argument to a function that expected a pointer. The given number indicates which argument was in error.

C2173 *function* **: actual is not a pointer : parameter** *number*, **parameter list** *number*

An attempt was made to pass a non-pointer argument to a function that expected a pointer. This error occurs in calls that return a pointer to a function. The first number indicates which argument was in error; the second number indicates which argument list contained the invalid argument.

C2174 *function* **: actual has type 'void' : parameter** *number*, **parameter list** *number*

An attempt was made to pass a **void** argument to a function. Formal parameters and arguments to functions cannot have type **void**; they can, however, have type **void *** (pointer to **void**). This error occurs in calls that return a pointer to a function. The first number indicates which argument was in error; the second number indicates which argument list contained the invalid argument.

C2175 *function* **: unresolved external**

The given function is not defined in the source file, or built into the QuickC programming environment, or present in the Quick library (if any) that was loaded.

This error occurs only for single-module programs created in the QuickC environment. To solve this program, either define the function in the source file, load a Quick library containing the function, or (if the function is a standard C library function) create a program list for the program.

C2176 static huge data not supported

You cannot declare data items with the huge attribute in the QuickC environment. Declare a huge pointer to the data item instead.

C2177 constant too big

Information was lost because a constant value was too large to be replaced in the type to which it was assigned.

C2180 controlling expression has type 'void'

The controlling expression in an **if**, **while**, **for**, or **do** statement was a function with **void** return type.

Number	Compilation-Error Message
C2181	**pragma requires command line option '***option***'**

The given option must be used when compiling the **check_pointer** pragma with the QuickC compiler.

C2182	**'***identifier***' : has type 'void'**

The given variable was declared with the **void** keyword. The **void** keyword can be used only in function declarations.

C2183	**'interrupt' function must be 'far'**

The given interrupt function was implicitly or explicitly declared as near. You must declare the function without the **near** attribute; furthermore, if you compile the program with the small (default) or compact memory model, you must explicitly declare the function with the **far** attribute.

C2184	**'***identifier***' function cannot be 'pascal/fortran'**

The given function was declared with the FORTRAN/Pascal calling conventions, either because the **fortran** or **pascal** attribute was used in the declaration or because the /Gc option was used at compile time. Functions declared with a variable number of arguments or with the **interrupt** attribute must use the C calling conventions. To correct this error, either remove the **pascal** or **fortran** attribute from the function declaration (if you compile without the /Gc option), or declare the function with the **cdecl** attribute (if you compile with the /Gc option).

C2186	**'saveregs/interrupt' modifiers mutually exclusive**

A function may be declared as saveregs or interrupt but not both.

C2187	**cast of near function pointer to far function pointer**

You attempted an illegal type case.

C2188	**#error :** *message*

The **#error** directive was used to terminate compilation and display a message.

C2205	**'***identifier***' : cannot initialize 'extern' block scoped variables**

A block-scoped variable with **extern** storage class may not be initialized in a function.

Number	Compilation-Error Message
C2206	*'function-name'* **: typedefs cannot be used for function definition**

A **typedef** was used to define a function type, as in the following example:

```
typedef int functyp();
.
.
.
functyp foo
        {
        .
        .
        .
        }
```

C2400 **inline syntax error '*context*', found '*token*'**

The given token caused a syntax error caused an error within the given context.

C2401 *'identifier'* **: register must be base '*context*'**

The register used within an indirect memory operand must be a base register in this context.

C2402 *'identifier'* **: register must be index '*context*'**

The register used within an indirect memory operand must be an index register in this context.

C2403 *'identifier'* **: register must be base/index '*context*'**

The register used within an indirect memory operand must be either a base or index register in this context.

C2404 *'identifier'* **: illegal register '*context*'**

This register in this context is illegal.

C2405 **illegal short forward reference with offset**

Short forward references must refer only to a label. An additional offset cannot be used.

C2406 *'identifier'* **: name undefined '*context*'**

The *identifier* used with the **SIZE** or **LENGTH** operator, or as a specifier with the member-selection operator (.), was not defined.

C2407 **illegal float register '*context*'**

An NDP register was specified in an illegal context.

Number	Compilation-Error Message
C2408	**illegal type on PTR operator** '*context*'

The first argument of the **PTR** operator was not a legal type specification.

C2409 **illegal type used as operator** '*context*'

A type was used within the given context as an operator.

C2410 '*identifier*' **: ambiguous member name** '*context*'

The given identifier within the given context is a member of more than one structure or union. To correct this problem, use a structure or union specifier in the operand that caused the error. A structure or union specifier is the name of a structure or union (as given in a **typedef** declaration) or a variable of the same type as the structure or union you are trying to reference.

C2411 '*identifier*' **: illegal struct/union member** '*context*'

The given identifier used with this context is not a member of a visible structure or union; or the identifier is not a member of the structure or union specified with the member-selection operator (.)

C2412 '*identifier*' **: label redefined**

The given label was defined more than once within the current function. Change the spelling of the label and its references.

C2413 '*token*' **: illegal align size**

The alignment size used with the **ALIGN** directive was either missing or outside the valid range.

C2414 **illegal number of operands**

The opcode does not support the number of operands used.

C2415 **improper operand type**

The opcode does not use operands of this type.

C2416 '*identifier*' **: illegal opcode for processor**

An identifier was used that is supported as an opcode on a different processor but not on the current processor. See Section 4.3.13 for information on the use of certain processor-specific instructions.

C2417 **divide by zero** '*context*'

The second argument to the division (/) operator used within the given context is zero.

Number	Compilation-Error Message
C2419	**mod by zero** *'context'*
	The second argument to the MOD operator used within the given context is zero.
C2420	*'identifier'* **: illegal symbol** *'context'*
	The given identifier is illegal within the given context.
C2421	**PTR operator used with register** *name*
	The PTR operator may not be used with a register operand.
C2422	**illegal segment override** *'context'*
	An illegal segment override was used within the given context.
C2424	*'token'* **: improper expression** *'context'*
	The given token is used to form an improper expression within the given context.
C2425	*'token'* **: non-constant expression** *'context'*
	The given token is used to form a nonconstant expression within the given context.
C2426	*'token'* **: illegal operator** *'context'*
	The given token may not be used as an operator within the given context. For example, array dimension tokens ([ ]) may not be nested.
C2427	*'identifier'* **: jump referencing label is out of range**
	The specified change of control is farther than allowed.

D.1.3 Warning Messages

The messages listed in this section indicate potential problems but do not hinder compiling and linking. The number in parentheses at the end of an error-message description gives the minimum warning level that must be set for the message to appear.

Number	Warning Message
C4000	**UNKNOWN WARNING** **Contact Microsoft Technical Support** The compiler detected an unknown error condition. Please report this condition to Microsoft Corporation, using the Product Assistance Request form at the back of this manual.
C4002	**too many actual parameters for macro '***identifier***'** The number of actual arguments specified with the given identifier was greater than the number of formal parameters given in the macro definition of the identifier. (1)
C4003	**not enough actual parameters for macro '***identifier***'** The number of actual arguments specified with the given identifier was less than the number of formal parameters given in the macro definition of the identifier. (1)
C4004	**missing close parenthesis after 'defined'** The closing parenthesis was missing from an **#if defined** phrase. (1)
C4005	**'***identifier***' : redefinition** The given identifier was redefined. (1)
C4006	**#undef expected an identifier** The name of the identifier whose definition was to be removed was not given with the **#undef** directive. (1)
C4009	**string too big, trailing chars truncated** A string exceeded the compiler limit on string size. To correct this problem, break the string into two or more strings. (1)
C4011	**identifier truncated to '***identifier***'** Only the first 31 characters of an identifier are significant. (1)

Number	Warning Message
C4012	**float constant in a cross compilation**

The compiler found a **float** constant in a file being compiled for a different processor. Floating-point constants may be represented differently on different processors. (1)

C4014 **'*identifier*' : bit-field type must be unsigned**

The given bit field was not declared as an **unsigned** type.

Bit fields must be declared as **unsigned** integral types. The compiler converted the given bit field accordingly. (1)

C4015 **'*identifier*' : bit-field type must be integral**

The given bit field was not declared as an integral type.

Bit fields must be declared as **unsigned** integral types. A conversion has been supplied. (1)

C4016 **'*identifier*' : no function return type, using 'int' as default**

The given function had not yet been declared or defined, so the return type was unknown.

The default return type (**int**) is assumed. (2)

C4017 **cast of int expression to far pointer**

A far pointer represents a full segmented address. On an 8086/8088 processor, casting an **int** value to a far pointer may produce an address with a meaningless segment value. (1)

C4020 **'*identifier*' : too many actual parameters**

The number of arguments specified in a function call was greater than the number of parameters specified in the argument-type list or function definition. (1)

C4021 **'*identifier*' : too few actual parameters**

The number of arguments specified in a function call was less than the number of parameters specified in the argument-type list or function definition. (1)

C4022 **'*identifier*' : pointer mismatch: parameter *n***

The pointer type of the given parameter was different from the pointer type specified in the argument-type list or function definition. (1)

Number	Warning Message
C4024	*'identifier'* **: different types : parameter** *n*

The type of the given parameter in a function call did not agree with the type given in the argument-type list or function definition. (1)

C4026 **function was declared with formal argument list**

The function was declared to take arguments, but the function definition did not declare formal parameters. (1)

C4027 **function was declared without formal argument list**

The function was declared to take no arguments (the argument-type list consisted of the word **void**), but formal parameters were declared in the function definition, or arguments were given in a call to the function. (1)

C4028 **parameter** *n* **declaration different**

The type of the given parameter did not agree with the corresponding type in the argument-type list or with the corresponding formal parameter. (1)

C4029 **declared parameter list different from definition**

The argument-type list given in a function declaration did not agree with the types of the formal parameters given in the function definition. (1)

C4030 **first parameter list is longer than the second**

A function was declared more than once with different argument-type lists in the declarations. (1)

C4031 **second parameter list is longer than the first**

A function was declared more than once with different argument-type lists. (1)

C4032 **unnamed struct/union as parameter**

The type of the structure or union being passed as an argument was not named, so the declaration of the formal parameter cannot use the name and must declare the type. (1)

C4033 **function must return a value**

A function is expected to return a value unless it is declared as **void**. (2)

Number	Warning Message
C4034	**sizeof returns 0**

The **sizeof** operator was applied to an operand that yielded a size of zero. (1)

C4035 *identifier* **: no return value**

A function declared to return a value did not do so. (2)

C4036 **unexpected formal parameter list**

A formal-parameter list was given in a function declaration. The formal-parameter list is ignored. (1)

C4037 **'***identifier***' : formal parameters ignored**

No storage class or type name appeared before the declarators of formal-parameters in a function declaration, as in the following example:

```
int *f(a,b,c);
```

The formal parameters are ignored. (1)

C4038 **'***identifier***' : formal parameter has bad storage class**

The given formal parameter was declared with a storage class other than **auto** or **register**. (1)

C4039 **'***identifier***' : function used as an argument**

A formal parameter to a function was declared to be a function, which is illegal. The formal parameter is converted to a function pointer. (1)

C4040 **near/far/huge on '***identifier***' ignored**

The **near**, **far**, or **huge** keyword has no effect in the declaration of the given identifier and is ignored. (1)

C4041 **formal parameter '***identifier***' is redefined**

The given formal parameter was redefined in the function body, making the corresponding actual argument unavailable in the function. (1)

C4042 **'***identifier***' : has bad storage class**

The specified storage class cannot be used in this context (for example, function parameters cannot be given **extern** class). The default storage class for that context was used in place of the illegal class. (1)

Number	Warning Message

C4044 **huge on '*identifier*' ignored, must be an array**

The compiler ignored the **huge** keyword on the specified identifier. Only arrays may be declared **huge**. (1)

C4045 **'*identifier*' : array bounds overflow**

Too many initializers were present for the given array. The excess initializers are ignored. (1)

C4046 **'&' on function/array, ignored**

An attempt was made to apply the address-of operator (&) to a function or array identifier. (1)

C4047 **'*operator*': different levels of indirection**

An expression involving the specified operator had inconsistent levels of indirection. (1)

The following example illustrates this condition:

```
char **p;
char *q;
.
.
.
p = q;
```

C4048 **array's declared subscripts different**

An array was declared twice with different sizes. The larger size is used. (1)

C4049 **'*operator*' : indirection to different types**

The indirection operator (*) was used in an expression to access values of different types. (1)

C4051 **data conversion**

Two data items in an expression had different types, causing the type of one item to be converted. During the conversion, a data item was truncated. (2)

C4053 **at least one void operand**

An expression with type **void** was used as an operand. (1)

Number	Warning Message

C4058 **address of frame variable taken, DS != SS**

The program was compiled with the default data segment (DS) not equal to the stack segment (SS), and the program tried to point to a frame variable with a near pointer. (1)

C4060 **conversion of long address to short address**

The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) resulted in the loss of the segment address. (1)

C4061 **long/short mismatch in argument: conversion supplied**

The base types of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)

C4062 **near/far mismatch in argument: conversion supplied**

The pointer sizes of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)

C4067 **unexpected characters following '*directive*' directive - newline expected**

Extra characters followed a preprocessor directive, as in the following example:

```
#endif    NO_EXT_KEYS
```

This is accepted in some versions of the Microsoft C Compiler, but not in Version 2.0 of the Microsoft QuickC Compiler. (1)

C4068 **unknown pragma**

The compiler did not recognize a pragma and ignored it. (1)

C4069 **conversion of near pointer to long integer**

The compiler has converted a 16-bit near pointer to a long integer, which involves first extending the high-order word with the current data-segment value, not 0. (1)

C4071 **'*identifier*' : no function prototype given**

The given function was called before the compiler found the corresponding function prototype. (3)

Note that although C4071 is a level 3 warning, the compiler issues it any time you compile with the /Gi option but omit function prototypes.

Number	Warning Message

C4074 **non standard extension used - '*extension*'**

The given nonstandard language extension was used when the Language Extensions option in the Compile dialog box was turned off, or when the /Ze option was in effect. (If the /Za option is in effect, this condition generates an error.) (3)

C4075 **size of switch expression or case constant too large - converted to int**

A value appearing in a **switch** or **case** statement was larger than an **int** type. The compiler converts the illegal value to an **int** type. (1)

C4076 **'*type*' : may be used on integral types only**

The **signed** or **unsigned** type modifier was used with a nonintegral type. (1)

The following example shows this condition:

```
unsigned double x;
```

C4077 **unknown check_stack option**

An unknown option was given with the old form of the **check_stack** pragma, as in the following example:

```
#pragma check_stack yes
```

In the old form of the **check_stack** pragma, the argument to the pragma must be empty, +, or –. (1)

C4079 **unexpected token '*token*'**

The compiler encountered a new-line character in a position where it expected to find some other token. (1)

C4082 **expected an identifier, found '*token*'**

There was a missing identifier in the list of arguments to a pragma. (1)

C4083 **expected '(', found '*token*'**

An opening left parenthesis was missing from the argument list for a pragma. (1)

The following line causes this error:

```
#pragma check_pointer on)
```

C4084 **expected a pragma keyword found '*token*'**

The token following the keyword **pragma** was not an identifier. (1)

The following illustrates this error:

```
#pragma (on)
```

Number	Warning Message

C4085 **expected [on | off]**

An invalid argument was given for the new form of the **check_stack** pragma. (1)

The following is an example of this error:

```
#pragma check_stack(yes)
```

C4086 **expected [1 | 2 | 4]**

An invalid argument was given for a **pack** pragma, as in the following example (1):

```
#pragma pack(yes)
```

C4087 **'*name*' : declared with 'void' parameter list**

The given function was declared as taking no parameters, but a call to the function specified actual parameters, as in the following example (1):

```
int f1(void);
      .
      .
      .
f1(10);
```

C4088 **'*name*' : pointer mismatch : parameter *n*, parameter list *number***

The *n*th argument in the given function call has a different level of indirection, as in the following example (1):

```
int (*sample (void *)) (void *);
.
.
.
        main()
        {
            sample(10) (10); /* pointer mismatch: parameter 1,
                                parameter list 2 */
        }
```

C4089 **'*name*' : different types : parameter *n*, parameter list *number***

The argument in the given function call did not have the same type as the argument in the function prototype, as in the following example (1):

```
int (*sample(int,int))(char *);
   .
   .
   .
```

Number Warning Message

```
main()
   {
      int i;
      (*sample(10,20))(i);/* pointer mismatch : parameter 1,
                             parameter list 2. */
   }
```

C4090 **different '*const*' attributes**

A pointer to an item declared as **const** was passed to a function where the corresponding formal parameter was a pointer to a non-**const** item. This means the item could be modified by the function undetected. (1)

The following illustrates this condition:

```
const char *p = "abcde";
int str(char *s);
.
.
.
str(p);
```

C4091 **no symbols were declared**

The compiler detected an empty declaration, as in the following example (2):

```
int ;
```

C4092 **untagged enum/struct/union declared no symbols**

The compiler detected an empty declaration using an untagged structure, union, or enumerated variable, as in the following example (2):

```
struct {
       .
       .
       .
       };
```

C4093 **unescaped newline in character constant in non-active code**

The constant expression of an **#if**, **#elif**, **#ifdef**, or **#ifndef** preprocessor directive evaluated to 0, making the following code inactive, and a new-line character appeared between a single or double quotation mark and the matching single or double quotation mark in that inactive code. (3)

C4095 **expected ')', found '*token*'**

More than one argument appeared for a pragma that takes only one argument. (1)

Number	Warning Message

C4098 **void function returning a value**

A function declared with a **void** return type also returned a value, as in the following example (1):

```
void func()
{
        .
        .
        .
        return(10);
}
```

C4100 **'*name*' : unreferenced formal parameter**

The given formal parameter was never referenced in the body of the function for which it was declared. (3)

C4101 **'*name*' : unreferenced local variable**

The given local variable was never used. (3)

C4102 **'*name*' : unreferenced label**

The given label was defined but never referenced. (3)

C4103 **'*name*' : function definition used as prototype**

A function definition appeared before its prototype in the program. (3)

C4105 ***name* : code modifiers only on function or pointer to function**

The given modifier was used to declare something other than a function or function pointer. (1)

C4109 **unexpected identifier '*token*'**

The line contains an unexpected token. (1)

C4110 **unexpected token 'int constant'**

The line contains an unexpected integer constant. (1)

C4111 **unexpected token 'string'**

The line contains an unexpected string. (1)

Number Warning Message

C4112 **macro name '*string*' is reserved, *command* ignored**

Using the given preprocessor command, you attempted to define a predefined macro name or the preprocessor operator **define**. This warning also occurs if you attempt to undefine a predefined macro name. If you attempt to define or undefine a predefined macro name using command-line options, *command* is still either **#define** or **#undef**. (1)

C4113 **function parameter lists differed**

You assigned a function pointer to a function pointer, but the parameter lists of the functions do not agree, as in the following example (1):

```
int (*sample) (int);
    int (*example) (char, char);
    main()
    {
        sample = example;
    }
```

C4114 **same type qualifier used more than once**

A type qualifier (**const**, **volatile**, **signed**, or **unsigned**) was used more than once in the same type. (1)

C4115 **'*tag*' : type definition in formal parameter list**

A structure, union, or enumerated type was defined or declared in the formal-parameter list of a function, as in the following example (1):

```
int funct1 (enum color {red, green, blue} col);
```

C4116 **'<no tag>' : type definition in formal parameter list**

A structure, union, or enumerated type was defined or declared in the formal-parameter list of a function, but no tag was given, as in the following example (3):

```
int funct1 (enum {red, green, blue} col);
```

C4118 **pragma not supported**

The pragma specified is not supported by QuickC, so the compiler ignored it. (1)

C4401 **'*identifier*' : member is bitfield**

The *identifier* is a bitfield. (1)

Number	Warning Message
C4402	**must use PTR operator** A type was used on an operand without a **PTR** operator. (1)
C4403	**illegal PTR operator** A type was used on an operand with the **PTR** operator. (1)
C4404	**period on directive ignored** The period preceding the directive was ignored. (2)
C4405	'*identifier*' **: identifier is reserved word** The identifier is a reserved word. (1)
C4406	**operand on directive ignored** The directive does not take any operands; however, an operand was specified. (1)
C4407	**operand size conflict** The size of the operands should match but didn't. (2)
C4408	'*label*' **: ambiguous label** The label referenced was ambiguous. This warning occurs when a label is defined twice in C source code with the same spelling but different capitalization and then referenced in in-line assembler. (1)
C4409	**illegal instruction size** The instruction does not have a form with the specified size. (1)
C4410	**illegal size for operand** One of the operands on this instruction has an incorrect size. (1)
C4411	'*identifier*' **: symbol resolves to displacement register** The identifier is a local symbol that resolves to a displacement register and therefore may be used on an operand with another symbol. (1)
C4412	'*identifier*' **: identifier is also assembler mnemonic** The given identifier is the same as a mnemonic instruction for the assembler. (1)

Number	Warning Message
C4413	*'function'* **redefined: preceding references may be invalid**

The compiler issues this error if a function definition changes between incremental compilations. For example:

```
main ()
{
func1 ();
}
int func1 ()
{
.
.
.
}
```

If you compile this program with the /Gi option, then change the definition of `func1` to `long func1`, the compiler issues this waring message to let you know that calls to `func1` may be of the wrong type.

You should be sure that the function calls reference the correct type; if not, recompile. To avoid the problem altogether, use function prototypes.

D.1.4 Compiler Limits

To operate the Microsoft QuickC Compiler, you must have sufficient disk space available for the compiler to create temporary files that are used in processing. The space required is approximately two times the size of the source file.

Table D.1 summarizes the limits imposed by the QuickC compiler. If your program exceeds one of these limits, an error message will inform you of the problem.

Table D.1 Limits Imposed by the QuickC Compiler

Program Item	Description	Limit
Constants	Maximum size of a constant is determined by its type; see Table A.1, "Range of Values of C Data Types," in the *C For Yourself* manual for details.	
Identifiers	Maximum length of an identifier	31 bytes (additional characters are discarded)

Table D.1 *(continued)*

Program Item	Description	Limit
Declarations	Maximum level of nesting for structure/union definitions	10 levels
Preprocessor directives	Maximum size of a macro definition	1024 bytes
	Maximum number of macro definitions in /D options	20 definitions
	Maximum number of formal parameters to a macro definition	31 parameters
	Maximum length of an actual preprocessor argument	256 bytes
	Maximum level of nesting for **#if**, **#ifdef**, and **#ifndef** directives	32 levels
	Maximum level of nesting for include files, counting the open source file	10 levels
	Maximum number of search paths for include files	20 paths

The compiler does not set explicit limits on the length of a string or on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

D.2 Command-Line Errors

Messages that indicate errors on the command line used to invoke the compiler have one of the following formats:

`command line error D2xxx:` *messagetext*	Error
`command line warning D4xxx:` *messagetext*	Warning

The compiler issues a warning message and, if possible, continues processing. In some cases, command-line errors are fatal and the compiler terminates processing.

D.2.1 Command-Line Error Messages

When the QCL compiler encounters any of the errors listed in this section, it terminates, producing no object file.

Number	Command-Line Error Message

D2000 **UNKNOWN COMMAND LINE ERROR**
Contact Microsoft Technical Support

The compiler detected an unknown error condition.

Please report this condition to Microsoft Corporation, using the Product Assistance Request form at the back of this manual.

D2001 **too many symbols predefined with -D**

Too many symbolic constants were defined by using the /D option on the command line.

The normal limit on command-line definitions is 15; you can use the /U or /u option to increase the limit to 20.

D2002 **a previously defined model specification has been overridden**

Two different memory models were specified; the model specified later on the command line was used.

D2003 **missing source file name**

You did not give the name of the source file to be compiled.

D2008 **too many** *option* **flags, '***string***'**

Too many letters were given with the specified option (for example, with the /O option).

Number	Command-Line Error Message
D2011	**only one floating point model allowed**
	You specified more than one floating-point (/FP) option on the command line.
D2012	**too many linker flags on command line**
	You tried to pass more than 128 separate options and object files to the linker.
D2015	**assembly files are not handled**
	You gave a file name with an extension of .ASM on the command line.
	Because the compiler cannot invoke the Microsoft Macro Assembler (MASM) automatically, it cannot assemble such files.
D2016	**'*option1*' and '*option2*' are incompatible**
	The two options conflict and may not be used together.
D2018	**cannot open linker cmd file**
	The response file used to pass object-file names and options to the linker could not be opened.
	This error may have occurred because another read-only file had the same name as the response file.
D2019	**cannot overwrite the source/object file, '*name*'**
	You specified the source file as an output-file name.
	The compiler does not allow the source file to be overwritten by one of the compiler output files.
D2020	**-Gc option requires extended keywords to be enabled (-Ze)**
	The /Gc option and the /Za option were specified on the same command line.
	The /Gc option requires the extended keyword **cdecl** to be enabled if library functions are to be accessible.
D2021	**invalid numerical argument '*string*'**
	A non-numerical string was specified following an option that required a numerical argument.
D2022	**cannot open help file, qcl.hlp**
	The /HELP option was given, but the file containing the help messages (QCL.HLP) was not in the current directory or in any of the directories specified by the PATH environment variable.

Number	Command-Line Error Message
D2027	**could not execute '*filename*'**

The compiler could not find the given file in the current working directory or in any of the other directories named in the PATH variable.

D2028 **too many open files, cannot redirect '*filename*'**

No more file handles were available to redirect the output of the /P option to a file.

Try editing your CONFIG.SYS file and increasing the value *num* on the line `files=`*num* (if *num* is less than 20).

D.2.2 Command-Line Warning Messages

The messages listed in this section indicate potential problems, but the errors do not hinder compilation and linking.

Number	Command-Line Warning Message
D4000	**UNKNOWN COMMAND LINE WARNING Contact Microsoft Technical Support**

An unknown command-line error condition has been detected by the compiler.

Please report this condition to Microsoft Corporation, using the Product Assistance Request form at the back of this manual.

D4002 **ignoring unknown flag '*string*'**

One of the options given on the command line was not recognized and is ignored.

D4003 **Different processors selected for code generation**

Both the /G0 option and the /G2 option were given; /G2 takes precedence.

D4005 **could not execute '*filename*';
Please enter new file name (full path) or Ctrl -C to quit:**

The QCL command could not find the specified executable file in the search path.

D4006 **only one of -P/-E/-EP allowed, -P selected**

More than one preprocessor output option was specified.

D4007 **-C ignored (must also specify -P or -E or -EP)**

The /C option must be used in conjunction with one of the preprocessor output flags, /E, /EP, or /P.

Number	Command-Line Warning Message
D4009	**threshold only for far/huge data, ignored** The /Gt option was used in a memory model that has near data pointers. It can be used only in compact, large, and huge models.
D4013	**combined listing has precedence over object listing** When /Fc is specified along with either /Fl or /Fa, the combined listing (/Fc) is created.
D4014	**invalid value** *number* **for '***string***'. Default** *number* **is used** An invalid value was given in a context where a particular numeric value was expected.

D.3 Run-Time Errors

Run-time error messages fall into four categories:

1. Floating-point exceptions generated by the 8087/80287 hardware or the emulator.
2. Error messages generated by the run-time library to notify you of serious errors.
3. Error messages generated by program calls to error-handling routines in the C run-time library—the **abort**, **assert**, and **perror** routines. These routines print an error message to standard error output whenever the program calls the given routine.
4. Error messages generated by calls to math routines in the C run-time library. On error, the math routines return an error value, and some print a message to the standard error output. See the on-line help or the *Microsoft C Run-Time Library Reference* (sold separately) for descriptions of the math routines and corresponding error messages.

D.3.1 Floating-Point Exceptions

The error messages listed below correspond to exceptions generated by the 8087/80287 hardware. Refer to the Intel documentation for your processor for a detailed discussion of hardware exceptions. These errors may also be detected by the floating-point emulator built into the standard QuickC library.

Using C's default 8087/80287 control-word settings, the following exceptions are masked and do not occur:

Exception	Default Masked Action
Denormal	Exception masked
Underflow	Result goes to 0.0
Inexact	Exception masked

The following errors do not occur with code generated by the Microsoft QuickC Compiler or provided in the standard C library:

```
Square root
Stack underflow
Unemulated
```

The floating-point exceptions have the following format:

```
run-time error M61nn: MATH
- floating-point error: messagetext
```

The floating-point exceptions are listed and described below:

Number	Floating-Point Exception

M6101 **invalid**

An invalid operation occurred. These usually involve operating on NANs or infinities. This error terminates the program with exit code 129.

M6102 **denormal**

A very small floating-point number was generated and may no longer be valid due to loss of significance. Denormals are normally masked, causing them to be trapped and operated on. This error terminates the program with exit code 130.

M6103 **divide by 0**

An attempt was made to divide by 0. This error terminates the program with exit code 131.

M6104 **overflow**

An overflow occurred in a floating-point operation. This error terminates the program with exit code 132.

M6105 **underflow**

An underflow occurred in a floating-point operation. (An underflow is normally masked so that the underflowing value is replaced with 0.0.) This error terminates the program with exit code 133.

M6106 **inexact**

Loss of precision occurred in a floating-point operation. This exception is normally masked because almost any floating-point operation can cause loss of precision. This error terminates the program with exit code 134.

M6107 **unemulated**

An attempt was made to execute an 8087/80287 instruction that is invalid or not supported by the emulator. This error terminates the program with exit code 135.

Number	Floating-Point Exception

M6108 **square root**

The operand in a square-root operation was negative. This error terminates the program with exit code 136. (Note: the **sqrt** function in the C run-time library checks the argument before performing the operation and returns an error value if the operand is negative; see the on-line help for details on **sqrt**.)

M6110 **stack overflow**

A floating-point expression caused a stack overflow on the 8087 or 80287 co-processor or the emulator. (Stack-overflow exceptions are trapped up to a limit of seven levels in addition to the eight levels normally supported by the 8087 or 80287 coprocessor.) This error terminates the program with exit code 138.

M6111 **stack underflow**

A floating-point operation resulted in a stack underflow on the 8087 or 80287 co-processor or the emulator. This error terminates the program with exit code 139.

M6112 **explicitly generated**

A signal indicating a floating-point error was sent using a `raise(SIGFPE)` call. This error terminates the program with exit code 140.

D.3.2 Run-Time Error Messages

The following messages may be generated at run time when your program has serious errors. Run-time error-message numbers range from R6000 to R6999.

A run-time error message takes the following general form:

`run-time error R6`*`nnn`*`-` *`messagetext`*

Number	Run-Time Error Message

R6000 **stack overflow**

Your program has run out of stack space. This can occur when a program uses a large amount of local data or is heavily recursive. The program was terminated with an exit code of 255.

To correct the problem, recompile using the /F option of the QCL command, or relink using the linker /STACK option to allocate a large stack.

Number	Run-Time Error Message

R6001 **null pointer assignment**

The contents of the NULL segment have changed in the course of program execution. The NULL segment is a special location in low memory that is not normally used. If the contents of the NULL segment change during a program's execution, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. Note that your program can contain null pointers without generating this message; the message appears only when you access a memory location through the null pointer.

This error does not cause your program to terminate; the error message is printed following the normal termination of the program. This error yields a nonzero exit code.

This message reflects a potentially serious error in your program. Although a program that produces this error may appear to operate correctly, it is likely to cause problems in the future and may fail to run in a different operating environment.

R6002 **floating point not loaded**

Your program needs the floating-point library, but the library was not loaded. The error causes the program to be terminated with an exit code of 255. This occurs in two situations:

- The program was compiled or linked with an option (such as /FPi87) that required an 8087 or 80287 coprocessor, but the program was run on a machine that did not have a coprocessor installed. To fix this problem, either recompile the program with the /FPi option or install a coprocessor.
- A format string for one of the routines in the **printf** or **scanf** families contains a floating-point format specification and there are no floating-point values or variables in the program. The QuickC compiler attempts to minimize the size of a program by loading floating-point support only when necessary. Floating-point format specifications within format strings are not detected, so the necessary floating-point routines are not loaded. To correct this error, use a floating-point argument to correspond to the floating-point format specification. This causes floating-point support to be loaded.

R6003 **integer divide by 0**

An attempt was made to divide an integer by 0, giving an undefined result. This error terminates the program with an exit code of 255.

Number	Run-Time Error Message

R6005 **not enough memory on exec**

Errors R6005 through R6007 occur when a child process spawned by one of the **exec** library routines fails and DOS could not return control to the parent process. This error indicates that not enough memory remained to load the program being spawned.

R6006 **bad format on exec**

The file to be executed by one of the **exec** functions was not in the correct format for an executable file.

R6007 **bad environment on exec**

During a call to one of the **exec** functions, DOS determined that the child process was being given a bad environment block.

R6008 **not enough space for arguments**

R6009 **not enough space for environment**

Errors R6008 and R6009 both occur at start-up if there is enough memory to load the program, but not enough room for the *argv* vector, the *envp* vector, or both. To avoid this problem, rewrite the **_setargv** or **_setenvp** routine.

R6012 **illegal near pointer use**

A null near pointer was used in the program.

This error occurs only if pointer checking is in effect (that is, if the program was compiled with the Pointer Check option in the Compile dialog box, the /Zr option on the QCL command line, or the **pointer_check** pragma in effect).

R6013 **illegal far pointer use**

An out-of-range far pointer was used in the program.

This error occurs only if pointer checking is in effect (that is, if the program was compiled with the Pointer Check option in the Compile dialog box, the /Zr option on the QCL command line, or the **pointer_check** pragma in effect).

R6014 **control-BREAK encountered**

You pressed CTRL+BREAK to stop the execution of a program within the QuickC environment.

Number	Run-Time Error Message
R6015	**unexpected interrupt**

The program could not be run because it contained unexpected interrupts.

When you create a program from a program list in the QuickC environment, QuickC automatically creates object files and passes them to the linker. If you compile with the Debug option turned on, the object files that QuickC passes to the linker contain interrupts that are required within the QuickC environment. However, you cannot run a program created from such object files outside of the QuickC programming environment.

D.3.3 Run-Time Limits

Table D.2 summarizes the limits that apply to programs at run time. If your program exceeds one of these limits, an error message will inform you of the problem.

Table D.2 Program Limits at Run Time

Item	Description	Limit
Files	Maximum file size	$2^{32}-1$ bytes (4 gigabytes)
	Maximum number of open files (streams)	20[a]
Command line	Maximum number of characters (including program name)	128
Environment table	Maximum size	32K

[a] Five streams are opened automatically (**stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**), leaving 15 files available for the program to open.

D.4 LINK Error Messages

This section lists and describes error messages generated by the Microsoft Overlay Linker (LINK), and the special incremental linker (ILINK) that is invoked when you compile QuickC programs with the /Gi or /Li option. Note that in most cases, QuickC will invoke LINK if the ILINK fails.

Fatal errors cause the linker to stop execution. Fatal error messages have the following format:

location `: error L1`*xxx*`:` *messagetext*

For LINK, fatal error numbers range from L1000 to L1199. For the incremental linker, fatal error numbers range from L1200 to L1249.

Nonfatal errors indicate problems in the executable file. LINK produces the executable file. Nonfatal error messages have the following format:

location `: error L2`*xxx*`:` *messagetext*

Nonfatal errors generated by the incremental linker are numbered from L1250 to L1299.

Warnings indicate possible problems in the executable file. LINK produces the executable file. Warnings have the following format:

location `: warning L4`*xxx*`:` *messagetext*

Warning numbers less than L4200 apply to LINK. Those numbers greater than L4200 apply to the incremental linker.

In all three kinds of messages, *location* is the input file associated with the error, or `LINK` if there is no input file. If the input file is an .OBJ or .LIB file, and a module name is associated with the error, the module name is enclosed in parentheses, as shown in the following examples:

```
SLIBC.LIB(_file)
MAIN.OBJ(main.c)
TEXT.OBJ
```

The following error messages may appear when you link object files:

Number	LINK Error Message
L1001	*option* **: option name ambiguous**

A unique option name did not appear after the option indicator (/). For example, the command

```
LINK /N main;
```

generates this error because LINK cannot tell which of the options beginning with the letter "N" was intended.

Number	LINK Error Message
L1002	*option* : **unrecognized option name** An unrecognized character followed the option indicator (/), as in the following example: `LINK /ABCDEF main;`
L1003	**/QUICKLIB, /EXEPACK incompatible** Quick libraries cannot be compressed using the /EXEPACK linker option.
L1004	*option* : **invalid numeric value** An incorrect value appeared for one of the linker options. For example, a character string was given for an option that requires a numeric value.
L1005	**/PACKCODE : packing limit exceeds 65536 bytes** The value supplied with the /PACKCODE option exceeds the limit of 65,536 bytes.
L1006	*option-text* : **stack size exceeds 65535 bytes** The value given as a parameter to the /STACKSIZE option exceeds the maximum allowed.
L1007	*option* : **interrupt number exceeds 255** A number greater than 255 was given as a value for the /OVERLAYINTERRUPT option.
L1008	*option* : **segment limit set too high** The /SEGMENTS option specified a limit greater than 3072 on the number of segments allowed.
L1009	*number* : **CPARMAXALLOC : illegal value** The *number* specified in the /CPARMAXALLOC option was not in the range 1–65,535.
L1020	**no object modules specified** No object-file names were specified to the linker.
L1021	**cannot nest response files** A response file occurred within a response file.

Number	LINK Error Message
L1022	**response line too long**

A line in a response file was longer than 127 characters.

L1023 **terminated by user**

You entered CTRL+C.

L1024 **nested right parentheses**

The contents of an overlay were typed incorrectly on the command line.

L1025 **nested left parentheses**

The contents of an overlay were typed incorrectly on the command line.

L1026 **unmatched right parenthesis**

A right parenthesis was missing from the contents specification of an overlay on the command line.

L1027 **unmatched left parenthesis**

A left parenthesis was missing from the contents specification of an overlay on the command line.

L1043 **relocation table overflow**

More than 32,768 long calls, long jumps, or other long pointers appeared in the program.

Try replacing long references with short references where possible, then recreate the object module.

L1045 **too many TYPDEF records**

An object module contained more than 255 TYPDEF records. These records describe communal variables. This error can appear only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables. (TYPDEF is a DOS term. It is explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)

L1046 **too many external symbols in one module**

An object module specified more than the limit of 1023 external symbols.

Break the module into smaller parts.

L1047 **too many group, segment, and class names in one module**

Reduce the number of groups, segments, or classes, and recreate the object file.

Number	LINK Error Message
L1048	**too many segments in one module**
	An object module had more than 255 segments.
	Split the module or combine segments.
L1049	**too many segments**
	The program had more than the maximum number of segments. (The /SEGMENTS option specifies the maximum legal number; the default is 128.)
	Relink by using the /SEGMENTS option with an appropriate number of segments.
L1050	**too many groups in one module**
	LINK encountered more than 21 group definitions (GRPDEF) in a single module.
	Reduce the number of group definitions or split the module. (Group definitions are explained in the *Microsoft MS-DOS Programmer's Reference* and in other reference books on DOS.)
L1051	**too many groups**
	The program defined more than 20 groups, not counting DGROUP.
	Reduce the number of groups.
L1052	**too many libraries**
	An attempt was made to link with more than 32 libraries.
	Combine libraries or use modules that require fewer libraries.
L1053	**out of memory for symbol table**
	The program had more symbolic information (such as public, external, segment, group, class, and file names) than could fit in available memory.
	Try freeing memory by linking from the DOS command level instead of from an NMAKE file or an editor. Otherwise, combine modules or segments and try to eliminate as many public symbols as possible.

Number	LINK Error Message

L1054 **requested segment limit too high**

The linker did not have enough memory to allocate tables describing the number of segments requested. (The default is 128 or the value specified with the /SEGMENTS option.)

Try linking again by using the /SEGMENTS option to select a smaller number of segments (for example, use 64 if the default was used previously), or free some memory by eliminating resident programs or shells.

L1056 **too many overlays**

The program defined more than 63 overlays.

L1057 **data record too large**

A LEDATA record (in an object module) contained more than 1024 bytes of data. This is a translator error. (LEDATA is a DOS term, which is explained in the *Microsoft MS-DOS Programmer's Reference* and in other DOS reference books.)

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances under which the error occurred. Please report this error to Microsoft Corporation using the Microsoft Product Assistance Request form at the back of one of your manuals.

L1061 **out of memory for /INCREMENTAL**

The linker ran out of memory when trying to process the additional information required for ILINK support.

If you were linking from within an editor or NMAKE, try linking directly.

L1062 **too many symbols for /INCREMENTAL**

The program had more symbols than can be stored in the .SYM file.

Reduce the number of symbols.

L1063 **out of memory for CodeView information**

The linker was given too many object files with debug information, and the linker ran out of space to store the debug information.

Reduce the number of object files that have debug information.

Number	LINK Error Message
L1070	*name* : **segment size exceeds 64K**
	A single segment contained more than 65,535 bytes of code or data.
	Try compiling and linking using the large model.
L1071	**segment _TEXT larger than 65520 bytes**
	This error is likely to occur only in small-model C programs, but it can occur when any program with a segment named _TEXT is linked using the /DOSSEG option of the LINK command. Small-model C programs must reserve code addresses 0 and 1; this range is increased to 16 for alignment purposes.
	Try compiling and linking using the large model.
L1072	**common area longer than 65536 bytes**
	The program had more than 64K of communal variables. This error occurs only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.
L1073	**file-segment limit exceeded**
	The number of physical segments exceeds the limit of 254 imposed when linking incrementally.
	Reduce the number of segments or group more of them, and use the /PACKCODE option.
L1074	*name* : **group larger than 64K bytes**
	The given group exceeds the limit of 65,536 bytes.
	Reduce the size of the group, or remove any unneeded segments from the group (refer to the map file for a listing of segments).
L1080	**cannot open list file**
	The disk or the root directory was full.
	Delete or move files to make space.
L1081	**out of space for run file**
	The disk on which the .EXE file was being written was full.
	Free more space on the disk and restart the linker.

Number	LINK Error Message
L1082	*filename* : **stub not found**

The linker cannot find the special stub file that is required for incremental linking under QuickC. The given file must be on the current path or in the current directory. For QuickC, the special stub file is named ILINKSTB.OVL, and it is shipped as part of the QuickC package.

L1083 **cannot open run file**

The disk or the root directory was full.

Delete or move files to make space.

L1084 **cannot create temporary file**

The disk or root directory was full.

Free more space in the directory and restart the linker.

L1085 **cannot open temporary file**

The disk or the root directory was full.

Delete or move files to make space.

L1086 **scratch file missing**

An internal error has occurred.

Note the circumstances of the problem and contact Microsoft Corporation using the Microsoft Product Assistance Request form at the back of one of your manuals.

L1087 **unexpected end-of-file on scratch file**

The disk with the temporary linker-output file was removed.

L1088 **out of space for list file**

The disk (where the listing file was being written) is full.

Free more space on the disk and restart the linker.

L1089 *filename* : **cannot open response file**

LINK could not find the specified response file. This usually indicates a typing error.

Number	LINK Error Message
L1090	**cannot reopen list file**
	The original disk was not replaced at the prompt.
	Restart the linker.
L1091	**unexpected end-of-file on library**
	The disk containing the library was probably removed.
	Replace the disk containing the library and run the linker again.
L1093	*file* **: object not found**
	One of the object files specified in the linker input was not found.
	Restart the linker and specify the object file.
L1094	*file* **: cannot open file for writing**
	The linker was unable to open the file with write permission.
	Check file permissions.
L1095	*file* **: out of space on file**
	The linker ran out of disk space for the specified output file.
	Delete or move files to make space.
L1100	**stub .EXE file invalid**
	The special stub file ILINKSTB.OVL, which is required for incremental linking with QuickC, has somehow become corrupted and is not a valid DOS executable file.
	Restore the file from your distribution disk.
L1101	**invalid object module**
	One of the object modules was invalid.
	If the error persists after recompiling, please contact Microsoft Corporation using the Microsoft Product Assistance Request form at the back of one of your manuals.
L1102	**unexpected end-of-file**
	An invalid format for a library was encountered.

Number LINK Error Message

L1103 **attempt to access data outside segment bounds**

A data record in an object module specified data extending beyond the end of a segment. This is a translator error.

Note which translator (compiler or assembler) produced the incorrect object module and the circumstances in which it was produced. Please report this error to Microsoft Corporation using the Microsoft Product Assistance Request form at the back of one of your manuals.

L1104 *filename* **: not valid library**

The specified file was not a valid library file. This error causes LINK to abort.

L1105 **invalid object due to aborted incremental compile**

The object file is incomplete, and therefore cannot be linked, because an error occurred during the incremental compile.

Correct the error, recompile, and then relink.

L1113 **unresolved COMDEF; internal error**

This is an internal error. Note the circumstances of the failure and contact Microsoft Corporation using the Software Problem Report form at the back of any manual.

L1114 **file not suitable for /EXEPACK; relink without**

For the linked program, the size of the packed load image plus packing overhead was larger than that of the unpacked load image.

Relink without the /EXEPACK option.

L1123 *name* **: segment defined both 16- and 32-bit**

You defined the same segment with the **use32** attribute in one module and with the **use16** attribute in another module.

Define the segment with the same attribute in both places.

L1127 **far segment references not allowed with /BINARY**

The /BINARY option produces a .COM file, which is not permitted to have any run-time relocations. For example, the following code causes this error:

```
mov     ax, seg mydata
```

Number	LINK Error Message
L1200	**SYM seek error**
L1202	**SYM write error**
	Errors L1200 and L1202 both have the same cause. Either the disk is full or the .SYM file already exists and has the READONLY attribute.
L1203	**map for segment** *name* **exceeds 64K**
	The symbolic information associated with the given segment exceeds 64K bytes, an amount greater than ILINK can handle.
L1204	**.ILK write error**
	The disk is full or the .SYM file already exists and has the READONLY attribute.
L1205	**fixup overflow at** *address* **in segment** *name*
	A FIXUPP object record with the given location referred to a target too far away to be correctly processed. This message indicates an error in translation by the compiler or assembler.
L1206	**.ILK seek error**
	The .ILK file is corrupted.
	Recompile without the /Gi or /Li option; or use LINK directly to perform a full link.
	If the error persists, note the circumstance of the error and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
L1207	**.ILK file too large**
	The .ILK file is too large for the incremental linker to process.
	Recompile without the /Gi or /Li option or use LINK directly to perform a full link.
L1208	**invalid .SYM file**
	The file *program* .SYM is invalid for the given program.
	To correct the problem, do a full (not incremental) link of the program. If the error persists, contact Microsoft using one of the Product Assistance Request forms at the back of a manual.

Number	LINK Error Message

L1209 **.OBJ close error**

The operating system returned an error when the linker attempted to close one of the .OBJ files.

L1210 **.OBJ read error**

The .OBJ file has an unreadable structure. This message indicates an error in translation by the compiler or assembler.

Try to rebuild the .OBJ file by recompiling without the /Gi or /Li option.

L1211 **too many LNAMES**

An object module has more than 255 LNAME records.

L1212 **too many SEGDEFs**

The given object module has more than 100 SEGDEF records. A SEGDEF record defines logical segments.

L1213 **too many GRPDEFs**

The given object module has more than 10 GRPDEF records. A GRPDEF record defines physical segments.

L1214 **too many COMDEFS**

L1215 **too many EXTDEFS**

The total number of COMDEF and EXTDEF records exceeded the limit. The limit on the total of COMDEF records (communal data variables) and EXTDEF records (external references) is 1023.

Use fewer communal or external variables in your program.

L1216 **symbol *name* multiply defined**

The given symbol is defined more than once.

L1217 **internal error #3**

Note the circumstance of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

Number	LINK Error Message
L1218	**.EXE file too big, change alignment**

The segment-sector alignment value in the .EXE file is too small to express the size of one of the segments.

Recompile without the /Gi or /Li option, or use LINK directly to perform a full link. You may also need to increase the alignment value with the /ALIGNMENT option to LINK.

L1219 **too many library files**

The number of libraries exceeded the limit of 32 libraries (.LIB files).

Reduce the number of libraries.

L1220 **seek error on library**

A library (.LIB file) is corrupted.

Check your .LIB files for consistency, and perform a full (not incremental) link.

L1221 **library close error**

The operating system returned an error when the linker attempted to close one of the libraries (.LIB files).

Do a full link. If the error persists, note the circumstance of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.

L1223 **could not update time on** *file*

The operating system returned an error when ILINK attempted to update the time on the given file. Possibly the file had the READONLY attribute set.

L1224 **invalid flag** *character*

A command-line option contained an invalid character.

L1225 **only one -e command allowed**

You used incorrect syntax on the command line.

L1226 **terminated by user**

You pressed CTRL+C or CTRL+BREAK, which interrupts and terminates the linker.

L1227 **file** *name* **write protected**

The .EXE, .ILK, or .SYM file that ILINK attempted to update has the READONLY attribute.

Number	LINK Error Message
L1228	**file** *name* **missing**

The linker could not find one of the .OBJ files specified on the command line.

L1229 **invalid .OBJ format**

There may be one of several problems: error in compiler translation, object file was corrupted, object file not valid (possibly text file), or object file could not be read or found.

L1230 **invalid** *file* **record: position** = *address*

The given .OBJ file has an invalid format or one that is not recognized by the linker. This message may indicate an error in translation by the compiler or assembler.

L1231 **file** *name* **was not full linked**

You specified an .OBJ file in the ILINK command line that was not in the list of files in the most recent full link.

L1232 **cannot run** *program*

ILINK is unable to execute a program specified for execution with the /e command-line option.

Make sure the program is in the search path and is an .EXE or .COM file.

L1233 *program* **returned** *return-code*

The given program was specified with the /e option. When ILINK executed this program, it terminated with the given nonzero return code. ILINK cannot continue to the next commands, if any.

L1234 **error creating** *file*

ILINK was unable to create the batch file for executing the /e commands.

Make sure the directory given in TMP or TEMP, or the current directory, exists and can be written to.

L1235 **error writing to** *file*

ILINK experienced an error while writing the batch file for executing the /e commands.

Make sure the drive for TMP or TEMP or the current drive has enough free space.

Number LINK Error Message

L1240 **far references in STRUC fields not supported**

ILINK currently does not support STRUC definitions like the following:

```
extrn   func:FAR
    rek     STRUC
    far_adr DD      func    ; Initialized far address
                            ;   within a STRUC
    rek     ENDS
```

Change your code to get rid of the far address within the STRUC. Alternatively, do not attempt to incrementally link.

L1241 **too many defined segments**

ILINK has a limit of 255 physical segments (that is, segments defined in the object module as opposed to groups or logical segments).

Reduce the number of segments and relink. Alternatively, do not use the /Li or /Gi option.

L1242 **too many modules**

The program exceeds ILINK's limit of 1204 modules. Reduce the number of modules.

L1243 **cannot link 64K-length segments**

The program has a segment larger than 65,535 bytes.

L1244 **cannot link iterated segments**

ILINK cannot handle programs linked with the /EXEPACK linker option.

L1250 *number* **undefined symbols**

A number of symbols were referred to in fixups but never publicly defined in the program. The given number indicates how many of these undefined symbols were found.

L1251 **invalid module reference** *library*

The program makes a dynamic-link reference to a dynamic-link library that is not recognized or declared by the .EXE file.

L1252 **file** *name* **does not exist**

The linker could not find the given file.

Number	LINK Error Message
L1253	**symbol** *name* **deleted** A symbol was deleted from an incrementally linked module.
L1254	**new segment definition** *name* A segment was added to the program.
L1255	**changed segment definition** *name* The segment contribution changed for the given module; it contributed to a segment to which it did not previously contribute, or a segment contribution was removed.
L1256	**segment** *name* **grew too big** The given segment grew beyond the padding for the given module.
L1257	**new group definition** *name* A new group was defined, via the GROUP directive in assembly language or via the C compiler option /ND.
L1258	**group** *name* **changed to include** *segment* The list of segments included in the given group changed.
L1259	**symbol** *name* **changed** The given data symbol moved (is now at a new address).
L1260	**cannot add new communal data symbol** *name* A new communal data symbol was added, as an uninitialized variable in C or with the COMM feature in MASM.
L1261	**communal variable** *name* **grew too big** The given communal variable changed size too much.
L1262	**invalid symbol type for** *symbol* A symbol that was previously a code symbol became a data symbol or vice versa.
L1263	**new Codeview symbolic info** A module previously compiled without /Zi was compiled with /Zi.
L1264	**new line-number info** A module previously compiled without /Zi or /Zd was compiled with /Zi or /Zd.

Number	LINK Error Message
L1265	**new public CodeView info**
	New information on public symbol addresses was added.
L1266	**invalid .EXE file**
	The .EXE file is invalid. Make sure you are using an up-to-date linker.
	If the error persists, note the circumstance of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
L1267	**invalid .ILK file**
	The .ILK file is invalid. Make sure you are using an up-to-date linker.
	If the error persists, notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
L1268	**.SYM/.ILK mismatch**
	The .SYM and .ILK files are out of sync. Make sure you are using an up-to-date linker. If the error persists, note the circumstance of the failure and notify Microsoft Corporation by following the directions in the Microsoft Product Assistance Request form at the back of one of your manuals.
L1269	**library *name* has changed**
	The given library has changed.
L1270	**entry table expansion not implemented**
	The program call tree changed in such a way that ILINK could not process it correctly.
L1271	**segment *index* with relocs exceeds 64K; cannot move**
	The given segment, referred to by its index within the program's segment table, is too big along with its run-time relocations for ILINK to process the segment correctly.
L1272	**.ILK read error**
	The .ILK file does not exist or was not in the expected format.

Number LINK Error Message

L1273 **out of memory**

ILINK ran out of memory for processing the input.

If you are linking incrementally from within an NMAKE description file, try linking from DOS command level instead. Otherwise, do a full link.

L2001 **fixup(s) without data**

A FIXUPP record occurred without a data record immediately preceding it. This is probably a compiler error. (See the *Microsoft MS-DOS Programmer's Reference* for more information on FIXUPP.)

If the error persists after recompiling, please contact Microsoft Corporation using the Microsoft Product Assistance Request form at the back of one of your manuals.

L2002 **fixup overflow at** *number* **in segment** *segname*

The following conditions can cause this error:

- A group is larger than 64K.
- The program contains an intersegment short jump or intersegment short call.
- The name of a data item in the program conflicts with that of a library subroutine included in the link.
- An **EXTRN** declaration in an assembly-language source file appeared inside the body of a segment, as in the following example:

```
code    SEGMENT public 'CODE'
        EXTRN   main:far
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

The following construction is preferred:

```
EXTRN    main:far
code    SEGMENT public 'CODE'
start   PROC    far
        call    main
        ret
start   ENDP
code    ENDS
```

Revise the source file and re-create the object file. (For information about frame and target segments, see the *Microsoft MS-DOS Programmer's Reference*.)

Number	LINK Error Message
L2003	**intersegment self-relative fixup at** *number* **in segment** *segname*

An intersegment self-relative fixup is not allowed.

L2005 **fixup type unsupported at** *number* **in segment** *segname*

A fixup type occurred that is not supported by the Microsoft linker. This is probably a compiler error.

Note the circumstances of the failure and contact Microsoft Corporation using the Microsoft Product Assistance Request form at the back of one of your manuals.

L2010 **too many fixups in LIDATA record**

The number of far relocations (pointer- or base-type) in a LIDATA record exceeds the limit imposed by the linker. This is typically produced by the **DUP** statement in an .ASM file. The limit is dynamic: a 1024-byte buffer is shared by relocations and the contents of the LIDATA record; there are eight bytes per relocation.

Reduce the number of far relocations in the **DUP** statement.

L2011 *name* **: NEAR/HUGE conflict**

Conflicting **near** and **huge** attributes were given for a communal variable. This error can occur only with programs produced by the Microsoft FORTRAN Compiler or other compilers that support communal variables.

L2012 *name* **: array-element size mismatch**

A far communal array was declared with two or more different array-element sizes (for instance, an array was declared once as an array of characters and once as an array of real numbers). This error occurs only with the Microsoft FORTRAN Compiler and any other compiler that supports far communal arrays.

L2013 **LIDATA record too large**

A LIDATA record contained more than 512 bytes. This is probably a compiler error.

Note the circumstances of the failure and contact Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.

Number	LINK Error Message

L2024 *name* **: special symbol already defined**

Your program defined a symbol name already used by the linker for one of its own low-level symbols. (For example, the linker generates special symbols used in overlay support and other operations.)

Choose another name for the symbol in order to avoid conflict.

L2025 *name* **: symbol defined more than once**

The given symbol was defined more than once.

L2029 *name* **: unresolved external**

One or more symbols were declared to be external in one or more modules, but they were not publicly defined in any of the modules or libraries. The symbol name at the start of the message is the unresolved external symbol. This message is also written to the map file, if one exists.

L2041 **stack plus data exceed 64K**

If the total of near data and requested stack size exceeds 64K, the program will not run correctly. The linker checks for this condition only when /DOSSEG is enabled, which is the case in the library start-up module.

Reduce the stack size.

L2043 **QuickLibrary support module missing**

When creating a Quick library, you did not link with the required QUICKLIB.OBJ object module.

L2044 *name* **: symbol multiply defined, use /NOE**

The linker found what it interprets as a public-symbol redefinition, probably because you have redefined a symbol defined in a library.

Relink with the /NOEXTDICTIONARY (/NOE) option. If error L2025 results for the same symbol, then you have a genuine symbol-redefinition error.

L2045 *segname* **: segment with >1 class name not allowed with /INC**

When linking incrementally, segments cannot be defined with more than one class. The following segment definition causes this error:

```
MYCODE  segment word public 'mycode-class'
MYCODE  ends

MYCODE  segment word public 'code-class'
MYCODE  ends
```

Number	LINK Error Message
L2048	**Microsoft Overlay Manager module not found**

A needed module could not be found for the Microsoft Overlay Manager.

L4000 **seg disp. included near *offset* in segment *name***

This is the warning generated by the /WARNFIXUP option. See Section 5.5.6, "Fixups," for more information.

L4001 **frame-relative fixup, frame ignored near *offset* in segment *name***

A reference is made relative to a segment that is different from the target segment of the reference. For example, if `_foo` is defined in segment `_TEXT`, the instruction `call DGROUP:_foo` produces this warning. The frame DGROUP is ignored, so the linker treats the call as if it were `call _TEXT:_foo`.

L4002 **frame-relative absolute fixup near *offset* in segment *name***

A reference is made similar to the type described in `L4001`, but both segments are absolute (defined with AT). The linker treats the executable file as if the file were to run in real mode only.

L4003 **intersegment self-relative fixup at *offset* in segment *name***

The linker found an intersegment self-relative fixup. This error may be caused by compiling a small-model program with the /NT option.

L4010 **invalid alignment specification**

The number specified in the /ALIGNMENT option must be a power of 2 in the range 2–32,768, inclusive.

L4011 **PACKCODE value exceeding 65500 unreliable**

The packing limit specified with the /PACKCODE option was between 65,500 and 65,536. Code segments with a size in this range are unreliable on some steppings of the 80286 processor.

L4012 **load-high disables /EXEPACK**

The /HIGH and /EXEPACK options cannot be used at the same time.

L4013 **invalid option for new-format executable file ignored**

You specified an option that is incompatible with incremental linking. The conflicting options are /CPARMAXALLOC, /DSALLOCATION, and /NOGROUPASSOCIATION. This message also appears if you specify overlays when linking incrementally.

Number	LINK Error Message
L4014	*option* **option ignored for realmode executable file**

The given option does not apply to real-mode executable files, so the linker ignored it.

L4015 **/CODEVIEW disables /DSALLOCATE**

The /CODEVIEW and /DSALLOCATE options cannot be used at the same time.

L4016 **/CODEVIEW disables /EXEPACK**

The /CODEVIEW and /EXEPACK options cannot be used at the same time.

L4020 *name* **: code-segment size exceeds 65500**

Code segments of 65,501–65,536 bytes in length may be unreliable on the Intel 80286 processor.

L4021 **no stack segment**

The program did not contain a stack segment defined with the **STACK** combine type. This message should not appear for modules compiled with the Microsoft FORTRAN Compiler, but it could appear for an assembly-language module.

Normally, every program should have a stack segment with the combine type specified as **STACK**. You may ignore this message if you have a specific reason for not defining a stack or for defining one without the **STACK** combine type. Linking with versions of LINK earlier than Version 2.40 might cause this message because these linkers search libraries only once.

L4022 *group1*, *group2* **: groups overlap**

The named groups overlap. Because a group is assigned to a physical segment, groups cannot overlap.

Reorganize segments and group definitions so that the groups do not overlap. Refer to the map file. Normally this message applies only to OS/2 and Windows programs. If you receive this message when linking a DOS file, the incremental linker is probably being used because you specified either /Gi or /Li on the QCL command line. Unless you change your program as instructed above, you cannot link incrementally.

L4029 *name* **: DGROUP segment converted to type data**

The given logical segment in the group DGROUP was defined as a code segment. (DGROUP cannot contain code segments because the linker always considers DGROUP to be a data segment. The name DGROUP is predefined as the automatic data segment.) The linker converts the named segment to type "data."

Number	LINK Error Message
L4031	*name* : **segment declared in more than one group** A segment was declared to be a member of two different groups. Correct the source file and create new object files.
L4032	*name* : **code-group size exceeds 65500 bytes** The given code group has a size between 65,500 and 65,536 bytes, a size that is unreliable on some steppings of the 80286 processor.
L4034	**more than 239 overlay segments; extra put in root** Your program designated more than the limit of 239 segments to go in overlays. Starting with the 234th segment, they are assigned to the root (that is, the permanently resident portion of the program).
L4036	**no automatic data segment** The application did not define a group named DGROUP. DGROUP has special meaning to the linker, which uses it to identify the automatic or default data segment used by the operating system. Most OS/2 protected-mode and Windows applications require DGROUP. This warning is not issued if DATA NONE is declared or if the executable file is a dynamic-link library. If you get this message when linking a DOS program, the incremental linker is probably being used. If you are using the QCL command, try omitting the /Gi and /Li options.
L4038	**program has no starting address** The linker can find no starting address for the program. If you get this message when linking a DOS program, the incremental linker is probably being used. Try compiling with neither the /Gi nor /Li option.
L4039	**memory model mismatch** Linked objects were compiled with different memory models.
L4050	**too many public symbols for sorting** The linker uses the stack and all available memory in the near heap to sort public symbols for the /MAP option. If the number of public symbols exceeds the space available for them, this warning is issued and the symbols are not sorted in the map file but listed in an arbitrary order.

Number	LINK Error Message
L4051	*filename* **: cannot find library** The linker could not find the specified file. Enter a new file name, a new path specification, or both.
L4053	**VM.TMP : illegal file name; ignored** VM.TMP appeared as an object-file name. This is the name the linker uses for its temporary file. Rename the file and rerun the linker.
L4054	*filename* **: cannot find file** The linker could not find the specified file. Enter a new file name, a new path specification, or both.
L4201	**fixup frame relative to an (as yet) undefined symbol - assuming ok** See documentation for LINK error messages L4001 and L4002.
L4202	**module contains TYPEDEFs - ignored** The .OBJ file contains type definitions. ILINK ignores these records.
L4203	**module contains BLKDEFs - ignored** The .OBJ file contains records no longer supported by Microsoft language compilers.
L4204	**old .EXE free information lost** The free list in the .EXE file has been corrupted. The free list represents "holes" in the .EXE file made available when segments moved to new locations.
L4205	**file** *name* **has no useful contribution** The given module makes no contribution to any segment.
L4206	**main entry point moved** The program starting address changed. You may want to consider doing a full link.

D.5 LIB Error Messages

Error messages generated by the Microsoft Library Manager, LIB, have one of the following formats:

{*filename* | `LIB`} : `fatal error U1`*xxx*: *messagetext*
{*filename* | `LIB`} : `warning U4`*xxx*: *messagetext*

The message begins with the input-file name (*filename*), if one exists, or with the name of the utility. If possible, LIB prints a warning and continues operation. In some cases, errors are fatal and LIB terminates processing. LIB may display the following error messages:

Number	LIB Error Message
U1150	**page size too small** The page size of an input library was too small, which indicates an invalid input .LIB file.
U1151	**syntax error : illegal file specification** A command operator such as a minus sign (–) was given without a following module name.
U1152	**syntax error : option name missing** A forward slash (/) was given without an option after it.
U1153	**syntax error : option value missing** The /PAGESIZE option was given without a value following it.
U1154	**option unknown** An unknown option was given. Currently, LIB recognizes the /PAGESIZE, /NOEXTDICTIONARY, /NOIGNORECASE, and /IGNORECASE options.
U1155	**syntax error : illegal input** The given command did not follow correct LIB syntax as specified in Chapter 6, "LIB."
U1156	**syntax error** The given command did not follow correct LIB syntax as specified in Chapter 6, "LIB."

Number LIB Error Message

U1157 **comma or new line missing**

A comma or carriage return was expected in the command line but did not appear. This may indicate an inappropriately placed comma, as in the following line:

```
LIB math.lib,-mod1+mod2;
```

The line should have been entered as follows:

```
LIB math.lib -mod1+mod2;
```

U1158 **terminator missing**

Either the response to the "Output library" prompt or the last line of the response file used to start LIB did not end with a carriage return.

U1161 **cannot rename old library**

LIB could not rename the old library to have a .BAK extension because the .BAK version already existed with read-only protection.

Change the protection on the old .BAK version.

U1162 **cannot reopen library**

The old library could not be reopened after it was renamed to have a .BAK extension.

U1163 **error writing to cross-reference file**

The disk or root directory was full.

Delete or move files to make space.

U1170 **too many symbols**

More than 4609 symbols appeared in the library file.

U1171 **insufficient memory**

LIB did not have enough memory to run.

Remove any shells or resident programs and try again, or add more memory.

U1172 **no more virtual memory**

The current library exceeds the 512K limit imposed by LIB.

Reduce the number of object modules.

Number	LIB Error Message
U1173	**internal failure**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1174	**mark: not allocated**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1175	**free: not allocated**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1180	**write to extract file failed**
	The disk or root directory was full.
	Delete or move files to make space.
U1181	**write to library file failed**
	The disk or root directory was full.
	Delete or move files to make space.
U1182	*filename* **: cannot create extract file**
	The disk or root directory was full, or the specified extract file already existed with read-only protection.
	Make space on the disk or change the protection of the extract file.
U1183	**cannot open response file**
	The response file was not found.
U1184	**unexpected end-of-file on command input**
	An end-of-file character was received prematurely in response to a prompt.
U1185	**cannot create new library**
	The disk or root directory was full, or the library file already existed with read-only protection.
	Make space on the disk or change the protection of the library file.

Number	LIB Error Message
U1186	**error writing to new library**
	The disk or root directory was full.
	Delete or move files to make space.
U1187	**cannot open VM.TMP**
	The disk or root directory was full.
	Delete or move files to make space.
U1188	**cannot write to VM**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1189	**cannot read from VM**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1200	*name* **: invalid library header**
	The input library file had an invalid format. It was either not a library file, or it had been corrupted.
U1203	*name* **: invalid object module near** *location*
	The module specified by *name* was not a valid object module.
U2152	*filename* **: cannot create listing**
	The directory or disk was full, or the cross-reference-listing file already existed with read-only protection.
	Make space on the disk or change the protection of the cross-reference-listing file.
U2155	*modulename* **: module not in library; ignored**
	The specified module was not found in the input library.
U2157	*filename* **: cannot access file**
	LIB was unable to open the specified file.
U2158	*libraryname* **: invalid library header; file ignored**
	The input library had an incorrect format.

Number	LIB Error Message
U2159	*filename* : **invalid format** *hexnumber*; **file ignored**

The signature byte or word *hexnumber* of the given file was not one of the following recognized types: Microsoft library, Intel library, Microsoft object, or XENIX archive.

U4150 *modulename* : **module redefinition ignored**

A module was specified to be added to a library, but a module with the same name was already in the library, or a module with the same name was found more than once in the library.

U4151 *'name'* : **symbol defined in module** *name*, **redefinition ignored**

The specified symbol was defined in more than one module.

U4153 *number1*:*number2* : **page size too small; ignored**

The value specified in the /PAGESIZE option was less than 16.

U4155 *modulename* : **module not in library**

A module specified to be replaced does not already exist in the library. LIB adds the module anyway.

U4156 **output-library specification ignored**

An output library was specified in addition to a new library name. For example, specifying

```
LIB new.lib+one.obj,new.lst,new.lib
```

where `new.lib` does not already exist, causes this error.

U4157 **insufficient memory, extended dictionary not created**

Insufficient memory prevented LIB from creating an extended dictionary. The library is still valid, but the linker will not be able to take advantage of the extended dictionary to speed linking.

U4158 **internal error, extended dictionary not created**

An internal error prevented LIB from creating an extended dictionary. The library is still valid, but the linker will not be able to take advantage of the extended dictionary to speed linking.

D.6 NMAKE Error Messages

Error messages from the NMAKE utility have one of the following formats:

{*filename* | NMAKE} : fatal error U1*xxx*: *messagetext*
{*filename* | NMAKE} : warning U4*xxx*: *messagetext*

The message begins with the input-file name (*filename*) and line number, if one exists, or with the name of the utility.

NMAKE generates the following error messages:

Number	NMAKE Error Message

U1000 **syntax error : ')' missing in macro invocation**

A left parenthesis (( ) appeared without a matching right parenthesis ( )) in a macro invocation. The correct form is $(*name*), or $*n* for one-character names.

U1001 **syntax error : illegal character '*character*' in macro**

A nonalphanumeric character other than an underscore (_) appeared in a macro.

U1002 **syntax error : bad macro invocation '$'**

A single dollar sign ($) appeared without a macro name associated with it. The correct form is $(*name*). To use a dollar sign in the file, type it twice or precede it with a caret (^).

U1003 **syntax error : '=' missing in macro**

The equal sign (=) was missing in a macro definition. The correct form is '*name* = *value*'.

U1004 **syntax error : macro name missing**

A macro invocation appeared without a name. The correct form is $(*name*).

U1005 **syntax error : text must follow ':' in macro**

A string substitution was specified for a macro, but the string to be changed in the macro was not specified.

U1016 **syntax error : closing '"' missing**

An opening double quotation mark (") appeared without a closing double quotation mark.

U1017 **unknown directive '*directive*'**

The directive specified is not one of the recognized directives.

Number	NMAKE Error Message
U1018	**directive and/or expression part missing**
	The directive is incompletely specified. The expression part is required.
U1019	**too many nested if blocks**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1020	**EOF found before next directive**
	A directive, such as !ENDIF, was missing.
U1021	**syntax error : else unexpected**
	An !ELSE directive was found that was not preceded by !IF, !IFDEF, or !IFNDEF, or was placed in a syntactically incorrect place.
U1022	**Missing terminating char for string/program invocation : '*character*'**
	The closing double quotation mark (") in a string comparison in a directive was missing, or the closing bracket (]) in a program invocation in a directive was missing.
U1023	**syntax error present in expression**
	An expression is invalid. Check the allowed operators and operator precedence.
U1024	**illegal argument to !CMDSWITCHES**
	An unrecognized command switch was specified.
U1031	**file name missing**
	An include directive was found, but the name of the file to include was missing.
U1033	**syntax error : '*string*' unexpected**
	The specified string is not part of the valid syntax for a makefile.
U1034	**syntax error : separator missing**
	The colon (:) that separates target(s) and dependent(s) is missing.
U1035	**syntax error : expected separator or '='**
	Either a colon (:), implying a dependency line, or an equal sign (=), implying a macro definition, was expected.

Number	NMAKE Error Message
U1036	**syntax error : too many names to left of '='**
	Only one string is allowed to the left of a macro definition.
U1037	**syntax error : target name missing**
	A colon (:) was found before a target name was found. At least one target is required.
U1038	**internal error : lexer**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1039	**internal error : parser**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1040	**internal error : macro-expansion**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1041	**internal error : target building**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1042	**internal error : expression stack overflow**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1043	**internal error : temp file limit exceeded**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1044	**internal error : too many levels of recursion building a target**
	Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.

Number	NMAKE Error Message
U1050	*user-specified text* The message specified with the !ERROR directive is displayed.
U1051	*'progname'* **usage : [-acdeinpqrst -f makefile -x stderrfile] [macrodefs] [targets]** An error was made trying to invoke NMAKE. Use the specified form.
U1052	**out of memory** The program ran out of space in the far heap. Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
U1053	**file '***filename***' not found** The file was not found. The file name might not be properly specified in the makefile.
U1054	**file '***filename***' unreadable** The file cannot be read. The file might not have the appropriate attributes for reading.
U1055	**can't create response file '***filename***'** The response file cannot be created.
U1056	**out of environment space** The environment space limit was reached. Restart the program with a larger environment space.
U1057	**can't find command.com** The COMMAND.COM file could not be found.
U1058	**unlink of file '***filename***' failed** Unlink of the temporary response file failed.
U1059	**terminated by user** Execution of NMAKE aborted because you typed CTRL+C or CTRL+BREAK.

Number	NMAKE Error Message
U1070	**cycle in macro definition '***macroname***'** A circular definition was detected in the macro definition specified. This is an invalid definition.
U1071	**cycle in dependency tree for target '***targetname***'** A circular dependency was detected in the dependency tree for the specified target. This is invalid.
U1072	**cycle in include files** *filenames* A circular inclusion was detected in the include files specified. That is, each file includes the other.
U1073	**don't know how to make '***targetname***'** The specified target does not exist and there are no commands to execute or inference rules given for it. Hence it cannot be built.
U1074	**macro definition too long** The macro definition is too long.
U1075	**string too long** The text string would overflow an internal buffer.
U1076	**name too long** The macro name, target name, or build-command name would overflow an internal buffer. Macro names may be at most 128 characters.
U1077	*'program'* **: return code** *value* The given program invoked from NMAKE failed, returning the error code *value*.
U1078	**constant overflow at '***directive***'** A constant in *directive*'s expression was too big.
U1079	**illegal expression: divide by zero present** An expression tries to divide by zero.
U1080	**operator and/or operand out of place: usage illegal** The expression incorrectly uses an operator or operand. Check the allowed set of operators and their precedence.

Number	NMAKE Error Message
U1081	**'*program*' : program not found** NMAKE could not find the given program in order to run it. Make sure that the program is in the current path and has the correct extension.
U1082	**'*command*' : cannot execute command: out of memory** NMAKE cannot execute the given command because there is not enough memory available. Free some memory and run NMAKE again.
U1085	**can't mix implicit and explicit rules** A regular target was specified along with the target for a rule (which has the form *.fromext.toext*). This is invalid.
U1086	**inference rule can't have dependents** Dependents are not allowed when an inference rule is being defined.
U1087	**can't have : and :: dependents for same target** A target cannot have both a single-colon (:) and a double-colon (::) dependency.
U1088	**invalid separator on inference rules: '::'** Inference rules can use only a single-colon (:) separator.
U1089	**can't have build commands for pseudotarget '*targetname*'** Pseudotargets (for example, .PRECIOUS, .SUFFIXES) cannot have build commands specified.
U1090	**can't have dependents for pseudotarget '*targetname*'** The specified pseudotarget (for example, .SILENT, .IGNORE) cannot have a dependent.
U1091	**invalid suffixes in inference rule** The suffixes being used in the inference rule are invalid.
U1092	**too many names in rule** An inference rule cannot have more than one pair of extensions (*.fromext.toext*) as a target.

Number	NMAKE Error Message
U1093	**can't mix special pseudotargets** It is illegal to list two or more pseudotargets together.
U4011	**command file can only be invoked from command line** A command file cannot be invoked from within another command file. Such an invocation is ignored.
U4012	**resetting value of special macro '***macroname***'** The value of a macro such as $(MAKE) was changed within a description file. The name by which this program was invoked is not a tagged section in the TOOLS.INI file.
U4015	**no match found for wildcard '***filename***'** There are no file names that match the specified target or dependent file with the wild-card characters asterisk (*) and question mark (?).
U4016	**too many rules for target '***targetname***'** Multiple blocks of build commands are specified for a target using single colons (:) as separators.
U4017	**ignoring rule** ***rule*** **(extension not in .SUFFIXES)** The rule was ignored because the suffix(es) in the rule are not listed in the .SUFFIXES list.
U4018	**special macro undefined '***macroname***'** The special macro *macroname* is undefined.
U4019	**Filename '***filename***' too long; truncating to 8.3** The base name of the file has more than eight characters or the extension has more than three characters. NMAKE truncates the name to an eight-character base and a three-character extension.
U4020	**removed target '***target***'** Execution of NMAKE was interrupted while it was trying to build the given target, and therefore the target was incomplete. Because the target was not specified in the .PRECIOUS list, NMAKE has deleted it.

D.7 HELPMAKE Error Messages

HELPMAKE generates the following error messages:

Number	HELPMAKE Error Message
H1000	**/A requires character**

The /A option requires an application-specific control character.

The correct form is /A*character*, where *character* is the control character.

H1001 **/E compression level must be numeric**

The /E option requires either no argument or a numeric value.

The correct form is /E*value*, where *value* specifies the amount of compression requested.

H1002 **Multiple /O parameters specified**

Only one output file can be specified with the /O option.

H1003 **Invalid /S filetype identifier**

The /S option requires specification of the type of input file. There was an invalid file-type identifier specified.

The correct form is /S*filetype*, where *filetype* specifies the format of the input help text file. The only valid values are 1 (RTF), 2 (QuickHelp format), and 3 (minimally formatted ASCII).

H1004 **/S requires filetype identifier**

The /S option requires specification of the type of input file. There was no file-type identifier specified.

The correct form is /S*filetype*, where *filetype* specifies the format of the input help text file. The only valid values are 1 (RTF), 2 (QuickHelp format), and 3 (minimally formatted ASCII).

H1005 **/W fixed width invalid**

The /W option requires a width specification. An invalid width was specified. The valid range is 11-255.

H1050 **Improper arguments for /DS**

The /O, /L, and /C options are invalid with the /DS option.

Number	HELPMAKE Error Message
H1051	**Improper arguments for /D**
	The /D option permits either no argument, an "S," or a "U." In addition, it may not be used with /L or /C.
H1052	**Encode requires /O option**
	You have requested data-base encoding without specifying an output-file name for the operation.
H1097	**No operation specified**
	There is no operation specified on the HELPMAKE command line.
	Either the /D or the /E option must be specified.
H1098	**Unknown Switch**
	There is an invalid switch specified on the command line.
H1099	**Syntax error on command line**
	HELPMAKE cannot interpret the command line.
H1100	**Cannot open file**
	One of the files specified on the HELPMAKE command line could not be found or created.
H1101	**Error writing file**
	The output file could not be written, probably because the disk is full.
H1102	**No Input File Specified**
	In an encoding operation, no input help text file was specified.
H1103	**No context strings found**
	No context strings were found in the input stream while encoding. Either the file is empty, or the specified /S value does not correspond to the help text formatting.
H1104	**No topic text found**
	No topic text was found in the help text file. Either the file is empty, or the specified /S value does not correspond to the help text formatting.
H1200	**Insufficient memory to allocate context buffer**
	There is insufficient memory to run HELPMAKE. It requires 256K free memory.

Number	HELPMAKE Error Message
H1201	**Insufficient memory to allocate utility buffer** There is insufficient memory to run HELPMAKE. It requires 256K free memory.
H1250	**Not a valid compressed help file** The input file specified for a decompression operation is not a valid help data-base file.
H1251	**Cannot decompress locked help file** The help data-base file you are attempting to decompress is locked (that is, the /L option was specified when the help file was created).
H1300	**Word too long in RTF processing** A single word was longer than the specified format width (set by /W option).
H1301	**too much back-up required while formatting RTF** While attempting to reformat a paragraph, HELPMAKE had to back up more than 128 characters to find a word break.
H1302	**Attribute stack overflow processing RTF** RTF attributes are nested too deeply. HELPMAKE supports a maximum of 50 levels of attribute nesting.
H1303	**Unknown RTF attribute** An unknown RTF formatting command was encountered.
H1900	**Internal Virtual Memory Error** This message indicates an internal HELPMAKE error. Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.
H1901	**Out of local memory** This message indicates an internal HELPMAKE error. Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.

Number	HELPMAKE Error Message

H1902 **Out of disk space for swap file**

HELPMAKE uses a temporary swapping file which is written to the current drive and directory. That drive is full. The temporary file may grow to 1.5 times the size of the input files (for large help files) and is not removed until the final help file is completed.

H1903 **Cannot open swapping file**

HELPMAKE uses a temporary swapping file, which is written to the current drive and directory. It cannot create the swapping file because the disk drive or directory is full.

H1990 **Character not found in cmpr**

This message indicates an internal HELPMAKE error.

Note the circumstances of the failure and notify Microsoft Corporation by using the Microsoft Product Assistance Request form at the back of one of your manuals.

H2000 **line too long, truncated**

A line exceeded the fixed width specified by the /W option. The extra characters have been truncated.

H2001 **duplicate context string**

The same context string appeared preceding more than one block of topic text. A context string may be associated with one and only one block of topic text.

H4000 **keyword compression analysis table size exceeded.**

This error occurs in conjunction with HELPMAKE error H4001. The maximum number (16,000) of unique keywords has been encountered during keyword compression. This happens only in very large help files. No further keywords will be included in the analysis.

H4001 **No further new words will be analyzed.**

This error occurs in conjunction with HELPMAKE error H4000. There is no more room for keywords in the analysis tables. HELPMAKE continues to analyze how frequently words occur that it has already encountered.

Glossary

The definitions in this glossary are intended primarily for use with this manual. Neither individual definitions nor the list of terms is comprehensive.

8087 or 80287 coprocessor Intel® hardware products that provide very fast and precise floating-point number processing.

ANSI (American National Standards Institute) The national institute responsible for defining programming-language standards to promote portability of these languages between different computer systems.

argument A value passed to a function. In the QuickC Tool Kit, a string or value that modifies the effects of a compiler, linker, or utility option.

arithmetic conversion Conversion operations performed on items of integral and floating-point types used in expressions.

ASCII (American Standard Code for Information Interchange) A set of 256 codes that many computers use to represent letters, digits, special characters, and other symbols. Only the first 128 of these codes are standardized; the remaining 128 are special characters that are defined by the computer manufacturer.

base name The portion of the file name that precedes the file-name extension. For example, `samp` is the base name of the file `samp.c`.

batch file A text file containing DOS commands that can be invoked from the DOS command line.

block A sequence of declarations, definitions, and statements enclosed within braces ({ }).

canonical frame number Part of the starting address for a segment. The canonical frame number specifies the address of the first paragraph in memory that contains one or more bytes of the segment.

child process A new process started by a currently running process.

compact memory model A memory model that allows for more than one data segment and only one code segment.

constant expression Any expression that evaluates to a constant. A constant may involve integer constants, character constants, floating-point constants, enumeration constants, type casts to integral and floating-point types, and other constant expressions.

context A key word or phrase that is recognized by the help system and that defines a topic.

cross-reference A string of text that is associated with a hyperlink or location in the displayed help text. When activated, the cross-reference string may reference another help context or another help file.

declaration A construct that associates the name and the attributes of a variable, function, or type.

decoding The process of decompressing a help-data-base file into its component parts, thereby creating one or more QuickHelp text files.

definition A construct that initializes and allocates storage for a variable or that specifies the name, formal parameters, body, and the return type of a function.

dependency line A line in an NMAKE description file that defines one or more targets and the files they depend on.

dependents The files that, when modified, cause NMAKE to update a target.

description block A dependency line in an NMAKE description file and all the statements (commands, comments, and directives) that apply to it.

description file The text file that NMAKE reads to determine what to do. A description file is also called a makefile.

directive An instruction to the C preprocessor to perform a specific action on source-program text before compilation. For the NMAKE utility, an instruction that gives information about which commands to execute or how to execute them.

emulator A floating-point-math package that provides software emulation of the operations of a math coprocessor.

encoding The process of compressing a help text file into a help data base.

environment variable A variable stored in the environment table that provides DOS with information (where to find executable files and library files, where to create temporary files, etc.).

errorlevel code See "exit code."

escape character A character that, when used immediately preceding a special character, causes the special character to lose its special meaning.

executable image The code and data that make up an executable file; that is, a compiled, linked program that DOS can execute.

exit code A code returned by a program to DOS indicating whether the program ran successfully.

external level The parts of a C program outside the function declarations.

expression A combination of operands and operators that yields a single value.

formal parameters Variables that receive values passed to a function when the function is called.

function A collection of declarations and statements that has a unique name and can return a value.

function body A statement block containing the local variable declarations and statements of a function.

function call An expression that passes control and arguments (if any) to a function.

function declaration A declaration that establishes the name, return type, and storage class of a function that is defined explicitly elsewhere in the program.

function definition A definition that specifies a function's name, its formal parameters, the declarations and statements that define what it does, and (optionally) its return type and storage class.

function prototype A function declaration that includes a list of the names and types of formal parameters in the parentheses following the function name.

global region The area of a source file between the beginning of the file and the first curly brace of a function, or between the ending curly brace of a function and the beginning curly brace of another function. If no edits have occurred within a global region, incremental compilation is usually possible.

heap An area of memory set aside for dynamic allocation by a program.

help screen An application window that displays information on a single topic. The help screen can be sized and may be scrolled onto additional topics.

huge memory model A memory model that allows for more than one code segment. Individual data items may exceed 64K in length

hyperlink A location in the topic text of an on-line-help file to which a cross-reference has been attached.

include file A text file that is merged into another text file through use of the **#include** preprocessor directive.

incremental compilation The compilation mode, specified by the /Gi option to the QCL command, in which only changed functions are recompiled.

inference rule A template that the NMAKE utility follows to update a target in the absence of explicit commands.

internal level The parts of a C program within function declarations.

keyword A word with a special, predefined meaning for the QuickC compiler.

large memory model A memory model that allows for more than one segment of code and more than one segment of data, but with no individual data items spanning a single segment.

level See "internal level"; "external level."

library A file containing compiled modules. Also called an object-code library. The linker extracts modules from the library and combines them with object files to create executable program files. A load library is a library specified in the object-files field as input to the linker. The linker links every module in a load library into the executable file.

load-time-relocation table A table of references, relative to the start of the program, that are resolved when the program is loaded into memory.

loop optimization Optimization that reduces the amount of code executed for each loop iteration in a program, thereby increasing the speed with which the loop executes.

lvalue An expression (such as a variable name) that refers to a memory location and is required as the left-hand operand of an assignment operation or as the single operand of a unary operator.

macro An identifier defined in a **#define** preprocessor directive to represent another series of tokens. For the NMAKE utility, a name defined on the command line or in a description file to represent another string.

medium memory model A memory model that allows for more than one code segment and only one data segment.

member One of the elements of a structure or union.

memory model One of the models that specifies how memory is set up for program code and data. (For descriptions of standard memory models, see "small memory model"; "medium memory model"; "compact memory model"; "large memory model"; "huge memory model.")

minimally formatted ASCII An ASCII text format that defines only contexts and topic text for the help system.

module-description table (MDT) A file created or updated during incremental compilation that saves information about changes to a source file.

NAN (Not a number) The 8087 or 80287 coprocessor generates NANs when the result of an operation cannot be represented in the IEEE format. For example, if you try to add two positive numbers whose sum is larger than the maximum value permitted by the processor, the coprocessor returns a NAN instead of the sum.

new-line character The character used to mark the end of a line in a text file, or the escape sequence (\n) used to represent this character.

null character The ASCII character encoded as the value 0, represented as an escape sequence (\0) in a source file.

null pointer A pointer to nothing, expressed as the value 0.

object code Relocatable machine code, created by a compiler.

object file A file containing relocatable machine code, created as the result of compiling a source file.

object module A component of a library. An object file becomes an object module when it is loaded into a library.

operand A constant or variable value that is manipulated in an expression.

operator One or more symbols that specify how the operand or operands of an expression are manipulated.

overlay Part of a program that is read into memory from disk only if and when it is needed.

parent process A process that generates a child process using one of the **spawn**, **exec**, or **system** families of run-time-library functions.

pointer A variable containing the address of another variable, function, or constant.

pragma An instruction to the compiler to perform an action at compile time.

precedence The relative position of an operator in the hierarchy that determines the order in which expressions are evaluated.

preprocessor A text processor that manipulates the contents of a C source file during the first phase of compilation.

preprocessor directive See "directive."

prototype See "function prototype."

pseudotarget A target, in an NMAKE description file, that is not a file but is used as a label for performing a set of commands.

QCL The command that invokes the Microsoft QuickC Compiler to compile and link programs.

QuickHelp An ASCII text format that supports implicit cross-references, hyperlinks, and screen formatting flags for input to a help data base.

relocatable Not containing absolute addresses; therefore, eligible to be placed in memory at any location.

response file A file that contains command-line arguments or responses to program prompts. Response files may be used as input to LINK, LIB, and NMAKE.

RTF (Rich Text Format) An ASCII text format for storing documents and their format information.

run time The time during which a previously compiled and linked program is executing.

run-time library A file containing the routines needed to implement certain functions of the Microsoft QuickC language.

scope The parts of a program in which an item can be referenced by name. The scope of an item may be limited to the file, function, block, or function prototype in which it appears.

segment An area of memory, less than or equal to 64K long, that contains code or data.

small memory model A memory model that allows for only one code segment and only one data segment.

source file A text file containing C-language code.

stack A dynamically shrinking and expanding area of memory in which data items are stored in consecutive order and removed on a last in, first out basis.

stack probe A short routine called on entry to a function to verify that there is enough room in the program stack to allocate local variables required by the function and, if so, to allocate those variables.

static A storage class that allows variables to keep their values even after the program exits the block in which the variable is declared.

string An array of characters, terminated by a null character (\0).

string constant A string of characters and escape sequences enclosed in double quotation marks (" "). Every string constant is an array of elements of type **char**.

subscript expression An expression, usually used to reference array elements, representing an address that is offset from a specified base address by a given number of positions.

target The object of an NMAKE description block.

topic text The text displayed as a help entry. Topic text may contain up to 64K of encoded text.

type cast An operation in which a value of one type is converted to a value of a different type.

type checking An operation in which the compiler verifies that the operands of an operator are valid or that the actual arguments in a function call are of the same types as the corresponding formal parameters in the function definition and function prototype.

type declaration A declaration that defines the name and members of a structure or union type, or the name and enumeration set of an enumeration type.

unary expression An expression consisting of a single operand preceded or followed by a unary operator.

unary operator An operator that takes a single operand. Unary operators in the C language are the complement operators (– ~ !), indirection operator (*), increment (++) and decrement (– –) operators, address-of operator (&), and **sizeof** operator. The unary plus operator (+) is legal but has no effect.

unresolved reference A reference to a global or external variable or function that cannot be found, either in the modules being linked or in the libraries that are linked with those modules.

white-space character A space, tab, line-feed, carriage-return, form-feed, vertical-tab, or new-line character.

wild card One of the DOS characters (? and *) that can be expanded into one or more characters in file-name references.

A

B

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

MICROSOFT PRODUCT ASSISTANCE REQUEST

Microsoft Product Support Services - Phone (206) 454-2030

Instructions

When you need assistance with a Microsoft product, call our Product Support Services group at (206) 454-2030. So that we can answer your question as quickly as possible, please gather all information that applies to your problem. Note or print out any on-screen messages you get when the problem occurs. Have your manual and product disks close at hand and have all the information requested on this form available when you call.

Diagnosing a Problem

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

1. Can you reproduce the problem?
   ❒ yes ❒ no

2. Does the problem occur with another copy of the original disk of your Microsoft Software?
   ❒ yes ❒ no

3. Does the problem occur with another system (if available)?
   ❒ yes ❒ no

4. If you were running other windowing or memory-resident software at the same time, does the problem also occur when you don't use the other software?
   ❒ yes ❒ no

Product

Product name

Version Number

Registration Number

Software

Operating System

Name/Version number

Windowing Environment

If you were running Microsoft Windows or another windowing environment, give name and number of windowing software:

CD ROM Software

Name/Version number

Other Software

Name/Version number of any other software you were running when problem occurred, including memory-resident software (such as keyboard enhancers or print spoolers):

Hardware

So that we can assist you more effectively, please be prepared to answer the following questions regarding your problem, your software, and your hardware.

Computer

_____ _____
Manufacturer/model Total memory

Floppy-disk drives
Number: ❒ 1 ❒ 2 ❒ Other
Size: ❒ 3 1/2" ❒ 5 1/4"
Number of Sides: ❒ 1 ❒ 2
Density: ❒ Single ❒ Double ❒ Quad
Capacity:
5 1/4": ❒ 160K ❒ 360K ❒ 1.2 megabytes
3 1/2": ❒ 360K ❒ 400K ❒ 720K ❒ 800K
❒ 1.4 megabytes

System Memory

_____ _____
Manufacturer/model Total memory
(If using DOS, you can run CHKDSK to determine the amount of memory available. If using Apple Macintosh Finder, select "About The Finder..." from the Apple menu to determine the amount of memory available.)

Peripherals

Hard Disk

_____ _____
Manufacturer/model Capacity(megabyte)

Printer/Plotter

_____
Manufacturer/model

❒ Serial ❒ Parallel

Printer peripherals, such as font cartridges, downloadable fonts, sheet feeders:

_____

_____

Mouse
Microsoft Mouse: ❒ Bus ❒ Serial ❒ InPort™ ❒ Other

_____
Manufacturer/model

Boards
❒ Add-on RAM board

_____
Manufacturer/model

❒ Graphics-adapter board

_____
Manufacturer/model

❒ Other boards installed

_____
Manufacturer/model

Modem

_____
Manufacturer/model

CD ROM Player

_____
Manufacturer/model

Version of Microsoft MS-DOS® CD ROM Extensions:

_____

Network

Is your system part of a network? ❒ Yes ❒ No

_____
Manufacturer/model

What hardware and software does your network use?

_____

_____

_____

Microsoft Corporation
16011 NE 36th Way
Box 97017
Redmond, WA 98073-9717

Microsoft®
Making it all make sense™

1088 Part No. 04320