C/2™ Version 1.10

Compile, Link and Run

Programming Family

15F0383

International Business Machines Corporation *Armonk, New York 10504*

IBM Program License Agreement

BEFORE OPENING THIS PACKAGE, YOU SHOULD CAREFULLY READ THE FOLLOWING TERMS AND CONDITIONS. OPENING THIS PACKAGE INDICATES YOUR ACCEPTANCE OF THESE TERMS AND CONDITIONS. IF YOU DO NOT AGREE WITH THEM, YOU SHOULD PROMPTLY RETURN THE PACKAGE UNOPENED AND YOUR MONEY WILL BE REFUNDED.

This is a license agreement and not an agreement for sale. IBM owns, or has licensed from the owner, copyrights in the Program. You obtain no rights other than the license granted you by this Agreement. Title to the enclosed copy of the Program, and any copy made from it, is retained by IBM. IBM licenses your use of the Program in the United States and Puerto Rico. You assume all responsibility for the selection of the Program to achieve your intended results and for the installation of, use of, and results obtained from, the Program.

The Section in the enclosed documentation entitled "License Information" contains additional information concerning the Program and any related Program Services.

LICENSE

You may:

1) use the Program on only one machine at any one time, unless permission to use it on more than one machine at any one time is granted in the License Information (Authorized Use);
2) make a copy of the Program for backup or modification purposes only in support of your Authorized Use. However, Programs marked "Copy Protected" limit copying;
3) modify the Program and/or merge it into another program only in support of your Authorized Use; and
4) transfer possession of copies of the Program to another party by transferring this copy of the IBM Program License Agreement, the License Information, and all other documentation along with at least one complete, unaltered copy of the Program. You must, at the same time, either transfer to such other party or destroy all your other copies of the Program, including modified copies or portions of the Program merged into other programs. Such transfer of possession terminates your license from IBM. Such other party shall be licensed, under the terms of this Agreement, upon acceptance of this Agreement by its initial use of the Program.

You shall reproduce and include the copyright notice(s) on all such copies of the Program, in whole or in part.

You shall not:

1) use, copy, modify, merge, or transfer copies of the Program except as provided in this Agreement;
2) reverse assemble or reverse compile the Program;
   and/or
3) sublicense, rent, lease, or assign the Program or any copy thereof.

LIMITED WARRANTY

Warranty details and limitations are described in the Statement of Limited Warranty which is available upon request from IBM, its Authorized Dealer or its approved supplier and is also contained in the License Information. IBM provides a three-month limited warranty on the media for all Programs. For selected Programs, as indicated on the outside of the package, a limited warranty on the Program is available. The applicable Warranty Period is measured from the date of delivery to the original user as evidenced by a receipt.

Certain Programs, as indicated on the outside of the package, are not warranted and are provided "AS IS."

Continued on inside back cover.

84X1712

C/2™ Version 1.10

Compile, Link and Run

Programming Family

First Edition (September 1988)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

Operating System/2 is a trademark of the International Business Machines Corporation.

C/2 is a trademark of the International Business Machines Corporation.

All rights reserved. No part of this publication may be reproduced or distributed in any form or by any means without prior permission in writing from the International Business Machines Corporation.

Preface

This book is Volume 2 of a four-volume set explaining the IBM C/2 compiler. It gives you information you need to write, compile, link, and run a program with IBM C/2. Sample sessions guide you through the steps of compiling, linking, and running a sample program that is included on one of the compiler diskettes.

This book assumes that first-time users of IBM C/2 have completed at least one year of computer science studies. This book is also intended for experienced applications programmers or system programmers. Users should also be familiar with their personal computer and operating system.

The following table lists some common tasks you may want information about and which book you can find the information in.

If You Want To...	**Refer to...**
Install IBM C/2	*Fundamentals*
Learn basic facts about IBM C/2	*Fundamentals*
Learn the format of a function	*Language Reference*
Understand error messages	*Compile, Link, and Run*
Debug a program	*Debug*
Compile a program	*Compile, Link, and Run*
Link a program	*Compile, Link, and Run*
Write a program	*Fundamentals* and *Language Reference.*

Related Publications

The following books cover topics related to the IBM C/2 Library:

- *IBM C/2 Compile, Link, and Run*
- *IBM C/2 Language Reference*
- *IBM Debug*
- *IBM MASM/2 Fundamentals*
- *IBM MASM/2 Assemble, Link, and Run*
- *IBM MASM/2 Language Reference*
- IBM Operating System/2 Version 1.00 (Standard and Extended Editions)
  - *Programmer's Guide*
- IBM Operating System/2 Version 1.10
  - *Programming Guide*
- The technical reference for your personal computer.
- The technical reference for your operating system.
- *IBM System Application Architecture Common Programming Interface C Reference*
- *iAPX 86, 88 User's Manual*, Copyright 1981, Intel Corp., Santa Clara, CA.
- *iAPX 286 Hardware Reference Manual*, Copyright 1983, Intel Corp., Santa Clara, CA.
- *iAPX 286 Programmer's Reference Manual*, Copyright 1985, Intel Corp., Santa Clara, CA.

Contents

Summary of Changes

Following are the differences between IBM C/2 Version 1.00 and Version 1.10 that affect this book.

Technical Changes

MAKE
The IBM MAKE/2 1.10 utility supports most of the functions of IBM MAKE/2 1.00 with the following enhancements:

- Uses the tools initialization file, TOOLS.INI.
- Accepts command-prompt arguments from a file.
- Does not require build lines to begin with a TAB.
- Does not require a blank line to separate a target's build lines from the next target in the makefile.
- Uses MAKEFILE as the default MAKE description file.
- Recognizes single-character macros without parentheses.
- Supports special macros.
- Supports substitution sequences in macro invocations.
- Allows you to redefine macros so that different targets can use different values for the same macro (this prevents MAKE from supporting forward referencing).
- Provides global filename character expansion in targets.
- Supports include files.
- Supports conditionals and other directives.
- Supports path searching for dependents.
- Allows both / and \ as path separators.
- Allows you to explicitly generate an input script file from within a MAKE description file.
- Provides a means of overriding option settings for specific targets or for changing applicable command option settings from within the MAKE description file.
- Allows the double colon (::) as a separator between targets and dependents.
- Allows you to redirect **stderr** by means of a command option.

Modifications

The IBM MAKE/2, Version 1.00, is not fully compatible with Version 1.10. IBM MAKE/2, Version 1.10, provides basic target and dependency features. Targets are no longer evaluated sequentially. When you invoke NMAKE, the targets you specify are updated regardless of their positions in the file. If you do not specify targets, NMAKE updates the first target in the file.

To convert these description files:

1. Create a new description block at the top of the file.
2. Give this block a pseudotarget names ALL.
3. Set the dependents for the block to all of the other targets in the file.

When MAKE executes the description, it assumes you want to build the target ALL and builds all targets in the file.

LINK Enhancements

IBM LINK Version 1.10:

- Provides a new option that prevents prompting for libraries and object modules in batch mode.
- Allows programs with 80386 object code to be linked.
- Reduces performance penalty for linking objects with CodeView.
- Issues a warning when a program includes an impossibly large stack size.
- Provides extended exit code values.
- Allows the /NOF and /FAR options for DOS mode programs.

LIB Utility Enhancements

IBM LIB Version 1.10:

- Has been modified so that library routines can be debugged at the source code level if source files are available.

Compiler Software Enhancements
Under Version 1.10, the compiler software:

- Allows you to write and debug dynamically linked applications.
- Allows you to write and debug programs that run under control of the Presentation Manager.
- Allows you to write and debug programs that contain multiple threads that are running under OS/2.
- Includes several new compiler options and pragmas.

Organizational Changes

- The installation and practice session information has been moved from this book to *IBM C/2 Fundamentals*.
- Error messages have been moved from *IBM C/2 Language Reference* to this book.

Chapter 1. Introducing IBM C/2

This book has the following organization:

Chapter 1, Introducing IBM C/2: Describes the notation conventions used in this book, and some of the features and functions of IBM C/2™.

Chapter 2, Compiling: Explains how to compile program source files into object modules using the CC command and its options.

Chapter 3, Linking A Program: Shows how to link object modules into an executable program using the LINK command and its options. It also tells how to compile and link source files into executable programs in one step, using the CL command and its options.

Chapter 4, Running A C Program: Tells how to run a program. It also explains how to pass data to a program, how to exit from the **main** function, how to suppress command-prompt processing, and how to suppress null-pointer checks.

Chapter 5, Using the Program Utilities: Tells how to maintain program modules using the MAKE utility, its options, and the MAKE description files. It also describes how to manage object module libraries using the LIB utility. It describes how to create and change a library file, how to add, delete, replace, or extract library modules, and how to combine libraries. It also shows how to create a cross-reference listing, perform consistency checks, and set the library page size. Additionally, it describes how to change file headers using the EXEMOD utility.

Chapter 6, Interfaces with the IBM Macro Assembler/2: Tells how to write programs to link with assembler language modules or to use as modules of assembler language programs.

C/2 is a trademark of the International Business Machines Corporation.

Appendix A, Error Messages: References the error messages that the IBM C/2 compiler can display and the error conditions that produce them.

Index: Includes entries for this book only.

Conventions Used In This Book

This book uses certain conventions in defining operating system commands, formats of functions, names, and terms.

Convention	Meaning
Boldface	Words or numerics printed in **bold** indicate procedural tasks, menu items, directives, function calls, library functions, statements, keywords, and values.
Italics	Words or numerics printed in *italics* represent information you supply, such as variables and filenames. Italics also introduce new terms or concepts.
Uppercase	Words printed in CAPITAL letters include DOS commands, OS/2 commands, dialog commands, options, programs, filenames, libraries, and utilities.
Color	Color indicates screen responses and programming examples.
Ellipses	Ellipses (...) indicate that you supply additional information in the form shown.
Brackets	Brackets [] indicate optional items supplied to commands.
Vertical Bars	Items separated by a vertical bar (|) mean that you can enter either one of the separated items. For example: `ON|OFF` means you can enter ON or OFF but not both.

The following terms have the specified reference:

Term	Reference
LINK	IBM Linker/2, Version 1.10
LIB	IBM Library Manager/2, Version 1.10
MAKE	IBM MAKE/2, Version 1.10
CodeView[1]	IBM CodeView
EXEMOD	IBM EXEMOD/2 Version 1.10
Assembler	IBM Macro Assembler/2.

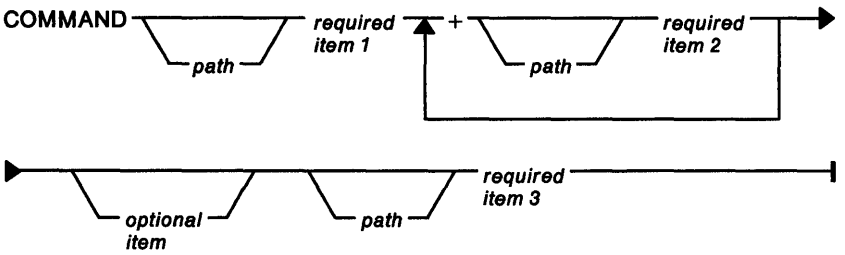
Hexadecimal Representation

This book represents hexadecimal numbers in three ways. The letter **H** (or **h**) shows hexadecimal system calls, such as **59H** (or **59h**), in DOS. All other hexadecimal numbers use the standard C representation **0x***hexdigits*, such as **0x1F**.

Syntax Diagrams

Syntax diagrams define the format of commands entered at the command prompt, the command name is at the beginning (top left corner).

The following shows the form of a syntax diagram:



Understanding Syntax Terms

Syntax Term	Description
syntax diagram	An illustration of the structure and options of a command.
baseline	A horizontal line that connects each of the required items in turn.

[1]CodeView is a trademark of the Microsoft Corporation.

branch lines	Multiple horizontal lines that show choices. Branch lines are below the baseline.
keyword	Words shown in all uppercase letters. Compiler command and utility names are keywords. You can type keywords in any combination of uppercase and lowercase letters.
variable	Items shown in lowercase italic letters mean that you are to substitute the item. For example, *filename* indicates that you should type the name of your file in place of *filename*.
required items	Items that must be included. Required items appear on the baseline. Command names are required items.
optional items	Items that you can include if you choose to do so. Optional items appear below the baseline.
repeat symbol	A symbol that indicates you can specify more than one choice or a single choice more than once.

Symbols show baseline continuation and completion as follows:


Indicates that the command syntax is continued.


Indicates that a line continues from the previous line.


Indicates the end of a command.


Indicates that you can specify a choice more than once.

Reading a Syntax Diagram

1. Start at the top left of the diagram.
2. Follow only one line at a time going from left to right and top to bottom. Items on the lines indicate what you must or can specify and the required sequence.
3. When you encounter one or more branch lines, you must make a choice of items. Follow the line you choose from left to right except where you encounter the repeat symbol.

With many commands, you can enter as many of a group of options as you want. These options are in a box that has a repeat symbol around it. You can follow the arrow through the box until you have selected all the options you want to use. Once you have chosen an option from the box, you cannot choose the same option again.

Operating Systems

Throughout these books, the references to operating systems have the following meaning:

Abbreviation	Meaning
DOS	DOS 3.30 or 4.00
DOS mode	DOS or the DOS mode of OS/2
OS/2	IBM Operating System/2™.

Operating System/2 and OS/2 are trademarks of the International Business Machines Corporation.

Chapter 2. Compiling

This chapter explains how to run the compiler using the CC command and how to use C compiler options.

The CC command is all you need to compile C source files with IBM C/2; it runs all of the compiler passes for you. The CC command also optimizes a program. You need not give an optimizing instruction except to change the way CC optimizes or to disable optimization altogether. See "Optimizing /O" on page 2-40 for more on these choices.

By drawing on the large set of CC options, you can control and modify the tasks performed by the command. For example, you can direct IBM C/2 to create an object listing file or a preprocessed listing. Compiler options also let you supply information that applies to the compiling process, such as the definitions for manifest (symbolic) constants, macros, and the kinds of warning messages you want to see.

To call both the compiler and the linker at the same time use the CL command instead of the CC and LINK commands. The CL command is described in Chapter 3, "Linking A Program."

For a brief description of the IBM C/2 compiler options see "Available Compiler Options" on page 2-12. Additional options are covered in "Advanced Topics" on page 2-48.

Running the Compiler

IBM C/2 requires two types of input: a command to start the compiler and responses to prompts. Start the compiler by typing CC at the command prompt. IBM C/2 prompts for the input it needs by displaying the following four messages, one at a time:

```
Source filename [.C]:
Object filename [filename.OBJ]:
Source listing [NUL.LST]:
Object listing [NUL.COD]:
```

To stop a compiling session for any reason, press either Ctrl + C or Ctrl + Break at any time to return to the command prompt area, where you can start IBM C/2 from the beginning.

File-Naming Conventions

Use any combination of uppercase and lowercase letters for the filenames you type in response to the prompts. For example, the filenames abcde.fgh, AbCdE.FgH, and ABCDE.fgh are equivalent.

You can include spaces before or after filenames but not within them. Options can appear anywhere spaces can appear. See "Using the Compiler Options" on page 2-10 for more information.

CC uses the default file extensions .C, .OBJ, .LST, and .COD when you do not supply extensions with your filenames. You can cancel the default extension for a particular prompt by specifying a different extension. To enter a filename that has no extension, type the name followed by a period. For example, typing `ABC.` in response to a prompt tells IBM C/2 that the specified file has no extension. Typing `ABC` (no period) tells IBM C/2 to use the default extension for that prompt.

To cancel defaults, type all or part of the filename. For example, if the current drive is B and you want the output file to be written to the diskette in drive A, type the response `A:`. The compiler writes the output file on drive A with the default filename.

If you type any part of a legal pathname following the source listing prompt, IBM C/2 produces a source-listing file. The default name is the filename of the source file with the extension .LST. The filename of a file is the portion of the name preceding the period. For example, if you compile a file named TEST.C and type `A:` following the source listing prompt, IBM C/2 produces a source-listing file on drive A with the name A:TEST.LST.

IBM C/2 handles your response to the object listing prompt in the same manner, using the extension .COD.

Special Filenames

You can use the following DOS device names as filenames with the CC command. This allows you to direct files to your terminal or to a printer. You cannot use these names for ordinary filenames.

Name	Device
AUX	An auxiliary device (usually the same as COM1)
CON	The terminal

PRN	The printer (usually the same as LPT1)
NUL	A null (nonexistent) file NUL as a filename means that the compiler does not create a file.
LPT1	First parallel printer
LPT2	Second parallel printer
LPT3	Third parallel printer
COM1	First serial port
COM2	Second serial port.

Even if you add device designations or filename extensions to these special filenames, they remain associated with the devices listed above. For example, A:CON.XXX refers to the terminal and is not the name of a file.

Note: Object files contain machine code and are not printable. When responding to the object filename prompt, do not give a filename that refers to a printer or terminal. When using device names, do not follow them with a colon. IBM C/2 does not recognize the colon. For example, use CON or PRN, not CON: or PRN:, in your responses to prompts.

Source Filename Prompt

Following the source filename prompt, you specify the file that IBM C/2 compiles. If the extension is .C, you do not have to give an extension. IBM C/2 looks for a file with that extension. Do not use the extensions .ASM or .OBJ in the names of C source files.

Pathnames are allowed with the source filename. You can specify the pathname of a source file in another directory or on another disk.

IBM C/2 displays an error message if you do not supply a source filename or if you use a reserved extension.

Object Filename Prompt

Following the object filename prompt, you can supply a name for the object file produced by compiling the source file. Give the object file any name and any extension. It is recommended that you use the conventional .OBJ extension because it simplifies the operation of LINK and LIB, both use .OBJ as the default extension when processing object files.

If you supply only a drive or directory specification following the object filename prompt, IBM C/2 creates the object file in the given

drive or directory and uses the default filename. You can use this option to create the object file in another directory or on another disk. When you give only a directory specification, it must end with a backslash (\) so that IBM C/2 can distinguish between a directory specification and a filename.

The default name displayed for the object file is the filename of the source file with an .OBJ extension. If you supply no pathname, IBM C/2 creates the object file in the current working directory.

Source Listing Prompt

Following the source listing prompt, you can tell IBM C/2 to create a source listing. If you supply any filename following this prompt, the compiler creates a source listing, using the filename you supply. By convention, these listings are given the extension .LST, but you are free to choose any extension. When you do not supply a filename, the default is the special name NUL.LST, which tells the compiler not to create a listing.

Object Listing Prompt

Following the object listing prompt, you can tell IBM C/2 to create an object listing for the compiled file. The object listing contains the machine instructions and assembled code for your program.

If you supply a filename following this prompt, IBM C/2 creates an object listing, using the filename you supply. By convention, these listings are given the extension .COD, but you can choose any extension you like. When you do not supply a filename, the default is the special name NUL.COD, which tells IBM C/2 not to create a listing.

The CC command optimizes by default, so the object listing reflects the optimized code. Since optimization may involve rearrangement of code, the relationships between your source file and the machine instructions may not be clear. To produce a listing without optimizing, use the /Od option, discussed under "Preparing for Debugging /Zd, /Zi, /Od" on page 2-38.

To produce a combined source and object code listing, use the /Fc option. To produce an assembler listing, use the /Fa option. (An assembler listing is a listing of the assembler code, that can be used as input to IBM Macro Assembler/2.) See "Producing Listing Files /Fs, /Fl, /Fa, /Fc" on page 2-16 for additional information.

Selecting Default Responses

To select the default response for the current prompt, press Enter without giving any other response. The next prompt appears.

To select default responses for all remaining prompts, type a single semicolon (;) after the filename following the source filename, object filename, or source listing prompts. Once the semicolon is entered, you cannot respond to any of the remaining prompts for that compiling session. The compiler ignores any text appearing after the semicolon (such as an option). Use the semicolon to save time when the default responses are acceptable.

There is no default for the first prompt, source filename. The default for the object filename is the filename of the source file with an .OBJ extension. The default for the source listing prompt is the special name NUL.LST, which tells IBM C/2 not to create a source-listing file. The default for the object listing prompt is the special name NUL.COD, which tells the compiler not to create an object-listing file.

If you respond to the source filename prompt with a nonexistent filename, or to the object filename, object listing, or source listing prompts with an incorrect pathname, IBM C/2 displays an error message and ends. You must start the compiler again with the correct information.

Using the Source Listing

The information in the source listing helps debug programs as they are being developed and documents the structure of a finished program. The source listing contains the numbered source code lines, embedded error messages, and symbol tables. Error messages appear in the listing after the line that caused the error. The line number given in the error message corresponds to the number of the source line immediately above the message in the source listing. The compiler does not expand include files in the source listing. It places any errors detected in an include file in the source listing following the **#include** directive for that file.

At the end of each function, a table of local symbols is given. This table has a Name field, a Class field, an Offset field, and a Register field for each local symbol declared in the function. The *Class field* of a symbol is **auto** if the symbol is a non-static local variable or **param** if the symbol is a formal parameter. The *Offset field* of a symbol is its

offset address relative to the frame pointer (that is, the BP register). The Offset field is positive for **param** symbols and negative for **auto** symbols with **auto** storage class. The *Register field* indicates if the variable is stored in a register and, if so, in which one (SI or DI).

At the end of the source code, a table of global symbols is given. This table gives a Name field, a Type field, a Size field, a Class field, and an Offset field for each global symbol, external symbol, and statically-reserved variable declared in the source file.

The *Type field* of a symbol gives a simplified version of its type as declared in the source file. The Type entry for a function is either a **near** or a **far** function, depending on the storage model and how the function was declared. The Type entry for a pointer is **near** pointer, **far** pointer, or **huge** pointer. For enumeration variables, the Type entry is **int**. For structures, unions, and arrays, the Type entry is **struct/array**.

The *Size field* of a symbol is defined only for variables. This field specifies the number of bytes of storage reserved for the variable. The amount of storage reserved for an external array may not be known, so its Size field may be undefined.

The Offset field of a symbol is defined only for symbols with an entry of **global** or **static** in the Class field. For variables, the Offset field gives the relative offset of the location of the stored variable in the data segment for the program file being compiled. Because the linker combines several logical data segments into a physical segment, this number is useful only for determining the relative position of storage of variables. For functions, the Offset field gives the relative offset of the start of the function in the logical code segment. For small- and compact-model programs, logical code segments from different program files are combined into a single physical segment by the linker. The Offset field is useful for determining the relative positions of different functions defined in the same source file. However, for medium-, large-, and huge-model programs, each logical code segment becomes a unique physical segment. In these cases, the Offset field gives the actual offset of the function in its run-time code segment.

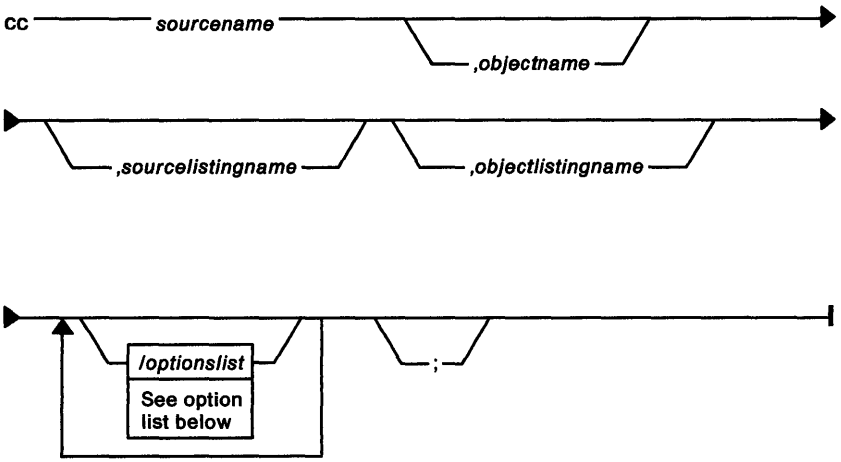
Using the Command-Prompt Method

After you understand how IBM C/2 prompts and responses work, you can use the command-prompt method to run the compiler. With this method, you type all the filenames and options in the command you use to start the compiler.

The entries following CC are responses to the command prompts. You can include spaces before or after filenames but not within them. Options can appear anywhere spaces can appear. See "Using the Compiler Options" on page 2-10 for more information.

Note: The command-prompt form of the CC command is used for the examples of options in this book.

The command-prompt method has the following form:



Leave the *objectname*, *sourcelistingname*, and *objectlistingname* fields blank to select the default responses. Options can be placed anywhere at the command prompt, but they must be placed before the semicolon. When the compiler finds a semicolon at the command prompt, it uses the default responses to the remaining prompts. The compiler ignores any text after the semicolon at the command prompt. See "Using the Compiler Options" on page 2-10 for more information about options at the command prompt.

The comma serves as a separator and also has a special function at the command prompt. If you place a comma after the *objectname*

field at the command prompt (whether or not an *objectname* is given), the default for the source listing field is changed from NUL.LST to the filename of the source file plus .LST. Similarly, if a comma follows the *sourcelistingname* field, the default for the *objectlistingname* field is changed from NUL.COD to the filename of the source file plus .COD.

Example
The following two commands are equivalent:

```
CC TEST, TEST, TEST, TEST;
CC TEST, , , ;
```

In the first command, the name TEST is explicitly specified for all prompts, so TEST.C is compiled and three files are produced: TEST.OBJ, TEST.LST, and TEST.COD.

In the second command, only the source filename is supplied. The default name TEST.OBJ is used for the object filename because none is specified. The comma following the object filename field causes the default for the source listing file to be changed to TEST.LST. Because no alternative name is supplied at the command prompt, the compiler creates a listing file named TEST.LST. It creates the object-listing file TEST.COD in a similar fashion.

The following example uses the semicolon to tell CC not to produce listing files:

```
CC TEST;
```

The command tells the compiler to create an object file named TEST.OBJ but not to create a listing file. No comma is present in the command to change the default from NUL.COD to TEST.COD and NUL.LST to TEST.LST.

You can combine the prompt method and command-prompt methods by giving CC a partial command. It prompts for fields not supplied. You can end a partial command with a semicolon, a filename, or a comma:

semicolon CC uses the default responses for the remaining prompts.

filename CC prompts you for the remaining responses, if any.

comma If you give a source filename followed by a comma, CC prompts you for the object filename, source listing name, and object listing name, as usual. However, if you supply both a source filename and an object filename, then end the command with a comma, CC changes the default source-listing name from NUL.LST to the filename of the source file plus .LST. CC then prompts you for a source-listing name to allow you to cancel the default. (You can give the name NUL.LST to suppress the creation of a source listing). The default object-listing name is changed in a similar fashion if the command ends with a source listing name followed by a comma.

Options can also be placed at the end of a partial command.

Example

The following partial command tells IBM C/2 to compile a source filename ASK.C and place the results in the object filename TELL.OBJ:.

```
CC ASK.C, TELL.OBJ
```

The command also causes CC to display the following prompt, because you supplied the source filename and object filename but not the source or object listing filenames:

```
Source listing [NUL.LST]:
```

The next command tells CC to use the default response (no file) for the source and object listings. No further prompts appear in this case:

```
CC ASK, TELL;
```

In the following command, the trailing comma (after TELL.OBJ) has a special meaning:

```
CC ASK.C, TELL.OBJ,
```

The trailing comma causes CC to prompt as follows:

```
Source listing [ASK.LST]:
```

The default name in brackets is ASK.LST rather than NUL.LST. In this case an object listing is created by default, unless you cancel the default to specify a different listing name (or specify the name NUL to suppress the listing).

The last command tells CC to start with the object filename prompt because only the source filename is supplied:

```
CC ASK
```

C Compiler Error Messages

The error messages produced by the C compiler fall into five categories: warning messages, fatal error messages, compilation error messages, command area messages, and compiler internal error messages. The messages for each category are listed in numeric order and described in detail in Appendix A of this book.

Note: You can control the level of warnings produced by the compiler using the /W option, as described under "Setting the Warning Level /W /w" on page 2-37.

Using the Compiler Options

A number of command options are available to control and modify the operation of the compiler. (See "Available Compiler Options" on page 2-12.) Options must begin with a slash (/) or a hyphen (-) and contain one or more letters. For example, /Zg and -Zg are both acceptable forms of the Zg option.

Note: The CC command is case sensitive to the option letter, so you must specify options exactly as shown in this book. For more information about case, see "Allowing Case-Insensitive Options /CASE" on page 2-16.

You can place options anywhere you can place a space when you use the CC command, but the command ignores options following a semicolon. Thus, you can place options before or after any of the four filenames (source filename, object filename, sourcelisting, and objectlisting). The options apply to the entire compiling process, not only to the lines they appear on.

For CL, the options must appear before the source file they apply to. For example,

```
CL /Ox prog1.c
```

uses maximum optimization when compiling PROG1.C, but

```
CL prog1.c /Ox prog2.c
```

uses default optimization for PROG1.C and maximum optimization for PROG2.C. Only the linking options, such as -c or -link, may be given following the name of the source file.

Some options use arguments, such as filenames, strings, or numbers. In most of these cases, spaces are allowed between the option letter and the argument. For example, /W 3 and /W3 are both acceptable forms of the /W option.

The /Gt, /Fa, and /Fc options are the only exceptions to the previous example. The /Gt option accepts an optional numerical argument, whereas the /F options accept an optional pathname or partial pathname argument. When you supply an argument to one of these options, no spaces can be placed between the option and the argument. For example, /FcMINGLE is acceptable but /Fc MINGLE is not.

Note: Options that consist of more than one letter, such as /Fc, cannot have spaces between the letters. But you must put at least one space between separate options in the list. For example: /Oas /Fc.

The order options are specified in is not important. They can be given following any prompt or at any command prompt. The default for the prompt is still used if you supply an option but no filename in response to the prompt.

Example

The following examples produce exactly the same results. The source file LOAD.C on drive A is compiled. The object file is named OUT.OBJ. The source listing file is named LOAD.SRC.

```
Example 1
CC
Source filename [.C]:  A:\LOAD.C
Object filename [LOAD.OBJ]: OUT
Source listing [NUL.LST]: LOAD.SRC
Object listing [NUL.COD]: /Oas /Fc

Example 2
CC A:\LOAD.C,  OUT,LOAD.SRC /Oas /Fc;
```

The /Fc option produces a combined source and assembly code listing. Because no argument was given with the /Fc option, the listing is given the default name LOAD.COD, formed by adding .COD to the filename of the source file. The object file and combined listing are both created on the default drive because no drive was specified. The /Oas option tells the compiler how to optimize the object file.

The /Fc and /Oas options are explained under "Producing Listing Files /Fs, /Fl, /Fa, /Fc" on page 2-16 and "Optimizing /O" on page 2-40.

Available Compiler Options

The following table lists the IBM C/2 compiler options, a brief description of what the options do and whether you can use the options with the CC command, the CL command, or both.

Option	Description	Use with	Page
A*string*	Creates customized storage model.	CL only	2-65
A*storage model*	Sets up storage for your program and determines how the program is loaded.	Both	2-44
c	Suppresses linking.	CL only	3-63
C	Preserves comments during preprocessing.	Both	2-28
/CASE	Allows use of IBM C 1.00 options.	CC only	2-16
D*name*	Defines a constant or macro in a source file.	Both	2-24
E	Produces a listing of pre-processed files.	Both	2-27
EP	Produces a listing of pre-processed files.	CL only	2-27
Fa	Produces an assembler listing.	Both	2-16
Fb*name*	Creates bound applications.	CL only	3-63
Fc	Produces mixed source and object listing.	Both	2-16
Fe*programname*	Names the executable file.	CL only	3-57
F *hexnumber*	Sets stack size to *hexnumber*.	CL only	3-61
Fl	Produces object listing.	CL only	2-16

Option	Description	Use with	Page
FPa, FPc, FPi, FPi87, FPc87	Handle floating-point operations.	Both	2-32
Fm*mapname*	Creates the map file.	CL only	3-57
Fo	Names the object file.	CL only	3-57
Fs	Produces source listing.	CL only	2-16
G0, G1, G2	Enable the appropriate instruction set.	Both	2-37
Gc	Controls function-calling sequence.	Both	2-69
Gm	Allocates some data items.	CL only	3-61
Gs	Removes stack probes.	Both	2-55
Gt*number*	Allocates data to a new data segment.	Both	2-57
Gw	Compiles a Presentation Manager application.	CL only	3-65
HELP	Lists the compiler options.	Both	2-15
H*number*	Restricts the length of external names.	CL only	3-65
I*dirname*	Searches for **include** directories.	Both	2-28
J	Changes the default **char** type.	Both	2-53
Lc	Creates a DOS mode executable file.	CL only	2-47
link	Passes data to LINK.	CL only	3-61
Lp	Creates an OS/2 mode executable file.	CL only	2-47
ND*name*	Sets the data segment name.	CL only	3-68
NM*name*	Sets the module name.	CL only	3-68
nologo	Suppresses writing logo and copyright lines.	Both	3-66
NT	Sets the text segment name.	CL only	3-68

Option	Description	Use with	Page
Oa	Cancels alias checking.	Both	2-40
Od	Turns off optimization.	Both	2-38
Oi	Declares some functions as intrinsic.	CL only	3-66
Ol	Turns on loop optimization.	Both	3-71
On	Turns off loop optimizations.	CL only	2-40
Op	Enforces precision in floating-point operations.	Both	2-40
Os	Favors code size during optimization.	Both	2-40
Ot	Favors run time during optimization.	Both	2-40
Ow	Restricts assumptions.	CL only	2-40
Ox	Performs maximum optimization.	Both	2-40
P	Produces listing of preprocessed files.	Both	2-27
S	Causes the compiler to perform a syntax check.	CC only	2-30
Sl*length*, Sp*length*	Specify line width and page length.	CL only	3-72
St*title*, Ss*title*	Specify title and subtitle.	CL only	3-74
Tc*filename*	Indicates a C source file.	CL only	3-74
u	Undefines names.	CL only	2-26
U	Turns off definition of predefined identifiers.	Both	2-26
V*string*	Labels the object file.	CL only	2-26
w	Sets the level of warning messages.	CL only	2-37
W*number*	Sets the level of warning messages.	Both	2-37

Option	Description	Use with	Page
X	Excludes directory from search.	CL only	2-28
Za	Enables language extensions.	Both	2-48
Zc	Enables language extensions.	CL only	2-48
Zd	Enables language extensions.	Both	2-38
Ze	Enables language extensions.	Both	2-48
Zg	Produces function declarations.	Both	2-30
Zi	Produces an object file containing debugging information.	Both	2-38
Zl	Suppresses selection of default libraries.	Both	2-52
Zp	Packs structure data.	Both	2-51
Zp*n*	Specifies placement of struct members.	CL only	2-51
Zs	Causes the compiler to perform a syntax check.	CL only	2-30

Listing the Compiler Options /HELP

This option displays a list of the compiler options. You can specify /HELP as part of the CC command or as part of the response to a CC prompt. In either case, CC processes all information on the line containing /HELP, prints the command list, and, if needed, reissues the current prompt for further input.

Format

```
/HELP
```

All of the input you have given up to this point has been processed. For example, if you have typed a filename followed by /HELP, that filename appears as the default value when the prompt is reissued.

The only exception to these rules is for source filenames. If you type the source filename with /HELP, the source-file prompt is not reissued. Instead, the object-file prompt is displayed after the command list.

Allowing Case-Insensitive Options /CASE

This option permits you to use options with IBM C/2 that you used with the IBM C Compiler Version 1.00. This option works with the CC command only.

Format

/CASE

When you specify /CASE, all options passed to the compiler can use uppercase or lowercase letters. This provides a subset of compiler options that are available in the IBM C Compiler Version 1.00. Because the CC command is case-insensitive, CC cannot support some case-sensitive switches. To handle CC commands correctly in make or batch files written for IBM C Version 1.00, the /CASE switch lets CC run without regard to case sensitivity in order to avoid misinterpreting options with case-sensitive meanings. The /CASE option is itself case insensitive. The compiler can also read this option from the CC environment variable.

Producing Listing Files /Fs, /Fl, /Fa, /Fc

In addition to the command-prompt method of creating listing files, you can use options to create source and object listings. You can also use options to create assembler listings as well as mixed source and assembler listings that are not available through prompts.

To set the format of generated listings, see "Setting Line Width and Page Length /Sl, /Sp" on page 3-72 and "Setting Titles and Subtitles /St, /Ss" on page 3-74.

Format

```
/Fs[listfile] Produces source listing
/Fl[listfile] Produces object listing
/Fa[listfile] Produces assembler listing
/Fc[listfile] Produces mixed source and object listing.
#pragma page([n]) Skips pages in source listing output.
#pragma skip([n]) Skips lines in source listing output.
```

The *listfile*, if given, must immediately follow the option. It can be any of the following entries.

Entry	Result
filename	CC uses the given filename, adding the default extension if the filename has no extension. The filename can include a path to tell CC where to create the listing.
Directory specification	CC uses the default listing name (the base name of the source file plus the default extension) to create the listing in the given directory. The directory specification must end with a backslash so that CC can distinguish between it and a filename.
Omitted	When you give no *listfile*, CC uses the default listing name (the base name of the source file plus the default extension) to create the listing in the current working directory.

The default extension is .LST for the /Fs option, .COD for the /Fc and /Fl options, and .ASM for the /Fa option.

The **#pragma page** and **#pragma skip** pragmas default to 1. For both, *n* must be a non-negative integer constant. The following example causes the following source line to start at the top of the next page.

Example

```
#pragma page (1)
```

The compiler can produce at most one source-listing file and one variation of the object listing each time you compile. If you use both the /Fa and the /Fl options in one command, the compiler produces only one file. The /Fc option cancels other listing options. Whenever you use /Fc, the compiler produces a combined listing. If you give conflicting names for a listing file (for example, one following the prompt and one with the option), the name specified last has precedence. The /Fs option is recognized only by the CL command.

/Fs Option

This option produces a source listing file with the default extension .LST.

The following example shows a section of code from a source listing file:

```
  Line#  Source Line

 1  #include  <stdio.h>
 2
 3  main(argc, argv)
 4  int  argc;
 5  char *argv[];
 6  {
 7      FILE *infile;
 8      char *name, line[100];
 9      int  nlines;
 10
 11      if (argc > 1)
 12      {
 13          nane = argv[argc - 1];
***** c.c(13) : error C2065: 'nane'
: undefined
***** c.c(13) : warning C4047: '=' :
different levels of indirection
 14          if ((infile = fopen(name, "r")) == NULL)
 15          {
 16              fprintf(stderr,
 17                  "%s couldn't open file %s\n",
 18                  argv[0], name);
 19              exit(1);
 20          }
 21      }
 22  }

1 errors detected
```

The error messages result from the misspelling of the variable *name* on line 13.

If the source file compiles with no errors more serious than warnings, the compiler includes tables of segments, local symbols, and global symbols in the source listing. The compiler does not include symbol tables if the compiler is unable to finish compiling.

At the end of each function, the compiler gives a table of local symbols, as shown below for the function **main**:

```
main  Local Symbols

Name      Class  Offset    Register

name      auto   -006a
line      auto   -0068
infile    auto   -0004
nlines    auto   -0002
argc      param   0004
argv      param   0006
```

The Name column lists the name of each local symbol in the function. The Class column contains either **auto** if the symbol is a nonstatic local variable or **param** if the symbol is a formal parameter. The Offset column shows the offset address of the symbol relative to the frame pointer (that is, the **BP** register). The Offset number is positive for **param** symbols and negative for **auto** symbols with **auto** storage class. The Register column is blank unless the compiler stores the variable in a register. If the variable is in a register, the column shows the register **SI** or **DI**.

At the end of the source code, the compiler gives a table of global symbols, as shown below:

```
Global Symbols

Name      Type           Size  Class   Offset

_iob      struct/array   160   extern  ***
exit      near function  ***   extern  ***
fopen     near function  ***   extern  ***
fprintf   near function  ***   extern  **
main      near function  ***   global  0000
```

The Name column lists each global symbol, external symbol, and statically-reserved variable declared in the source file. The Type column shows a simplified version of the type of the symbol as declared in the source file. The Type entry for a function is either **near function** or **far function**, depending on the storage model and how you declared the function. The Type entry for a pointer is **near pointer**, **far pointer**, or **huge pointer**. For enumeration variables, the Type entry is **int**. For structures, unions, and arrays, the Type entry is **struct/array**.

The compiler uses the Size column only for variables. This column specifies the number of bytes of storage reserved for the variable. You might not know the amount of storage reserved for an external array; its Size field might be undefined. The Class column contains

either **global, common, extern,** or **static,** depending on how you defined the symbol in the source file.

The compiler uses the Offset column only for symbols with an entry of **global** or **static** in the Class field. For variables, the Offset field gives the relative offset of the location in storage of the variable in the logical data segment for the program file you are compiling. Because the linker generally combines several logical data segments into a physical segment, this number is useful only for telling the relative position of storage of variables. For functions, the Offset field gives the relative offset of the start of the function in the logical code segment. For small- and compact-model programs, the linker combines logical code segments from different program files into a single physical segment. The Offset field is useful for telling the relative positions of different functions defined in the same source file. However, for medium-, large-, and huge-model programs, each logical code segment becomes a unique physical segment. In these cases, the Offset field gives the actual offset of the function in its run-time code segment. (For more information about storage models see "Working with Storage Models /A" on page 2-44.)

The last table in the source listing shows the segments used and their size, as shown below:

```
Code size = 0057 (87)
Data size = 001c (28)
Bss size  = 0000 (0).
```

The table gives the byte size of each segment first in hexadecimal, and then in decimal (in parentheses).

/Fl Option

This option produces an object listing file and is available only with the CL command. The object listing contains the machine instructions and assembler code for your program, as shown in the following example:

```
; Line 12
    *** 00000a 83 7e 04 01  cmp  WORD PTR [bp+4],1 ;argc
    *** 00000e 7e 42        jle  $I68
; Line 13
    *** 000010 8b 76 04     mov  si,[bp+4]     ;argc
    *** 000013 d1 e6        shl  si,1
    *** 000015 8b 5e 06     mov  bx,[bp+6]     ;argv
    *** 000018 8b 40 fe     mov  ax,[bx-2][si]
    *** 00001b 89 46 96     mov  [bp-106],ax     ;name
; Line  14
    *** 00001e b8 00 00     mov  ax,OFFSET DGROUP:$SG67
    *** 000021 50           push ax
    *** 000022 ff 76 96     push WORD PTR [bp-106] ;name
    *** 000025 e8 00 00     call _fopen
    *** 000028 83 c4 04     add  sp,4
    *** 00002b 89 46 fc     mov  [bp-4],ax     ;infile
    *** 00002e 0b c0        or   ax,ax
    *** 000030 75 32        jne  $I68
```

The sample shows the line numbers in the source code as comments. The machine instructions are on the left and assembler code is on the right.

/Fa Option

This option produces an assembler listing of your program. The assembler listing contains the assembler code corresponding to your C file, as shown below:

```
; Line 12
 cmp WORD PTR [bp+4],1 ;argc
 jle $I68
; Line 13
 mov si,[bp+4]  ;argc
 shl si,1
 mov bx,[bp+6]  ;argv
 mov ax,[bx-2][si]
 mov [bp-106],ax  ;name
; Line 14
 mov ax,OFFSET DGROUP:$SG67
 push ax
 push WORD PTR [bp-106] ;name
 call _fopen
 add sp,4
 mov [bp-4],ax  ;infile
 or ax,ax
 jne $I70
```

The sample shows the same code as in the object listing sample except that it omits the machine instructions. This is to ensure the listing is suitable as input for IBM Macro Assembler/2 (MASM/2).

/Fc Option

This option produces a line-by-line combined source and assembler code listing, showing one line of your source program followed by the corresponding line (or lines) of machine instructions, as shown in the following example:

```
;|***   if (argc > 1) {
; Line 12
    *** 00000a 83 7e 04 01  cmp  WORD PTR [bp+4],1  ;argc
    *** 00000e 7e 42      jle  $I68
;|***           name = argv[argc - 1];
; Line 13
    *** 000010 8b 76 04     mov  si,[bp+4]      ;argc
    *** 000013 d1 e6      shl  si,1
    *** 000015 8b 5e 06     mov  bx,[bp+6]      ;argv
    *** 000018 8b 40 fe     mov  ax,[bx-2][si]
    *** 00001b 89 46 96     mov  [bp-106],ax      ;name
;|***      if ((infile = fopen(name,"r")) == NULL) {
; Line 14
    *** 00001e b8 00 00     mov  ax,OFFSET DGROUP:$SG67
    *** 000021 50       push ax
    *** 000022 ff 76 96     push WORD PTR [bp-106]  ;name
    *** 000025 e8 00 00     call _fopen
    *** 000028 83 c4 04     add  sp,4
    *** 00002b 89 46 fc     mov  [bp-4],ax      ;infile
    *** 00002e 0b c0      or   ax,ax
    *** 000030 75 32      jne  $I70
```

This sample is like the object-listing sample except that the sample provides the C source line in addition to the line number.

In a listing file, the names of globally visible functions and variables begin with an underscore, as shown below (this part is the same for all three kinds of listings):

```
EXTRN _exit:NEAR
EXTRN _fopen:NEAR
EXTRN _fprintf:NEAR
EXTRN __chkstk:NEAR
EXTRN __iob:BYTE
```

IBM C/2 adds an underscore as a prefix to all global names to preserve compatibility with other C compilers. If you write assembler language routines to work with your C program, this naming convention is important.

The listing can also contain names that begin with more than one underscore (for example, **__chkstk** and **__iob** in the sample). The compiler reserves identifiers with more than one leading underscore for internal use, and you should not use them in your programs except for those documented in *IBM C/2 Language Reference*, such as **_psp**,

_amblksiz, and **_fpreset( )**. Avoid creating global names that begin with an underscore in your C source files. Because the compiler adds another leading underscore, these names could have two leading underscores, possibly causing conflicts with the names reserved by the compiler.

The CC command optimizes by default, so listing files reflect the optimized code. Because optimizing code can involve rearranging it, the correspondence between your source file and the machine instructions might not be clear, especially when you use the /Fc option to combine the source and assembler code. To produce a listing without optimizing, use the /Od option along with the listing option.

Example

```
CC HELLO.C /FsHELLO.SRC /FcHELLO.CMB;

CC HELLO /FsHELLO.SRC, ,HELLO.LST, HELLO.COD;
```

In the first example, CC creates a source listing called HELLO.SRC and a combined source and assembler listing called HELLO.CMB. The object file has the default name HELLO.OBJ.

The second example produces a source listing called HELLO.LST rather than HELLO.SRC because the last name provided has precedence. This example also produces an object listing file named HELLO.COD. The object file in this example has the default name HELLO.OBJ.

Controlling the Preprocessor

The CC command provides a number of options that give you control over the operation of the C preprocessor. You can define macros and manifest (symbolic) constants from the command prompt, change the search path for include files, and stop compiling a source file after the preprocessing stage to produce a preprocessed source file listing.

The C preprocessor recognizes only preprocessor directives. It treats the source file as a text file, processing substitutions and definitions as directed. See Chapter 9 of *IBM C/2 Fundamentals* for a complete discussion of C preprocessor directives.

Defining Constants and Macros
/D

This option lets you define a constant or a macro in a source file.

Format

/Didentifier[=[*string*]]

The *identifier* is the name of the constant or macro, and the *string* is its value or meaning. */Didentifier*= and */Didentifier*=*string* cannot be defined in the CL environment because the SET command does not accept the = sign, a special character in the CL environment.

If you leave out both the equal sign and the *string*, the given constant or macro is assumed to be defined and its value is set to 1. For example, /DSET is sufficient to define SET.

If you give the equal sign with an empty *string*, the given constant or macro is considered defined as the empty *string*. This definition effectively removes all occurrences of the identifier from the source file. For example, */Dregister*= removes all occurrences of *register* from the source file. The identifier *register* is still considered to be defined.

The effect of the /D option is the same as a preprocessor **#define** directive at the beginning of your source file. The identifier is defined throughout the source file being compiled.

You can supply a command prompt definition for an identifier that is also defined within the source file. The command prompt definition remains in effect until the point of the redefinition in the source file.

Up to 16 definitions can appear in the command, each preceded by the /D option. If you need to define more than 16 identifiers, see the discussion of the /U option under "Removing Definitions of Predefined Identifiers /U, /u" on page 2-26.

Example

The following example defines the manifest constant NEED in the source file MAIN.C:

```
CC MAIN.C /DNEED=2;
```

Spaces are permitted (but not required) between /D and the identifier. This definition is equivalent to placing the directive at the beginning of the source file as follows:

```
#define NEED    2
```

The /D option is especially useful with the **#If** directive to control the compiling of statements in the source file. For example, suppose a source file named OTHER.C contained the following fragment:

```
#if defined(NEED)
⋮
#endif
```

Also suppose that OTHER.C does not explicitly define NEED (that is, no **#define** directive for NEED is present). Then, all statements between the **#If** and the **#endIf** directives are compiled only if you supply a definition of NEED by using /D. You can do this by entering this command:

```
CC MAIN.C /DNEED;
```

This command is sufficient to compile all statements following the **#If** directive. NEED does not have to be set to a specific value to be considered defined. The following command causes the compiler to ignore (not compile) the statements in the **#If** block:

```
CC MAIN.C;
```

Predefined Identifiers

The compiler defines identifiers that are useful in writing portable programs. Use these identifiers to compile code sections conditionally, depending on the current processor and operating system. The predefined identifiers and their functions are:

Identifier	Function
DOS	Always defined. Identifies the target operating system as DOS.
M_I86	Always defined. Identifies the target machine as a member of the I86 family.
M_I8086	Identifies the target processor as an 8086. Defined by default or when the /G0 option is given explicitly.
M_I86xM	Always defined. Identifies the storage model, where *x* is either S (small-model), C (compact-model), M (medium-model), L (large-model), or H

	(huge-model). Small-model is the default. See "Working with Storage Models /A" on page 2-44 for more information about storage models.
M_I286	Identifies the target processor as an 80286. Defined when you use the /G1 or /G2 option.
NO_EXT_KEYS	No extended keywords. Defined only when you use the /Za option, disabling special keywords, such as **far** and **fortran**.
_CHAR_UNSIGNED	Type **char** is **unsigned**. Defined only when you use the /J option to make the **char** type **unsigned** by default.

Removing Definitions of Predefined Identifiers /U, /u

This option turns off the definition of one or more of the predefined identifiers.

Format

/Uidentifier
/u

For each definition of a predefined identifier you remove, you can substitute a definition of your own at the command prompt.

For example, the following line removes the definitions of three predefined identifiers. You give the /U option three times to do this.

```
CC WORK /UDOS /UM_I86 /UM_I86SM;
```

The /u form of this option turns off all predefined names except M_I86xM. This form of the option is available with the CL command only, not with the CC command. The definition of the C constant NULL is dependent on the memory model, which M_I86xM identifies. Without a value for memory model, NULL might have an incorrect value and cause program errors. Use the /U form to turn off the definition of M_I86xM, if needed.

```
CL /u WORK.C
```

If you remove the definitions of all predefined names, you can define up to 20 identifiers at the command prompt with the /D option.

Producing a Preprocessed Listing /P, /E, /EP

These options produce listings of preprocessed files and let you examine the output of the C preprocessor. The /EP and /P options are available only with the CL command, not with CC.

Format

```
/P
/E
/EP
```

The preprocessed listing file carries out preprocessor directives, performs macro expansions, and removes comments. These options suppress compiling; no object file, source listing, or object listing is produced, even if you supply a name following the object filename, source filename, or object listing prompt.

The /P option writes the preprocessed listing to a file with the same prefix as the source file but with an .I extension.

The /E option copies the preprocessed listing to the standard output (usually your display) and places a **#line** directive in the output at the beginning and end of each included file. You can save this output by redirecting it to a file, using the redirection symbols > or >>. See the user's reference information for the operating system for a description of these symbols.

The /E option is useful to see how macros are expanded, particularly when compiling errors occur on lines containing macros. The **#line** directives renumber the lines of the preprocessed file so that errors produced in later stages of processing refer to the original source file rather than to the preprocessed file. You can also resubmit the preprocessed listing for compiling.

The /EP option combines features of the /E and /P options. The compiler preprocesses the file and copies it to the standard output, but adds no **#line** directives.

Example

The first example creates the preprocessed file MAIN.I from the source file MAIN.C:

```
CL /P main.c
```

The second example creates a preprocessed file with inserted **#line** directives from the source file ADD.C. DOS redirects the output to the file PREADD.C:

```
CC ADD.C /E ;  > PREADD.C
```

The third example produces the same preprocessed output as the second example without the **#line** directives. The output appears on the display:

```
CL /EP add.c
```

Preserving Comments /C

This option preserves comments during preprocessing. Normally, the compiler removes comments from a source file in the preprocessing stage because they do not serve any purpose in later stages of compiling. This option is valid only when you use the /E, /P, or /EP options. It is available only with the CL command, not with CC.

Format

```
/C
```

Example

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

```
CL /P /C SAMPLE.C
```

Searching for Include Files /I, /X

These options temporarily change the effects of the environment variable INCLUDE. The /I option causes the compiler to search the directory you specify before searching the standard places given by the INCLUDE environment variable. You can add more than one include directory by giving the /I option more than once in the CC command. The compiler searches the directories in order of their appearance in the command. The /X option is available only with the CL command.

Format

/Idirectory
/X

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

You can prevent the C preprocessor from searching the standard places for include files by using the /X (exclude) option. When CL finds the /X option, it considers the list of standard places to be empty. You use this option often 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 the second example below.

Example

The first example directs the compiler to search for include files requested by MAIN.C, first in directory A:\INCLUDE, second in directory B:\MY\INCLUDE, and finally in the directory or directories assigned to the INCLUDE environment variable:

```
CC MAIN.C /I A:\INCLUDE /I B:\MY\INCLUDE;
```

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

```
CL /X /I B:\ALT\INCLUDE MAIN.C
```

Syntax Checking

The options described in this section are useful in the early stages of program development. With the /Zs option, you can quickly check a program for syntax errors. With the /Zg option, you can produce function declarations, which you can then use to enhance the syntax-checking capabilities of the compiler.

Identifying Syntax Errors /Zs, /S

These options cause the compiler to perform a syntax check. Use the /Zs form with the CL command and the /S form with the CC command.

Format

```
/Zs
/S
```

If the source file has syntax errors, error messages appear on the standard output device, which is usually your screen. If a source listing file is requested, error messages are embedded following the source line they were detected in.

Example

The following command causes the compiler to perform a syntax check on PRELIM.C and display messages about any errors it finds on the screen as well as in PRELIM.LST:

```
CC /S PRELIM.C, ,PRELIM.LST;
```

Generating Function Declarations /Zg

This option produces a function declaration for each function defined in the source file. The function declaration includes the function return type and an argument type list created from the types of the formal parameters of the function. Any function declarations already present in the source file are ignored.

Format

```
/Zg
```

The produced list of declarations is written to the standard output. It can be saved in a file by means of the redirection symbols > or >>.

When you use the /Zg option, the source file is not compiled. As a result, no object file or object listing is produced. The source listing is also suppressed.

The list of declarations is helpful for verifying that actual arguments and formal function parameters are compatible. You can save the list

and include it in the source file to cause the compiler to perform type-checking. The presence of a declared argument type list for a function enables compiler type-checking between actual arguments to a function (given in the function call) and the formal parameters of a function.

Type-checking can be helpful when writing and debugging C programs, especially if you are working with older C programs. See the "Function Declarations" section in Chapter 5 of *IBM C/2 Fundamentals* for details on function declarations and argument type lists.

You can use the /Zg option even if the source program already contains some function declarations. The compiler accepts more than one occurrence of a function declaration, as long as the declarations do not conflict. No conflict occurs when one declaration has an argument type list and another declaration of the same function does not, as long as the declarations are otherwise identical.

Note: If you use the /Zg option and the program contains formal parameters that have structure, enumeration, or union types (or pointers to such types), then the declaration of each structure, enumeration, or union must have a tag. For example, use the following form:

```
struct tagA {

⋮
} A;
```

Your program can include calls to IBM C/2 run-time library routines. The INCLUDE files provided with IBM C/2 contain function declarations so that you can enable type-checking on library calls.

Example

The following command causes the compiler to produce argument-type lists for functions defined in FILE.C:

```
CC FILE.C /Zg; > FUN_DCL.H
```

The output is redirected to the file FUN_DCL.H; later, the argument type lists can be included in the FILE.C to enable argument-type checking for FILE.C.

Using the Floating-Point Options

IBM C/2 offers several methods of handling floating-point operations. This section provides an overview of the floating-point options available and discusses the default floating-point behavior.

Selecting the Floating-Point Options /FPa, /FPc, /FPc87, /FPi, /FPi87

Select a floating-point operation by using one of the following options.

Format

```
/FPa    Produces floating-point calls and selects alternate math library
/FPc    Produces floating-point calls and selects emulator library
/FPc87  Produces floating-point calls and selects 8087/80287/80387 library
/FPi    Produces in-line instructions and selects emulator library
/FPi87  Produces in-line instructions and selects 8087/80287/80387 library.
```

IBM C/2 can use a numeric coprocessor or emulate numeric operation through an emulator library. The emulator library (EM.LIB) provides a large subset of the software functions of a numeric coprocessor. The emulator can perform basic operations as accurately as a numeric coprocessor. However, the emulator routines used for transcendental math functions differ slightly from the corresponding coprocessor functions, causing a slight difference (usually within 2 bits) in the results of these operations. If you selected the emulator library when you installed IBM C/2, the installation program used EM.LIB to build combined libraries *x*LIBCEz.LIB.

By default, IBM C/2 handles floating-point operations by generating inline instructions (/FPi option). The emulator library is loaded, but if a numeric coprocessor is present at run time, the coprocessor is used instead of the emulator. This method of handling floating-point operations works whether or not you have a coprocessor installed. You do not have to give a floating-point option at compile time unless you want to use one of the other options described here.

When you compile a source file using one of the floating-point options, the name of the required floating-point library (or libraries) is placed in the object file. At link time, the linker refers to the names in the object file to link with the appropriate libraries. You can override the library name given in the object file at link time and link with a different library. (See "Changing the Default Libraries" on page 3-15

for more information.) The only restriction on overriding at link time is that you are not allowed to change to the alternate math library after you have compiled by means of the /FPi or /FPi87 option.

If You Have a Numeric Coprocessor

The /FPi87 option produces in-line instructions for a numeric coprocessor. It is the fastest and smallest option available for floating-point operations.

The /FPc87 option is slower than /FPi87 because it makes function calls instead of using in-line instructions. However, /FPc87 is more flexible. It allows you to change your mind at link time (without recompiling the file) and use either the emulator or the alternate math library instead of relying on a numeric coprocessor. This is made possible because calls to floating-point instructions are interchangeable with calls to the emulator and the alternate math library. (See "Changing the Default Libraries" on page 3-15 for instructions on changing libraries at link time.)

Both the /FPi87 and /FPc87 options select the *x*LIBC7*z*.LIB, where *x* is the storage model and *z* is the addressing mode you chose during installation. Whenever you use these options, a numeric coprocessor must be present at run time. If no coprocessor is present, the program does not run and the following message is displayed:

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

The /FPi option produces in-line instructions for a numeric coprocessor and selects the *x*LIBCE*z*.LIB, where *x* is the storage model and *z* is the addressing mode you chose during installation. If a numeric coprocessor is present at run time, it is used. If not, the emulator is used.

The emulator requires approximately 7KB of additional space when loading, so programs that use the /FPi option are larger than programs that use /FPi87. However, /FPi is a useful option when you do not know whether a numeric coprocessor will be available at run time but wish to use it if it is present.

You may not want to use a numeric coprocessor even though one is present. For example, you may be developing programs to run on systems that lack coprocessors. Conversely, you may want to write programs that can take advantage of a coprocessor at run time, even

though you do not have one installed. There are two ways to control the use of a coprocessor:

1. Use the /Fpi (the default) or the /FPc option to specify either the use of a numeric coprocessor or the emulator. To use the emulator even when a coprocessor is present, set the **NO87** environmental variable. See "Using the NO87 Environment Variable" on page 2-54 for more information.
2. Use the /FPc87 or /FPi87 option if you always want to use a coprocessor. Programs compiled with these options fail if a coprocessor is not present at run time.

If You Do Not Have a Numeric Coprocessor

The /FPi option produces in-line instructions for a numeric coprocessor and selects the *x*LIBCE*z*.LIB library, where *x* is the storage model and *z* is the addressing mode you chose during installation. If a numeric coprocessor is present at run time, it will be used. If not, the emulator library is used. Because this option uses in-line instructions, it is the most efficient way to get maximum precision in floating-point operations without a coprocessor.

The /FPi option is the default when you do not specify a floating-point option. The /FPc option produces floating-point calls to the emulator routines in library *x*LIBCE*z*.LIB, where *x* is the storage model and *z* is the addressing mode you chose during installation.

The /FPc option is slower than /FPi because it makes function calls instead of using in-line instructions. However, /FPc is more flexible than /FPi. When you use the /FPc option, you can change your mind at link time (without recompiling the file) and use a numeric coprocessor or the alternate math library instead of using the emulator. This is made possible because the same function call interface is provided in all three libraries: the 8087/80287/80387 library, the alternate math library, and the emulator library. See "Changing the Default Libraries" on page 3-15 for instructions on changing libraries at link time.

The /FPa option produces floating-point calls and selects the combined library *x*LIBCA*z*.LIB, where *x* is the storage model and *z* is the addressing mode you chose during installation. The alternate math library uses a subset of the IEEE (Institute of Electrical and Electronics Engineers, Inc.) standard format numbers, sacrificing some

accuracy for speed and simplicity. Infinities, NaNs, and denormal numbers are not used.

The alternate math routines use exactly the same format for the exponent and mantissa as the full IEEE format with the following exceptions:

Notes:

1. IEEE Infinites and NaNs are presented as numbers with the exponent field containing all bits on. This combination is not generated by the alternate math package. If one is seen (passed in through a data file or constructed), it is treated as a normal number that happens to be larger than the normal finite precision range. If an IEEE infinity is seen, it will be treated as 2*maximum finite IEEE number for that precision because the exponent field is 1 larger.
2. Denormal numbers are numbers that have all bits off in the exponent field and some nonzero mantissa bits. The alternate math package treats this number as if it were zero.
3. Alternate math numbers have exactly the same precision as the full IEEE numbers: 23 mantissa bits in single and 53 mantissa bits in double. The maximum range for alternate math numbers is the same as that of the full IEEE representation.

   The minimum nonzero values are :

   ± 1.175e-38 (single)

   ± 2.226e-308 (double)

   The full IEEE representation is given in *IBM C/2 Fundamentals*, Chapter 5.

Calls to the alternate math library provide the fastest and smallest option when you do not have a numeric coprocessor. With the /FPa option, you can change your mind at link time and use either the emulator or a numeric coprocessor.

In some cases, you may want to write programs that can take advantage of a numeric coprocessor at run time, even though you do not have one installed. See "If You Have a Numeric Coprocessor" on page 2-33 for a description of the appropriate options.

Compatibility Between Floating-Point Options

Each time you compile a source file, you can specify a floating-point option. When you link more than one source file to produce an executable program file, you are responsible for ensuring that floating-point operations are handled consistently and that the environment is set up properly to allow the linker to find the required libraries.

Note: To build libraries of C routines that contain floating-point operations, the /FPc floating-point option is recommended for all compiling because it offers the most flexibility.

When you compile a file using the /FPi or /FPi87 option, in-line instructions are produced. When you use the /FPi87 option, the combined library *x*LIBC7*z*.LIB must be present at link time, and a numeric coprocessor must be present at run time. When you use the /FPi option, the combined library *x*LIBCEz.LIB must be present at link time. When these requirements are satisfied, object files produced by using the /FPi and /FPi87 options can be linked together without compatibility problems. These object files also can be linked with object files produced by using the /FPa, /FPc, or /FPc87 options.

When a file is compiled with the /FPa, /FPc, or /FPc87 options, floating-point function calls are produced. Each option places the name of the appropriate library file or files in the object file. However, when linking several of these object files, you must be aware of the process used to resolve the function calls.

Floating-point calls to the emulator, the alternate math library, and numeric coprocessor instructions are interchangeable. Only one library is used at link time to resolve the calls; the same program cannot make calls to more than one library. You must indicate to the linker which of your combined libraries *x*LIBC*yz*.LIB you want to link with the program.

At link time, give the /NOD (no default library search) option; then give the name of the combined library file you want to use in the libraries field. This library overrides the names in the object files, and all floating-point calls then refer to the named library.

Using 80186/80188, 80286 or 80386 Processors /G0, /G1, /G2

This option enables the appropriate instruction set for the processor type you are using. Although it is usually advantageous to enable the appropriate instruction set, you are not required to do so. If you have an 80286 or 80386 processor, for example, but want your code to run on an 8086, use the 8086/8088 instruction set.

Format

```
/G0
/G1
/G2
```

/G0 Enables the instruction set for the 8086/8088 processor. You need not specify this option explicitly because the 8086/8088 instruction set is used by default. Programs compiled in this way also run on the 80186/80188, 80286, and 80386.

/G1 Enables the instruction set for the 80186 or 80188 processors.

/G2 Enables the instruction set for the 80286 and 80386 processors.

Setting the Warning Level /W /w

These options set the level of warning messages produced by the compiler and direct the compiler to display messages about statements that may not compile as the programmer intends. Warnings indicate potential problems, not necessarily actual errors.

Format

```
/Wnumber
/w
```

To use the /W option, choose one of the warning levels described in the following table and specify the level *number* after the option. The /w option provides a shorter way to say /W 0 and has the same effect. The /w option is available only with the CL command, not with CC.

Warning Levels

Level	Warning
0	Suppresses all warning messages. Only messages about actual syntactic or semantic errors are displayed.

1	The default. Warns about potentially missing statements, unsafe conversions, and other structural problems. Also, warns about overt type mismatches.
2	Warns about all automatic data conversions.
3	Warns of usage that does not comply with American National Standards Institute (ANSI) standards.

The higher number levels are especially useful in the earlier stages of program development when messages about potential problems are most helpful. The lower number levels are best for compiling programs whose questionable statements are intentionally designed.

Example
The following example directs the compiler to perform the highest level of checking and produces the greatest number of warning messages:

```
CC /W 3 MAIN.C;
```

Compiler Exit Codes

The C compiler control program returns an exit code of 0, 2, or 4 to indicate the status of compiling. The exit code is useful with the DOS BATCH command IF ERRORLEVEL; it allows you to test for the success or failure of compiling before proceeding with other tasks in the batch file. The exit codes are defined as follows:

Code	Meaning
0	Successful compiling. Compiling can be successful even if warning messages are produced.
2	Unsuccessful compiling due to program errors.
4	Unsuccessful compiling due to system-level errors (such as insufficient disk space) or compiler internal errors.

See Appendix A for information about specific error messages.

Preparing for Debugging /Zd, /Zi, /Od

The following commands produce object-file characteristics necessary for debugging with CodeView.

Format

/Zd
/Zi
/Od

/Zd Produces an object file containing line number records that correspond to the line numbers of the source file. Useful for passing an object file to a symbolic debugger. The debugger can use the line numbers to refer to program locations. /Zd has no effect on the generated code but increases the size of .OBJ files because of the line number information.

/Zi Produces an object file containing full symbolic debugging information for use with the CodeView symbolic debugger. This object file includes full symbol table information and line numbers.

/Od Tells the compiler not to perform optimization. Without the /Od option, the default is to optimize. Optimization can involve rearrangement of instructions. If you optimize before debugging, it may be difficult to recognize and correct your code.

/Zi implies /Zd also, so you do not need to give /Zd when you give /Zi. /Zi does not affect code generation either except for the one case where /Zi is specified and no optimization flags are given; for example:

```
CL /AM /I\include /Zi /c foo.c
```

In this case, the /Zi suppresses some of the optimizations that involve code motion. This has less impact than /Od, which suppresses all optimizations. You are not required to use /Od when compiling for CodeView. Certain optimizations such as jump shortening make the code much easier to follow and debug. Using /Zi without explicit optimization flags does most of the default optimizations, but not the ones that make the code hard to follow. Using /Zi in conjunction with any explicit optimization flag (/O<*anything*>) performs all requested optimizations; nothing is suppressed.

See "Optimizing /O" on page 2-40 for information about other optimization options.

Example

The following command produces an object file named TEST.OBJ that contains line numbers corresponding to the line numbers of TEST.C. A listing file TEST.COD is created and the source listing is suppressed. No optimization is performed.

```
CC TEST.C, ,NUL /Zd /Od, TEST.COD;
```

The following command produces an object file named TEST.OBJ that contains line numbers and symbol table information corresponding to TEST.C. It also creates a source listing file TEST.LST.

```
CC TEST.C,, TEST /Zi/Zd/Od;
```

Optimizing /O

The optimizing procedures available with IBM C/2 can reduce the storage space and run time required for a compiled program by eliminating unnecessary instructions and rearranging code. The compiler performs some optimization by default. You can use the /O options to exercise greater control over the optimizations performed. See "Advanced Optimizing" on page 2-55 for information about additional advanced optimizing procedures.

Format

/Ostring

The *string* influences how the compiler performs optimization. The *string* is formed from the following:

Letter	Optimizing Procedure
a	Cancels alias checking
d	Disables optimization
l	Enables loop optimization
n	Disables unsafe loop optimizations
p	Disables certain floating point optimizations
s	Favors code size during optimization
t	Favors run time during optimization
w	Assumes called functions can change local variables
x	Performs maximum optimization.

The letters can appear in any order. For example, /Oat and /Ota have the same effect. You can also specify more than one /O option. For example, /Oa /Ot is equivalent to /Oat.

When you do not give an /O option to the CC command, it uses /Ot to favor run time speed during optimizing. To cause the compiler to favor code size instead, use the /Os option.

The /Od option turns off optimization. This option is useful in the early stages of program development to avoid optimizing code that will be changed. Because optimization may involve rearrangement of instructions, you also may want to specify the /Od option when you use a debugger with your program or when you want to examine an object file listing. If you optimize before debugging, the code can be difficult to recognize and correct.

The /Ol option enables loop optimization. The compiler performs special checks to test for constants across loops. This often improves run-time performance of programs with repetitive calculations. Option /Ox also implies /Ol.

The /Ox option combines optimizing options to produce the fastest possible program. Its effect is equivalent to the following for CL:

```
/Oailt /Gs
```

Its effect is equivalent to the following for CC:

```
/Oalt /Gs
```

The /Ox option removes stack probes, relaxes alias checking, favors code run time over code size, and enables loop optimization. When you use the CL command, it also generates all intrinsics for the functions listed under "Declaring Functions as Intrinsic /Oi" on page 3-66.

The /Op option disables certain floating-point optimizations. The compiler tries to keep floating-point values in extended-precision form and stores them only when necessary. This is not desirable for some algorithms, such as convergence algorithms that try to find the minimum or maximum floating point values recognized by an implementation. Optimization with the /Op option forces writing to storage when the source requests it, which limits the precision of floating-point calculations.

The /Oa option cancels alias checking. The compiler performs alias checking to make sure that it does not eliminate instructions incorrectly when you refer to the same storage location by more than one name. Include the /Oa option only when you are sure that your program does not use aliases.

For example, consider the following code fragment:

```
int count, *pc;
pc = &count;
count = 0;
  .
  .
  .
(*pc)++;
  .
  .
  .
count = 0;
```

The reference to count by means of a pointer, (*pc), is known as an alias for count because it provides another way to get access to the same storage location. When the compiler performs alias checking, it detects the indirect reference to count through *pc* and does not eliminate the second instruction that assigns zero to count.

The /Oa option tells the compiler that your program does not use aliases. Therefore, the compiler does not check for indirect references, such as the reference to count through a pointer. It would be an error to use the /Oa option with the above example. The compiler sees only that the same value, 0, is assigned to count twice, without any intervening assignments that change its value. The second assignment would be considered redundant and would be eliminated in the optimization stage, possibly causing the program to give incorrect results.

The /Ow option has the same effect as /Oa except that the compiler assumes that any function call can potentially alter the value of any variable, including local variables. For example, the following program prints the word "pass" when compiled with the /Ow option and prints the word "fail" when compiled with the /Oa option.

```
main()
{
    int *p, i;
    i = 5;
    sample((int)&i);
    if (i != 5)   puts("pass");
    else  puts("fail");
}

#ifdef EXEC
sample( a )
int a;
{
  *(int *)a = 2;
}
#endif
```

The /On option turns off potentially unsafe loop optimizations in programs compiled with the /Ol or /Ox options. When you use the /On option the compiler does not perform the following types of loop optimizations:

Hoisting division operations out of loops: This type of optimization can cause problems in code such as the following:

```
for (i=0; i<=99; i+=10)
{
    if (denom != 0)
    {
      array[i] += (numer/denom);
         printf("%f ", array[i]);
    }
}
```

When loop optimizations are turned on, the compiler knows that *numer/denom* does not change within the loop. Therefore, it calculates *numer/denom* only once: before the start of the loop, which is before the **if** statement within the loop can check for division by zero.

Loop-induction optimizations: When loop optimizations are turned on, this code:

```
int  larray[400];
unsigned char k, top_val, inc_val, var;

for( k = 3; k < top_val; k += 8 )
{
     larray[k*4] = k*4;
}
```

optimizes to code such as the following:

```
unsigned char t;
for( t = 12; t < top_val*4; t += 32 )
{
     larray[t] = t;
}
```

If the loop-control variable *top_val* in the original code is 64, the induction expression

```
top_val*4
```

overflows the limit for type **unsigned char**, and the loop never terminates. To avoid this problem, use the /On option. For example:

```
unsigned char t;
for( t = 12, k=3; k < top_val; k += 8,
                                t += 32 )
{
     larray[t] = t;
}
```

Use the /On option to solve similar overflow problems in cases where induction variables result from array or pointer references and the offset part of the address is close to wrapping.

Generally, you may want to compile with /On if your programs use arrays that are larger than 32KB or if divide-by-zero or infinite-loop errors occur in programs compiled with the /Oi option.

Example

The following command tells the compiler to relax alias checking and to optimize for faster execution time when it compiles FILE.C:

```
CC FILE.C /Ota;
```

Compiling Large Programs

If you are compiling a large program or linking compiled files together that form a large program with more than 64KB of data or code, use one of the storage models described in the next section.

Working with Storage Models /A

The CC command lets you create programs of a variety of sizes and purposes using the /A options. These storage model options allow you to set up storage that is best for your program and determine how the system loads the program for execution.

Format

```
/AS    (small)
/AM    (medium)
/AC    (compact)
/AL    (large)
/AH    (huge)
```

The compiler uses the small model by default.

The terms **near**, **far**, and **huge** are important for understanding the concept of storage models. Depending on its size (and the use of **near**, **far**, and **huge** as explained under "Using the Near, Far, and

Huge Keywords" on page 2-58), a program may require more segments for its code or data. The program size includes any data and code required for library routines.

Five commonly used storage models are available to the CC command: the small model, the medium model, the compact model, the large model, and the huge model. Library support is provided for each of these standard models. Each model defines a different type of program structure and storage. Only the memory models chosen when you installed the compiler are available for use. To build libraries for additional memory models, repeat the SETUP installation using the /L option.

Program	Description
Small-model	Typically short or make limited use of memory. Code and data for these programs each occupy one segment and are limited to 64KB each (128KB maximum total). Most programs fit easily in this model; that is why small-model is the default.
Medium-model	Typically have a large number of program statements but a relatively small amount of data.
Compact-model	Typically have a large amount of data but a relatively small number of program statements.
Large-model	Use a large amount of data storage during normal processing.
Huge-model	Similar to large-model programs but may contain arrays that require more than 64KB of storage.

In all the models, no matter how large the program, no single object file can exceed 64KB. When you choose one of these storage models, the compiler operates with certain assumptions about the addresses of code and data for your program.

A small-model program stores all code in a single segment, and all data in a single segment. Because the segment addresses are constant for all code items and all data items, the segment address is not required each time an item is addressed. Instead, any items in the program can be addressed with an offset from the segment address. Only 16 bits are required to store an offset from an address, as opposed to 32 bits for a full segment plus offset address. Thus, the compiler produces 16-bit (**near**) pointers for use in small-model programs. This is the smallest and fastest option.

A medium-model program uses multiple segments for code and a single segment for data. The address of a function, for example, in a medium-model program must include the address of the appropriate code segment and the offset of the beginning of the function from the base of that segment. Full 32-bit (**far**) pointers are produced by the compiler to access code items in a medium-model program. However, an offset is sufficient for data items because all data resides in one segment. Data items are accessed with **near** pointers in a medium-model program. The medium model provides a useful trade-off of speed and space because most programs refer more frequently to data items than to code.

In compact-model programs, the default is that the compiler gets access to code items with **near** addresses and to data items with **far** addresses. You can cancel the default by using the **far** keyword for code and the **near** and **huge** keywords for data.

A large-model program requires the compiler to produce **far** pointers for both code and data items because multiple segments are allotted for both code and data. The large model is useful because it can accommodate very large programs. The 64KB limitation on array size in the large-model program allows the compiler to perform address arithmetic on just 16 bits (the offset portion) of the address to refer to individual elements of the array. This is more efficient than using a full 32-bit address and is possible because all elements of an array are known to reside in the same segment.

The huge model removes the 64KB restriction on arrays, allowing an array to span more than one segment. This means that address arithmetic for array elements in huge-model programs is not limited to the offset portion of the address but must take into account the segment address. Thus, in addition to using **far** pointers for both code and data items (as in a large model), a huge-model program also produces **far** pointers for individual elements of an array.

Some restrictions apply to arrays composed of structures or unions. To provide efficient addressing, structures and unions are not permitted to cross segment boundaries. This has the following implications:

- No structure or union element can be larger than 64KB.
- For arrays larger than 128KB, a structure or union element of the array must have a size in bytes equal to a power of 2 (for

example, 2 bytes, 4 bytes, 8 bytes, 16 bytes, and so on). If the array is smaller than 128KB, this rule does not apply.

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 a **size_t** value, (which in IBM C/2 is equal to an **unsigned int**, but the size in bytes of a huge array is a **long int** value.) To solve this discrepancy, the IBM C/2 compiler produces the correct size of a huge array when the following type cast is used:

```
(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 C compiler gives the correct result when you use the following type cast:

```
(long)(huge_ptr1 - huge_ptr2)
```

To provide additional flexibility within the standard storage models, IBM C/2 allows you to override the default addressing conventions for individual program items by using the special **near**, **far**, and **huge** keywords. These keywords let you access an item with either a **near**, **far**, or **huge** pointer. This is particularly useful when you have a very large or infrequently used data item that you want to access from a small- or medium-model program.

Specifying a Combined Library for Linking /Lc, /Lp

These options allow you to build an executable file that runs in DOS mode when compiling in OS/2 mode and to create an OS/2 mode executable file when compiling with DOS. These options are available only with the CL command, not with CC.

Format

```
/Lc
/Lp
```

/Lc creates a DOS mode executable file, and /Lp creates an OS/2 mode executable file. To use each option, your combined libraries must have the names *x*LIBC*yz*.LIB. For example, to use /Lc, the DOS mode library *x*LIBC*y*R.LIB, depending on the memory model and floating-point option, must be visible to the linker. If you have

renamed one of your combined libraries to xLIBCy.LIB, you link to the functions in that library by not specifying the /Lc or /Lp options. To create family applications, you must first create an OS/2 mode executable file. See "Creating Bound Applications /Fb" on page 3-63.

Example

```
CL  /Lc   main.c
```

Advanced Topics

IBM C/2 offers a number of advanced programming options that give you control over compiling and the final form of the executable program. This section describes the advanced options.

Enabling Language Extensions /Ze, /Za, /Zc

Format

```
/Ze
/Za
/Zc
```

/Ze enables C language extensions and is the default. The /Ze option allows:

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

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

- Casting data pointers to function pointers, as in the following example.

```
int *ip;
int sample();

((int (*)())ip)();
```

- Casting function pointers to data pointers, as in the following example.

```
int *ip;
int sample();

ip = (int *)sample;
```

Note that the preceding example generates an illegal cast error in programs compiled with the /Za option. Generally, casts generate a "non-standard extension used" warning at level 3 or higher.

According to the American National Standards Institute (ANSI) standard, the way to cast from data pointers to function pointers and from function pointers to data pointers is to

1. Cast to an integral type (**int** or **long** depending on pointer size)
2. Cast to the final pointer type.

In a small-model program, the preceding examples could be rewritten as follows to conform to the ANSI convention:

```
ip = (int *)(int)foo;    /* cast function
                  pointer to data pointer */

((int (*)())(int)ip)(); /* cast data
             pointer to function pointer */
```

Examples like those above work correctly whether or not the program is compiled with the /Za option. Note that the compiler generates identical code in both cases; the only difference is that one form is ANSI-compatible and the other is not.

When you use the /Za option, the compiler issues an error message whenever you use a valid construction that does not conform to the ANSI standard.

The /Za option causes the compiler to define the identifier NO_EXT_KEYS. In the include files provided with the run-time library functions, the compiler uses this identifier with **#ifndef** to conditionally compile blocks of text containing the keywords **cdecl**, **far**, **interrupt**, or **near**.

You can also use the /Za option to restrict the valid base types of bit-fields. When you use /Za, any bit-field must be either **int**, **signed int**, or **unsigned int**. Bit-fields of width zero force alignment to an **int** boundary.

IBM Extension

IBM C/2 considers the identifiers in the following list to be keywords when processing a file:

cdecl	**fortran**	**_loadds**	**_saveregs**
_export	**huge**	**near**	**pascal**
far	**interrupt**		

To transfer C programs from other systems these are not keywords in, use the /Za option to tell the compiler to treat these words as ordinary identifiers.

End of IBM Extension

/Zc forces the compiler to allow names declared with the **pascal** and **fortran** modifiers to be used without regard for case.

Example

```
int pascal foo(void);
int pascal Foo(void);
int pascal FOO(int); /* error: redefinition */
```

Under the /Zc option, the second declaration is equivalent to the first and the third produces a compiler error because it tries to redeclare the same name with an argument list. Such a declaration is inconsistent with the first two declarations.

```
Foo();  /* A reference to foo */
FOO();  /* Another reference to it /*
```

You can also use the /Ze option to allow the use of casts to produce **lvalues**, as in this example:

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

The preceding example could be rewritten to conform to the ANSI standard as follows:

```
p = (int *)((char *)p + sizeof(long));
```

You can also use the /Ze option to allow redefinitions of **extern** items as static, as in this example:

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

Packing Structure Members /Zp

When storage is reserved for structures, structure members larger than a **char** are ordinarily stored beginning at an **int** boundary. To conserve space you can store your structures more compactly. The /Zp option and the **pack** pragma cause structure data to be packed tightly into storage. These options are also useful to read existing packed structures from a data file. /Zp is available with both CC and CL, but /Zp*n* is available only with CL.

Format

```
/Zp[1|2|4.]
#pragma pack([1|2|4])
```

Use /Zp 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 is stored on *n*-byte boundaries. The default is 1-byte.

On most processors, using the /Zp option causes a program to run slower because of the time required to unpack structure members when accessing them. This option also reduces efficiency when a program gets access to 16-bit members (with **int** type) that begin on odd boundaries.

Use the **pack** pragma to specify packing other than that specified in the /Zp option for particular structures. Give the **pack(*n*)** pragma (where *n* is 1, 2, or 4) before the structures you want packed differently. To reinstate the packing given in the /Zp option type the **pack()** pragma with no arguments.

Example

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

```
CC /Zp PROG.C;
```

In IBM C/2 Version 1.10, a structure or union whose members have only the types **char** or **unsigned char**, or any array of those types, is byte-aligned. That is, every such structure or union is sized exactly, not padded to an even number of bytes as in IBM C/2 Version 1.00. If a structure or union contains any member whose type is not **char** or **unsigned char**, there is no difference between the versions. This

change makes a difference only if the sum of the sizes of the members is odd; if the sum is even, then no alignment byte is added. This example demonstrates the difference:

```
char a;
struct {char a, b, c} y;
struct {char a[3]} z;
main()
{
    printf( "Size of y = %d\n", sizeof(y) );
    printf( "Size of z = %d\n", sizeof(z) );
}
```

If you compile the preceding with IBM C/2 Version 1.00 (not using /Zp), the output is:

```
Size of y = 4
Size of z = 4
```

If you compile the preceding code with IBM C/2 Version 1.10 (not using /Zp), the output is:

```
Size of y = 3
Size of z = 3
```

Programs affected by this change should not mingle object files generated by the two versions. You should recompile all of the objects with IBM C/2, Version 1.10. If that remedy is impossible (for instance, if you do not have source code for the objects in question), it is possible to change include files to add an extra **char**-type member at the end of any odd-sized structure or union. This change adds the alignment member that the compiler no longer adds.

Suppressing Default Library Selection /Zl

Ordinarily, the compiler places the names of the default libraries (containing the standard C library plus the selected floating-point library) in the object file for the linker to read. This allows the default libraries to be linked with a program.

This option suppresses the selection of default libraries. No library names are placed in the object file. As a result, the object file is slightly smaller.

Format

```
/Zl
```

Use the /Zl option when you are building a library of routines. It is not necessary for every routine in the library to contain the default library information. Although the /Zl option saves only a small amount of space for a single object file, the total space savings is significant in a library containing many object modules. When you link a library of object modules created with the /Zl option with a C program file compiled without the /Zl option, the default library information is supplied by the program file.

Example

The following two commands create an object file named ONE.OBJ that contains the name of your combined library, x*LIBCyz*.LIB, and an object file named TWO.OBJ that contains no default library information.

```
CC ONE.C;
CC /Zl TWO.C;
```

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

```
LINK ONE+TWO;
```

Changing the Default char Type /J

In IBM C/2, the **char** type is signed by default. If you widen a **char** value to an **int**, the result is sign extended. To have an unsigned default, use the /J option. This causes the compiler to zero-extend the **char** type when you widen it to an **int** type.

Format

```
/J
```

If you declare a **char** value explicitly **signed**, the /J option does not affect it, and the compiler sign extends the value when you widen it to an **int**. When you invoke CL with the /J option, a new predefined identifier, _CHAR_UNSIGNED, is defined. This identifier is used with **#ifndef** in the file LIMITS.H to define the range of the default **char** type. Note that compiling with /J reduces by 1 the number of constant and macro definitions that you can give at the command prompt.

Controlling Floating-Point Operations

By default, the compiler handles floating-point operations by using calls to an emulator library, which emulates the operation of a numeric coprocessor. If a numeric coprocessor is present at run time, it is used. The floating-point (/FP) options give you a choice of five different methods of handling floating-point operations.

The advantages and disadvantages of each of the five /FP options are described under "Selecting the Floating-Point Options /FPa, /FPc, /FPc87, /FPi, /FPi87" on page 2-32. You should read that discussion of floating-point options before reading this section. This section discusses two additional ways to control floating-point operations: by changing libraries at link time and by using the NO87 environment variable.

Using the NO87 Environment Variable

Programs compiled using the /FPc or /FPi option use a numeric coprocessor at run time if one is installed. You can override the selection of the coprocessor at run time and force the use of the emulator by setting an environment variable named NO87.

If NO87 is currently set to any value when the program runs, use of the numeric coprocessor is suppressed. The value of the NO87 setting is printed on the standard output as a message. The message prints only if a numeric coprocessor is present and suppressed; if no coprocessor is present, no message appears. If you do not want a message to print, set NO87 equal to one or more spaces; nothing will be printed.

Only the presence or absence of the NO87 definition is important in suppressing use of the coprocessor. The actual value of the NO87 setting is used only for printing the message.

The NO87 variable takes effect with any program linked with the emulator library, that is, a combined library with name of the form *x*LIBCEz.LIB. It has no effect on programs linked with combined libraries named *x*LIBC7z.LIB.

Example

The following command causes the message Use of coprocessor suppressed to appear on the screen when a program that can use a coprocessor is run.

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

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

```
SET NO87=space
```

Advanced Optimizing

This section describes additional optimizing procedures that can be used with the optimizing options described in "Optimizing /O" on page 2-40 to create more efficient programs from your code.

Removing Stack Probes /Gs

You can reduce the size of a program and speed up performance slightly by using the /Gs option to remove all stack probes. A *stack probe* is a short routine called by a function to check the program stack for available space.

Format

```
/Gs
#pragma check_stack[+|-]
#pragma check_stack[-]
```

The stack probe routine is called at every entry point. Ordinarily, the stack probe routine generates a message when it detects a stack overflow. When you use the /Gs option no message prints.

Use the /Gs option when a program is known not to exceed the available stack space. For example, stack probes might not be needed for programs that make very few function calls.

Although the /Gs option, combined with the /Osa option (described with the /O *string* options under "Optimizing /O" on page 2-40) makes the smallest possible program, use it with care; removing stack probes from a program can cause some execution errors to go undetected.

Use the **check_stack** pragma when you want to turn stack checking on or off only for selected routines, leaving the default, determined by the presence or absence of the /Gs option, for the remaining routines. To turn off stack checking, put the following line before the definition of the function that you do not want to check.

```
#pragma check_stack(OFF)
```

The preceding line disables stack checking for all routines that follow it, not only the routines on the same line. To reinstate stack checking, insert the following line:

```
#pragma check_stack(ON)
```

If you omit the trailing + or –, or the empty parentheses of the pragma **check_stack**, stack checking follows the default. The following table shows the relationship between the **check_stack** pragma and the /Gs option.

Syntax	Compiled with /Gs?	Result
#pragma check_stack **#pragma check_stack–** **#pragma check_stack()** **#pragma check_stack(OFF)**	Yes	Stack checking off for the routines that follow.
#pragma check_stack **#pragma check_stack+** **#pragma check_stack()** **#pragma check_stack(ON)**	No	Stack checking on for the routines that follow.
#pragma check_stack+ **#pragma check_stack(ON)**	Yes	Stack checking on for the routines that follow.
#pragma check_stack– **#pragma check_stack(OFF)**	No	Stack checking off for the routines that follow.

Example

The following command optimizes the file FILE.C by removing stack probes with the /Gs option and relaxing alias checking with the /Ota option. The letter *t* in the /Ota option tells the compiler to favor execution time over code size in the optimization.

```
CC FILE.C /Ota /Gs;
```

Setting the Data Threshold /Gt

By default, the compiler reserves all static and global data items to the default data segment in the small- and medium-storage models. In compact-, large-, and huge-model programs, the compiler assigns only initialized static and global data items to the default data segment. This option causes all data items whose size is greater than or equal to *number* bytes to be allocated to a new data segment. When you specify *number*, it must follow the /Gt option, with no intervening spaces. When you omit *number*, the default threshold value is 256.

Format

`/Gt[number]`

You can use the /Gt option only with compact-, large-, and huge-model programs because small- and medium-model programs have only one data segment. This option is particularly useful with programs that have more than 64KB of initialized static and global data in small data items.

Mixed-Model Programming

IBM C/2 defines five standard storage models (small, medium, compact, large, and huge) to accommodate programs with differing storage requirements. For an introduction to storage models (using the /A option), see "Working with Storage Models /A" on page 2-44.

One limitation of the predefined storage-model structure is that all pointers for code or data change size at once when you change storage models. To overcome this limitation, IBM C/2 lets you cancel the default-addressing convention for a given storage model and access an item with either a **near**, **far**, or a **huge** pointer. This is particularly useful with a very large or infrequently used data item that you want to get from a small- or medium-model program. You can get access to that item in another segment, saving space in the default data segment.

You can use the special keywords **near**, **far**, and **huge** to declare near, far, and huge data items and pointers. See "Declaring Data with Near, Far, and Huge" on page 2-59 for more information.

The **near** keyword defines an object with a 16-bit address. The **far** keyword defines an object with a 32-bit segmented address. You can get access to any data item or function with a **far** pointer. However, the size of a **far** data item is restricted to 64KB maximum (one segment). The address arithmetic required to refer to individual elements of a **far** item is performed on just 16 bits (the offset portion) of the address, because all elements are known to reside in the same segment.

The **huge** keyword identifies a data object with a full 32-bit segmented address. A **huge** data item can exceed 64KB. Because elements of a **huge** array occupy more than one segment, full 32-bit address arithmetic is required to refer to individual elements of the object. Certain restrictions apply to **huge** objects; these restrictions are outlined in "Optimizing /O" on page 2-40.

In a small-model program, the **far** keyword lets you get access to data and functions in segments outside the program.

In medium- and large-model programs, **near** lets you get access to data with just an offset. In small-, medium-, or large-model programs, the **huge** keyword lets you declare and get access to an array spanning more than 64KB (one segment).

Using the Near, Far, and Huge Keywords
Use the **near** and **far** keywords to create mixed-model programs. These keywords are particularly useful with a very large or infrequently used data item that you want to access from a small- or medium-model program. Use the **far** keyword to reserve a new segment for the data item, and then get access to that item with a **far** pointer, while still using **near** pointers (the default) for other data.

When using the **near**, **far**, and **huge** keywords to change addressing conventions for particular items, you can usually use one of the standard libraries (small, compact, medium, or large) with your program. The large-model libraries are also appropriate for use with huge-model programs. However, you must take care when calling library routines; in general, you cannot pass **far** pointers or addresses of **far** data items to a small-model library routine. Some exceptions to this statement are the library routines **halloc**, **hfree**, and the **printf** family.

You can always pass the *value* of a **far** item to a small-module library routine. For example:

```
long far time_val;

time(&time_val);/*Illegal */
printf("%ld\n", time_val); /* Legal */
```

If you use the **near**, **far**, or **huge** keywords, it is recommended that you use function declarations with argument-type lists to ensure that pointers are passed to functions correctly; see "Generating Function Declarations /Zg" on page 2-30.

For more information on library routines and memory models, see "Using Huge Arrays with Library Functions" in Chapter 1 of *IBM C/2 Language Reference*.

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, 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 such as **char far* *p**, think of the **far** keyword and the item to its right as being a single unit. In this case, **p** is a pointer to a far pointer to **char**. The size of **p** depends on the memory model being used. See *IBM C/2 Fundamentals* for complete rules for using special keywords in complex declarations.
- If the item immediately to the right of the keyword is an identifier, the keyword determines whether the item is allocated in the default data segment (**near**) or a separate data segment (**far** or **huge**). For example:

  ```
  char far a;
  ```

  allocates *a* as an item of type **char** with a far address.
- If the item immediately to the right of the keyword is a pointer, the keyword determines whether the pointer holds 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**.

Example

The following examples show data declarations using the **near**, **far**, and **huge** keywords:

```
char a [3000];      /* Example 1: small-model program */
char far b[30000]; /* Example 2: small-model program */
```

The declaration in the first example allocates the array *a* in the default segment; in contrast, the array *b* in the second example may be allocated in any segment. Since these declarations are made in a small-model program, array *a* probably represents frequently used data that was deliberately placed in the default segment for fast access, while array *b* probably represents seldom-used data that might make the data segment exceed 64KB. This forces the programmer to use a larger memory model if it is declared with the **far** keyword. The second example 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];      /* Example 3: large-model program */
char near b[30000]; /* Example 4: large-model program */
```

In Example 3, the speed of access would probably not be critical for array *a*; even though it may or may not be allocated to the default data segment, it is always referenced with a 32-bit address. In Example 4, array *b* is explicitly allocated **near** to improve speed of access in this memory model (large).

```
char huge a[70000]; /* Example 5: small-model program */
char huge *pa;       /* Example 6: small-model program */
```

In Example 5, *a* must be declared as **huge** because it is larger than 64KB. Using the **huge** keyword instead of the standard huge memory model means that the price for using huge data is paid only for this one large item. Other data can be accessed within the default segment. The pointer *pa* in Example 6 could be used to point to *a*. Any arithmetic done with *pa* uses 32-bit arithmetic.

```
char *pa;                  /* Example 7: small-model program */
char far *pb;              /* Example 8: small-model program */
```

In Example 7, *pa* is declared as a near pointer to **char**. The pointer is near by default since the example is in a small-model program. In contrast, *pb* in Example 8 is allocated as a far pointer to **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 the array *a* in Example 1, while *pb* might be used to point to the array *b* in Example 2.

```
char far * *pa;              /* Example 9: small-model program */
char far * *pa;              /* Example 10: large-model program */
```

The pointer declarations in Examples 9 and 10 show the interaction between the memory model chosen and the **near** and **far** keywords; although the declarations for *pa* in these two examples are identical, Example 10 declares *pa* as a far pointer to an array of far pointers to type **char**.

```
char far * near *pb;         /* Example 11: any model */
char far * far *pb;          /* Example 12: any model */
```

In Example 11, *pb* is declared as a near pointer to an array of far pointers to type **char**. In Example 12, *pb* is declared as a far pointer to an array of far pointers to type **char**. Note that in these final two examples, the inclusion of **far** and **near** keywords overrides the model-specific addressing conventions shown in Examples 9 and 10; the declarations for *pb* would have the same effect, regardless of the memory model.

Declaring Functions with Near and Far

The rules for using the **near** and **far** keywords for functions are similar to those for using them with data:

- The keyword always modifies the function or pointer immediately to its right. See "Declarators with Special Keywords" in *IBM C/2 Fundamentals* for more information about rules for evaluating complex declarations.
- If the item immediately to the right of the keyword is a function, the keyword determines whether the function is 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 is 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.

Example

```
char far fun ( );                    /* Example 1: small model */
char far fun ( )
          {

⋮
          }
```

In this example, *fun* is declared as 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 ( );      /* Example 2: large model */
static char far * near fun ( )
          {

⋮
          }
```

In the second example, *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. Since 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 *fun* was declared in.

```
void far fun ( ) ;                   /* Example 3: small model */
void (far * pfun) ( ) = fun;
```

This example declares *pfun* as a far pointer to a function that has a **void** return type 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: If the function pointed to by *pfun* has not been declared **far** or if it is not far by default, then calling that function through *pfun* would cause the program to fail.

```
double far * (far fun) ( );      /* Example 4: compact model */
double far * (far *pfun) ( ) = fun;
```

This example 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 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 as **far**.

Pointer Conversions

Passing pointers as arguments to functions may cause automatic conversions in the size of the pointer argument, since 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 storage 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 argument.

If the forward declaration of a function includes declared argument types, 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 converts pointer arguments to the larger of the default type of the type of the argument. To avoid mismatch arguments, always give the argument types in a forward declaration.

Example

This program produces unexpected results in compact-, large-, or huge-model programs.

```
                        Example 1
main ( )
        {
        int near *x;
        char far *y;
        int z = 1;

        test_fun(x, y, z);   /* x will be coerced to far
                                ** pointer in compact, large
                                ** or huge model
                                */
        }
int test_fun(ptr1, prt2, a)
        int near *ptr1;
        char far *ptr;
        int a;

        {
        printf("Value of a = %d\n", a);
        }
```

If the preceding example is compiled as a small-model program (no memory model options or /AS at the command prompt) or medium-model program (/AM option), 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*.

Since *prt1*, 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 of the 32 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, since both the function call and the function definition are legal; but in this case the program does not work as intended, since 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. In the following example, *test_fun* was declared so the compiler knows in advance about the **near** pointer argument.

Example

```
                        Example 2
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 *prt1;
        char far *ptr2;
        int a;

        {
        printf("Value of a = %d\n", a) ;

        }
```

Note: Reversing the definition order for *test_fun* and *main* in the first example does not avoid pointer coercions; the pointer arguments must be declared in a forward declaration, as in the second example.

Creating Customized Storage Models

A method of managing storage models is to combine features of the standard storage models to create your own customized storage model. You should have a thorough understanding of C storage models and the 8086 architecture before creating your own non-standard storage models, since there is no library support other than the C start-up routines for any of the options that follow. These options are available only with the CL command, not with CC.

The /A*string* option lets you change the attributes of the standard storage models to create your own storage models. The three fields of the string correspond to the code pointer size, the data pointer size, and the stack and data segment setup. The letters allowed in each field are unique, so you can give them in any order after /A. All three letters must be present.

The standard-storage-model options (/AS, /AM, /AC, /AL, and /AH) can be specified in the /A*string* form. As an example of how to construct storage models, the standard-storage-model options are listed below with their /A*string* equivalents:

Standard	Custom Equivalents
/AS	/Asnd
/AM	/Alnd
/AC	/Asfd
/AL	/Alfd
/AH	/Alhd

As an example of the use of customized models, suppose that you want to create a huge-compact model. You want this model to allow huge data items but only one code segment. Then, the option for specifying this model is /Ashd.

Note: For the descriptions that follow, the letters *l* for *long* and *s* for *short* are for code pointers to distinguish them in the storage-model string from the letters for data pointers. The term *short* is the same as *near*, and *long* is the same as **far**.

Producing Code Pointers

Format

```
/Aszz
/Alzz
```

The letter *s* tells the compiler to produce **near** (16-bit) pointers and addresses for all code items. This is the default for small- and compact-model programs.

The letter *l* means that **far** (32-bit) pointers and addresses address all code items. **Far** pointers are the default for medium-, large-, and huge-model programs.

Producing Data Pointers

Format

```
/Anzz
/Afzz
/Ahzz
```

Three sizes are available for data pointers: **near**, **far**, and **huge**. The letter *n* tells the compiler to use **near** (16-bit) pointers and addresses for all data. This is the default for small- and medium-model programs.

The letter *f* specifies that all data pointers and addresses are **far** (32-bit). This is the default for compact- and large-model programs.

The letter *h* specifies that all data pointers and addresses are **huge** (32-bit). This is the default for huge-model programs.

When **far** data pointers are used, no single data item may be larger than a segment (64KB) because address arithmetic is performed only on 16 bits (the offset portion) of the address. When **huge** data pointers are used, individual data items can be larger than a segment (64KB) because address arithmetic is performed on the entire 32 bits of the address.

Setting Up Segments

Format

```
/Adzz
/Auzz
/Awzz
```

The letter *d* tells the compiler that **SS** equals **DS**; that is, the stack segment and the default data segment are combined into a single segment. This is the default for all programs. In small- and medium-model programs, the stack and all data combined must occupy less than 64KB; you get access to any data item with only a 16-bit offset from the segment address in the **SS** and **DS** registers.

In compact-, large-, and huge-model programs, initialized global and static data are placed in the default segment. The address of this segment is stored in the **SS** and **DS** registers. All pointers to data, including pointers to local data (the stack), are full 32-bit addresses. This is important to remember when passing pointers as arguments in large-model programs. Although you may have more than 64KB bytes of total data in these models, there can be no more than 64KB of data in the default segment. The /Gt and /ND options can be used to control allocation of items in the default data segment if a program exceeds this limit. See "Naming Modules and Segments /NM, /NT, /ND" on page 3-68 and "Setting the Data Threshold /Gt" on page 2-57 for more information about these options.

The letter *u* reserves different segments for the stack and the data segments. A segment for global and static data items is reserved for each object file. When you specify the letter *u*, the address in the **DS** register is saved upon entry to each function, and the new **DS** value for the module the function was defined in is loaded into the register. The previous **DS** value restores on exit from the function. Therefore, only one data segment is accessible at any given time.

A single segment must be reserved for the stack and its address stored in the stack register. The stack cannot be placed in a data segment because it must be available throughout the entire program.

The letter *w*, like the letter *u*, sets up a separate stack segment but does not load the **DS** register at each module entry point. This option is typically used when writing application programs that run with an operating system (such as a Presentation Manager application) or

with a program running at the operating-system level. The operating system or the program running under the operating system receives the data intended for the application program and places it in a segment; then, it must load the **DS** register with the segment address for the application program.

Even though *u* and *w* set up a separate segment for the stack, the size of the stack is still fixed at the default size unless this is canceled with the /F*hexnumber* compiler option (CL only) or the /STACK linker option.

Library Support

Most C programs make function calls to the routines in the C run-time library. Library support is provided for the five standard storage models (small, medium, compact, large, and huge) through four separate run-time libraries. When you write mixed-model programs, you are responsible for determining which library (if any) is suitable for your program and for ensuring that the appropriate library is used.

When using the **near**, **far**, and **huge** keywords to change addressing conventions for particular items, you can use one of the standard libraries (small, medium, compact or large) with your program. However, you must take care when calling library routines. For example, you cannot pass **far** data items to a small model library routine.

Controlling the Function Calling Sequence /Gc

The **pascal** and **cdecl** keywords, and the /Gc option let you control the function-calling sequence and naming conventions, permitting your C programs to call and be called by functions written in IBM Pascal.

Format

```
/Gc      /* As a command-prompt option */

[pascal] /* As keywords in a source file */
[cdecl]
```

Because C, unlike Pascal, lets you write functions that take a variable number of arguments, it must handle function calls differently than these languages do. Languages such as Pascal normally push actual parameters to a function in left-to-right order. The last argument in

the list is the last argument pushed. C functions do not know the number of actual parameters. They must push the arguments from right to left. The first argument in the list is the last argument pushed. Also, the calling function in C must remove the arguments from the stack instead of having the called function do it, as in Pascal. If the code for removing the arguments is in the function definition, it appears only once, as in Pascal. If the code for removing the arguments is in the calling function, it appears every time there is a function call (as in C). Because function calls are more numerous than function definitions, the method that Pascal uses often produces slightly smaller, more efficient object modules.

IBM C/2 can produce the Pascal call/return sequence in any one of several ways. For the first method, you may use the the **pascal** keyword. Applied to functions or pointers to functions, this keyword shows that the corresponding function is a Pascal style function, requiring the compiler to use the correct call/return sequence. The following example shows **sort** declared as a function using the alternative call/return sequence:

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

The second method for producing the Pascal call/return sequence uses the /Gc option. If you use the /Gc option, the entire module compiles using the alternative call/return sequence. You might use this method to call all the functions in a C module from another language or to gain the performance and size improvement provided by this call/return sequence. However, if you use the /Gc option, you cannot define functions that take variable numbers of parameters, nor can you call functions, such as the C library functions, that use the C calling sequence. When you use /Gc to compile a module, the compiler assumes that all functions called from that module use the Pascal calling sequence, even if the functions are defined outside that module.

The **cdecl** keyword overcomes these restrictions. When you apply it to a function or a pointer to a function, it shows the compiler that it is to call the associated function using the normal C call/return sequence. This lets you write C programs that take advantage of the more efficient call/return sequence while retaining access to the entire C library, to other C objects, and to your own functions that contain variable-length argument lists.

For convenience, IBM has applied the **cdecl** keyword to the run-time library function declarations in the include files distributed with this compiler.

Using the **pascal** keyword or the /Gc option also affects the naming convention for the associated item (for /Gc, all items). The compiler converts the name to uppercase characters. It does not add the leading underscore character that C usually places before the name. You can apply the **pascal** keyword to data items and pointers, as well as to functions. When applied to data items and pointers to data items, these keywords force the naming convention described above for that item or pointer.

The **pascal** and **cdecl** keywords, like the **near**, **far**, and **huge** keywords are disabled when you use the /Za option. If you use this option, these keywords are treated as ordinary identifiers instead of keywords.

Note: IBM C/2 also supports the keyword **fortran**, which is identical in meaning and function with the keyword **pascal**. You cannot call a C/2 program directly from a main program written in IBM FORTRAN/2. Both the **pascal** and **fortran** keywords mean:

- Fixed length parameter lists are pushed left to right, not variable length parameter lists right to left as usually done in C.
- The called subroutine must remove the fixed number of parameters from the stack, not the caller who would normally adjust the stack pointer in C.
- Names are shifted to all caps; no underscore is prefixed as is usually done in C

Example

In the following example, **var_print** can have a variable number of arguments because it is declared as a function using the normal, right-to-left, C function, call/return, sequence-and-naming convention. The **cdecl** keyword cancels the left-to-right calling sequence set with the /Gc option when compiling a source file this declaration appears in. If this file compiles without the /Gc option, **cdecl** has no effect because it is the same as the default C convention.

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

Chapter 3. Linking A Program

The IBM Segmented Executable Linker (LINK) links object files compiled with the IBM C/2 compiler. The linker produces executable program modules or dynamic link libraries from object files created with IBM C/2. The linker also combines application or dynamic link object files with object files in the .LIB libraries. The linker runs in OS/2 and DOS mode. For information about linking programs to run as OS/2 multi-thread programs, see Chapter 6 of *IBM C/2 Language Reference*.

The linker can produce the following:

- An OS/2 executable module
- An OS/2 dynamic link library
- A DOS executable module.

The linker supports dynamic linking, executable file compression, embedded debug information, and module definition files.

Note: Before linking your C programs, make sure your environment variables are set. See Chapter 2 in *IBM C/2 Fundamentals* for information about setting the appropriate environment variables for your operating system.

The linker's ability to create files to run in the OS/2 mode depend on the following:

Dynamic links: If an external reference is an entry point in a dynamic link library the linker produces an OS/2 mode executable module. The OS/2 mode library DOSCALLS.LIB (supplied with OS/2) contains entries for all of the OS/2 dynamic link routines. You must supply this library to LINK when creating OS/2 mode executable program modules or dynamic link libraries.

Module definition files: If you supply a module definition file, LINK produces an OS/2 mode executable module. Module definition files are optional when creating program modules, but they are required when creating dynamic link libraries. For more information on module definition files, see "Module Definition Files" on page 3-5.

If no module definition file is supplied and no dynamic link entries are found, the linker produces a DOS mode executable program module. Whether you are linking an application or a group of dynamic link routines, the output is one file. If you link an application, the resulting file is called a program module. The linker gives the default extension .EXE to program modules. If you are linking a group of dynamic link routines, the resulting file is called a dynamic link library. When it creates a dynamic link library, LINK uses the extension .DLL.

Note: Dynamic link libraries and OS/2 mode executable files can be used only in OS/2 mode. DOS mode executable files will run only in DOS mode.

How the Linker Works

The linker performs the following steps to combine object modules and produce an executable module:

1. Reads the object modules you submit.
2. Searches the given libraries, if necessary, to resolve external references.
3. Assigns addresses to the segments.
4. Assigns addresses to the public symbols.
5. Reads data in the segments.
6. Reads all relocation references in the object modules.
7. Performs fix-ups (See "Fix-ups" on page 3-52 for more information.)
8. Creates an executable image and relocation information.

The linker produces a list file that shows segment and public symbol addresses and error messages.

The *executable image* contains the code and data that make up the executable file. The *relocation information* is a list of references to locations in the program; the final address of the locations is decided after the operating system loads the program. The format and processing of these relocations is different for the DOS and OS/2 modes.

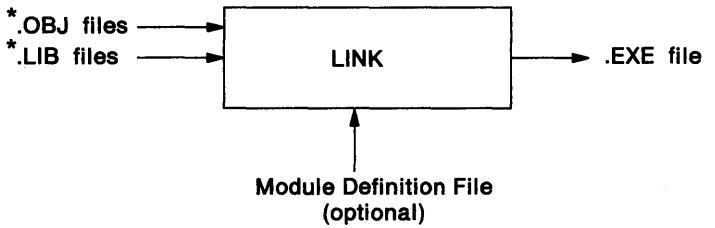
Creating DOS Mode Applications

Create a DOS mode program module by linking the compiled source files (.OBJ) with LINK. The linker uses the compiled source files and a list of library files (.LIB) to produce a program module. The following diagram illustrates the steps required to create a DOS mode application .EXE file:

Creating OS/2 Mode Applications

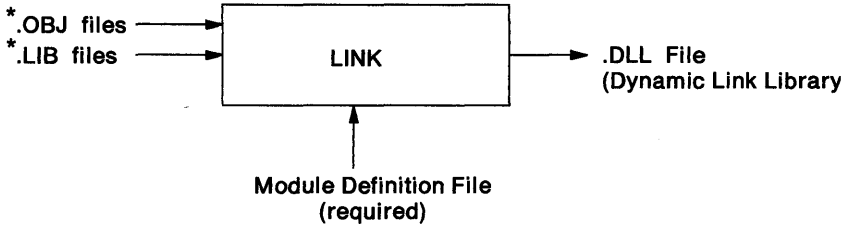
Create an OS/2 mode program module by linking the compiled source files (.OBJ) with LINK. The linker uses the compiled source files, a list of library files (.LIB), and an optional module definition file to produce a program module. A *module definition file* is an ASCII text file containing information about your application. The linker uses this information to help build the .EXE file.

The following diagram illustrates the steps required to create an OS/2 mode application .EXE file:



Creating Dynamic Link Libraries

Create dynamic link libraries by linking compiled dynamic link source files (.OBJ) with LINK. The linker uses the compiled source files and a module definition file to produce a dynamic link library. The module definition file is required because it tells the linker to produce a dynamic link library rather than an executable file. The resulting dynamic link library contains entry points to its dynamic link routines that can be called by other applications.



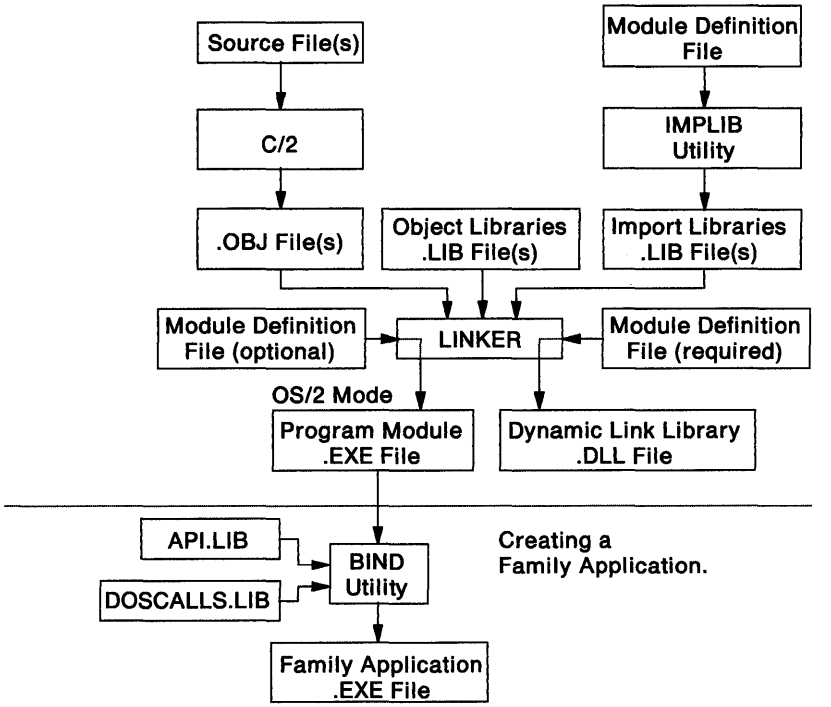
You can use the IMPLIB utility to create a .LIB file for the dynamic link library. The .LIB file resolves external references to the dynamic link routines. Ordinary .LIB files resolve external references by supplying the object code referenced. The .LIB files built by IMPLIB resolve external references by supplying special records that contain pointers to the target dynamic link library and entry points.

For more information about the IMPLIB utility, see the programming information for OS/2. For more information about dynamic linking see Chapter 6 of the *IBM C/2 Language Reference*.

Creating Family Applications

A *Family Application* is a program module that can run in DOS mode, and in the OS/2 mode. To create a Family Application you must use the BIND utility. For more information about BIND, see the programming information for OS/2.

The following diagram illustrates the steps required to create a Family Application .EXE file:



Creating a Family Application.

Module Definition Files

A *module definition file* is an ordinary ASCII text file that provides additional information about the .EXE or .DLL file being created. A module definition file is required to create a dynamic link library and is optional to create a program module. The module definition file contains one or more definition statements. Each statement defines some aspect of the program module or dynamic link library, such as segment attributes or functions exported. For a full description of each statement see "Module Definition File Statements" on

page 3-34. You can choose any filename and extension for a module definition file. The default extension LINK searches for is .DEF.

Creating Module Definition Files

The following sections explain how to create module definition files for dynamic link modules and program modules.

Module Definition Files for Dynamic Link Libraries

A module definition file for a dynamic link library must contain a LIBRARY statement specifying that the file being created is a dynamic link library. The file must also contain an EXPORTS statement that lists the routines within the dynamic link library to be exported. Routines in the dynamic link library cannot be used if they are not listed.

The following example shows a module definition file for a dynamic link library:

```
;Sample Module Definition File
LIBRARY

DESCRIPTION  'Sample .DEF file for Dynamic Link Library'

CODE     LOADONCALL

EXPORTS
 Init  @1
 Start @2
 End   @3
 Load  @4
 Save  @5
```

The first line of the sample module definition file is a comment. A comment can appear on a line by itself or on the same line as a definition statement, as long as it appears after the definition.

Note: A semicolon must precede a comment. In this example, the LIBRARY statement tells LINK that the file being created is a dynamic link library (.DLL). The CODE statement instructs the system to load code segments on demand. The EXPORTS statement lists the dynamic link routines to be exported by name and ordinal.

Dynamic link routines use the stack of their caller. OS/2 moves and discards code segments of dynamic link routines and moves and swaps data segments to take the best advantage of memory.

Module Definition Files for Program Modules

A module definition file for a program module is optional. For example, you can use it to change the stack size or to cancel the default segment characteristics. The following example shows a module definition file for a program module:

```
;Sample Module Definition File
NAME

DESCRIPTION  'Sample .DEF file for Application'

CODE       LOADONCALL

STACKSIZE 2048
```

Note: A semicolon must precede a comment. In this example, the NAME statement tells LINK that the file being created is a program module (.EXE). The CODE statement specifies that code segments are loaded on demand. The STACKSIZE is 2048 bytes.

The first line of the sample module definition file is a comment. A comment can appear on a line by itself or on the same line as a definition statement, as long as it appears after the definition.

Using the Linker

There are three different ways to use LINK:

The prompt method: You supply information by responding to several prompts after you call LINK.

The command-prompt method: You supply all input to LINK on one line.

The response file method: You create a file that contains all the necessary options and filenames, then supply this file to LINK.

You can also mix these three methods.

To use LINK, create one or more object files and submit these files and any required library files to LINK for processing. The linker combines the code and data in the object files and searches the library files that you name to resolve external references. The linker then copies a relocatable and executable image and the relocation infor-

mation to the executable (.EXE or .DLL) file. Using the relocation information, the operating system can load the .EXE or .DLL image to any convenient location in memory and run it. You can run these programs by typing the name of the file at the command prompt.

File-Naming Conventions

Use any combination of uppercase and lowercase letters for the filenames you give in response to the prompts. For example, abcde.fgh, AbCdE.FgH, and ABCDE.fgh are all acceptable filenames.

LINK uses the default file extensions .OBJ, .EXE, .MAP, .LIB, and .DEF when you do not supply any extensions. To cancel or replace the default extension specify a different extension.

To enter a filename that has no extension, type the name followed by a period. For example, if you type *ABC.* in response to a prompt, it tells LINK that the given file has no extension, but typing just *ABC* tells LINK to use the default extension for that prompt.

Type the name or names of the object files that you want to link. If you do not supply filename extensions, LINK uses .OBJ by default. If you have more than one name, separate each name with spaces or a plus sign. If you have more names than can fit on a single line, type a plus sign as the last character on the line and press Enter. LINK asks you for additional object files.

Selecting Default Responses

To select the default response to a prompt, press Enter without typing a filename. The next prompt appears.

Use the semicolon character to save time when the default responses are acceptable. LINK does not allow the semicolon character with the first prompt, Object Modules [.OBJ]:, because that prompt has no default. To select all the remaining default responses at once, type a semicolon at the next prompt and press Enter.

Note: When you use the semicolon, you cannot respond to any of the remaining prompts for that link session.

Defaults for other linker prompts follow:

- **Run File**: The name of the first object file specified for the previous prompt, LINK replaces the .OBJ extension with the .EXE extension.
- **List File**: The special filename NUL.MAP, which tells LINK not to create a map file.
- **Libraries**: For C programs, the floating point library and the library for the appropriate memory model. The names of these default libraries are imbedded in the .OBJ file by the compiler. Specifying libraries at this prompt does not override the default but adds to it. To override the default, use the /NOD option. (See "Ignoring Default Libraries /NODEFAULTLIBRARYSEARCH" on page 3-27.)
- **Definitions File**: The special filename NUL.DEF, which tells LINK not to search for a definitions file.

Ending the LINK Session

You can end LINK at any time by pressing Ctrl + Break.

Note: When you use Ctrl + Break, you must restart LINK.

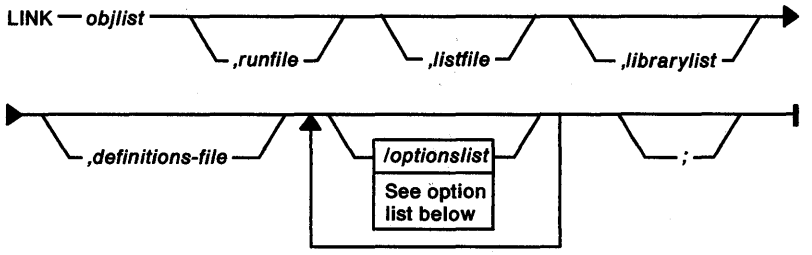
Using LINK Exit Codes

LINK returns an exit code of 0, 2, or 4 to indicate the status of linking. The exit code is useful with the operating system batch command IF ERRORLEVEL: it allows you to test for the success or failure of linking before running other tasks that depend on the LINK step. The exit codes from LINK are defined as:

Code	Meaning
0	No error
2	Program error, something was wrong with the commands or files input to LINK.
4	System error. The linker ran out of space on output files, was unable to reopen the temporary file, experienced an internal error, or was interrupted by the user.

Using a Command to Specify LINK Files

To run the linker with a single command, supply all your responses to link prompts on a single line. Separate your responses with commas, as shown:



objlist The object files that you want to link, separated by plus signs or spaces. The linker requires at least one object file. If you do not supply an extension, LINK provides the extension .OBJ.

runfile The name of the file to receive the output that LINK creates. You can prefix the filename with a drive specifier and path. If you do not supply a runfile, LINK creates one using the first object filename. It adds the extension .EXE for program modules or .DLL for dynamic link libraries.

listfile The name of the file to receive the map listing. The filename can be prefixed with a drive specifier and path. If you do not supply an extension, the linker uses the extension .MAP. If you use the /MAP or /LINENUMBERS option, LINK creates a map file even if a map file is not listed in the command.

librarylist A list of libraries for LINK to search, separated by plus signs or spaces. LINK searches these in addition to the default libraries specified in the .OBJ file.

definitions-file An optional module definition file you can give to the linker.

/optionslist A list of linker options. If you specify options, you can put them anywhere in the command.

Example

This example uses an object module FILE.OBJ to create the executable file FILE.EXE. The linker searches the library ROUTINE.LIB, in addition to the default libraries specified in the .OBJ file, for routines and variables used within the program. It also creates a file called FILE.MAP containing a list of the segments of the program and groups.

```
LINK file.obj,file.exe,file.map,routine.lib;
```

It is equal to the following line:

```
LINK file , ,file, routine;
```

In the following example, the linker loads and links the object modules FUN.OBJ, TEXT.OBJ, TABLE.OBJ, and CARE.OBJ and searches for unresolved references in the library file COBLIB.LIB, in addition to the default libraries imbedded in the ,OBJ file. By default, the executable file produced is named FUN.EXE. A listfile named FUNLIST.MAP is also produced.

```
LINK FUN+TEXT+TABLE+CARE,,FUNLIST,COBLIB.LIB /MAP;
```

The next example uses the two object modules STARTUP.OBJ and FILE.OBJ on the current drive to create an executable file named FILE.EXE on drive B. The linker creates a map file in the **\MAP** directory of the current drive and searches only the libraries indicated within the .OBJ file.

```
LINK startup+file,b:file,\map\file;
```

The following example links the application object file SAMPLE.OBJ using the module definition file SAMPLE.DEF and the libraries LIB1.LIB and LIB2.LIB.

```
LINK sample/A:4,sample.exe, sample.map/LI, lib1+lib2/NOD, sample
```

This command creates the file SAMPLE.EXE. It also creates the map file SAMPLE.MAP. The command searches the library files LIB1.LIB and LIB2.LIB to resolve any external references made in SAMPLE.OBJ. The /NOD option directs LINK to ignore any default libraries specified in the object file.

The linker uses default filename extensions if you do not explicitly provide your own. In the example above, LINK extends the first occurrence of the filename SAMPLE to SAMPLE.OBJ and the final occurrence to SAMPLE.DEF. LINK extends the library files with the .LIB extension. The /A:4 option sets the segment alignment factor to

16. The /LI option copies the line number information from the object files.

Using Prompts to Specify LINK Files

The linker prompts for the information it needs by displaying the following lines, one at a time.

```
Object Modules [.OBJ]:
Run File [filename.EXE]:
List File [NUL.MAP]:
Libraries [.LIB]:
Definitions File [NUL.DEF]:
```

The linker waits for a response to each prompt before displaying the next one.

1. To start LINK, type: `LINK`

   LINK displays the following message:

   ```
   Object Modules [.OBJ]:
   ```

   List the names of the object files you want to link at the Object Modules prompt. You must respond to this prompt; there is no default. LINK supplies the .OBJ extension when you give a filename without an extension; if your object file has a different extension, you must supply it.

   You can use pathnames with object filenames. You can give LINK the pathname of an object file in another directory or on another diskette. If LINK cannot find a given object file, it displays a message and waits for you to change diskettes or directories.

   You must separate each object filename from the next by blank spaces or a plus sign. If a plus sign is the last character typed on the line, the Object Modules prompt reappears on the next line, allowing you to type more object files.
2. Press Enter after you type all object filenames.

   LINK displays the following prompt:

   ```
   Run File [filename.EXE]:
   ```

   The *filename* in this prompt is the first one entered in response to the Object Modules [.OBJ]: prompt. This is the default name if no new name is supplied.

   Enter the name for the executable (.EXE) file. The Run File prompt allows any filename you prefer. However, .EXE is the

recommended extension because DOS runs files with this extension. If you give no extension, LINK uses .EXE by default.

3. If you want LINK to supply a default executable filename, press Enter.

   LINK displays the prompt:

   ```
   List File [NUL.MAP]:
   ```

   Following the List File prompt you can:

   - Enter the name of the map (.MAP) file that you want to create. If you do not give a filename extension, LINK uses .MAP by default. By adding the /MAP option you may also list all external (public) symbols and their addresses (see "Producing a Public Symbol Map /MAP" on page 3-26).
   - Press Enter without giving a name. When you skip this prompt, LINK uses the special filename NUL.MAP, which tells LINK not to create a map file.

   At the bottom of the map file, LINK gives you the address of the program entry point.

4. Press Enter.

   LINK displays the prompt:

   ```
   Libraries [.LIB]:
   ```

   Following the Libraries prompt you can:

   a. Press Enter to go to the next prompt.
      or
   b. Use one search path or up to 16 search paths by listing one or more search paths with the library names.

      You can assign the search paths to the environment variable LIB before you call LINK.

   Environment variables are explained under the SET command in the user's reference information for the operating system. Each search path can be either a directory specification or a library name. Directory specifications must end with a backslash (\) so that LINK can distinguish the directory names from the library names.

   IBM C/2 encodes object files with the names of the default combined libraries, *x*LIBCyz.LIB, for the appropriate memory model. These combined libraries contain the C library and the floating-point library or libraries selected at compile time. This encoded information lets LINK search for the default library files and link them with your C program.

Typing the directory specifications causes LINK to search for the default libraries and for any other libraries without a pathname on the same line. LINK searches for default libraries in the following order:

a. The current working directory
b. The directories in the order listed following the Libraries prompt
c. The libraries specified by the LIB environment variable.

When you give a library name, LINK searches for the library and links it with your program. If the library name includes a directory specification, LINK searches only that directory for the library. If you give no directory specification, LINK searches for the library in the order just described. LINK searches all libraries until it finds the first definition of a symbol. LINK searches default libraries after the libraries given at the command prompt. For information about changing the default libraries see "Changing the Default Libraries" on page 3-15.

5. After typing the names, press Enter.

   LINK displays the prompt:

   ```
   Definitions File [NUL.DEF]:
   ```

   Following the Definitions File, prompt you can:

   - Enter the name of a module definition file for the executable module.
   - If you do not want to use a module definition file, do not enter any names; just press Enter. LINK creates an executable file.

When you enter filenames, you must give a pathname for any file not in the current drive directory. Select the LINK options by typing them after the filename at any linker prompt. If LINK cannot find an object file, it displays a message and waits for you to change diskettes, if necessary.

You can type the rest of the filenames at the command prompt at any linker prompt. Choose the default response for all remaining prompts by typing a semicolon after any linker prompt. If you type a semicolon at the Object Modules prompt, supply at least one object filename. You can use commas, instead of pressing Enter, to immediately specify files for the next prompt.

Example

The following example links the object modules MODA.OBJ, MODB.OBJ, MODC.OBJ, and STARTUP.OBJ. LINK searches the library file MATH.LIB in the \LIB directory on drive B for routines and data used in the programs. LINK then creates an executable file named MODA.EXE and a map file named ABC.MAP. The /PAUSE option at the Object Modules prompt causes LINK to pause while you change diskettes. LINK then creates the executable file.

```
LINK

Object Modules [.OBJ]: moda + modb +
Object Modules [.OBJ]: modc + startup /PAUSE
Run File [MODA.EXE]:
List File [NUL.MAP]:abc
Libraries [.LIB]: b:\lib\math
Definitions File [NUL.DEF]:
```

Changing the Default Libraries

If you use the /FPa, /FPc87, or /FPc (default) option when you compile, you can switch to a different floating-point library at link time. Do this by entering the name of each library you want to use following the Libraries prompt.

If you do not want to use the C library *x*LIB*yz*.LIB, you must give the /NOD (no default library) option when you link. This option tells LINK to ignore the encoded information in the C object files. Use this option with caution. Be sure to specify all libraries necessary to compile the program. See the /NOD option under "Ignoring Default Libraries /NODEFAULTLIBRARYSEARCH" on page 3-27.

Type the names of any library files containing routines or variables your program refers to but does not define. If you do not supply filename extensions, LINK uses .LIB by default.

Using a Response File

A *response file* contains the names of all the files that you want processed. To operate the linker with a response file, you must create the file, then type the following:

```
LINK @filename
```

The *filename* is the name of the response file you created. The response file can have any name. If the file is in another directory or on another disk drive, you must provide a pathname.

The response file has the following general form:

```
objectfiles
[runfile]
[listfile]
[libraryfiles]
[definitionsfile]
```

Omit elements you have already provided at prompts or with a partial command.

The responses must be in the same order as the LINK prompts. Each response to a prompt must begin on a separate line, but you can extend long responses across more than one line by typing a plus sign as the last character of each incomplete line. You can place options on any line. Use options and command characters in the response file as if they were typed at the keyboard.

You can type a semicolon on any line in the response file. When LINK reads the semicolon, it automatically supplies default filenames for all files that you have not yet named in the response file. LINK ignores the remainder of the response file. 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.

When you use the LINK command with a response file, LINK displays each prompt on your screen with the corresponding response from your file. If the response file does not contain responses for all the prompts (in the form of filenames, the semicolon command character, or carriage returns), LINK displays the appropriate prompts and waits for responses. When you type an acceptable response, LINK continues the session.

Note: End a response file with either a semicolon or a carriage return/line feed combination. If you do not provide a final carriage return/line feed in the file, the linker displays the last line of the response file and waits for you to press Enter.

Example

The following lines in a response file tell LINK to load the four object modules FUN, TEXT, TABLE, and CARE. LINK produces two output files named FUN.EXE and FUNLIST.MAP. The /PAUSE option causes LINK to pause before producing the executable file (FUN.EXE). This permits a diskette change if necessary. The linker also searches the COBLIB.LIB, in addition to the defaults libraries listed in the .OBJ files. See the /PAUSE option under "About LINK Options" on page 3-18 for more information.

```
FUN TEXT TABLE CARE
/PAUSE
FUNLIST
COBLIB.LIB;
```

The response file below tells the linker to link the four object modules MODA, MODB, MODC, and STARTUP. The linker pauses for a diskette change before producing the runfile MODA.EXE. The linker also creates a map file ABC.MAP and searches the library MATH.LIB, in addition to the default libraries listed in the .OBJ files, in the \LIB directory of drive B.

```
moda modb modc startup /PAUSE

abc
b:\lib\math
```

The following example combines all three methods of supplying filenames. Assume that you have a response file called LIBRARY that contains the following line:

```
lib1+lib2+lib3+lib4;
```

Now start LINK with a partial command:

```
LINK object1 object2
```

LINK takes OBJECT1.OBJ and OBJECT2.OBJ as its object files and asks for the next line with the following:

```
Run File [object1.EXE]:  exec
List File [NUL.MAP]:
Libraries [.LIB]:  @library
```

1. Type **exec** so that the linker names the run file EXEC.EXE.
2. Press Enter to show that you do not want a map file.
3. Type **@ library** for the linker to use in the response file containing the four library filenames.

Temporary Disk File

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 storage. LINK creates the temporary file in the current working directory and displays the following message:

```
Temporary file name has been created.
Do not change diskette in drive letter:
```

The *name* is a unique temporary filename created by the linker. After this message appears, do not remove the diskette from the given drive (*letter*) until the link session ends. If you remove the diskette, the operation of LINK is unpredictable. If LINK unexpectedly ends, you may see the following message:

```
Unexpected end of file on name
```

If you get this message, you must restart LINK from the beginning. After LINK creates the executable file, it automatically deletes the temporary file.

About LINK Options

When you start the linker at a command prompt, you can specify LINK options anywhere, except before the last comma on the response line. If you are using a response file, you can place options after the individual responses on the same line of the file or by themselves on a separate line.

When specifying more than one option, you can group them at the end of a single response to a prompt or distribute them among several responses for different prompts. Every option must begin with the slash character, even if other options precede it on the same line.

LINK options are named according to their function. You can abbreviate the names to save space and typing. Be sure your abbreviations are at least as long as the minimum stated in the individual descriptions.

All linker options must begin with a slash (/).

LINK allows no gaps or transpositions.

Some linker options take numerical arguments. A numerical argument can be any of the following:

- A decimal number from 0 to 65535.
- An octal number from 0 to 0177777. LINK interprets a number as octal if it starts with a 0. For example, the number "10" is a decimal number, but the number "010" is an octal number, equal to 8 in decimal.
- A hexadecimal number from 0 to 0xFFFF (X'FFFF'). LINK interprets a number as hexadecimal if it starts with "0x." For example, "0x10" is a hexadecimal number, equivalent to 16 in decimal.

Using LINK Options

The linker options control what tasks LINK performs. You may use an option anywhere in a LINK command. You can abbreviate option names as long as your abbreviations contain enough letters to distinguish the specified option from other options. Minimum abbreviations are listed for each option.

The following is a list of LINK options. Those followed by a B are valid for creating DOS mode and OS/2 mode executable files. Those followed by a D are valid for creating DOS mode executable files, and an O following the option means it is valid for creating OS/2 mode executable files.

Option	Description
/ALIGNMENT (O)	Sets segment alignment factor
/CODEVIEW (B)	Includes symbolic debugging information
/CPARMAXALLOC (D)	Changes value of maximum number of reserved paragraphs
/DOSSEG (B)	Forces ordering of segments
/EXEPACK (D)	Packs executable files
/FARCALLTRANSLATION (O)	Optimizes intrasegment far calls
/HELP (B)	Writes a list of the available LINK options to the screen
/INFORMATION (B)	Displays information during the link process
/LINENUMBERS (B)	Copies line numbers to the map file
/MAP (B)	Lists all public symbols in your program
/NODEFAULTLIBRARYSEARCH (B)	Ignores default libraries

/NOFARCALLTRANSLATION (O)	Disables far call translations
/NOIGNORECASE (B)	Case sensitive
/NOPACKCODE (O)	Disables code segment packing
/PACKCODE (O)	Packs code segments
/PACKDATA (O)	Packs data segments
/PAUSE (B)	Pauses before writing the executable
/SEGMENTS (B)	Sets the maximum number of logical segments
/STACK (B)	Sets the stack size
/WARNFIXUP (O)	Issues warning message on certain fix-ups.

Linker Options for Other IBM Language Files

These options are documented here, along with the other LINK options because you may need them if you use LINK to link files written by other IBM languages. They are not recommended for use with C.

Option	Description
/DSALLOCATE	Loads all data starting at the high end of the data segment
/HIGH	Places runfile as high as possible in storage
/NOGROUPASSOCIATION	Provides some fix-up routines compatibility with previous versions of other IBM language compilers
/OVERLAYINTERRUPT	Allows selection of an interrupt number for overlays.

Aligning Segments /ALIGNMENT

This option directs LINK to set the segment alignment factor in the executable file to the number given, which must be a power of 2. The default alignment is 512. This is valid for OS/2 mode executable files only.

Format

`/ALIGNMENT:`*`number`*

The minimum abbreviation is /A.

Comments
The *number* can be a hexadecimal, decimal, or octal number.

Preparing Files for CodeView /CODEVIEW

This option directs LINK to include symbolic debugging information for CodeView in the output .EXE file.

Format

```
/CODEVIEW
```

The minimum abbreviation is /CO.

Comments

Note: /CODEVIEW cannot be used with /EXEPACK.

Reserving Paragraph Space /CPARMAXALLOC

This option allows you to change the default value of the MAXALLOC field, which controls the maximum number of paragraphs reserved in storage for your program. A paragraph is defined as the smallest storage unit (16 bytes) addressable by a segment register.

This is valid for DOS mode executable files only.

Format

/CPARMAXALLOC:*number*

The minimum abbreviation is /CP.

Comments
The maximum number of paragraphs reserved for a program is determined by the value of the MAXALLOC field at offset 0CH in the EXE header. For more information about EXE file structuring and loading, see the technical reference information for DOS.

The default for the MAXALLOC field is 65535 (decimal), or 64KB minus 1. You can reset the default to any number between 1 and 65535 (decimal, octal, or hexadecimal). Changing the number is helpful because:

- Program efficiency is not increased by reserving all available memory.
- You may need to run another program within your program and need to reserve space for the run program.

If the value specified is less than the computed value of MINALLOC (at offset 0AH), the linker uses the value of MINALLOC instead.

Ordering Segments /DOSSEG

This option forces segments to be ordered according to the following rules:

1. All segments with a class name ending in CODE.
2. All other segments outside of DGROUP.
3. DGROUP segments in the following order:
    a. Any segments of class BEGDATA. (This class name is reserved for IBM use.)
    b. Any segments not of class BEGDATA, BSS, or STACK.
    c. Segments of class BSS.
    d. Segments of class STACK.

Format

```
/DOSSEG
```

The minimum abbreviation is /DO.

Controlling Data Loading /DSALLOCATE

By default, LINK loads all data starting at the low end of the data segment. At run time, LINK sets the DS (data segment) pointer to the lowest possible address to allow the entire data segment to be used.

This option tells LINK to load all data starting at the high end of the data segment. To do this, at run time, set the DS pointer to the lowest data segment address that contains program data.

This is valid for DOS mode executable files only.

Format

```
/DSALLOCATE
```

The minimum abbreviation is /DS.

Comments

This option is typically used with the /HIGH option to take advantage of unused storage within the data segment. You can reserve any available storage below the area specifically reserved for DGROUP, using the same DS pointer.

Packing Executable Files /EXEPACK

This option directs LINK to remove sequences of repeated bytes (typically nulls) and to optimize the load-time relocation table before creating the DOS executable file.

This is valid for DOS mode executable files only.

Format

```
/EXEPACK
```

The minimum abbreviation is /E.

Comments

Executable files linked with this option are usually smaller and load faster than files linked without this option. However, you cannot use symbolic debugging programs with packed files.

This option does not always save a significant amount of disk space and may sometimes actually increase file size. Programs that have a large number of load-time relocations (about 500 or more) or long streams of repeated characters are usually shorter if packed.

Note: /EXEPACK cannot be used with /CODEVIEW.

Example

This example creates a packed version of file PROGRAM.EXE.

```
LINK  program  /E;
```

Optimizing Intrasegment Far Calls /FARCALLTRANSLATION

This option directs LINK to optimize intrasegment far calls into the sequence:

```
NOP
PUSH    CS
CALL    NEAR    address
```

Format

```
/FARCALLTRANSLATION
```

The minimum abbreviation is /F.

This is valid for OS/2 mode executable files only.

Note: In most medium- and large-model programs, this option yields significant savings in executable size and load time. However, there is a small chance that the linker, during this optimization, will mistakenly identify a byte with a value of Ox9a as a far-call, when, in fact, it is an assembled constant. Be cautious when using this option.

Viewing the Options List /HELP

This option causes LINK to write a list of the available options to the screen. This may be convenient if you need a reminder of the available options. Do not give a filename when using the **/HELP** option.

Format

```
/HELP
```

The minimum abbreviation is /HE.

Example

```
LINK /HELP
```

Controlling Run File Loading /HIGH

You can place the run file as low or as high in storage as possible. This option directs LINK to cause the loader to place the run file as high as possible in storage without overlaying the transient portion of COMMAND.COM. The COMMAND.COM file occupies the highest area of storage when loaded. Without the /HIGH option, the loader places the run file as low as possible in storage.

Use the /HIGH option in association with the /DSALLOCATE option.

This is valid for DOS mode executable files only.

Format

```
/HIGH
```

The minimum abbreviation is /HI.

Note: Do not use this option with C programs.

Displaying LINK-Time Information /INFORMATION

This option causes the linker to display which phase of processing it is running and the name of each input module as it is linked. This is useful during debugging.

Format

```
/INFORMATION
```

The minimum abbreviation is /I.

Copying Line Numbers to the Map File /LINENUMBERS

This option directs the linker to copy the starting address of each program source line to a map file. The starting address is the address of the first instruction that corresponds to the source line.

Format

```
/LINENUMBERS
```

The minimum abbreviation is /LI.

Comments

LINK copies the line number data only if you give a map filename in the LINK command and only if the given object file has line number information. Line numbering is available in some high-level languages.

The IBM Macro Assembler/2 does not copy line number information to the object file. If an object file has no line number information, the linker ignores the /LINENUMBERS option.

Note: If you do not specify a map file in a LINK command, you can still use the /LINENUMBERS option to force the linker to create a map file. Just place the option at or before the List File prompt. LINK gives the forced map file the same filename as the first object file specified in the command and gives it the default extension .MAP.

Example

This example causes the line number information in the object file FILE.OBJ to be copied to the map file FILE.MAP.

```
LINK file/LINENUMBERS,,file
```

Producing a Public Symbol Map /MAP

This option causes LINK to produce a listing of all public symbols declared in your program. This list is copied to the map file created by the linker.

Format

/MAP:number

The minimum abbreviation is /M.

For a complete description of the listing file format, see "The Map File" on page 3-47.

Comments

The *number* parameter specifies the maximum number of public symbols that the linker can sort in the map file. If you give no number, the limit is 2048. Valid values are 1 through 32767. They can be specified in hex, decimal, or octal. If the limit is exceeded, the linker produces an unsorted list and issues the following fatal error message:

```
MAP symbol limit too high
```

If you get this error, link again with a lower number. The limit varies according to how many segments the program has and how much memory is available.

Specifying a number also causes the sorting of public symbols by address only, not by name, regardless of the number. If you want to reduce the size of your map files by removing the list sorted by name, link with /MAP followed by a small number that is large enough to accommodate the number of public symbols in your program.

Note: If you do not specify a map file in a LINK command, you can use the /MAP option to force the linker to create a map file. LINK gives the forced map file the same name as the first object file specified in the command and the default extension .MAP.

Ignoring Default Libraries /NODEFAULTLIBRARYSEARCH

This option directs the linker to ignore any library names it may find in an object file. A high-level language compiler may add a library name to an object file to ensure that a default set of libraries is linked with the program. Using this option bypasses these default libraries and lets you name the libraries you want by including them at the LINK command prompt.

Format

```
/NODEFAULTLIBRARYSEARCH[:libname]
```

The *libname* causes the linker to ignore the library search record for that library only. For example:

```
LINK prog1 /NOD:SLIBCE
```

causes the linker to ignore SLIBCE.LIB but search all others. The minimum abbreviation is /NOD.

Example

This example links the object files STARTUP.OBJ and FILE.OBJ with routines from the libraries EM.LIB, SLIBFP.LIB, and SLIBC.LIB. Any default libraries that may have been named in STARTUP.OBJ or FILE.OBJ are ignored.

```
LINK startup+file/NOD,,,em+slibfp+slibc;
```

Disabling Far Call Translations /NOFARCALLTRANSLATION

This option directs LINK to disable translation of intrasegment far calls. This option is in effect by default.

This is valid for OS/2 mode executable files only.

Format

```
/NOFARCALLTRANSLATION
```

The minimum abbreviation is /NOF.

Comments

When you use this option, the linker does not translate intrasegment far calls.

Preserving Compatibility /NOGROUPASSOCIATION

This option causes the linker to process a certain class of fix-up routines in a manner compatible with previous versions of the linker. This option is provided primarily for compatibility with previous versions of other IBM language compilers.

This is valid for DOS mode executable files only.

Format

```
/NOGROUPASSOCIATION
```

The minimum abbreviation is /NOG.

Preserving Lowercase /NOIGNORECASE

This option directs LINK to treat uppercase and lowercase letters in symbol names as distinct letters. Normally, LINK considers uppercase and lowercase letters to be identical, treating the names *TWO*, *Two*, and *two* as the same. When you use the /NOIGNORECASE option, the linker treats *TWO*, *Two*, and *two* as three different names.

Format

```
/NOIGNORECASE
```

The minimum abbreviation is /NOI.

Comments

This option is typically used with object files created by high-level language compilers. Some compilers treat uppercase and lowercase letters as distinct letters and assume that the linker does the same.

Example

This command causes the linker to treat uppercase and lowercase letters in symbol names as distinct letters. The object file FILE.OBJ is linked with routines from the C language library \SLIBC.LIB located in the \LIB directory.

Note: The C language does not treat uppercase and lowercase letters as the same.

```
LINK file/NOI,,,\lib\slibc;
```

Disabling Packing /NOPACKCODE

This option directs the linker to disable the packing of code segments. /NOPACKCODE is the opposite of /PACKCODE. This option is valid for OS/2 mode executable files only.

Format

```
/NOPACKCODE
```

The minimum abbreviation is /NOP.

Setting the Overlay Interrupt /OVERLAYINTERRUPT

By default, the DOS interrupt number used for passing control to overlays is 3FH. This option allows you to select a different interrupt number.

This is valid for DOS mode executable files only.

Format

/OVERLAYINTERRUPT:*number*

The minimum abbreviation is /O.

Comments

The *number* can be a decimal number from 0 to 255, an octal number from 0 to 0377, or a hexadecimal number from 0 to 0xFF. Numbers that conflict with DOS interrupts are not prohibited, but their use is not recommended.

Packing Code Segments /PACKCODE

This option directs LINK to try to pack neighboring logical code segments into one physical segment. LINK performs this option by default.

This is valid for OS/2 mode executable files only.

Format

/PACKCODE[:*number*]

The minimum abbreviation is /PACKC.

Comments

The *number* is the limit at which to stop packing; it can be any number between 0 and 65536. If no number is given LINK uses 65536. For more information on packing, see "Rules for Segment Packing in LINK" on page 3-53.

Packing Data Segments /PACKDATA

This option directs LINK to pack neighboring logical data segments into one physical segment.

This option is valid for OS/2 executable programs only.

Format

/PACKDATA[:*packlimit*]

The minimum abbreviation is /PACKD.

Comments

By default, the linker does not try to pack neighboring logical data segments into one physical segment. The *packlimit* is the limit at which to stop packing. If no *packlimit* is given, LINK uses 65536. For more information about packing, see "Rules for Segment Packing in LINK" on page 3-53.

Consider using this option if you have a large-model program with many modules and you get LINK error L1073- file-segment limit exceeded.

Pausing to Change Disks /PAUSE

This option causes LINK to pause before writing the executable file to disk so that you can change disks.

Format

/PAUSE

The minimum abbreviation is /PAU.

Comments

If you choose the /PAUSE option, the linker displays the following message before creating the run file:

```
About to generate .EXE file
Change diskette in drive letter and press <ENTER>
```

The *letter* is the proper drive name. This message appears after the linker has read data from the object and library files and after it has

written data to the map file, if one was specified. LINK resumes processing when you press Enter. After LINK writes the executable file to disk, the following message appears:

```
Please replace original diskette
in drive letter and press <ENTER>
```

Note: Do not remove the disk used for the temporary file, if one has been created. If the temporary disk message appears when you have specified the /PAUSE option, you should press Ctrl+C to end the LINK session. Rearrange your files so that LINK can write the temporary file and the executable file to the same disk; then try again.

Example

This command causes the linker to pause just before creating the executable file FILE.EXE. After creating the executable file, LINK pauses again to let you replace the original disk.

```
LINK file/PAUSE,file,,\lib\math;
```

Setting the Maximum Number of Segments /SEGMENTS

This option directs the linker to process no more than *number* segments per program. If it finds more than the given limit, the linker displays an error message and stops linking. The /SEGMENTS option bypasses the default limit of 128 segments.

Format

/SEGMENTS:*number*

The minimum abbreviation is /SE.

Comments

If you do not specify /SEGMENTS, the linker reserves enough storage space to process up to 128 segments. If your program has more than 128 segments, set the segment limit higher. Set the segment limit lower if you get the following LINK error message:

```
Segment limit set too high
```

The *number* can be any integer value in the range 0 to 3072. It must be a decimal, octal, or hexadecimal number. Octal numbers must have a leading zero. Hexadecimal numbers must start with a leading zero followed by a lowercase *x*. For example, 0x4B.

Example

This example sets the segment limit to 192:

```
LINK file/SE:192;
```

The next example sets the segment limit to 255 (X'FF'):

```
LINK moda+modb,run/SEGMENTS:0xff,ab,em+mlibfp;
```

Setting the Stack Size /STACK

This option sets the program stack to the number of bytes given by *size*. The linker automatically calculates the stack size, basing it on the size of any stack segments given in the object files. If you specify /STACK, the linker uses the given *size* in place of any value it may have calculated.

Format

/STACK:*size*

The minimum abbreviation is /ST.

Comments

The *size* can be any positive integer value in the range 0 to 65535. The value can be a decimal, octal, or hexadecimal number. Octal numbers must begin with a zero. Hexadecimal numbers must begin with a leading zero followed by a lowercase *x*. For example, Ox1B.

The stack size can also be changed after linking with the EXEMOD utility.

Example

The first example sets the stack size to 512 bytes.

```
LINK file/STACK:512;
```

The second example sets the stack size to 255 (X'FF') bytes.

```
LINK moda+modb,run/ST:0xFF,ab.\lib\start;
```

The final example sets the stack size to 24 (30 octal) bytes.

```
LINK startup+file/ST:030;
```

Warning of Fix-ups /WARNFIXUP

This option directs the linker to issue a warning for each segment-relative fix-up of location-type offset when the segment is contained within a group, but not at the beginning. The linker includes the displacement of the segment from the group in determining the final value of the fix-up, contrary to what happens with DOS mode executable files.

This is valid for OS/2 mode executable files only.

Format

```
/WARNFIXUP
```

The minimum abbreviation is /W.

Module Definition File Statements

The module definition file defines the contents and system requirements of an OS/2 mode executable module. The file contains one or more module statements, each defining a specific attribute of the module, such as module name, number and type of program segments, and the number and names of exported and imported functions.

The following section describes these module statements:

Statement	Description
CODE	Code segment attributes
DATA	Data segment attributes
DESCRIPTION	One line description of the module
EXPORTS	Exported functions
HEAPSIZE	Local heap size in bytes
IMPORTS	Imported functions
LIBRARY	Specifies dynamic link library
NAME	Specifies program module
OLD	Module for export ordinals
PROTMODE	Specifies OS/2 mode
SEGMENT	Segment attributes per segment name
STACKSIZE	Local stack size in bytes
STUB	Adds a DOS mode executable file.

Notes:

1. Any line, or part of a line, in the definition file beginning with a semicolon is a comment and is ignored by the linker and IMPLIB.
2. All statements in the module definition file must be entirely in uppercase.

Defining the Code Segment Default Attributes CODE

This statement defines the default attributes of all code segments in the module.

Format

CODE [*load option*][*execute option*][*privilege option*][*conforming option*]

Parameters

The *load option* is an optional keyword specifying when the segment is to be loaded. It must be one of the following:

PRELOAD The segment is loaded immediately.

LOADONCALL The segment is loaded when called.

The default is LOADONCALL.

The *execute option* specifies whether the segment can be read as well as run. It must be one of the following:

EXECUTEONLY The segment can only be run.

EXECUTEREAD The segment can be run and read.

The default is EXECUTEREAD.

Note: If you use C to create a code segment of a dynamic link module and the source code contains a switch statement, do not specify EXECUTEONLY on the CODE statement of the dynamic link module definition file. Use the default execute option EXECUTEREAD for such segments. In many cases of the switch statement, the compiler generates a table lookup through the CODE segment. The table lookup method is taken for switch statements with more than six distinct cases when the ratio of number-of-cases to range-of-cases exceeds one-third.

The *privilege option* is an optional keyword specifying if the segment has I/O privilege. It must be one of the following:

IOPL The segments have I/O privilege.

NOIOPL The segments do not have I/O privilege.

The default is NOIOPL.

Conforming-option is an optional keyword specifying the access rights to code segments.

CONFORMING Sets access rights to conforming.

NONCONFORMING Sets access rights to nonconforming.

The default is NONCONFORMING. Set the access rights to CONFORMING if the segment contains at least one routine that has all of the following qualities:

- Will be called from both IOPL and non-IOPL segments
- Will not be called through call gates
- Does not need I/O privilege to operate (that is, does not access ring 2 data, disable interrupts, or use the IN and OUT instructions.

Defining Data Segment Default Attributes DATA

This statement defines the default attributes of the data segments of the application. The automatic data segment contains the local stack and heap of the module.

Format

```
DATA [instance option][shared option][write option]
  [privilege option][load option]
```

Parameters

The *instance option* is an optional keyword that describes the sharing of the automatic data segment, which is any group named DGROUP. It can be any one of the following:

NONE There is no automatic data segment.

SINGLE The automatic data segment is shared by all instances of the module (valid only for dynamic link libraries).

MULTIPLE The automatic data segment is copied for each instance of the module.

The default for *instance option* is MULTIPLE for program modules and SINGLE for dynamic link libraries.

The *shared option* is an optional keyword specifying the need to have a unique copy of the READWRITE data segments loaded for each process using the dynamic link library. Values are:

SHARED A single copy of each data segment is loaded.

NONSHARED A unique copy of each READWRITE data segment is loaded for each process using the dynamic link library.

The default is NONSHARED for program modules and SHARED for dynamic link libraries.

The *write option* is an optional keyword specifying whether the segment can be written to.

READONLY The segment can only be read from.

READWRITE The segment can be read from or written to.

The default is READWRITE.

The *privilege option* is an optional keyword specifying if the segment has I/O privilege. It must be one of the following:

IOPL The segments have I/O privilege.

NOIOPL The segments do not have I/O privilege.

The default is NOIOPL.

The *load option* is an optional keyword specifying when the segment is to be loaded. It must be one of the following:

PRELOAD The segment is loaded immediately.

LOADONCALL The segment is loaded when called.

If you give a CODE statement, the default is LOADONCALL.

Inserting Text
DESCRIPTION

This statement inserts *text* into the module of the application. It is useful for embedding source control or copyright information.

Format

```
DESCRIPTION text
```

Parameters

The *text* is one or more ASCII characters. You must enclose the string in single quotation marks.

Example

```
DESCRIPTION 'Template Application'
```

Exporting Functions
EXPORTS

This statement defines the names and characteristics of the functions in the dynamic link library to export to other applications. The EXPORTS keyword marks the beginning of the definitions. Following the EXPORTS keyword are up to 3072 export definitions, each on a separate line.

Format

```
EXPORTS
exportname[ordinal option][RESIDENTNAME][iopl-parmwords]
```

Parameters

The *exportname* is one or more ASCII characters defining the function name. It has the form:

```
entryname[=internalname]
```

The *entryname* is the name that other applications use to get access to the exported function. It is a required parameter.

The *internalname* defines the actual name of the function if *entryname* is not the actual name. It is an optional parameter.

The *ordinal option* defines the ordinal value of the function. It has the form:

@ordinal

where *ordinal* is an integer number specifying the ordinal value of the function. The ordinal value defines the index of the name of the function in the entry table of the dynamic link library. A blank space must precede the @ character. It is an optional parameter.

RESIDENTNAME indicates that the entry point name of the function is kept resident in memory. It is an optional parameter, applicable only when *ordinal-option* is specified. DOS normally keeps name strings resident in memory, enabling it to rapidly resolve calls to frequently-used entry points.

The *iopl-parmwords* is an optional numeric value that must be specified for functions that run with I/O privilege. Each such function is allocated a 512-byte stack. When the function is invoked, the number of parameters (words) specified by *iopl-parmwords* are copied from the caller's stack to the new stack.

Example

```
EXPORTS
 SampleRead= @1 8
 StringIn=strl @2
 CharTest @3
```

Defining Local Storage
HEAPSIZE

This statement defines the number of bytes needed by the application for its local heap. An application uses the local heap whenever it reserves local storage. The default heapsize is 0.

Format

HEAPSIZE *bytes*

Parameters

The *bytes* parameter is an integer number specifying the heap size in bytes. It must not exceed 65536 (64KB), the size of a single physical segment.

Example

```
HEAPSIZE 4096
```

Importing Functions
IMPORTS

This statement defines the names and attributes of functions to be imported from existing dynamic link modules. The IMPORTS keyword marks the beginning of the definitions. There can be any number of IMPORTS statements, each on a separate line.

Format

```
IMPORTS
[internal-name=] libraryname.entry
```

Parameters

The *internal-name* is an optional specification of one or more ASCII characters. It specifies the name that the application uses to call the function. It must be a unique identifier. The *libraryname* is the name of the dynamic link library containing the function. The *entry* specifies the function to import. It is one of the following:

entryname — The actual name of the function.

entryordinal — The ordinal value of the function; corresponds to the entry point in the dynamic link library. For each entry of this type you must specify an internal-name.

Note: Named references to entry points in DOSCALLS are not supported by the IMPORTS statement; these calls fail during the module load. To dynamically link to DOSCALLS, code the *entrynames* in your program and include DOSCALLS.LIB when linking, or code the *entryordinals* in your program, and use the IMPORTS statement to import the DOSCALLS functions.

If *internal-name* is not given, the *entry* must be *entryname*, which is used for the internal name.

Example

```
IMPORTS
   Sample.SampleRead
   write2hex=Sample.SampleWrite
   read=Read.1
```

Naming Library Modules
LIBRARY

This statement specifies the creation of a dynamic link library. It also specifies the type of library initialization required for the dynamic link library.

Note: The LIBRARY statement tells LINK to build a dynamic link library. Do not associate the LIBRARY statement with .LIB files that are libraries of object modules.

Format

LIBRARY[*libraryname*][*initialization-type*]

Comments

The LIBRARY statement is mutually exclusive with the NAME statement. If the LIBRARY statement is used, it must appear as the first statement in the definitions file. If neither LIBRARY nor NAME is included in a module definition file, the default is NAME.

Parameters

The *libraryname* defines the name of the dynamic link library. Although the *libraryname* is not normally specified, it can be up to eight characters. The actual library name created by the linker is based on the external name of the executable file. The linker uses the *libraryname* to validate the library name being generated. If the internal library name does not match the one specified by *libraryname*, the linker issues a warning message. When used, LIBRARY must be the first statement specified in the module definition file. Once a dynamic link library has been loaded, OS/2 knows it by the internal library name.

Initialization-type is an optional keyword that specifies the type of initialization required by the dynamic link library. If no library initialization routine is defined for the library, this keyword is ignored. It must be one of the following:

INITGLOBAL The library module initialization routine is called once when the library module is initially loaded.

INITINSTANCE The library module initialization routine is called once for each process that gains access to the library module.

The default is INITGLOBAL.

Note: The starting address of the module is determined by the object files and refers to a function that must conform to the library initialization convention that DOS requires.

Example

```
LIBRARY User
```

Naming Executable Modules
NAME

This statement specifies creation of a program module.

Format

NAME[*module name*][*applicationtype*]

Parameters

The *modulename* is optional. The actual module name created by the linker is based on the external name of the executable file. The linker uses the *modulename* to validate the module name being generated. If the internal module name does not match *modulename*, the linker issues a warning message. *Applicationtype* is optional and defines the type of application being linked. This information is kept in the executable header and is used by the Presentation Manager during program load and execution. If specified, it must be one of the following:

WINDOWAPI	Uses the API provided by Presentation Manager. It must be executed in a Presentation Manager window.
WINDOWCOMPAT	Runs (compatible) in a Presentation Manager window or in a separate screen group. An application is of this type if it uses the proper subset of OS/2 VIO, KBD, and MOU functions.
NOTWINDOWCOMPAT	Does not run (not compatible) in a Presentation Manager window. It must execute in a separate screen group. All DOS mode applications and those that use the VIO, KBD, and MOU functions that are not compatible with Presentation Manager are of this type.

If *applicationtype* is not provided, the executable header will indicate that the application type was unspecified.

Example

```
NAME Calendar WINDOWCOMPAT
```

Comments

The NAME statement is mutually exclusive with the LIBRARY statement. If the NAME statement is used, it must appear as the first statement in the definitions file. If neither NAME nor LIBRARY is included in a module definition file, the default is NAME.

Preserving Export Ordinals
OLD

This statement preserves export ordinals across successive versions of a dynamic link library.

Format

```
OLD 'libraryname'
```

Comments

The *libraryname* is the name of the dynamic link library to be used for extracting export ordinals. The *libraryname* must be within single quotation marks.

Exported names in this library that match exported names in the OLD library are assigned ordinal values from the OLD library, unless:

1. The name in the OLD library did not have an assigned ordinal, or
2. An ordinal was explicitly assigned to a name in this library.

Note: If the linker cannot find *libraryname* in the current directory, it looks in the directories listed in the LIB PATH environment variable.

Setting OS/2 Environment
PROTMODE

This statement causes the linker to set the OS/2 mode only bit in the file header of the executable (.EXE) file. Without a PROTMODE statement, the OS/2 mode only bit is not set.

By default, LINK does not set the OS/2 mode only bit.

Format

```
PROTMODE
```

Comments

If the linker recognizes floating-point instructions in the object module and the OS/2 mode only bit is not requested to be set, the linker produces run-time relocations of type OSFIXUP. If you use the resultant executable program as input to the BIND utility, BIND can emulate the floating-point instructions for the DOS mode if no coprocessor is present. If you request the OS/2 mode only bit be set, the linker does not produce OSFIXUP relocations and the BIND utility cannot be used to emulate the floating-point instructions for DOS mode.

For programs that heavily use floating-point instructions, the amount of space in the executable file that OSFIXUP relocations occupy can be significant. By using a definitions file containing the PROTMODE statement, you can save space in the executable file. (Do this only if you intend to run the program only in OS/2 mode.)

Defining Segments
SEGMENTS

This statement defines code and data segment attributes on a per-segment basis. The parameters specified override the defaults on the CODE and DATA statements. The SEGMENTS statement marks the beginning of the definitions. You can give any number of segment definitions, each on a separate line.

Format

```
SEGMENTS['']segmentname['][class option][segflags]
```

Parameters

Each segment definition consists of a combination of the following parameters.

The *segmentname* is a character string naming the new segment. Optionally, you can enclose the *segmentname* in single quotation marks. If the *segmentname* is CODE or DATA or any other definitions file keyword, you must enclose it in single quotations marks to avoid conflict with the CODE and DATA keywords.

The *class option* is an optional keyword specifying the class of the segment:

```
CLASS 'classname'
```

Note: If you do not give a class name, the linker assumes class CODE. Because any segment whose class name ends in CODE (case-insensitive) is given the type CODE by the linker, if a segment is defined here without a CLASS directive, it is recognized as a code segment.

The *segflags* is any combination of these options that are described under the CODE and DATA keywords above:

Option	Default
SHARED, NONSHARED	NONSHARED
PRELOAD, LOADONCALL	LOADONCALL
EXECUTEONLY, EXECUTEREAD (for code segments only)	EXECUTEREAD
READONLY, READWRITE (for data segments)	READWRITE
IOPL, NOIOPL	NOIOPL
CONFORMING, NONCONFORMING (for code segments only)	NONCONFORMING

Example

```
SEGMENTS
    CSEG  LOADONCALL
    CSEG2 EXECUTEREAD
    DSEG1 READONLY IOPL
    DSEG2 SHARED
```

Defining Local Stack
STACKSIZE

This statement defines the number of bytes needed by the application for its local stack. An application uses the local stack whenever it calls its own functions. A minimum stack size of 4096 bytes is recommended. The default stack size is zero if the application makes no function calls. Otherwise, it is 4096. The maximum stack size permitted is 65535 bytes.

Format

```
STACKSIZE bytes
```

Parameters

The *bytes* parameter is an integer specifying the stack size in bytes.

Example

```
STACKSIZE 4096
```

Adding an Executable File to a Module
STUB

This statement adds the DOS mode executable file named in *filename* to the beginning of the OS/2 mode module being created.

Format

```
STUB 'filename'
```

Parameters

The filename is the name of the DOS mode executable file to add to the module. The name must have the DOS filename format and be enclosed in single quotation marks. The stub is started if the OS/2 mode program is run in the DOS mode.

The Map File

The map file lists the names, load addresses, and lengths of all segments in a program. It also lists the names and load addresses of any groups in the program, the program's start address, and messages about any errors the linker may have encountered. If you use the /MAP LINK option, the map file lists the names and load addresses of all public symbols.

In the map file for OS/2 mode executable files, segment information has the general form:

```
PROGRAM_A

Start       Length     Name                   Class

0001:0000   02C24H     _TEXT                  CODE
0001:2C30   02502H     EMULATOR_TEXT          CODE
0001:5132   00000H     C_ETEXT                ENDCODE
0002:0000   00170H     EMULATOR_DATA          FAR_DATA
0003:0000   00036H     NULL                   BEGDATA
0003:0036   00708H     _DATA                  DATA
```

The Start column shows the address of the first byte in the segment, in the form *segment number:offset*. The segment numbers are indexes into the segment table of the executable file and start from 1. The Length column shows the length of the segment in bytes. The Name column shows the name of the segment and the Class column shows the class name of the segment.

Group information has the general form:

```
Origin         Group

0003:0         DGROUP
```

At the end of the listing file, the linker shows the address of the program entry point.

```
Program entry point at 0001:02A0
```

In the map file for DOS mode executable files, segment information has the following general form.

```
Start   Stop     Length   Name              Class

00000H  02B86H   02B87H   _TEXT             CODE
02B90H  05091H   02502H   EMULATOR_TEXT     CODE
05092H  05092H   00000H   C_ETEXT           ENDCODE
050A0H  0520FH   00170H   EMULATOR_DATA     FAR_DATA
05210H  05245H   00036H   NULL              BEGDATA
05246H  05761H   0051CH   _DATA             DATA
```

The Start column shows the address of the first byte in the segment. The number shown is the offset from the beginning of the program. This number is referred to as the frame number. The Stop column shows the address of the last byte in the segment. The Length is the length of the segment in bytes. The Name column shows the name of the segment and the Class column shows the class name of the segment.

Group information has the general form:

```
Origin          Group

0521:0          DGROUP

Program entry point at 0000:02A0
```

If you have specified the /MAP LINK option, the linker adds a public symbol list to the map file. Symbols are listed twice: once in alphabetic order, then in the order of their load addresses. This list has the general form shown in the following example. The form is the same for OS/2 mode and DOS mode programs. For each symbol address, the number to the left of the colon represents a segment number for OS/2 mode programs and a frame number for DOS mode programs.

```
Address         Publics by Name

0003:071A        $i8_implicit_exp
0003:0718        $i8_inpbas
0001:287C        $i8_input
0003:0719        $i8_input_ws
0001:0CF6        $i8_output

Address         Publics by Value

0000:0000  Imp  DOSWRITE      (DOSCALLS.138)
0000:0000  Imp  DOSDEVCONFIG     (DOSCALLS.52)
0000:0000  Imp  DOSEXIT      (DOSCALLS.5)
0001:0010        _main
0001:01B0        _countwords
0001:0238        _analyze
0001:02A0        __astart
0001:0368        __cintDIV
```

The first three symbols shown in the example under Publics by Value are imported public symbols and appear in map files created for OS/2 mode programs only.

Advanced LINK Topics

Using Overlays

You can direct LINK to create an *overlaid* version of your DOS mode program. This means that the loader loads parts of your program only when they are needed; the overlaid version shares the same space in storage. Your program should be able to run in less storage, but it usually runs more slowly because of the time needed to read and load the code into storage.

You specify the overlay structure to the linker in response to the Object Modules prompt. Loading is automatic. You specify the overlays in the list of modules that you submit to the linker by enclosing them in parentheses. Each parenthetical list represents one overlay. For example:

```
Object Modules [.OBJ] : a+(b+c)+(e+f)+g+(i)
```

The elements (b+c), (e+f), and (i) are overlays. The remaining modules and any drawn from the run-time libraries make up the resident or root part of your program. LINK loads your program or root overlays into the same region of storage, so only one can be resident at a time. Because LINK does not allow duplicate names in different overlays, each module can occur 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. The DOS interrupt number is 3FH.

Restrictions

LINK adds the name for the overlays to the .EXE file and encodes the name of this file into the program so the overlay manager can get access to it. If, when the program is initiated, the overlay manager cannot find the .EXE file (perhaps it was renamed or is not in a directory specified by the PATH environment variable), the overlay manager prompts you for a new name.

You can only overlay modules to which control is transferred and returned by a standard 8086 long (32-bit) CALL/RETURN instruction.

You cannot use long jumps or indirect calls (through a function pointer) to pass control to an overlay. When a pointer calls a function, the called function must either be in the same overlay or in the root.

Overlay Manager Prompts

In the following example, suppose that B is the default drive and the following message is displayed:

```
Cannot find PAYROLL.EXE
Please enter new program spec:
```

The response

```
EMPLOYEE\DATA\
```

causes the overlay manager to look for EMPLOYEE\DATA\PAYROLL.EXE on drive B.

Suppose that you must change the diskette in drive B. If the overlay manager needs to swap overlays, it finds that PAYROLL.EXE is no longer on B and gives the following message:

```
Please put diskette containing
B:\EMPLOYEE\DATA\PAYROLL.EXE
in drive B and strike any key when ready.
```

After the overlay is read from the diskette, the overlay manager gives the following message:

```
Please restore the original diskette.
Strike any key when ready.
```

Generating OS/2 Mode Applications

If you are generating an OS/2 mode application, LINK searches for the file named OSO001.MSG. It takes message 129 from that file and imbeds it in your program so it will be displayed if you attempt to run your program under DOS mode. If LINK cannot find OSO001.MSG or message 129 within this file, the following message is displayed (in English):

```
This program cannot be run in DOS mode.
```

To override that default message you can:

- Provide an OSO001.MSG file containing the message you want to use as message 129.

- Use the STUB definition module statement to specify a program to replace the default stub. This STUB program can print the message you want or perform other functions.

Order of Segments

LINK copies segments to the executable file in the same order that it meets them in the object files. This order is maintained throughout the program unless the linker finds two or more segments having the same class name. Segments having identical class names belong to the same class type and are copied to the executable files as adjoining blocks.

Combined Segments

LINK uses combine types to tell if two or more segments that share the same segment name should be one segment. The combine types are public, stack, common, and private.

- If a segment is combine type public, the linker combines it with any segments of the same name and class. When LINK combines segments, it makes the segments adjoining in storage; you can reach each address in the segments using an offset from one frame address. The result is the same as if the segments were defined as a whole in the source file.

  The linker preserves the align type of each segment in the combined segment. So, even though the individual segments compose a single, larger segment, the code and data in each segment retain the original align type of the segment. If LINK tries to combine segments that total more than 64KB, it displays an error message.

- If a segment is combine type stack, the linker combines individual segments as it does for public combine types. For stack segments, LINK copies 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) that LINK meets. If you use the stack type for stack segments, you need not give instructions to load the segment into the SS register.

- If a segment is combine type common, the linker combines it with any segments of the same name and class. When LINK combines common segments, it places the start of each segment at the same address. This creates a series of overlapping segments.

The resulting combination segment has a length equal to the length of the longest individual segment.

- LINK assigns a default combine type private to any segments with no explicit combine type definition in the source file. LINK does not combine private segments.

Groups

A *group* gives addressability to non-adjoining segments of various classes relative to the same frame address. When LINK encounters a group, it adjusts all storage references to items in the group so that they are relative to the same frame address.

Segments of a group need not be adjoining, belong to one class, or have the same combine type. All segments of the group must fit within 64KB of storage. For OS/2 mode executable objects, a group is synonymous with a *selector* or a physical segment.

Groups do not affect the order of loading of segments. You must use class names and enter object files in the correct order to guarantee adjoining segments. If the group is smaller than 64KB of storage, LINK may place segments that are not part of the group. in the same storage area. LINK does not specifically check that all segments in a group fit within 64KB of storage. If the segments are larger than the 64KB maximum, the linker can produce a fix-up overflow error.

A description of groups and defining groups is in *IBM Macro Assembler/2 Language Reference.*

Fix-ups

Once the linker knows the starting address of each segment in a program and establishes all segment combinations and groups, it can resolve any unresolved references to labels and variables. The linker computes an appropriate offset and segment address and replaces the temporary address values with the new values.

The size of the value that LINK computes depends on the type of reference. If LINK discovers an error in the anticipated size of the reference, it displays a fix-up overflow error message. This happens, for example, when a program tries to use a 16-bit offset to address an instruction in a segment that has a different frame address. It also occurs when the segments in a group do not fit within a single, 64KB block of storage.

LINK resolves four types of references:

Short: Occurs in JMP instructions that try to pass control to labeled instructions that are in the same segment or group. The target instruction must be no longer than 128 bytes from the point of reference. The linker computes a signed, 8-bit number for this reference. It displays an error message if the target instruction belongs to a different segment or group (different frame address). The linker also displays an error message if the distance from the frame address to the target is more than 128 bytes in either direction.

Near self-relative: Occurs in instructions that access data relative to the same segment or group. The linker computes a 16-bit offset for this reference. It displays an error message if the data resides in more than one segment or group.

Near segment-relative: Occurs in instructions that attempt to access data either in a specified segment or group or relative to a specified segment register. LINK computes a 16-bit offset for this reference. It displays an error message if the offset of the target within the specified frame is greater than 64KB or less than 0 bytes. LINK also displays an error message if LINK cannot address the beginning of the sanctioned frame of the target.

Long: Occurs in CALL instructions that try to get access to an instruction in another segment or group. LINK computes a 16-bit frame address and a 16-bit offset for this reference. The linker displays an error message if the computed offset is greater than 64KB or less than 0 bytes. The linker also displays an error message if LINK cannot address the beginning of the sanctioned frame of the target.

Rules for Segment Packing in LINK

When the linker produces an OS/2 mode executable object, it can pack distinct, adjacent segments into the same physical or file segment. Physical or file segments are represented by entries in the program segment table. The rules that LINK uses when packing segments follow:

- The limit of the total size of a set of segments packed into a file segment is 64KB. LINK starts a new file segment while packing segments into a file when the size of the file segment reaches 64KB.
- LINK packs adjacent segments only into a file segment.

- LINK packs segments in the same group into a file segment. LINK does not pack segments in different groups into a file segment. If a segment in one group occurs between segments in another group, there is an error.
- If you use the /PACKCODE:*packlimit* or /PACKDATA:*packlimit* options, LINK packs code segments to the size you specify in *packlimit*. The default *packlimit* is 64KB.
- LINK will not pack code and data together with /PACKCODE or /PACKDATA. If the program declares code in the same group, all segments in the group are forced to type **CODE**.
- If LINK packs any segments that are EXECUTEREAD or READWRITE, it marks the entire file segment as EXECUTEREAD (if code) or READWRITE (if data).
- If LINK packs any segments that are PRELOAD, it designates the entire file segment as PRELOAD.

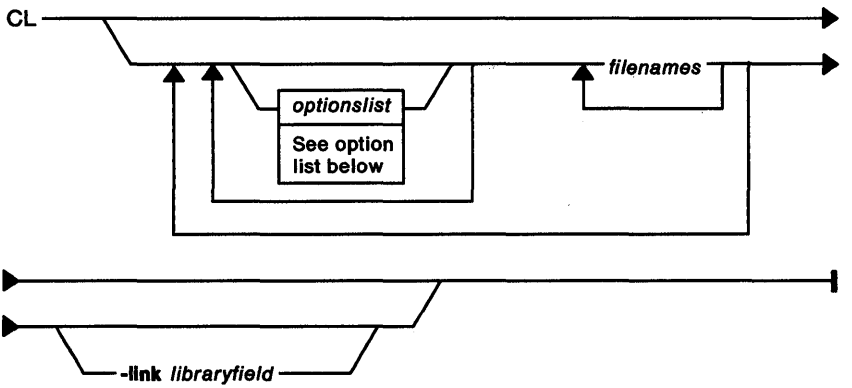
Compiling and Linking in One Step Using the CL Command

This section summarizes the CL command. You can use the CL command instead of the CC and LINK commands to call the compiler and LINK.

CL uses environment variables to locate the files it needs. It also uses the CL environment variable to read the options you want to be in effect for every compilation. Before calling CL, use the PATH and SET commands to assign a pathname or names to the following variables:

Variable	Types of Files
PATH	Executable compiler files
INCLUDE	Include files
TMP	Temporary files
LIB	Library files
CL	Option list.

Format



Parameters

optionslist A hyphen followed by a combination of one or more letters that have special meaning to the CL command. You can use a slash instead of the hyphen if you prefer. Most options available with the CC command are also available with the CL command.

CL reads the option list in the CL environment variable before the list at the command prompt and uses all the options it finds.

filenames The name of the file that the CL command processes.

-link The parameter that lets you pass information to LINK.

libraryfield The kinds of data you can pass to LINK. See "Linking with the CL Command" on page 3-59 for a description of the kinds of data you can pass.

Comments

You can give more than one *option* or *filename*, but you must set off each item with one or more spaces.

If you use the CL command without an argument, CL displays a summary of the CL command syntax. If you provide an argument, each *filename* must be the name of a C language source file or an object file. If the name is a source file, it must include the extension .c or .C. When CL processes the file, it looks at the filename extension to determine whether it should start compiling or linking the file. It compiles any files ending with .c or .C.

If the file extension is .ASM, CL does not compile the file. CL displays an error message indicating it cannot start the assembler. CL assumes any files with other extensions or no extensions are object files.

You can use the DOS global filename characters (? and *) in filenames in the CL command. The CL command expands these characters in the same manner that DOS does. (See the reference information for DOS for a description of the global filename characters.)

Because you can process more than one file at a time with the CL command, the order you give listing options in (the -F group of options) is important.

The -F options that can be used with the CL command are summarized in the chart below:

Default Option	Default Task	Filename
-Fs	Produces the source listing	Base name of source file plus .LST
-Fa	Produces the assembly listing	Base name of source file plus .ASM
-Fb	Binds the executable file	Base name of first source or object in the command plus .EXE
-Fc	Produces the combined source-assembly listing	Base name of source file plus .COD
-Fe	Names the executable file	Base name of first source or object file at command prompt plus .EXE
-Fl	Produces the object listing	Base name of source file plus .COD
-Fm	Creates the map file	Base name of first source or object file at the command prompt .MAP
-Fo	Names the object file	Base name of source file plus .OBJ

IBM Extension

LINK uses the default filename for the -Fs, -Fa, -Fc, -Fl, and -Fm options when the option has no argument or has a directory name as an argument. The default filename for the -Fe and -Fo options is used when the option is not given or when a directory name is given as the argument to the option.

End of IBM Extension

Some additional rules that apply to arguments of the -F options when used with the CL command are in the chart below:

File Name	Pathname Options	Argument	No Argument
-Fa, -Fc, -Fl, -Fs	Creates a listing for next source file at command prompt; uses default extension if no extension is supplied	Creates listings in the given directory for every source file listed after the option at the command prompt; uses default names	Creates listings in the default directory for every source file listed after the option at the command prompt; uses default names
-Fe	Uses given filename for the executable file; uses default extension if no extension is supplied	Creates executable file in the given directory; uses default name	Not applicable; argument is required
-Fm	Uses given filename for the map file; uses default extension if no extension is supplied	Creates map file in the given directory; uses default name	Uses default name
-Fo	Uses given filename as the object filename for the next source file at the command prompt; uses default extension if no extension is supplied	Creates object files in the given directory for every source file listed after the option at the command prompt; uses default names	Not applicable; argument is required
-Fb	Uses given name for bound executable	Binds executable file in the given directory	Uses same name as .EXE file

Note: You cannot put a space between a -F*x* option and any argument.

Unlike the CC command, the CL command calls LINK as well as the compiler. By default, CL automatically links. You can cancel this with the -c option, described in "Additional Options" on page 3-61. You can pass your own arguments to LINK. For a description of how to pass your own arguments to LINK, read the following section.

Linking with the CL Command

By default, the CL command calls LINK after compiling. To cancel the default and cause the CL command to stop after it compiles, use the -c (compile only) option. (See "Compiling without Linking /c" on page 3-63 for more information.)

The CL command uses the response-file method of calling LINK. By default, it builds the following response file:

```
LINK objectfiles [/CO]
basename /NOI
NUL;
```

The *libraryfield* does not have to be specified; if it is not, none is assumed by default. The names of the default combined libraries (the C library of the appropriate memory model plus the appropriate floating-point library as determined by the floating-point option you are using) are encoded in the object file. LINK searches for the default combined libraries in the current working directory, then in the directories specified in the LIB environment variable, if any.

The *objectfiles* are all object files produced in the compiling stage of the CL command, plus any object files specified in the CL command. The /CO option (for the CodeView symbolic debugger) is added to the first line of the response file if the -Zi option is used in the CL command. The /NOI option tells LINK *not* to ignore case; uppercase and lowercase letters are considered different. By default, *basename* is the name supplied for the executable file; it corresponds to the base name of the first source or object file in the CL command. However, you can provide a different name by using the -Fe option. By default, no map file is produced because the name *NUL* is provided in the third field. The -Fm option can be used in the CL command to override the default and produce a map file. A map file is also produced when the -Zd option is given in the CL command; with -Zd, CL builds the following response file:

```
LINK objectfiles [/LI]
basename /NOI
basename;
```

You can supply your own responses for the *libraryfield* by using the -link option. This option, if included, must be the last item in the CL command. Any libraries specified in the *libraryfield* are searched before the default libraries.

The *libraryfield* can contain one or more of the following:

A pathname: LINK searches the given pathname for the default libraries *before* searching directories given by the LIB variable.

Additional or alternate library names: If a pathname is included with the library name, only that pathname is searched. Otherwise, LINK uses the standard library search path.

Floating-point library or libraries: Any floating-point calls in your program refer to the given floating-point library instead of the default floating-point library.

Options: You can supply any of the LINK options described in this chapter.

Compiling and Linking Combined Libraries

If you link with the CL command, using the /Lc or /Lp option, CL automatically links the correct libraries. If you link separately by directly invoking LINK, then you must specify the name of your combined library at the LINK command prompt. To link an OS/2 mode program, specify both the name of the library and the file DOSCALLS.LIB at the LINK command prompt.

Example

```
LINK sample.obj,,SLIBCER.LIB;
LINK psample.obj,,MLIBCAP.LIB+DOSCALLS.LIB;
```

This applies if you installed IBM C/2 to create programs for both target operating environments or if you installed for one target environment without renaming the combined libraries to the default combined-library names. If you installed for a single target operating environment and renamed the combined libraries to the default combined-library names, invoke LINK without specifying a combined library.

Additional Options

The CL command also recognizes the options listed below. You can begin the options with the slash (/) character, or the hyphen (-) character.

Option	Task
A*string*	Creates a customized storage model.
c	Creates an object file for each source file at the command prompt; suppresses linking.
C	Preserves comments when preprocessing a file (only when -P or -E).
EP	Preprocesses each source file, copying the results to the standard output. Does not put ***#line*** directives in the output.
Fb[*bound-exe*]	Produces a bound executable file with the name *bound-exe*.
F *hexnumber*	Forces stack size to be set to *hexnumber* bytes; space required between /F and *hexnumber*.
Fe*programname*	Names the executable program file as *programname*.
Fm[*mapname*]	Creates a map file.
Gm	Allocates data items declared with the const keyword and all string literals in segments of type CONST or FAR_CONST.
Gw	Compiles a Presentation Manager application.
H	Restricts the length of external names.
link *libraryfield*	Passes the specified *libraryfield* to LINK.
Lc	Creates a DOS mode executable file.
Lp	Creates an OS/2 mode executable file.
nologo	Suppresses writing logo and copyright lines in the compiler output stream.
ND *name*	Sets the data segment name.
NM *name*	Sets the module name.
NT *name*	Sets the text segment name.
Oa	Cancels alias checking.
Oi	Declares some functions as intrinsic. See "Declaring Functions as Intrinsic /Oi" on page 3-66 for more information.
On	Disables loop optimizations.
Op	Enforces consistent precision in floating point operations, disables certain optimizations in floating-point expressions.

Ow	Restricts the compilers assumptions about where an alias can occur when /Oa is in effect.
Sl *linewidth*	Specifies *linewidth* as the number of characters-per-line used for source listings.
Sp *pagelength*	Specifies *page length* as the number of lines-per-page used for source listing.
Ss *subtitle*	Specifies *subtitle* as the subtitle used for source listings.
St *title*	Specifies *title* to be used for source listings.
Tc *filename*	Indicates that *filename* is a C source file. Used to compile files without .c (.C) extensions.
u	Undefines names.
V*string*	Labels the object file.
X	Removes the standard directories from the list of directories to be searched for *#include* files.
Zc	Allows names declared with the **pascal** and **fortran** modifiers to be used without regard for case.
Zp	Specifies that struct members are to be aligned on *n*-byte boundaries.

Advanced CL Topics

The following sections describe options that you use with CL to further specialize your program.

Specifying Overlays ( )

Specify overlays in the CL command by enclosing the names of the overlay source or object files in parentheses.

Example

```
CL src_1.c (src_2.c obj_1) obj_2
```

The preceding command causes the SRC_2.OBJ and OBJ_1.OBJ modules to be overlays in the SRC_1.EXE file. They are read into memory from disk only when they are needed.

Compiling without Linking
/c

This option suppresses linking. Source files are given at the command prompt, but the resulting object files are not linked. The option does not create an executable file, and object files specified at the command prompt are ignored.

Format

```
/c
```

This option is useful in compiling individual source files that do not make up a complete program.

The /c option applies to the entire CL command. The position of the option at the command prompt has no effect.

Example

```
CL /c *.C
```

This command compiles, but does not link, all files with the extension .C in the current working directory.

Creating Bound Applications
/Fb

This option allows you to bind a program after compiling and linking. Binding allows a program to run in both OS/2 mode and DOS mode. Some special rules apply to programs that must run in both modes.

See "Creating Family Applications" on page 3-5 for more information about creating bound applications.

Format

/Fb[*bound-exe*]

where *bound-exe* is a string literal that specifies the name you want for the bound program.

If you do not specify a name, the bound program uses the same base name as the unbound program and overwrites it. In other respects, the /Fb option follows the same file naming conventions as the /Fe

option. (See "Additional Options" on page 3-61.) This option causes the compiler to invoke the BIND utility with the following command:

```
BIND exe-file API.LIB
   <full-path-for-DOSCALLS.LIB> /o bound-exe
  /m map-name
```

If *bound-exe* is not specified, then exe-file is used for the output filename. If the /Fm option is specified, then the /m *map-name* option is also specified to BIND. *Map-name* defaults according to the usual rules for -Fm if not specified explicitly. A map file is only produced if some form of /Fm is specified.

To bind EXE files that have API calls that are not part of Family API, you must run BIND from the command prompt using the /n command for BIND. In this case you cannot use the /Fb option.

The /c option causes the /Fb option to be ignored; binding does not occur. You cannot bind files that have been linked with the /EXEPACK linker option.

You must be sure that the EXE file being bound was linked with the OS/2 mode libraries (either by default or with the /Lp command) and that API.LIB and DOSCALLS.LIB are in the current working directory or somewhere in the paths specified by the LIB environment variable. The CL command will find DOSCALLS.LIB and create a full pathname for the invocation of BIND.

Example

The following example creates the OS/2 mode executable file T.EXE and the bound executable file TB.EXE. No map file is produced.

```
cl /Fbtb /Ox /Lp t.c
```

This example compiles T.C and creates T.OBJ, but does not link or bind. The /Fb is ignored.

```
cl /Ox /c /Fb t.c
```

The next example creates the bound executable file T.EXE. The map file, T.MAP, will be the one created by BIND, not the one created by the linker for the unbound T.EXE. This example assumes that you have named the OS/2 mode libraries as *x*LIBC*y*.LIB rather than *x*LIBC*y*P.LIB, as in the dual mode case.

```
cl /Ox /Fb /Fm t.c
```

Compiling Presentation Manager Applications /Gw

This option causes the compiler to generate a special code sequence at the entry to a function. Use this option for developing applications to run in the Presentation Manager environment.

Format

```
/Gw
```

Presentation Manager recognizes the code and changes the instruction at runtime to set up a correct value in DS. This allows Presentation Manager applications to move data segments at runtime.

The /Aw option is also needed to control the segment setup. For more information about /Aw see "Setting Up Segments" on page 2-68.

Restricting the Length of External Names /H

This option lets you restrict the length of external (public) names.

Format

/Hnumber

The *number* is an integer specifying the maximum number of significant characters in external names. When you use the /H option, the compiler considers only the first *number* characters used in the program. The program can contain external names longer than *number* characters, but the compiler ignores the extra characters.

Use the /H option to conserve space or aid in creating portable programs. IBM C/2 places no restrictions on the length of external names (although it uses only the first 31 characters), but other compilers or linkers might produce errors when they find names longer than a certain limit.

The /H option is not available with the CC compiler command.

Suppressing Logo Lines
/nologo

This option turns off the display of the compiler copyright notice.

Format

```
/nologo
```

Use the /nologo option to save space on your compiler output stream. This is especially useful for MAKE files or when you redirect the compiler output to another file.

Declaring Functions as Intrinsic
/Oi

This option tells the compiler to generate intrinsic functions instead of function calls for certain functions.

Format

```
/Oi
#pragma intrinsic (function1[,function2]...)
#pragma function (function1[,function2]...)
```

Intrinsic functions may be in-line functions, use special argument-passing conventions, or, in some cases, do nothing. Programs that use intrinsic functions are faster because they do not include the overhead associated with function calls. However, they may be larger because they generate additional code.

The /Oi option declares the following list of functions as intrinsic:

abs	**fabs**	**memcpy**	**strcat**
acos	**fmod**	**memset**	**strcpy**
asin	**inp**	**min**	**strcmp**
atan	**inpw**	**outp**	**strlen**
atan2	**log**	**outpw**	**strset**
cos	**log10**	**pow**	**sqrt**
cosh	**_lrotl**	**_rotl**	**tan**
_disable	**_lrotr**	**_rotr**	**tanh**
_enable	**max**	**sin**	
exp	**memcmp**	**sinh**	

In the case of the non-floating-point functions, the compiler generates the code for the particular function directly in line rather than calling a function. In the case of the floating-point functions, the compiler passes the arguments on the coprocessor stack rather than the 80x86 stack.

Intrinsic versions of the **memset**, **memcpy**, and **memcmp** functions in compact- and large-model programs cannot handle **huge** arrays or **huge** pointers. To use **huge** arrays or **huge** pointers with these functions, you must compile your program, using the /AH option in the command.

If you use intrinsic forms of the floating-point functions listed above (that is, if you compile with the /Oi option or specify any of the functions in an **intrinsic** pragma), you cannot link with an alternate math library (*x*LIBCAz.LIB). Any attempt to do so will cause unresolved external errors at link time. Note that this restriction applies even if you link with an alternate math library after compiling with the /FPc or /FPc87 option.

To compile with **/Oi** (to use the intrinsic forms of non-floating-point functions) and then link with an alternate math library, specify any floating-point functions that appear in a program in a **function** pragma.

The **intrinsic** pragma affects the specified functions from the point where the pragma appears until either the end of the source file or the next **function** pragma specifying any of the same functions.

You can use the **function** pragma selectively to generate function calls instead of intrinsic functions when you compile a program with the /Oi option.

Naming Modules and Segments /NM, /NT, /ND

The following options of the CL command let you name modules and segments:

Format

```
/NM modulename
/NT textsegmentname
/ND datasegmentname
```

The space between each of the preceding options and the *name* is required. A *module* is an object file that the C compiler creates. Every module has a name. The compiler uses this name in error messages if it finds problems during processing. The name of the module is usually the same as the name of the source file. You can change this name using the /NM (name-module) option. The new *modulename* can be any combination of letters and digits.

A *segment* is an adjoining block of binary information (code or data) produced by the C compiler. Every module has at least two segments: a text segment containing the program instructions and a data segment containing the program data. Each segment in every module has a name. The linker uses this name to define segment storage order when you load the program for running.

Note: The segments in the group DGROUP are an exception. For more information about DGROUP, see "Groups" on page 6-3.

The C compiler creates text and data segment names. These default names depend on the storage model chosen for the program. For example, in small-model programs the compiler names the text segment _TEXT and the data segment _DATA. The names are the same for all small-model modules, so the compiler loads all text segments from all modules as one adjoining block, and all data segments from all modules form another adjoining block.

In medium-model programs, the compiler places the text from each module in a separate segment with the suffix _TEXT. The data segment is named _DATA as in the small model.

In compact-model programs, the compiler places the data from each module in a separate segment with a distinct name, formed by using

the module base name along with the suffix _DATA. An exception to this is initialized global and static data, which the compiler puts in the default data segment, _DATA. The code segment is named _TEXT, as in the small model.

In large- and huge-model programs, the compiler loads the text and data from each module into separate segments with distinct names. Each text segment is given the name of the module plus the suffix _TEXT. The compiler places the data, except for initialized global and static data placed in the default data segment, from each segment in a private segment with a unique name.

Even if you have used the /NM command, the compiler uses the name of the file itself, not the name of the module, in error messages. The effect of the /NM option is limited to changing the name of the text segment in a program with a large code segment _TEXT to *module*_TEXT. In a small-model program, the /NM option has no effect.

The following table summarizes the naming conventions for text and data segments:

Model	Text	Data	Module
Small	**_TEXT**	**_DATA**	*filename*
Medium	*module*_**TEXT**	**_DATA**	*filename*
Compact	**_TEXT**	**_DATA**	*filename*
Large	*module*_**TEXT**	**_DATA**	*filename*
Huge	*module*_**TEXT**	**_DATA**	*filename*

Note: For compact, large, and huge models, _DATA is the name of the default data segment. Other data segments have unique, private names.

You can cancel the default names used by the C compiler, thus canceling the default loading order, by using the /NT (name text) and /ND (name data) options. These options set the names of the text and data segments, in each module being compiled, to a given name. The *textsegmentname* used with the /NT option, and *datasegmentname* used with the /ND option can be any combination

of letters and digits. To specify *name* as the data segment name loaded by subsequent load DS functions, use the following.

```
#pragma data_seg([name])
```

All subsequent initialized static and global data will be allocated into segment *name*. If omitted, *name* defaults to the name specified by the /ND command, if any; otherwise DGROUP is assumed.

Note: The /ND command no longer requires that you set /Au also. If you supply /ND without /Au, it applies to all functions in the module that have the **_loadds** attribute and whose data segment name is not overridden by the **data_seg** pragma. If you also supply /Au, the data segment will be loaded and you can use the **data_seg** pragma to change the segment name at the source level. The #pragma **data_seg** does not cause a function to load a data segment. Use /Au or the **_loadds** keyword to cause a function to load a data segment upon entry.

The /NM, /ND, and /NT options are not available with the CC command.

Placing Variables and Functions in Segments

The **alloc_text** pragma gives you source-level control over the segment particular functions are allocated in. The **same_seg** pragma provides information the compiler can use to generate better code.

Format

```
# pragma alloc_text(textsegment, function1[, function2]...)
# pragma same_seg(variable1[,variable2]...)
```

If you use overlays or swapping techniques to handle large programs, **alloc_text** allows you to tune the contents of their text segments for maximum efficiency. The **alloc_text** pragma must appear before the definitions of any of the specified functions, but it can appear either before or after the functions are declared or called. Any functions specified in an **alloc_text** pragma must either be explicitly declared with the **far** keyword or assumed to be **far** because of the memory model used (medium, large, or huge).

The **same_seg** pragma tells the compiler to assume that the specified external variables are allocated in the same data segment. You are responsible for making sure that these variables are put in the same data segment; one way to do this is to specify the /ND option when you compile the program. The **same_seg** pragma must appear before any of the specified variables are used in executable code and after the variables are declared. Variables specified in a **same_seg** pragma must be explicitly declared with **extern** storage class, and they must either be explicitly declared with the **far** keyword or assumed to be **far** because of the memory model used (compact, large, or huge).

Loop Optimization /Ol

This option tells the compiler to perform loop optimizations. For best performance, the /Ol option should be specified along with the *a* option letter /Oal, since the compiler can detect more loop optimizations when it relaxes its assumptions about the use of aliases.

Use the **loop_opt** pragma to turn loop optimization on or off for selected functions. To turn off loop optimization, put the following line before the code:

Format

```
#pragma loop_opt (off)
```

The preceding line stops loop optimization for all code that follows it in the source file. The routines on the same line also stop. To reinstate loop optimization, insert the following line:

```
#pragma loop_opt (on)
```

The interaction of the **loop_opt** pragma with the /Ol and /Ox options is explained in the following table:

Format	Compiled with /Ox or /Ol?	Loops Optimized?
#pragma loop_opt()	No	No
#pragma loop_opt()	Yes	Yes
#pragma loop_opt (on)	Yes or No	Yes

Format	Compiled with /Ox or /Oi?	Loops Optimized?
#pragma loop_opt (off)	Yes or No	No

Setting Line Width and Page Length /Sl, /Sp

These options are useful in preparing source listings for a printer that uses nonstandard page lengths.

Format

```
/Sl linewidth
/Sp pagelength
#pragma linesize([n])
#pragma pagesize([n])
```

The *linewidth* is the width of the listing line in columns (on line printers, columns usually correspond to characters). It must be a positive integer between 79 and 132; any number outside that range causes the compiler to issue a diagnostic message and default to 79. Any line that exceeds the listing width is truncated.

The *pagelength* argument gives the number of lines to appear on each page of the listing. The number given must be a positive integer between 15 and 255. Any number outside this range causes the compiler to generate a diagnostic message and default to 63.

The /Sl or /Sp option applies to the remainder of the command or until the next occurrence of /Sl or /Sp in the command. These options do not create source listings; they take effect only if you also specify the /Fs option to create a source listing.

The **#pragma linesize** sets the number of characters per line to *n*. It must be an integer constant between 79 and 132. If *n* is omitted, the line size is set to the value specified by the /Sl linesize command, if any, or to 79, which is the default.

The **#pragma pagesize** sets the number of lines per page to *n*, which must be an integer constant between 15 and 255. If *n* is omitted, the line size is set to the value specified by the **/Sl** pagesize command, if any, or to 63, which is the default.

Example

```
CL /c /Fs /Sl 90 /Sp 70 *.C
```

The previous example compiles all C source files with the default extension (.C) in the current working directory, creating a source-listing file for each source file. Each page of the source-listing file is 90 columns wide and 70 lines long.

Setting Titles and Subtitles /St, /Ss

Format

```
/St "title"
/Ss "subtitle"
#pragma title(string)
#pragma subtitle(string)
```

The /St and /Ss options set the title and subtitle, respectively, for source listings. The quotation marks around the *title* and *subtitle* argument can be omitted if the title or subtitle does not contain space or tab characters. The space between /St or /Ss and their arguments is optional.

The title appears in the upper-left corner of each page of the source listing. The subtitle appears below the title.

The /St or /Ss option applies to the remainder of the command or until the next occurrence of /St or /Ss in the command. These options do not create source listings.

Example

```
CL /St "INCOME TAX" /Ss 4-14 /Fs TAX*.C
```

The previous example compiles and links all source files beginning with TAX and ending with the default extension (.C) in the current working directory. Each page of the source listing contains the title INCOME TAX in the upper-left corner. The subtitle 4-14 appears below the title on each page.

The following example compiles two source files and creates two source listings.

```
CL /c /Fs /St"CALC PROG" /Ss"COUNT" CT.C /Ss"SORT" SRT.C
```

Each source listing has a unique subtitle, but both listings have the title CALC PROG.

Specifying Source Files /Tc

This option tells the CL command that the *sourcefile* is a C source file.

Format

/Tc sourcefile

Using this option causes CL to treat the *sourcefile* as a C source file, regardless of its extension. A separate /Tc option must appear for each source file that has an extension other than .C.

Labeling the Object File /V

This option imbeds a given text *string* into an object file. You can omit the quotation marks surrounding the string if the string does not contain white-space characters.

Format

/V"string"

Use the /V option to label an object file with a version number or a copyright notice. The /V option is available only with the CL command, not with the CC compiler command.

Example

```
CL  /V"IBM C/2"MAIN.C
```

The above command places the string: IBM C/2 into the object file MAIN.OBJ.

Creating Special Object File Records #pragma comment

The following pragma puts a comment record into an object file.

Format

#pragma comment(comment-type[, char-sequence])

The type of comment record is specified by the *comment-type*, which can be any of the following:

compiler Inserts a general comment record into the object module containing the name and version number of the compiler used to build the object module. The *char-sequence*, if any, is ignored.

date Inserts a general comment record into the object module containing the date and time of compilation. This comment record is ignored by the linker; the *char-sequence*, if any, is also ignored.

lib Inserts a library-search record into the object file. It must be followed by a char-sequence that is the (path) name of the library to be imbedded in the search record. The library name will be emitted into the object file prior to the default library search records (so that the behavior at link time is the same as if the library name were specified at link time). Multiple comment records of this type are inserted in the order they are encountered in the source.

timestamp Inserts a general comment record into the object module containing the date and time of the last modification to the source file. The date and time in the comment record is the same as that returned by the **asctime** function.

user Inserts a general comment record into the object module, which contains the *char-sequence* specified in the pragma. The linker ignores this comment record.

Writing Output Messages #pragma message

The **message** pragma is similar to the **#error** directive.

Format

```
#pragma message(string)
```

Unlike the **#error** directive, the **message** pragma writes the *string* to **stdout** instead of **stderr** and does not end compiling. See Chapter 9 of *IBM C/2 Fundamentals* for more information about the **#error** directive.

Chapter 4. Running C Programs

After you create an executable file, you can run your program by giving the name of the file without the extension. However, if you plan to run your program in the OS/2 mode, the name of the file should have an .EXE extension.

DOS uses the PATH environment variable to find executable files. You can run a program from any directory, as long as the executable program file is either in your current working directory or in one of the directories on the PATH.

The spawn, exec, and system routines provided in the IBM C run-time library let you run other programs and DOS commands from within a program. See *IBM C/2 Language Reference* for a description of these routines.

In the following example, the file named MYPROG can run as an OS/2 mode executable file named MYPROG.EXE or it can run as a DOS mode executable file named MYPROG.COM:

Example

```
MYPROG
```

Passing Data to a Program argc, argv, envp

You can get access to data at the command prompt or in the environment table at run time by declaring arguments to the **main** function. *Command prompt data* is any data that appears on the same line as the program name when you run the program. The environment table contains all environment settings in effect at run time. Because a program run begins at the **main** function, the **main** function controls data passed at run time.

To pass data to your program by way of the command prompt, give one or more arguments after the program name when you run it. Each argument must be separated from other arguments by one or more spaces or tab characters and can be enclosed in double quotation marks. To give a single argument that includes spaces or tab

characters, you must enclose the argument in double quotation marks. For example:

```
TEST 42 "de f" 16
```

This command runs the program named TEST.EXE (or TEST.COM) and passes three arguments: 42, de f, and 16.

In the DOS mode, each command-prompt argument, regardless of its data type, is stored as a null-terminated string in an array of strings. The combined length of all arguments in the command (including the program name) cannot exceed 128 bytes.

To set up your program to receive the command prompt data, declare arguments to the **main** function as shown in the following example. By declaring these variables as arguments to the **main** function, you make them available as local variables in the **main** function.

```
main (argc, argv, envp)

int argc;
char *argv[];
char *envp[];
```

You need not declare all three arguments, but if you do, they must be in the order shown. To use the *envp* arguments, you must declare *argc* and *argv*, even if you do not use them.

The command is passed to the program as the *argv* array of strings. The number of arguments appearing at the command prompt is passed as the integer variable *argc*.

The first argument of any command is the name of the program to run. The program name is the first string stored in *argv*, at *argv*[0]. Because you must always give a program name, the integer value of *argc* is at least 1.

The compiler stores the first argument after the program name at *argv*[1], the second at *argv*[2], and so on through the end of the arguments. It stores the total number of arguments, including the program name, in *argc*. The third argument passed to the **main** function, *envp,* is a pointer to the environment table. You can get access to the value of environment settings through this pointer. However, the putenv and getenv routines from the C run-time library accomplish the same task and are easier and safer to use.

Using the putenv routine might change the location of the environment table in storage, depending on storage requirements. The value given to *envp* at the beginning of the program run might not be correct throughout the running of the program. On the other hand, the putenv and getenv routines get access to the environment table properly, even when its location changes. These routines use the global variable *environ*, described in *IBM C/2 Language Reference*, which points to the correct table location.

Example
The following command runs the program MYPROG. It also passes the four command-prompt arguments to the **main** function of MYPROG. The arguments are stored as null-ended strings, and the number of arguments is stored in *argc*.

```
MYPROG ABC "abc e" 3 8
```

To get access to the last argument, for example, use an expression similar to the following:

```
argv[argc - 1]
```

Because the value of *argc* is 5 (counting the program name as an argument), this expression is equivalent to *argv*[4], or the fifth string of the array.

Exiting from the Main Function

The **main** function, like any other C function, returns a value. The return value of **main** is an int value that is passed to DOS as the return code of the program that has been run. You can check this return code with the IF ERRORLEVEL command in DOS batch files.

To cause the **main** function to return a specific value to DOS, use a return statement or **exit** function to specify the value to be returned. For example, the statement return(6); causes the value 6 to be returned. If you do not use either method, the return code is undefined.

Expanding Global Filename Arguments

You can use the DOS global filename characters, the question mark (?), and the asterisk (*) to specify the filename and pathname arguments at the command prompt. To prepare for using global filename characters, you must link with the special routine SETARGV.OBJ. This object file is included with your compiler software. If you do not link with SETARGV.OBJ, the compiler treats global filename characters literally.

SETARGV.OBJ expands the global filename characters in the same manner that DOS does. (See the system reference information for DOS to learn more about global filename characters). If you are enclosing an argument in quotation marks, global filename expansion is suppressed. Within quoted arguments, you can literally represent quotation marks by preceding the double quote character with a backslash.

If the compiler finds no matches for the global filename arguments, the argument is passed literally. For example, if the argument B:*.C is given, but the compiler finds no files with the extension of .C in the root directory of drive B, it passes the argument as the string B:*.C.

If you frequently use global filename expansion, place the global filename routine in whichever *x*LIB*yz*.LIB combined libraries you use. That way the routine is linked with your program automatically. To do this, use the LIB utility (see "Starting LIB" on page 5-24) to extract the module named STDARGV from the library (the module name is the same in any combined library) and insert SETARGV. When you replace STDARGV with the SETARGV routine, global filename expansions are performed on command-prompt arguments.

Example

In the following example, SETARGV.OBJ is linked with BETA.OBJ, producing the executable file BETA.EXE. When you run BETA.EXE, the compiler expands the global filename character (*), causing all filenames with the extension .INC in the current working directory to be passed as arguments to the BETA program:

```
LINK BETA SETARGV;
BETA *.INC "WHY?" \"HELLO\"
```

The argument WHY? is enclosed in double quotation marks, so expansion of the global filename character (?) is suppressed and the

argument WHY? is passed literally. In the third argument, the backslashes cause the quotation marks to be represented literally so the argument "HELLO" is passed.

Suppressing Command Processing

If your program does not take command-prompt arguments, you can save a small amount of space by suppressing the library routine that performs command processing. This routine is called _setargv. To suppress its use, define a routine that does nothing in the same file that contains the **main** function and name it _setargv. The call to _setargv is satisfied by your definition of _setargv, and the library version is not loaded.

If you do not get access to the environment table through the *envp* argument, you can provide your own empty routine to be used in place of _setenvp (the environment processing routine). In the same file that contains the **main** function, define a routine that does nothing, and name it _setenvp.

If your program makes calls to the spawn or exec routines in the C run-time library, do not suppress the environment processing routine. This routine is used to pass an environment from the parent process to the child process.

Example

This example shows how to define the _setargv and _setenvp functions to suppress command and environment processing. It is recommended that you place these definitions in the file containing the **main** function.

```
_setargv()
{
}

_setenvp()
{
}
```

Suppressing Null Pointer Checks

When you run your C program in the DOS mode, a special error-checking routine is automatically called to determine whether the content of the NULL segment has changed. This error-checking routine displays the following error message if the NULL segment has changed: **Null pointer assignment.**

The NULL segment is a special location in low storage that programs normally do not use. If the contents of the NULL segment changes during the program run, it means that the program has written to this area, usually by an inadvertent assignment through a null pointer. Notice that your program can contain null pointers without generating the preceding message; the message appears only when you write to a storage location through the null pointer.

This error message does not cause your program to end. The error is detected and the error message is printed after the normal ending of the program.

Note: The **Null pointer assignment** message tells you that there is a potentially serious error in your program. Although a program that produces this error might appear to operate correctly, it can cause problems later and might fail to run in a different operating environment, such as OS/2.

The library routine that performs the null pointer check is named _nullcheck. Suppress the null pointer check for a particular program by defining a routine that does nothing and naming it _nullcheck. The call to _nullcheck is satisfied by your definition of _nullcheck, and the library version is not loaded. It is recommended that you place the _nullcheck definition in the file containing the **main** function.

To suppress the null pointer check for all programs, replace the corresponding error-checking routine in the standard C library. The routine is stored in a module called chksum in all four libraries (SLIBC.LIB, MLIBC.LIB, CLIBC.LIB and LLIBC.LIB). Do not remove the routine entirely or there will be an unresolved reference in your program. Instead, use the LIB utility (see "Starting LIB" on page 5-24) to replace the chksum module with a module containing the empty definition of _nullcheck. This replacement satisfies the call to _nullcheck, and null pointer checking is not performed.

Chapter 5. Using the Program Utilities

MAKE

The IBM Program Maintainer MAKE automates the process required to maintain programs written in Macro Assembler, C, and other high-level languages. MAKE carries out all tasks needed to update a program after one or more of the source files in the program have changed.

MAKE compares the last modification date of the file or files that need updating with the modification dates of files that these target files depend on. MAKE then carries out a given task only if a target file is out of date. MAKE does not compile and link all files just because one file was updated. This can save much time when creating programs that have many source files or that take several steps to complete.

The following sections explain how to use MAKE and illustrate how to maintain a sample C program. A list of MAKE error messages is in Appendix A of this book. For a complete listing of the differences between this version of IBM MAKE/2, Version 1.00, and IBM MAKE/2, Version 1.10, see page xi.

Using MAKE

To use MAKE, you must create a MAKE description file that defines the tasks you want to accomplish and the files these tasks depend on. You can create a MAKE description file using a text editor that produces ASCII files.

Format

A MAKE description file consists of one or more description blocks. Each description block has the general form:

```
target-list: [dependent-list][;command]
  [command]

:
```

target-list A list of one or more files that may need updating. If one of the files is older than any of the files specified in the *dependent-list*, MAKE runs the commands listed in this description block.

You can specify any number of target files. Separate the filenames with spaces. The first filename must begin in column one of the file. The target files must be valid filenames and must include drive specifications and pathnames if the files are not located in the current directory.

dependent-list A list of one or more files that are required to build the target. (For example, you need one or more .OBJ files to build an .EXE file.) You can specify any number of dependent files. Separate the filenames with spaces. For each file, you can specify a list of directories that MAKE should search for that file. Enclose the list in braces ({}) and separate the directory names with semicolons. For example, if you specify the following dependent file:

```
{\c2\src;d:\project}pass.obj
```

MAKE looks for PASS.OBJ in the current directory, then in \C2\SRC on the current drive, then in \PROJECT on drive D. If PASS.OBJ is not in any of these locations, MAKE searches for an inference rule to build the dependent file. See "Inference Rules" on page 5-17 for more information.

command An action to be taken if a target file is older than one of its dependent files. This can be any operating system command (internal or external) or any executable file (with an extension of .BAT, .CMD, .EXE, or .COM). If you provide a command on the same line as the *dependent-list*, separate the two with a semicolon. If you provide commands on separate lines, do not begin the commands in column one; precede each with at least one space or tab character. You can put only one command on a line.

If you do not specify any commands for a *target-list*, MAKE searches for an inference rule to build the target files. See "Inference Rules" on page 5-17 for more information.

Note: Make allows the use of both \ and / as path separators in target filenames, include filenames, and inference rules. Notice, however that MAKE does not convert / to \ on command lines; make sure that on command lines you provide

separators that are recognized by the operating system and command you are using.

You can think of the MAKE format as an if-then statement as follows:

- If the *target-file* is older than any of its dependent files,

  or

- If the *target-file* does not exist,
- Then execute the commands.

Example

The following example creates three target files. Each file has at least one dependent file and one command. MAKE examines WORK1.OBJ and WORK2.OBJ and creates them, if necessary, before WORK.EXE.

```
WORK.EXE:   WORK1.OBJ WORK2.OBJ \LIB\MATH.LIB
 LINK WORK1+WORK2,WORK,WORK,\LIB\MATH.LIB /CO

WORK1.OBJ:  WORK1.C WORK.H
 CC /Zi         WORK1,,,;

WORK2.OBJ:  WORK2.C WORK.H
 CC /Zi         WORK2,,,;
```

For information about how MAKE uses this description file see "Maintaining a Program" on page 5-9.

Using Comment Characters

Anything between a comment character (#) and a newline character is considered a comment. MAKE ignores all characters on a line after the comment character. A comment must not be on the same line as a command, but it can appear on the same line as the target-dependent-description line.

The comment character:

- Must be in the first column if the comment line is between lines containing commands, or the comment will be considered part of a command.
- Can be placed at the beginning of any of the lines in a command spanning several lines; this causes MAKE to ignore the part of the command on that line only.
- Can appear anywhere on any other lines.

Example

Some examples of comments are:

```
t.exe: d.obj  # Valid comment
# Valid comment
    # Invalid (must start in first column)
    link d,t; # Invalid (cannot follow command)
t2.exe: d2.obj; link d2,t2; # Invalid (cannot follow command)
```

Global Filename Character Expansion

Global filename characters, such as the asterisk and question mark, may be used in filenames. Beginning a filename with an asterisk (for example, *.TXT) matches all files with the specified suffix (or all files, if you enter *.*). Global filename characters in target names expand when the MAKE description file is read; in dependent names or commands, they expand when a target is built.

Continuing Long Lines

You can break long description block lines over several physical lines. If the last two characters on a line are a space and a backslash or a tab character and a backslash, MAKE uses the next line of the MAKE description file as if it were a continuation of that line. Comment lines cannot be continued in this manner.

Example

```
EEPATH=edit\e\          # This a path
```

In the above example, if the comment is excluded, the following line is taken as the continuation of the path description for EEPATH.

The following two description blocks are considered the same:

```
target.exe: \
  depend.obj; \
    link depend, \
         target;

target.exe: depend.obj; link depend,target;
```

In the following example, each piece of the LINK command is listed on a separate physical line. DEPEND2.OBJ is not included during the link step. DEPEND3.OBJ, however, is included.

```
target.exe: depend1.obj depend2.obj depend3.obj
  link \
       depend1+ \
#      depend2+ \
       depend3, \
       target;
```

Special Command Prefixes

You can alter the way MAKE handles commands by placing the following special characters in front of the commands:

- `-` Turns off error checking for that command only.
- *–n* Causes MAKE to halt if the error level returned by the command is greater than *n*.
- `@` Causes MAKE not to echo that command.
- `!` Calls a command containing a list of arguments (in the form of a macro, for example, $? or $**) repetitively for each argument in the list.

Example

The following example generates four COPY commands:

```
print:    file1.c file2.c file3.c file4.c
   !copy $** lpt1:
```

Using the Escape Character

Some characters have special meanings within the MAKE description file. There may be times when these special meanings conflict with the commands and arguments you need to build your targets. MAKE provides the ^ escape character to remove the special meaning of the character that follows it.

Example

Suppose you have a file named FILE#1.OBJ that is used to build PROGRAM.EXE. To make sure that the # in "FILE#1" is not interpreted as a comment character, precede it with the escape character.

```
PROGRAM.EXE: FILE^#1.OBJ LIBRARY.LIB
LINK FILE^#1,,,LIBRARY.LIB;
```

Specifying Multiple Description Blocks for a Target

You can use one set of commands to build some of a target's dependents and another set to build others. To do this, list the target twice in the MAKE description file, and use a double colon, instead of the usual single colon, to separate it from its dependents.

Example

The following example allows you to maintain an object library whose source files are partly in C and partly in assembler.

```
target.lib:: a.asm b.asm c.asm
     !masm  $**;
      lib  -+a-+b-+c;

target.lib:: d.c e.c
     !cl /c $**;
     lib -+d-+e;
```

MAKE executes the commands in the first description block if any of the .ASM files are out-of-date; similarly, it executes the commands in the second block only if any of the .C files need updating.

When you use a single colon as a separator, successive dependencies are cumulative, for example

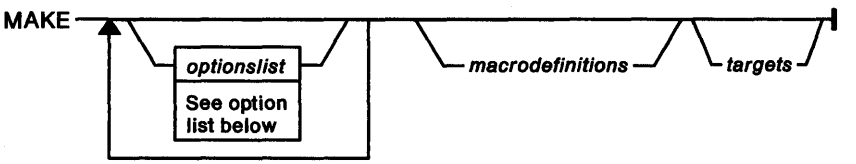
```
target.obj: target.c
target.obj: target.h
   cl  target.c
```

is equivalent to

```
target.obj: target.c target.h
    cl target.c
```

Starting MAKE

The MAKE command has the following syntax:



MAKE accepts command options preceded by either a slash or a dash, and the options may be given in either uppercase or lowercase (-a is the same as /A).

optionslist

The following options are available with the MAKE command:

/a Causes all targets to be rebuilt even if they are not out-of-date.

/c Starts cryptic output mode. *Cryptic mode* suppresses the MAKE copyright message and non-fatal error or warning messages. Cryptic mode does *not* override the /p or /d options, nor does it cause the /s option to be in effect.

/d Displays debugging information (the modification date of each file).

/e Causes environment variables to override assignments within MAKE description files.

/f Uses the next argument at the command prompt as the name of the MAKE description file to use. If the next argument is –, then MAKE reads from **stdin**. If you do not give a /f option, MAKE expects that a description file named MAKEFILE is present.

/i Causes MAKE to ignore error codes returned by programs called from within MAKE.

/n Displays the commands that would be executed but does not execute them.

/p Prints the complete set of macro definitions and target descriptions.

/q Returns a zero status code if the target file is up-to-date, nonzero if it is not.

/r Ignores rules and macros from TOOLS.INI. (See "Specifying Macros and Inference Rules with TOOLS.INI" on page 5-21 for more information.)

/s Causes MAKE to run in silent mode; commands are not displayed as they run.

/t Touches the out-of-date target files. *Touching* sets the date and time to the current date and time without modifying the contents of the files.

/x Uses the next argument as the name of the file to redirect **stderr** to. If the next argument is –, then **stderr** is redirected to **stdout**.

macro-definitions One or more macro definitions as described under "Using Macro Definitions" on page 5-12. Definitions that contain spaces must be enclosed in quotation marks.

targets The names of the targets listed in the MAKE description file that you want to build. If you do not specify a target, MAKE builds the first target listed in the MAKE description file.

Note: For compatibility with previous versions of IBM MAKE, if no /f option is specified and MAKE cannot find a file named MAKEFILE, it uses the first string in the command that is not an option or macro definition and does not contain an extension from the .SUFFIXES list as the name of the MAKE description file. See "Special Rules" on page 5-19 for more information on the .SUFFIXES list.

Example

The following example starts MAKE and instructs it to use the file UTILS.MAK as the MAKE description file. The targets that are built are SORT.EXE and SEARCH.EXE (and any targets that they are dependent on). The macro OPTIONS is defined for use within the MAKE description file. Suppose you have a MAKE description file named UTILS.MAK that contains the following lines:

```
SORT.EXE: SORT.C
  CL $(OPTIONS) SORT.C

SPLIT.EXE: SPLIT.C
  CL $(OPTIONS) SPLIT.C

SEARCH.EXE: SEARCH.C
  CL $(OPTIONS) SEARCH.C
```

Also suppose that you want to build SORT.EXE and SEARCH.EXE, but not SPLIT.EXE. You also want to define the macro OPTIONS so the /Zi and /Od compiler options are used when building these files. You would start MAKE as follows:

```
MAKE /F UTILS.MAK "OPTIONS=/Zi /Od" SORT.EXE SEARCH.EXE
```

If UTILS.MAK is the only MAKE description file in the current directory, you could take advantage of the default description filename that MAKE uses. Rename UTILS.MAK to MAKEFILE and start MAKE as follows:

```
MAKE "OPTIONS=/Zi /Od" SORT.EXE SEARCH.EXE
```

Because you have not given MAKE a MAKE description filename (using the /f option), MAKE assumes that the file it should use is called MAKEFILE.

Specifying MAKE Arguments from a File

If you specify macro definitions when starting MAKE, the command may exceed the DOS 128-character limit. To avoid this problem, put your command-prompt arguments in a file. For example, the command:

```
MAKE @makeargs
```

causes MAKE to read its command-prompt arguments from the file *makeargs*. This file is not a substitute for the makefile; it is merely a device that allows you to define long macros when starting MAKE.

The order for MAKE arguments stored in a command file is the same as the order for arguments given at the command prompt. Newline characters between arguments in the command file are treated as

spaces. Macro definitions may be split over several lines by ending each line (except the last line of the definition) with a backslash and the newline character. The backslash must be preceded by a space or tab to seperate it from the text on the line.

Example

If the following were the contents of a file named MAKE.ARG, then entering

```
make @make.arg
```

at the command prompt would have the same effect as the example shown on 5-8.

```
/f utils.mak
"OPTIONS \
= /Zi" /Od
sort.exe
search.exe
```

Maintaining a Program

Consider a program called WORK.EXE that is made from two source files, WORK1.C and WORK2.C. Both source modules use an include file called WORK.H. You must link both modules with routines in a library file called MATH.LIB. During development, you often compile and link to create WORK.EXE. To recompile only the source files that were changed use the following MAKE description file.

```
WORK.EXE:   WORK1.OBJ WORK2.OBJ \LIB\MATH.LIB
 LINK WORK1+WORK2,WORK,WORK,\LIB\MATH.LIB /CO

WORK1.OBJ:  WORK1.C WORK.H
 CC /Zi         WORK1,,,;

WORK2.OBJ:  WORK2.C WORK.H
 CC /Zi         WORK2,,,;
```

After each session of debugging and editing source files, start MAKE with the following command:

```
MAKE /F WORK WORK.EXE
```

MAKE checks to see if you have:

1. Changed WORK1.C or WORK.H since the last time that the compiler created WORK1.OBJ. If so, MAKE recompiles WORK1.C.
2. Changed WORK2.C or WORK.H since the last time that the compiler created WORK2.OBJ. If so, MAKE recompiles only WORK2.C.
3. Changed either of the object files since the last time the modules were linked. If the compiler compiled one or both of the files,

MAKE relinks the program. MAKE also relinks the program if MATH.LIB has changed.

When you first create the source files, MAKE compiles and links both source files because none of the target files exist. If you call MAKE again without changing any dependent files, MAKE skips all commands. If you change one of the source files, MAKE recompiles that file and relinks the program. If you change the library file MATH.LIB but make no other changes, MAKE skips the commands in the last two dependencies and relinks the program as specified in the first dependency.

Advanced MAKE Topics

Using Directives

MAKE supports directives for conditionals, file inclusion, and errors. Directives begin with an exclamation point. The exclamation point must be located in column one and cannot be preceded by a space. However, a space is allowed between the exclamation point and the directive keyword that follows it. The following directives are available.

!CMDSWITCHES	**!ERROR**	**!IFNDEF**
!ELSE	**!IF**	**!INCLUDE**
!ENDIF	**!IFDEF**	**!UNDEF**

!IF *constant-expression*: Checks to see whether *constant-expression* evaluates to nonzero. Constant expressions consist of integer constants, string constants, and program invocations. These constants may be connected by the following operators (except string constants, on which only the == and != operators are defined):

Binary: + − * / % & | ^^ && || << >> == != < > <= >=
Unary: − ~ !

Note: These are the same operators that are used by the C language, except for the bitwise exclusive OR operator (^). Because MAKE uses the ^ as an escape character, you must use ^^ to perform the bitwise exclusive OR operation within the MAKE utility.

Parentheses may be used to group operands.

Specify integer constants in decimal, octal, or hexadecimal notation (as in C notation, 7 is decimal, 07 is octal, and 0x7 is hexadecimal).

String constants must be enclosed in quotation marks. The following is a test for equality with string constants:

```
!IF "$(VER)" == "DOS3"
```

Invoke a program from within an **!IF** directive as follows:

```
!IF [prog arg1 arg2 ... argn]
```

Such program invocations evaluate to TRUE when the return code is non-zero. Note that programs that terminate without error usually return zero. The following example tests the value returned by the program by making the program invocation part of a constant expression:

```
!IF     ([prog1 arg1] != 0) || ([prog2] == -1)
```

!IFDEF *identifier*: Checks to see whether the macro *identifier* is currently defined. If so, the statements between the **!IFDEF** and the next **!ELSE** or **!ENDIF** directive are executed. Note that macros defined as the null string are still considered as defined by **!IFDEF**. To undefine a symbol, use the **!UNDEF** directive.

!IFNDEF *identifier*: Checks to see whether the macro *identifier* is currently defined. If it is not, statements between the **!IFNDEF** and the next **!ELSE** or **!ENDIF** directive are executed.

!ELSE: Causes MAKE to execute the following statements (up until the **!ENDIF** directive) if the preceding **!IF**, **!IFDEF**, or **!IFNDEF** evaluated to FALSE (zero).

!ENDIF: Marks the end of an **!IF**, **!IFDEF**, or **!IFNDEF** block of statements.

!UNDEF *identifier*: Undefines the macro *identifier*.

!ERROR *text*: Causes MAKE to print *text* and then stop execution.

!INCLUDE *filename*: Causes MAKE to read and evaluate *filename* before continuing with the current MAKE description file. If *filename* is enclosed between less-than and greater-than symbols (< >), MAKE searches for the file in the directories specified by the

INCLUDE macro; otherwise, MAKE looks for *filename* in the current directory.

The value of the INCLUDE macro is initially set the same as the value of the INCLUDE environment variable. If you redefine the INCLUDE macro, MAKE uses the new definition, ignoring the value of the environment variable.

Note: You can leave the exclamation point off of the **!INCLUDE** directive. If INCLUDE begins in column one, MAKE assumes this line is an **!INCLUDE** directive; otherwise, MAKE assumes that INCLUDE is a target or macro.

!CMDSWITCHES: [[+|–]*option*]...**:** Resets command switches from within the MAKE description file. The following switches can be reset: /d, /i, /n, and /s.

The string following **!CMDSWITCHES** specifies which switches should be turned on or off. Using **!CMDSWITCHES** alone on a line (for example, the argument string is null) restores the switch settings from the command that invoked MAKE. The argument string consists of + or –, followed by the letters of the switches to be changed.

The following example turns the debugging switch on, tells MAKE to ignore exit codes returned from programs it invokes, and then tells MAKE to start executing commands instead of just displaying them.

```
!CMDSWITCHES:     +di-n
```

The strings +di-n and +d+i-n are equivalent; space in between option letters is ignored. Only the switches specified in the argument string are affected, unless the argument string is null, which resets all switches to their original values.

Using **!CMDSWITCHES** on a line of a MAKE description file causes the new switch values to be in effect for all targets listed between that line and the next **!CMDSWITCHES** statement (or the end of the file if no other switch-setting statement is used). The MAKEFLAGS variable is updated accordingly whenever switch values change.

Using Macro Definitions

A macro definition associates a symbolic name with a particular value. Using macro definitions, you can change values in the description file without editing every line that uses a particular value.

Format

```
string1 = string2
```

The equal sign must not be preceded by a colon. Space following *string1* or preceding *string2* is stripped off. Note that *string2* may be null.

Invoke a macro by preceding the macro name with a $ sign. Macro names of length greater than 1 require parentheses around them, but single-character macros do not require parentheses. Once the macro is defined, *string2* replaces all subsequent appearances of $(*string1*) (or $*string1* if *string1* consists of only one character). Because macros are case-sensitive, *STRING1* and *string1* indicate two different macros.

Use a $$ sign to generate a $ sign within your MAKE description file without invoking a macro.

An undefined macro evaluates to the null string. Macros can be defined in the MAKE description file or at the command prompt, and they can be up to 64KB in length.

Change the value of a macro in your MAKE description file by redefining it. The new definition remains in effect until the macro is redefined again or until the end of the MAKE description file.

Example

This example shows a macro definition for the name *base* and its use in the description file. MAKE replaces each occurrence of $(*base*) with *abc*.

```
base=abc

$(base).obj:  $(base).c
 CC $(base),$(base),$(base),$(base)

$(base).exe:  $(base).obj \lib\math.lib
 LINK $(base),$(base),$(base) /map, \lib\math.lib;
```

Use the following command, to override the definition of *base* in the description file, causing *def* to be assembled and linked instead of *abc*:

```
MAKE "base=def" def.exe
```

Changing Macro Values

MAKE allows substitutions in macro invocations that allow you to change the value generated without changing the macro itself.

Format

name:string1 = string2

where *name* is the name of the macro whose value is being modified, *string1* is the character or characters to be modified, and *string2* is the replacement character or characters. As with the macro definition, space following *string1* or preceding *string2* is stripped off. If *string2* is NULL, *string1* is removed from *name*.

Example

The following macro:

```
FILES = file1.z file2.z file3.z
```

followed by this macro invocation:

```
$(FILES:.z=.c)
```

generates the value:

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

The actual value of FILES remains unchanged.

Using Special Macros

MAKE recognizes the following special macros:

$*: The target name with the extension deleted. For inference rules, this is the same as the filename part of the current dependent with the extension deleted, unless the target and dependent are in different directories). For example:

```
DIR\TARGET.EXE: DEPEND.OBJ
  ECHO $*
```

prints **.DIR\TARGET.**

$@: The full target name of the current target. When used in inference rules specifying a path for the target file, $@ evaluates to the target name with the specified path prepended. Use the $(@D) macro to derive the path given in the rule. For example, the following dependency block prints **DIR\TARGET.EXE.**

```
DIR\TARGET.EXE: DEPEND.OBJ
  ECHO $@
```

$:** The complete list of dependents. For example, the following dependency block prints **DEPEND.OBJ DIR\DEPEND2.OBJ**.

```
TARGET.EXE: DEPEND.OBJ DIR\DEPEND2.OBJ
  ECHO $**
```

$<: The dependent that is out of date with respect to the target (evaluated only for inference rules). For example, the following statements print **DIR\TARGET.DEP**.

```
.SUFFIXES: .TAR .DEP

{DIR}.DEP.TAR:
  ECHO $<

TARGET.TAR: DIR\TARGET.DEP
```

$?: The list of dependents that are out-of-date with respect to the target. When evaluating an inference rule, $? is equivalent to $<. For example, assume that DIR\DEPEND.OBJ has been updated since the last time TARGET.EXE was built. DIR\DEPEND2.OBJ has not been updated since then. The following dependency block prints **DIR\DEPEND.OBJ**.

```
TARGET.EXE: DIR\DEPEND.OBJ DIR\DEPEND2.OBJ
  ECHO $?
```

$$@: A dynamic dependency parameter referring to the current entry to the left of the colon (has meaning only on dependency lines). For example, the following dependency block:

```
TARGET.EXE TARGET2.EXE: DIR\$$@
  ECHO $?
```

prints the following:

```
DIR\TARGET.EXE
DIR\TARGET2.EXE
```

$(MAKE): Causes that line to be executed even if the /n flag is set. This macro has the default value MAKE and can be redefined if you want to execute some other program. For example, the following statements print **Prints anyway**, even if you use the /n option when starting MAKE.

```
MAKE=ECHO      # Redefine the special macro MAKE

TARGET.EXE: DEPEND.OBJ
  $(MAKE) Prints anyway
```

$(MAKEFLAGS): Holds the current input options. For example, if you start MAKE as follows:

```
MAKE /S /E ALL
```

then the following dependency block:

```
ALL:
  ECHO Current options: $(MAKEFLAGS)
```

prints **Current options: SE.**

$(CC): Predefined macro that is treated as if you had defined CC to equal CL. You can redefine this macro. For example, this command:

```
cl /c DEPEND,TARGET;
```

causes the following dependency block to compile the file DEPEND.C.

```
TARGET.OBJ: DEPEND.C
  $(CC) /c DEPEND.C
```

$(AS): Predefined macro that is treated as if you had defined AS to equal masm. You can redefine this macro. For example, the following command:

```
masm DEPEND,TARGET;
```

causes the following dependency block to assemble the file DEPEND.ASM.

```
TARGET.OBJ: DEPEND.ASM
  $(AS) DEPEND,TARGET;
```

The next examples are equivalent:

```
pgm.exe: mod1.obj mod2.obj mod3.obj
  link $**, $@

pgm.exe: mod1.obj mod2.obj mod3.obj
  link mod1.obj mod2.obj mod3.obj, pgm.exe
```

Changing the Meanings of Macros

You can change the meanings of the first six macros listed above by attaching the following:

D Causes the option to be the *directory* part of the filename. If there is no directory part, ".\" is used.

F Causes the option to be the *file* part.

B Causes the option to be the *base* portion of the filename (removes the directory and suffix).

R Gives the *root* of the filename (the directory part and the base part without the suffix).

Note: $(*B) is the same as $(*F), and $(*R) is the same as $*.

Parentheses must be used to enclose the macro when one of these suffixes is appended.

Example
The following example compiles or assembles each of the out-of-date files. By using the R with the $? macro, you can pass the filename without the extension to the compiler or assembler. Notice that the command prefix is used so MAKE will continue if an error occurs. (For example, when MASM attempts to assemble one of the C source files.)

```
target: file1.c  file2.c  file3.c  file4.asm  file5.asm file6.asm
  -!masm $(?R)
  -!cc $(?R);
```

Note that the preceding commands will try to compile file4.asm. Use the "—" before the "!" to turn off error checking.

Inference Rules
MAKE lets you create inference rules that specify commands for target descriptions even when there is no explicit command in the MAKE description file. An inference rule is a way of telling MAKE how to produce a file with one type of extension from a file with the same base name and a second type of extension.

The MAKE assignment order is:

1. Built-in rules (See "Built-In Inference Rules" on page 5-18 for more information)
2. TOOLS.INI definitions and rules (See "Specifying Macros and Inference Rules with TOOLS.INI" on page 5-21 for more information.)
3. Environment variables
4. Definitions and rules in the MAKE description file.
5. Command prompt definitions.

Invoking MAKE with the /e flag switches the order of 3 and 4.

Format

.dependent-ext.target-ext:
        command
        [*command*]

⋮

where *dependent-ext* is the extension of the dependent file and *target-ext* is the extension of the target file (for example, .obj.exe). *Command* is the action required to carry out the rule. More than one *command* can be given, but each must be listed on a separate line. The pair of suffixes tells MAKE how to make a file with the second suffix from a file having the first suffix.

You can specify directory names in braces before each suffix. The following example takes source files from one directory and updates object files in another:

```
{\usr\target\dir1}.c{\usr\dir2}.obj:
        cl -c $*.c
```

The paths can be null; for example,

```
{}.c{}.obj
```

and

```
{.}.c{.}.obj
```

and

```
.c.obj
```

are all equivalent. Only one directory can be specified in a path given in such rules. To instruct MAKE to take source files from many directories and place all the resulting files together in a directory, a separate rule must be given for each source directory.

Built-In Inference Rules

MAKE automatically assumes the following inference rules. They have the lowest precedence and are overruled by redefinitions in the TOOLS.INI file, in the MAKE description file, or at the command prompt. (See "Specifying Macros and Inference Rules with TOOLS.INI" on page 5-21 for more information about TOOLS.INI.)

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

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

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

The macros CFLAGS and AFLAGS are not defined and evaluate to null strings unless you define them.

Example
The following example specifies the filename in the rule with the special macro name **$***.

```
.C.OBJ:
   CC $*.C,,,;

TEST1.OBJ :  TEST1.C

TEST2.OBJ :  TEST2.C
   CC TEST2.C;
```

In the preceding description file, the first line redefines the built-in inference rule for creating .OBJ files from .C files. The rule applies to any base name. When MAKE encounters the dependency for files TEST1.OBJ and TEST1.C, it looks first for commands on the next line. When it does not find any, MAKE checks for a rule that may apply and finds the rule defined in the first lines of the description file. MAKE applies the rule, replacing the **$*** macro with TEST1 when it performs the command:

```
CC TEST1.C,,,;
```

When MAKE reaches the description block for the TEST2 files, it does not search for a dependency rule because a command is explicitly stated for this.

Special Rules
The following special pseudo-targets are available (note that these pseudo-targets, like targets, must be followed by a colon).

.SUFFIXES: Lists the file extensions that can be used in inference rules. If a file with a dependent extension is in the current directory and a relevant rule exists, MAKE automatically assumes a target-dependent statement for that file. (Subsequent occurrences of this pseudo-target followed by a list of extensions add to the existing dependency list. .SUFFIXES followed by an empty list of extensions clears the list.) The order of suffixes in the list tells MAKE in what order to examine inference rules. By default, .SUFFIXES is defined as ".obj .exe .c .asm."

SILENT: Same as the /s command option.

IGNORE: Same as the /i command option.

Generating Response Files

Some programs, such as the IBM Linker/2 and MAKE itself, allow you to use a response file to pass arguments rather than entering them at the command prompt. This makes it possible to pass a list of arguments that is longer than the DOS 128-character limit. To create a response file from within a MAKE description file, use the following:

Format

```
target:   [dependency-list]
    command @<<[filename]
text
...
<<
```

The first < < tells MAKE that what follows is the definition of a local input script. Every line between the first and second set of less-than symbols goes into the script file. MAKE expands macros that come between the first and second less-than symbols.

If you enter a filename after the first < <, MAKE uses that as the name of the input script file (the file may then be used as an input script for subsequent commands). If no filename is specified, MAKE creates a uniquely named file and stores the information there, deleting the file when it terminates.

Example

The following is an example of how to construct an input file for the LINK program

```
target.exe: file1.obj file2.obj file3.obj file4.obj
    link @<<file.lrf
file1+
file2+
file3+
file4
file.exe
file.map
lib1+lib2;
<<
```

MAKE creates a file named FILE.LRF and then executes the command

```
link @file.lrf
```

Specifying Macros and Inference Rules with TOOLS.INI

If you have a set of macros or inference rules that you use often, you can specify them through a file named TOOLS.INI rather than adding them to each of your MAKE description files.

TOOLS.INI is an initialization file you create to contain customization information for MAKE and other utility programs that use TOOLS.INI. TOOLS.INI can hold blocks of statements for different programs. Each block begins with a marker (the name of the program that the statements are intended for enclosed in brackets). This marker must begin in Column 1. The end of the block is marked by either the end of the file or the marker for another program.

When MAKE starts, it searches for TOOLS.INI in the current directory, then the directories specified by the INIT environment variable. If it finds TOOLS.INI and finds the MAKE marker within TOOLS.INI, it processes the block of statements before processing your makefile.

Notice that MAKE searches TOOLS.INI for a marker of the same name as its executable file. For example, if you rename MAKE.EXE to BUILDER.EXE, MAKE searches TOOLS.INI for the marker [BUILDER], not [MAKE].

Example

The following sample TOOLS.INI file adds the file extension .PAS to the .SUFFIXES list and defines an inference rule for building .OBJ files from Pascal files.

```
[make]

.SUFFIXES: .pas

.pas.obj:
        pas1 $(PFLAGS) $*.pas;
        pas2
```

LIB

The IBM Library Manager (LIB) is a utility that helps you create, organize, and maintain run-time libraries. Run-time libraries are collections of compiled or assembled functions that provide a common set of useful routines. Any program can call a run-time routine as though the program includes the function. When you link the program with a run-time library file, LIB finds the routine in the library file and resolves the call to the run-time routine.

You create run-time libraries by combining separately compiled object files into one library file. A .LIB extension usually identifies a library file, but you may use other extensions.

When you incorporate an object file in a library, the object file becomes an object module. LIB makes a distinction between object files and object modules; an *object file* is an independent file, but an *object module* is part of a larger library file.

An object file can have a full pathname (including a drive designation and a directory pathname) and a filename extension (usually .OBJ). Object modules have only a name. For example, B:\RUN\SORT.OBJ is an object filename, but SORT is the name of the corresponding object module.

Overview of LIB Operation

You can perform a number of library management functions with IBM LIB:

- Create a library file
- Delete modules
- Extract a module and place it in a separate object file
- Extract a module and delete it
- Add an object file to a library as a module, or add the contents to a library
- Replace a module in the library file with a new module
- Produce a listing of all public symbols in the library modules.

For each library session, LIB first reads and interprets your commands. It determines whether you are creating a new library or if you are examining or changing an existing library.

LIB processes deletion and extraction commands (if any) first. It does not 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.

Next, LIB processes any addition commands. Like deletions, it does not perform additions on the original library file. Instead, it appends the additional modules to the new library file. (If there were no deletion or extraction commands, LIB creates a new library file in the addition stage by copying the original library file.)

As LIB 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. The linker uses the library index to search the library.

The listing file contains a list of all public symbols in the index and the names of the modules they are defined in. LIB produces the listing file only if you ask for it during the library session.

LIB never makes changes to the original library; it copies the library and makes changes to the copy. When you end LIB for any reason, you do not lose your original file. It also means that when you run LIB, enough space must be available on your disk for both the original library file and the copy.

When you change a library file, LIB gives you the option of specifying a different name for the file containing the changes. If you use this option, LIB stores the changed library under the name you give, and preserves the original, unchanged version under its own name. If you choose not to give a new name, LIB gives the changed file the original library name but keeps a backup copy of the original library file. This copy has the extension .BAK instead of .LIB.

The LIB command is easy to use. Its syntax is straightforward, and it prompts you for responses. After you know how LIB works and what its prompts mean, you can use one of the alternate methods of calling LIB, described in "Starting LIB" on page 5-24. These alternative methods let you give LIB commands without waiting for the LIB prompts. A list of LIB error messages is in Appendix A.

Example

The following command deletes a library module named HEAP from the library file LANG.LIB, then adds a file named HEAP.OBJ as the last module in the library:

```
LIB LANG-HEAP+HEAP;
```

This command can also be entered this way, with the same effect:

```
LIB LANG+HEAP-HEAP;
```

This command always performs delete operations before add operations without regard to the order of operations at the command prompt. This order prevents LIB from flagging the operation as an

error when a new version of a module replaces an old version in the library file.

After a library is changed, the command writes the changed file back to the library file LANG.LIB. LANG.BAK is the name of the original backup file of LANG.LIB.

Starting LIB

You can start the LIB program by using one of the following methods:

The prompt method: Displays a prompt for each response it needs in the LIB program. See "Prompts for LIB" for information on how to use the prompt method. When you understand the LIB prompts and operations, you can use the command-prompt method of running LIB.

The command-prompt method: Lets you type all commands, options, and filenames on the line you use to start LIB. See "Command-Prompt Method for LIB" on page 5-27 for information on the command-prompt method.

The response file method: Lets you create a file that contains all the necessary commands, then tell LIB where to find that file. See "Response File for LIB" on page 5-29 for information on the response file method.

All of the above methods require that you understand how LIB works and what your responses to its prompts mean. For this reason, it is recommended that you allow LIB to prompt you for responses until you are comfortable with its commands and operations.

Prompts for LIB

You start LIB at the command prompt by typing **LIB**.

LIB prompts you for the input it needs by displaying the following prompts, one at a time. LIB waits for you to respond to a prompt before it displays the next one.

```
Library name:
Operations:
List file:
Output library:
```

The following sections explain the responses you can make to each prompt.

Library Name Prompt

At the Library name prompt, give the name of the library file you want. Library filenames usually have a .LIB extension. You can omit the extension when you give the library filename because LIB assumes an extension of .LIB. However, if your library file does not have the .LIB extension, include the extension when you give the library filename; otherwise, LIB cannot find the file.

The program allows pathnames with the library filename, so you can give LIB the pathname of a library file in another directory or on another disk.

Because LIB manages only one library file at a time, it allows only one filename in response to this prompt. There is no default response, so LIB produces an error message if you do not give a filename.

If you give the name of a library file that does not exist, LIB displays the prompt:

```
Library file does not exist.  Create?
```

Respond with y if you want to create the library file or n if you do not. If you answer n, LIB returns control to the prompt.

If you type a library filename and follow it immediately with a semicolon, LIB performs only a consistency check on the given library. A consistency check tells you whether all the modules in the library are in usable form. LIB prints a message only if it finds an incorrect object module; no message is displayed if all modules are intact.

You can also set the library page size following this prompt. See "Setting the Library Page Size" on page 5-34 for more information.

Operations Prompt

Following the Operations prompt, you can type one of the command symbols for manipulating modules (+, −, −+, *, −*) followed immediately by the module name or the object filename. You can specify more than one operation following this prompt, in any order. The default for the Operations prompt is no changes.

When you have a large number of modules or files to manipulate (more than can be typed on one line), type an ampersand (&) as the last symbol on the line, immediately before pressing Enter. The

ampersand must follow a filename; you cannot give an operator as the last character on a line you want to continue. The ampersand causes LIB to repeat the Operations prompt, allowing you to specify more operations and names.

The following list describes command symbols and their meanings and uses.

Symbol Meaning and Use

+ The plus sign adds an object file to the library file. Give the name of the object file immediately after the plus sign. You can use pathnames for object files. LIB supplies the .OBJ extension so you can omit the extension from the object filename.

  You can also use the plus sign to combine two libraries. When you give a library name after the plus sign, it adds a copy of the contents of the library to the library file it is changing. You must include the .LIB extension when you give a library filename. Otherwise, LIB uses the default .OBJ extension when it looks for the file.

– The minus sign deletes a module from the library file. Give the name of the module you want to delete immediately after the minus sign. A module name has no pathname and no extension.

–+ A minus sign followed by a plus sign replaces a module in the library. Insert the name of the module to be replaced after the replacement symbol. Module names have no pathnames and no extensions.

  To replace a module, LIB first deletes the specified module, then adds to the object file having the same name as the module. The command assumes the object file has an .OBJ extension and resides in the current working directory.

* An asterisk followed by a module name copies a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the .OBJ extension, the drive designation, and pathname of the current working directory to the module name to form a complete object filename. You cannot override the .OBJ extension, drive

designation, or pathname given to the object file, but you can later rename the file or copy it to another location.

–* A minus sign followed by an asterisk moves an object module from the library file to an object file. This operation is equivalent to copying the module to an object file, as described above, then deleting the module from the library.

List File Prompt

After the List file prompt, you can give a filename for a cross-reference listing file. You can specify a full pathname for the listing file to create it outside your current working directory. You can give the listing file any name and any extension. LIB does not supply a default if you omit the extension.

A cross-reference listing file contains two lists. The first is an alphabetic listing of all external (public) symbols in the library. The name of the module that the symbol name refers to comes after that symbol name.

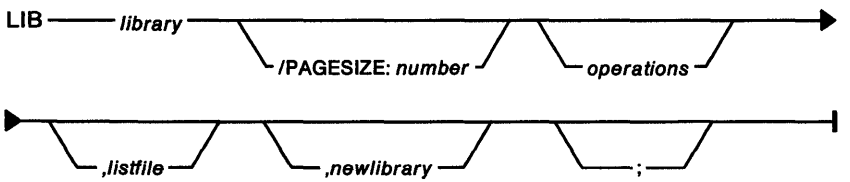
The second is a list of the modules in the library. Under each module name is an alphabetic listing of the public symbols defined in that module. The default when you omit the response to this prompt is the special filename NUL.LST, which tells LIB not to create a listing file.

Output Library Prompt

After the Output library prompt you can give the name of a new library file you want to create with the specified changes. The default is the current library filename; the original, unchanged library name remains the same, but the extension changes to .BAK, replacing the .LIB extension. This prompt is displayed only if you specify changes to the library following the Operations prompt.

Command-Prompt Method for LIB

The command-prompt method of starting LIB has the following form:



The entries following LIB are responses to the LIB command prompts.

library This parameter, with the optional /PAGESIZE:*number* specification, corresponds to the Library name prompt. If you want LIB to perform a consistency check on the library, follow the *library* entry with a semicolon.

operations These entries are any of the operations allowed following the Operations prompt.

If you want to create a cross-reference listing, you must separate the name of the listing file from the last *operations* entry by a comma. If you give a filename in the new library field, the library name must be separated from the listing filename or the last *operations* entry by a comma.

listfile If specified, LIB creates a listing file with the name.

newlibrary If specified, this is the name of the revised library.

You can use a semicolon after any entry to tell LIB to use the default responses for the remaining entries. The semicolon should be the last character at the command prompt.

Example

The following example instructs LIB to replace the HEAP module in the library LANG.LIB:

```
LIB LANG-+HEAP;
```

LIB first deletes the HEAP module in the library, then appends the object file HEAP.OBJ as a new module in the library. The semicolon command symbol at the end of the command tells LIB to use the default responses for the remaining prompts. It also tells LIB not to create a listing file and write the changes back to the original library file instead of creating a new library file.

The next example causes LIB to perform a consistency check of the library file C.LIB:

```
LIB C;
```

Does not perform any other action. LIB displays any consistency errors it finds and returns to the operating system level.

The last example tells LIB to perform a consistency check of the library file LANG.LIB and then to produce a cross-reference listing file named LCROSS.PUB:

```
LIB LANG,LCROSS.PUB;
```

Response File for LIB

The command to start LIB with a response file has the following form:

```
LIB @response-file
```

where *response-file* specifies the name of a response file. Qualify the *response-file* name with a drive and directory specification to name a response file from a directory other than the current working directory.

Before you use this method, you must set up a response file containing answers to the LIB prompts. This method lets you conduct the library session without typing responses at the keyboard.

A response file has one text line for each prompt. Responses must appear in the same order as the command prompts. Use command symbols in the response file the same way you type responses on the keyboard.

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 answers for all the prompts, LIB uses the default responses.

Example

```
SLIBC
+CURSOR+HEAP-HEAP*FOIBLES
CROSSLST
```

This response file causes LIB to delete the module HEAP from the SLIBC.LIB library file, extract the module FOIBLES and place it in an object file named FOIBLES.OBJ. It adds the object files CURSOR.OBJ and HEAP.OBJ as the last two modules in the library. Finally, LIB creates a cross-reference file named CROSSLST.

Extending Lines

If you have many operations to perform during a library session, use the ampersand command to extend the operations line. Type the ampersand symbol after an object module or object filename. Do not put the ampersand between an operations symbol and a name.

If you use the ampersand with the prompt method of calling LIB, the ampersand causes the Operations prompt to repeat, letting you type more operations. With the response file method, you can use the ampersand at the end of a line, then continue typing operations on the next line.

Ending the Library Session

At any time, you can use Ctrl+Break to end a library session. If you type an incorrect response, such as a wrong or incorrectly spelled filename or module name, you must press Ctrl+Break to leave LIB. You can then restart the program.

Selecting Default Responses to Prompts

After any entry but the first, use a single semicolon and then press Enter to select default responses to the remaining prompts. You can use the semicolon command symbol with the command prompt and response file methods of calling LIB. This is not necessary because LIB supplies the default responses wherever you omit responses.

The default response for the Operations prompt is no operation. The library file does not change.

The default response for the List file prompt is the special filename NUL.LST, which tells LIB not to create a listing file.

The default response for the Output library file is the current library name. This prompt appears only if you specify at least one operation following the Operations prompt.

Library Tasks

This section summarizes the library management tasks you can perform with LIB.

Creating a Library File

To create a new library file, type the name of the library file you want to create following the Library name prompt. LIB supplies the .LIB extension.

If the name of the new library is the name of an existing file, LIB assumes you want to modify the existing file. When you give the name of a library file that does not currently exist, LIB displays the following prompt:

```
Library file does not exist.  Create?
```

Type y (yes) to create the file or n (no) to end the library session.

Note: When you call LIB in such a way that no Operations prompt appears, the message above also does not appear. LIB assumes y (create the new library) by default. For example,

```
LIB new.lib+obj1;
```

Where *new.lib* does not exist, it creates the file NEW.LIB.

You can specify a page size for the library when you create it. The default page size is 16 bytes. See "Setting the Library Page Size" on page 5-34 for more information.

After you give the name of the new library file, you can insert object modules in the library by using the add operation (+) following the Operations prompt. You can also add the contents of another library. See "Adding Library Modules" on page 5-32 and "Combining Libraries" on page 5-33 for an explanation of these options.

Modifying a Library File

You can change an existing library file by giving the name of the library file following the Library name prompt. The Operations prompt performs all operations you specify on that library.

LIB lets you keep both the original library file and the newly changed version. You can do this by giving the name of a new library file following the Output library prompt. The library filename changes to the new library filename, while the original library file remains unchanged.

If you do not give a filename following the Output library prompt, the changed version of the library file replaces the original library file. LIB saves the original, unchanged library file. The original library file

has the extension .BAK instead of .LIB. At the end of the session, you have two library files: the changed version with the .LIB extension and the original, unchanged version with the .BAK extension.

Adding Library Modules
Use the plus sign following the Operations prompt to add an object module to a library. Give the name of the object file, without the .OBJ extension, that you want to add immediately after the plus sign.

LIB removes the drive designation and the extension from the object file specification, leaving only the filename. This becomes 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. LIB always adds object modules to the end of a library file.

Deleting Library Modules
Use the minus sign following the Operations prompt to delete an object module from a library. Give the name of the module you want to delete immediately after the minus sign. A module name has no pathname and no extension. It is only a name, such as CURSOR.

Replacing Library Modules
Use a minus sign followed by a plus sign to replace a module in the library. After the replacement symbol (−+), give the name of the module you want to replace. Module names have no pathnames and no extensions.

To replace a module, LIB deletes the given module and adds the object file with the same name as the module. The object file has an .OBJ extension and resides in the current working directory.

Extracting Library Modules
Use an asterisk followed by a module name to copy a module from the library file into an object file of the same name. The module remains in the library file. When LIB copies the module to an object file, it adds the .OBJ extension, the drive designation, and the pathname of the current working directory to the module name. This forms a complete object filename. You cannot override the .OBJ extension, drive designation, or pathname given to the object file. You can later rename the file or copy it to another location.

Use the minus sign followed by an asterisk (-*) 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.

Combining Libraries

You can add the contents of a library to another library by using the plus sign with a library filename instead of an object filename. Following the Operations prompt, give the plus sign and the name of the library with the contents you want to add to the library you are changing. When you use this option, you must include the .LIB extension of the library filename. Otherwise, LIB assumes that the file is an object file and looks for the file with an .OBJ extension.

LIB adds the modules of the library to the end of the library you are changing. The added library still exists as an independent library. LIB copies the modules without deleting them.

After you add the contents of a library or libraries, you can save the new, combined library under a new name by giving a new name following the Output library prompt. If you omit the Output library response, LIB saves the combined library under the name of the original library you are changing.

Creating a Cross-Reference Listing

Create a cross-reference listing by giving a name for the listing file following the List file prompt. If you omit the response to this prompt, LIB uses the special filename NUL.LST. It does not create a listing file.

You can give the listing file any name and any extension. You can specify a full pathname, including a drive designation, for the listing file to create it outside the current working directory. LIB does not supply a default if you omit the extension.

A cross-reference listing file contains two lists. The first is an alphabetic listing of all public symbols in the library. The name of the module the symbol name refers to comes after that symbol name.

The second list is an alphabetic list of the modules in the library. Under each module name is an alphabetic listing of the public symbols the module refers to.

Performing Consistency Checks

When you give only a library name followed by a semicolon at the Library name prompt, LIB performs a consistency check, displaying messages about any errors it finds. It does not make any changes to the library. This option is not usually necessary because LIB checks object files for consistency before adding them to the library.

To produce a cross-reference listing along with a consistency check, use the command-prompt method of calling LIB. Give the library name followed by a semicolon, then give the name of the listing file. LIB performs the consistency check and creates the cross-reference listing.

Setting the Library Page Size

The page size of a library affects the alignment of modules stored in the library. Modules in the library are aligned so that they always start at a position that is a multiple of *n* bytes from the beginning of the file. The value of *n* is the page size. The default page size is 16 for a new library or the current page size for an existing library.

Because of the indexing technique LIB uses, a library with a larger page size can hold more modules than a library with a smaller page size. However, for each module in the library, this indexing technique wastes an average of *n*/2 bytes of storage space (where *n* is the page size). In most cases a small page size is advantageous. You should use the smallest page size possible.

To set the library page size, add a page size option after the library filename in response to the Library name prompt:

library-name /PAGESIZE:*n*

The value of *n* is the new page size. It must be a power of 2 and be between 16 and 32768.

Another consequence of this indexing technique is that the page size determines the maximum possible size of the .LIB file. This limit is 65536 times *number*. For example, **/P:16** means that the .LIB file must be smaller than 1 megabyte (16 times 65536 bytes) in size.

EXEMOD

EXEMOD displays or changes fields in the DOS file header. To use this utility, you must understand the DOS conventions for file headers. They are explained in the technical reference information for DOS.

Some of the options available with EXEMOD are the same as the linker options, except that they work on files already linked. Unlike the linker options, the EXEMOD options require that you specify values in hexadecimal.

Note: EXEMOD uses only the DOS file header, so it is useful primarily for DOS executable files. EXEMOD can also be used to display or change fields in the DOS file header of a STUB program that can exist in an OS/2 executable file.

Displaying Current Status of Header Fields

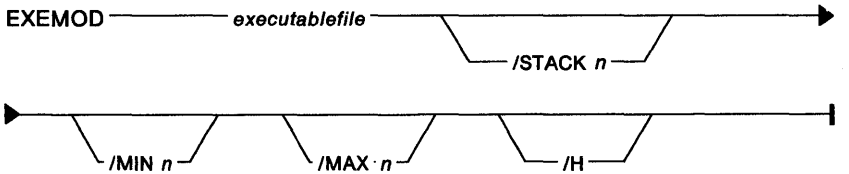
To display the current status of header fields, type:

```
EXEMOD executablefile
```

This command directs EXEMOD to display the current status of the header fields. See the example at the end of this chapter.

Changing Fields in the File Header

To change one or more fields in the file header, type:



This command directs EXEMOD to change one or more of the fields in the file header. EXEMOD expects the DOS executable file to be the name of an existing file with the .EXE extension. If you give the filename without an extension, EXEMOD adds .EXE and searches for that file. If you give a file with an extension other than .EXE, EXEMOD displays an error message.

Parameters

The examples in this chapter show parameters starting with a slash, but you can also use a dash to start a parameter. You can give EXEMOD parameters in either uppercase or lowercase, but you cannot abbreviate them.

EXEMOD parameters require that you give all values in hexadecimal. The available parameters and their meanings are:

/STACK *n* Sets the initial SP (stack pointer) value to *n*, where *n* is a hexadecimal value setting the number of bytes. EXEMOD adjusts the minimum allocation value upward, if necessary. This parameter has the same effect as the linker parameter /STACK.

/MIN *n* Sets the minimum allocation value to *n*, where *n* is a hexadecimal value setting the number of paragraphs. The actual value set may be different from the requested value if adjustments are necessary to accommodate the stack.

/MAX*n* Sets the maximum allocation to *n*, where *n* is a hexadecimal value setting the number of paragraphs. The maximum allocations value must be greater than or equal to the minimum allocation value. This option has the same effect as the linker parameter /CPARMAXALLOC.

/H Displays the current status of the DOS program header. Its effect is the same as entering EXEMOD with an executable file but with no parameters. Do not use the /H parameter with other parameters.

Note: The /STACK parameter can be used on programs assembled with the IBM Macro Assembler/2 or programs compiled with the IBM C Compiler, Version 1.00 or 1.10. Use of the /STACK parameter on programs developed with other compilers can cause the programs to fail or EXEMOD to return an error message.

Effect on Packed Files

EXEMOD can work on packed files. When it recognizes a packed file, it prints the following message:

```
exemod: (warning) packed file
```

It then continues to change the file header. When you load packed files, they are expanded to their unpacked state in storage. If you use the /STACK parameter on a packed file, the value changed is the value that the stack pointer (SP) is to have after expansion. If you use

either the /MIN or /STACK parameter, the value is corrected as necessary to accommodate unpacking of the modified stack. The /MAX parameter operates as it would for unpacked files.

If EXEMOD displays the header of a packed file, the CS:IP and SS:SP values appear as they will after expansion, which is not the same as the actual values in the header of the packed file.

Example

The first example shows the file header for the file named TEST.EXE. Suppose you enter the command:

```
EXEMOD test.exe
```

EXEMOD displays the following:

```
test.exe              (hex)    (dec)

.EXE size (bytes)     439D     17309
Minimum load size (bytes)    419D     16797
Overlay number                  0         0
Initial CS:IP         0403:0000
Initial SS:SP         0000:0000        0
Minimum allocation (para)       0         0
Maximum allocation (para)    FFFF     65535
Header size (para)             20        32
Relocation table offset        1E        30
Relocation entries              1         1
```

The following command shows how to change the header:

```
EXEMOD test.exe /STACK FF /MIN FF /MAX FFF
```

If you enter the same command as before, EXEMOD displays the file header values after the change.

```
EXEMOD test.exe
test.exe      (hex)      (dec)

.EXE size (bytes)     439D        17309
Minimum load size (bytes)    528D        20877
Overlay number                  0            0
Initial CS:IP          0403:0000
Initial SS:SP          0000:00FF  256
Minimum allocation (para)    FF          256
Maximum allocation (para)    FFF         4095
Header size (para)           20          32
Relocation table offset      1E          30
Relocation entries            1           1
```

Chapter 6. Interfacing with IBM Macro Assembler/2

This chapter explains how to use 8088 assembler language routines with C language programs and functions. It outlines the segment model that IBM C/2 uses and explains how to call assembler language routines from C language programs. It also explains how to call C programs from assembler language routines. This assembler language interface is especially useful for those assembler language programmers who want to use the functions of the standard C library and other C libraries.

Segment Model

This section describes the run-time structure of IBM C/2 programs. Storage for the 8088/80286/80386 series of processors is organized in segments, each containing up to 64KB.

The following list shows the order of segments of a C program in storage, from the highest storage location to the lowest. Because this is the default ordering for C programs, you do not need to use the segment order option with C programs, but it can be useful when linking assembler language routines. For more information about using the segment ordering option, see "Ordering Segments /DOSSEG" on page 3-22.

Note: A map file produced by linking a C program contains segments other than those listed below. Those additional segments have specialized uses for IBM languages and should not be used by other programs.

HEAP: The area of free storage available for dynamic allocation by the program. Its size varies, depending on the other storage requirements of the program.

STACK: Contains the stack that you use for all local data items.

_BSS segment: Contains all uninitialized static data items except those items explicitly declared as **far** or **huge** items in the source file.

c_common: Contains all uninitialized global data items for small- and medium-model programs. In large- and huge-model programs, the IBM C/2 places this type of data item in a data segment with class FAR_BSS.

CONST: Contains all constants that can only be read. These include floating-point constants and segment values for data items declared **far** in the source file or data items that are forced into their own segment by use of the /Gt option.

C lets you write to string literals. Thus, C programs store strings in the _DATA segment instead of the CONST segment.

_DATA: The default data segment. Initialized global and static data remain in this segment for all storage models, except for data explicitly declared **far** or for data forced into different segments by use of the /Gt option. For more information about the /Gt option, see "Setting the Data Threshold /Gt" on page 2-57.

NULL: A special-purpose segment that occurs at the beginning of DGROUP. The NULL segment contains the compiler copyright notice. The system checks this segment before and after the program runs. If the contents of the NULL segment change in the course of the program run, the program has written to this area, usually by an inadvertent assignment through a null pointer. You are notified of this by the error message **Null pointer assignment**.

Far Data Segments: IBM C/2 places initialized static and global **far** or **huge** data items in their own segments with class name FAR_DATA. This lets the linker combine these items so that they all come before DGROUP. Uninitialized static and global **far** or **huge** data items are placed in segments that have class FAR_BSS. This lets the linker place these items between the TEXT segment(s) and DGROUP. In large- and huge-model programs, the compiler treats global uninitialized data as if it were declared **far** or **huge** (unless specifically declared **near**) and given class FAR_BSS.

_TEXT: The code segment. In small- and compact-model programs, the linker combines the code for all modules in this segment. In medium-, large-, and huge-model programs, each module has its own reserved text segment. The linker does not combine segments; there are multiple text segments in medium- and large-model programs.

Each segment in a medium-, large-, or huge-model program has the name of the module plus the suffix _TEXT.

When implementing an assembler language routine to call or be called from a C program, you refer to the _TEXT and _DATA segments most frequently. Place the code for the assembler language routine in the _TEXT segment (or *module-name*_TEXT for medium-, large-, and huge-model programs). Place data in whichever segment is appropriate to its use, as described previously. Usually this segment is the default data segment, _DATA.

Groups

Segments with the same group name must fit into a single physical segment, which is up to 64KB long. This permits access to all segments in a group through the same segment register. IBM C/2 defines one group named DGROUP.

The NULL, _DATA, CONST, _BSS, c_common, and STACK segments are together in DGROUP. This lets the compiler produce code to get access to data in each of these segments without constantly loading the segment values or using many segment overrides on instructions. Address DGROUP using the DS or SS segment register. DS and SS always contain the same value.

Compact-, large- and huge-model programs, or small- and medium-model programs using **far** or **huge** data declarations, can change DS temporarily to a different value to let the program get access to data outside the default data segment. You can also use the ES register.

The stack segment (SS) is never changed; its segment registers always contain abstract segment values, and the contents are never examined or operated on. This provides compatibility with the 80286 processor.

In small- and compact-model programs, only one text segment is named _TEXT. In medium-and large-model programs, the names of all text segments must end with the suffix _TEXT. The text segments are not grouped.

Memory Model	Segment Name	Align Type	Combine Class	Class Name	Group
Small	_TEXT	byte	public	CODE	
	Data Segments[00]	para	private	FAR_DATA	
	Data Segments[00]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
Medium	*module*_TEXT	byte	public	CODE	
	Data Segments[00]	para	private	FAR_DATA	
	Data Segments[00]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
Compact	_TEXT	byte	public	CODE	
	Data segments[00]	para	private	FAR_DATA	
	Data segments[00]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP
Large **Huge**	*module*_TEXT	byte	public	CODE	
	.				
	.				
	.				
	Data Segments[00]	para	private	FAR_DATA	
	Data Segments[00]	para	public	FAR_BSS	
	NULL	para	public	BEGDATA	DGROUP
	_DATA	word	public	DATA	DGROUP
	CONST	word	public	CONST	DGROUP
	_BSS	word	public	BSS	DGROUP
	STACK	para	stack	STACK	DGROUP

The C Calling Sequence

To receive values from C language function calls or to pass values to C functions, assembler language routines must follow the C argument-passing conventions. In the C language, function calls pass their arguments to the given functions by pushing the value of each argument onto the stack. The call pushes the value of the last argument first and the first argument last. If an argument is an expression, the call computes the value of the expression before pushing it onto the stack.

Arguments with **char**, **short**, **int**, **unsigned char**, **unsigned short**, or **unsigned int** type occupy a single word (16 bits) on the stack. Arguments with **long** or **unsigned long** type occupy a double word (32 bits); the value's high-order word is pushed first.

The compiler can pass float arguments only when a function prototype specifying float is present. Otherwise, IBM C/2 converts arguments with **float** type to **double** type (64 bits). The **char** type arguments are sign-extended to **int** type before being pushed on the stack; **unsigned char** type arguments are zero-extended to **unsigned int** type.

Pointers occupy either 16 or 32 bits, depending on the storage model, the type of item addressed (code or data), and whether the pointer is changed with a **near** or **far** declaration. The segment value of far pointers is pushed first, then the offset.

If an argument is a structure, the function call pushes the last word of the structure first and each successive word in turn until the first word is pushed. C programs pass arrays by reference; the system evaluates the array identifier as the array address, which it uses to get access to the array.

After a function returns control to a routine, the calling routine is responsible for removing arguments from the stack.

Entering an Assembler Routine

Assembler language routines that receive control from C function calls should preserve the contents of the BP, SI, and DI registers and set the BP register to the current SP register value before proceeding with their tasks. You need not preserve the contents of the SI and DI registers if the assembler language routine does not change them.

If the assembler routine changes the contents of the SS (stack segment), DS (data segment), and CS (code segment) registers, it should save their values on entry and restore them at exit. The values of the SS and DS registers are equal in C programs, unless you specify the **u** or **w** flag of the **/A** option to set up separate stack and data segments.

The following example shows the recommended instruction sequence for entry to an assembler language routine.

```
entry:
        push    bp
        mov     bp,sp
        push    di
        push    si
```

This is the same sequence that IBM C/2 uses.

If you use this sequence, the last argument pushed by the function call, which is also the first argument given in the argument list of the call, is at address [bp+4] for a **near** function call or [bp+6] for a **far** function call. Subsequent arguments begin at [bp+6], [bp+8], or [bp+10], depending upon the size of the first argument and whether the function call is **near** or **far**. If the first argument is a single word and the function call is **near** the next argument starts at [bp+6]. If the first argument is a single word and the function call is **far**, or if the first argument is a double word and the function call is **near**, the next argument starts at [bp+8]. If the first argument is a double word and the function call is **far**, the next argument starts at [bp+10].

The last two **push** instructions in the above sequence are not necessary if the assembler language routine does not change the contents of the SI and DI registers, which the compiler uses to store **register** variables.

Note: It is recommended that you write macros to distinguish between **near** and **far** function calls and returns. Such macros

make the code more readable and can help to insulate a program from changes in the calling sequence.

Return Values

Assembler language routines that return values to a C language program or receive return values from C functions must follow the C return value conventions. The conventions are shown in the following table.

Return Value Type	Register
char	AX
short	AX
int	AX
unsigned char	AX
unsigned short	AX
unsigned int	AX
long	High-order word in DX; low-order word in AX
unsigned long	High-order word in DX; low-order word in AX
struct or **union**	Address of value in AX; value must be in a static area in storage
float or **double**	Address of value in AX; value must be in a static area in storage
near pointer	AX
far pointer	Segment selector in DX; offset in AX

Exiting from a Routine

Assembler language routines that return control to C programs should restore the values of the BP, SI, and DI registers before returning control. You need not restore the contents of the SI and DI registers if the entry sequence did not push them. The following example illustrates the recommended instruction sequence for exiting from a routine called by a small-model program.

```
pop     si
pop     di
mov     sp,bp
pop     bp
ret
```

This sequence does not change the AX, BX, CX, or DX registers or any of the segment registers. The sequence does not remove arguments from the stack; that is the responsibility of the calling routine.

The pop instructions for SI and DI in the above sequence are not necessary if the assembler language routine does not change the contents of the SI and DI registers and does not save them on entry.

Naming Conventions

An assembler language routine can access globally visible items (data or functions) in a C program by prefixing the item name with an underscore. Assembler language routines cannot access C items declared as static. For example, you can get access to a C function named **add** in an assembler language program by declaring the name **_add** as external.

For a C program to access an assembler language routine or data item, the name of the assembler language item must begin with an underscore. The C program refers to the assembler language item without the underscore. For example, a C program can call a publicly defined assembler language routine named **_mix** by the following declaration.

```
extern mix();
```

However, if the name of the assembler language routine does not begin with an underscore, the C program cannot get access to it.

IBM C/2 reserves some identifiers beginning with two underscores for internal use. Avoid using identifiers with two leading underscores in your assembler routines and identifiers with one leading underscore in your C source files; these identifiers might conflict with internal names.

If you are linking C modules with modules created by IBM Macro Assembler/2 (MASM), either assemble the MASM modules with the /MX option to preserve case sensitivity in these modules, or use the LINK command to link in a separate step. Do not specify the /NOI linker option.

Register Considerations

The SI and DI registers store the values of variables given register storage in a C program. An assembler language routine that changes the SI and DI registers must save the contents of these registers on entry and restore them before exiting.

IBM C/2 assumes that the assembler language routine clears the direction flag. If your assembler routine sets the direction flag, be sure to clear it, using the CLD instruction, before returning.

If the assembler routine changes the contents of the SS (stack segment), DS (data segment), and CS (code segment) registers, it should save their values on entry and restore them at exit. The values of SS and DS are equal in C programs.

Examples

The following example shows the assembler language interfaces. The example assumes that the C program is a small-model program.

```
int a=1, b=1, c;

main()
{
        c = add(a,b);
}

add(i,j)
int i,j;
{
        return(i+j);
}
```

If you write the **add** function as an assembler language routine instead of a C function, the routine must save the proper registers, retrieve the arguments from the stack, add the arguments, place the return value in the AX register, restore the registers, and return control. Here is a sample portion of how to write this routine. Preserving and restoring SI and DI is shown for illustration, although the procedure is not strictly necessary in this case. If the assembler routine is written to work with a medium-, compact-, or large-model C program, the **_add** procedure is declared **far** instead of **near**.

```
; i = [bp + 4]
; j = [bp + 6]

_add    PROC NEAR
        push    bp
        mov     bp,sp
        push    di
        push    si

        mov     ax,[bp+4]
        add     ax,[bp+6]

        pop     si
        pop     di
        mov     sp,bp
        pop     bp
        ret

_add    endP
```

On the other hand, if the C function is called by an assembler language routine, the routine must contain instructions that push the arguments onto the stack in the proper order, call the function, and clear the stack. It may then use the return value in the AX register. These instructions are shown in the following example:

```
push    [_b]
push    [_a]
call    _add
add     sp,4
mov     [_c],ax
```

Appendix A. Error Messages

The C/2 compiler produces a broad range of error and warning messages to help locate errors and potential problems in programs.

This appendix lists the error messages you might find as you develop a program and gives a brief description of the action required to correct the errors.

Run-time errors that may occur when you run your program are discussed first. The remaining sections describe error messages produced by the following programs:

- IBM C/2
- The IBM Linker
- The IBM LIB library management utility
- The EXEMOD header modification utility
- The MAKE program maintenance utility
- The *errno* variable.

Run-Time Error Messages

Run-time error messages are divided into four categories:

1. Error messages generated by the system to notify you of serious errors.
2. Floating-point exceptions generated by the Numeric Coprocessor or the NPX emulator.
3. Error messages generated by calls in the program to error-handling routines in the C run-time library.
4. Error messages generated by calls to math routines in the C run-time library. On an error, the math routines return an error value or print a message to the standard error data stream. See Chapter 5 in *IBM C/2 Language Reference* for a description of the math routines.

System Generated Error Messages

Programs with serious errors can cause the system to generate the following messages at run time:

Number	Message, Cause/Action
R6000:	**stack overflow** Program ran out of stack space. This can occur when a program uses a large amount of local data or is heavily recursive. The system stops the program with an exit status of 255. To correct the problem, recompile using the /F option of the CL command, or relink using the linker /STACK option to reserve a large stack or modify the stack information in the executable file header by using the EXEMOD program.
R6001:	**null pointer assignment** The contents of the NULL segment changed as the program ran. The NULL segment is a special location in low storage that is not normally used. If the contents of the NULL segment change during the running of a program, the program has written to this area, usually by an inadvertent assignment through a null pointer. Your program can contain null pointers without generating this message; the message appears only when you get access to a storage location through the null pointer. This error does not cause your program to stop; the system prints the error message following the normal end of the program. This message reflects a potentially serious error in the program. Although a program that produces this error can appear to run correctly, it is likely to cause problems in the future and might fail to run in a different operating environment.

Number	Message, Cause/Action
R6002:	**floating point not loaded** Program needs the floating-point library, but that library was not loaded. This error stops the program with an exit status of 255. This error occurs in three situations: 1. A format string for one of the routines in the **printf** or **scanf** family contains a floating-point format specification, and there are no floating-point values or variables in the program. The C compiler tries to minimize the size of the program by loading floating-point support only when necessary. It does not detect floating-point format specifications within format strings and, consequently, does not load the necessary floating-point routines. To correct this error, use a floating-point argument that corresponds to the floating-point format specification. This causes the C compiler to load floating-point support. 2. You specified *x*LIBFP.LIB or *x*LIBFA.LIB (where *x* is S, M, L, C, or H, depending on the storage model) after *x*LIBC.LIB in the linking stage. You must relink the program with the correct library specification. 3. The program uses floating point and is compiled and linked with options that require a numeric coprocessor (-FPi87, for example) but is run on a machine that does not have a numeric coprocessor. You should either recompile with the /FPi option, relink with emulator library EM.LIB, or install a coprocessor.
R6003:	**integer divide by 0** An attempt was made to divide an integer by 0, giving an undefined result.
R6005:	**not enough memory on exec**
R6006:	**bad format on exec**

Number	Message, Cause/Action
R6007:	**bad environment on exec** Errors R6005 through R6007 occur when a child process spawned by one of the exec library routines fails, and DOS was unable to return control to the parent process.
R6008:	**not enough space for arguments** See explanation under error R6009.
R6009:	**not enough space for environment** Error R6008 and R6009 both occur at start-up if there is enough memory to load the program but not enough room for the argv and/or envp vectors. To avoid this problem, you can rewrite the _setargv or _setenvp routines.

Floating-Point Exceptions

The error messages listed below correspond to exceptions produced by the numeric coprocessor. Refer to the Intel documentation for your processor for a detailed discussion of hardware exceptions.

When you use the default floating-point control word settings in C, 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 that IBM C/2 produces or code provided in the IBM C/2 run-time library:

- Square root
- Stack underflow
- Unemulated.

The floating-point exceptions have this format:

```
runtime-time error M61xx : MATH-floating-point error:  message text
```

The following list describes the floating-point exceptions:

Number	Message, Cause/Action
M6101:	**invalid** The operation is a non-valid operation. Usually this message appears when an operation tries to operate on NaNs or infinities.
M6102:	**denormal** The operation produced a very small floating-point number, which might no longer be correct due to loss of significance. Denormals are normally masked, causing them to be trapped and operated on.
M6103:	**divide by 0** The operation tried to divide by 0.
M6104:	**overflow** The operation produced an overflow in floating-point operation.
M6105:	**underflow** The operation produced an underflow in a floating-point operation. An underflow is normally masked so that the operation yields the result 0.0.
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.
M6107:	**unemulated** An attempt was made to run a floating-point instruction not supported by the emulator or a non-valid floating point instruction.
M6108:	**square root** The operand in a square root operation was negative. **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 chapter on Library Routines in *IBM C/2 Language Reference* the for details on **sqrt**.

Number	Message, Cause/Action
M6110:	**stack overflow** A floating-point expression has used too many stack levels on the numeric coprocessor or emulator. (Stack overflow exceptions are trapped up to a limit of seven additional levels beyond the eight levels normally supported by the numeric coprocessor.)
M6111:	**stack underflow** A floating-point operation resulted in a stack underflow on the numeric coprocessor or the emulator.

Error-Handling Routine Error Messages

The **abort**, **assert**, and **perror** routines print an error message to the standard error data stream (stderr) whenever the program calls the given routine. For a description of these routines, see Chapter 5 in the *IBM C/2 Language Reference*.

Math Errors

The following errors can be generated by the math routines of the C run-time library. These errors correspond to the exception types defined in **math.h** and returned by the **matherr** function when a math error occurs. See the chapter on Include Files in the *IBM C/2 Language Reference* for more information.

Error	Description
DOMAIN	An argument to the function is outside the domain of the function.
OVERFLOW	The result is too large to be represented in the return type of the function.
PLOSS	A partial loss of significance occurred.
SING	Argument singularity: an argument to the function has an illegal value (for example, passing the value 0 to a function that requires a nonzero value).
TLOSS	A total loss of significance occurred.
UNDERFLOW	The result is too small to be represented.

Compiler Error Messages

The C/2 compiler produces a broad range of error and warning messages to help you locate errors and potential problems in programs. Error messages produced by the compiler are sent to the standard output, that is usually your screen. You can redirect the messages to a file or printer by using a DOS redirection symbol, > or >>. This is especially useful in batch file processing. See "Redirecting Compiler Error Messages" on page A-54 for more information.

The error messages produced by IBM C/2 fall into five categories:

Fatal error messages (See "Fatal Error Messages" on page A-9) indicate severe problems, those that prevent the compiler from processing your program. After printing out a message about the fatal error, the compiler stops without producing an object file or checking for further errors.

Error messages during compiling (See "Error Messages During Compiling" on page A-16) identify actual program errors. No object file is produced for a source file that has such errors. When the compiler finds a nonfatal program error, it tries to recover from the error. If possible, the compiler continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Warning messages (See "Warning Error Messages" on page A-34) are informational only; they do not prevent compiling and linking. The list of warning messages includes a number for each message indicating the minimum level that must be set for the message to appear. You can control the level of warnings generated by the compiler by using the /W option. See "Setting the Warning Level /W /w" on page 2-37 for more information.

Command area error messages (See "Command Area Error Messages" on page A-48) give you information about non-valid or inconsistent command options. If possible, the compiler continues operation, printing a warning message to indicate which command options are in effect and which are disregarded. In some cases, command errors are fatal, and the compiler stops processing.

Compiler internal error messages (See "Compiler Internal Error Messages" on page A-53) indicate errors on the part of the compiler

instead of an error in your program. No matter what your source program contains, these messages should not appear. If they do, please report the condition to an IBM Authorized Dealer. Although these errors are not the fault of your program, you will probably want to rearrange your code so that the program can be compiled.

Error messages in the warning, fatal, and compiling error message categories have the same basic form:

filename (*linenumber*) :*msg-code error-number message*

The parts of the error message are as follows:

filename The name of the source file being compiled.

linenumber The line of the file containing the error.

msg-code The message code consists of two parts:

1. An initial letter that identifies the component that is reporting the error.
2. A single digit following the letter indicates the severity of the error.

The form of the message code with a number is:

<letter> <number> <###>

Letter	Error Type
C	C Compiler
D	CL/CC driver
M	Math run-time errors
R	General run-time errors

Number	Error Type
1	Fatal Error
2	Error
4	Warning
6	Run-time

The <###> is the 3-digit error number within the category.

error-number The number associated with the error.

message A self-explanatory description of the error or warning. A command error message gives a message about the command; it does not contain references to line numbers and filenames.

The messages for each category follow in numeric order, along with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number.

Fatal Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it stops after printing the error message.

Number	Message, Cause/Action
C1001:	**Internal Compiler Error** (**compiler file** '*name*',**line** *n*) The compiler has detected an internal error. Please report this error to an IBM Authorized Dealer. Include the compiler filename and line number information.
C1002:	**out of heap space** The compiler has run out of dynamic storage space. This usually means that your program has many symbols and complex expressions. To correct the problem, break the file into several smaller source files.
C1003:	**error count exceeds** *n*; **stopping compilation** Errors in the program are too numerous or too severe to allow recovery, and the compiler stops. Correct the other errors and recompile the program.
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.
C1006:	**write error on compiler intermediate file** The compiler is unable to create the intermediate files used in the compiling process. The exact reason is unknown.
C1007:	**unrecognized flag** '*string*' in'*option*' The given *string* in the command *option* is not valid. Specify a valid *string*.

Number	Message, Cause/Action
C1009:	**compiler limit : macros too deeply nested** The expansion of a macro exceeds the available space. Check whether 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 are not matched. The *character* is either (or).
C1013:	**cannot open source file '*filename*'** The given source file cannot be opened. Make sure you have given the correct pathname for the file. The system may have run out of file handles; the line FILES=20 should be in your CONFIG.SYS file.
C1014:	**too many include files** Nesting of #include directives exceeds the limit of 10 levels. Restructure your source files so that the #include directives are not so deeply nested.
C1015:	**cannot open include file '*filename*'** The given file cannot be opened. Make sure the include environment variable is correct. The system may have run out of file handles; the line FILES=20 should be in your CONFIG.SYS file. If include files are shared, they should be read-only.
C1016:	**#if[n]def expected an identifier** Specify an identifier with the #ifdef and #ifndef directives.
C1017:	**incorrect 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 directive. Correct the structure of your conditional preprocessing directives.

Number	Message, Cause/Action
C1019:	**unexpected '#else'** The #else directive is legal only when it appears within an #if, #ifdef, or #ifndef directive. Correct the structure of your conditional preprocessing directives.
C1020:	**unexpected '#endif'** An #endif directive appears without a matching #if, #ifdef, or #ifndef directive. Correct the structure of your conditional preprocessing directives.
C1021:	**bad preprocessor command '*string*'** The characters following the number sign (#) do not form a preprocessor directive.
C1022:	**expected '#endif'** An #if, #ifdef, or #ifndef directive does not end with an #endif directive.
C1026:	**parser stack overflow, please simplify your program** Your program cannot be processed because the space required to parse the program causes a stack overflow in the compiler. To solve this problem, simplify your program.
C1027:	**DGROUP data allocation exceeds 64KB** Large, compact, or huge model allocation of variables to the default segment exceeds 64KB. Use the /Gt option to move items into separate segments.
C1032:	**cannot open object listing file '*filename*'** One of the following statements about the *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.
C1033:	**cannot open assembly language output file '*filename*'** One of the conditions listed under C1032 prevents *filename* from being opened.

Number	Message, Cause/Action
C1034:	**cannot open source file '*filename*'** The filename or pathname given for the source file is not valid.
C1035:	**expression too complex, please simplify** The compiler cannot produce code for a complex expression. Break the expression into simpler subexpressions and recompile.
C1036:	**cannot open source-listing file '*filename*'** One of the conditions listed under C1032 prevents *filename* from being opened.
C1037:	**cannot open object file '*filename*'** One of the conditions listed under C1032 prevents *filename* from being opened.
C1039:	**unrecoverable heap overflow in Pass 3** The post-optimizer compiler pass has overflowed the heap and cannot continue. Try recompiling with the /Od option or breaking up the function containing the line causing the error.
C1040:	**unexpected EOF in source file '*filename*'** The compiler detected an unexpected end-of-file while creating a source listing or mingled a source/object listing. The probable cause is a source file edited during compiling. This error most likely occurs on a multitasking system where the compiling can be done as a background process.
C1041:	**cannot open compiler intermediate file - no more files** The compiler is unable to create intermediate files used in the compiling process because no more file handles are available. This can usually be corrected by changing the files = line in the CONFIG.SYS file to allow a larger number of open files (20 is the recommended setting).
C1042:	**cannot open compiler intermediate file - no such file or directory** The compiler is unable to create intermediate files used in the compiling process because the TMP environment variable is set to a non-valid directory or path. Correct the SET TMP=*pathname* command.

Number	Message, Cause/Action
C1043:	**cannot open compiler intermediate file** The compiler is unable to create intermediate files used in the compiling process. The exact reason is unknown.
C1044:	**out of disk space for compiler intermediate file** The compiler is unable to create intermediate files used in the compiling process because no more space is available. To correct the problem, make more space available on the disk and recompile.
C1045:	**floating-point overflow** The compiler has produced 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 case, the double-precision constant 1.0e100 exceeds the maximum allowable value for a floating-point data item. Be sure that each floating-point constant you write is within the limits of the type it is assigned to.
C1047:	**too many** *option* **flags,** '*string*' There are too many occurrences of the given *option*; *string* contains the occurrence of the *option* causing the error.
C1048:	**Unknown option** '*character*' **in** '*optionstring*' The specified *character* is not a valid letter for *optionstring*. Change the option specification to a valid letter.
C1049:	**Incorrect numerical argument** '*string*' A numerical argument was expected instead of *string*. Add the correct number in this argument.
C1050:	'*segname*'**: code segment too large** The code generated for the given segment exceeded 64KB. You must reduce the number of instructions compiled for this code segment.
C1051:	**program too complex** Simplify your program.

Number	Message, Cause/Action
C1052:	**too many #if/#ifdefs** Your #if or #ifdef directives are nested more than 32 levels deep. Restructure the conditional preprocessing directives to reduce the degree of nesting.
C1053:	**compiler limit : struct/union nesting** Nesting of structure and union definitions are limited to 10 levels. Change the structure or union definition to reduce the degree of nesting.
C1054:	**compiler limit : initializers too deeply nested** The compiler limit on nesting of initializers has been exceeded. The limit ranges from 10 through 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.
C1056:	**compiler limit : out of macro expansion space** The expansion of a macro (often nested macros and large actual parameters) has used up the available space in the macro expansion buffer.
C1057:	**unexpected EOF in macro expansion** The preprocessor encountered an end-of-file while collecting the actual arguments for a macro expansion. This is usually caused by a missing parenthesis to close the macro argument list.

C1058:	**floating point expression too complex, would overflow NDP stack**
	A floating-point expression was too complex for the compiler to handle, as in the following example:
	`double f(a, b, c, d, e, f, g, h, i, j)`
	`double a, b, c, d, e, f, g, h, i, j;`
	`{`
	`return ( a / ( b / ( c / ( d /`
	`( e / ( f / ( g / ( h /`
	`( i / j ) ) ) ) ) ) ) ) );`
	`}`
	Try breaking up the expression that caused the error and recompiling.

Number	Message, Cause/Action
C1059:	**out of near heap space** Program too large (too many symbols), and the compiler cannot allocate space in the near heap. Try breaking the program into smaller source modules.
C1060:	**out of far heap space** Program too large (too many symbols) and the compiler cannot allocate space in the far heap. Try removing other memory resident programs to create extra memory space. If your machine is a network server, you could reconfigure it so that it does not use network software during compiling. An alternate form of compiler pass 1 named C1L.EXE is provided for compiling programs that get "out of near/far heap space" errors. To invoke C1L.EXE, enter the CL command with the /B1 *path* option as illustrated below: `cl /B1 path\cl1.exe sourcefile.c` where *path* is the path (including drive and directory) where C1L.EXE resides and *sourcefile* is the name of the C source file you are compiling. During installation, C1L.EXE copies to C:\IBMC2\BIN, the default directory.
C1062:	**error writing to preprocessor output file** A -P, -E, or -EP option was entered to create a preprocessor listing file. However, there is no available space on the output directory. Clear some disk space before retrying.
C1064:	**too many text segments** Your program contains too many text segments. Try breaking it up into smaller modules.
C1065:	**compiler limit: declarator too complex** Occurs when you compile with the Zg option and a function definition would generate a prototype too large to hold an internal buffer.
C1067:	**compiler limit: identifier overflowed internal buffer** The object name is too long; use a shorter, unique name.

Number	Message, Cause/Action
C1126:	'*identifier*': **automatic allocation exceeds** *size* The space allocated for the local variables of a function exceeds the given limit.

Error Messages During Compiling

Receiving any of the following messages during compiling indicates errors in the program. When the compiler finds any of these errors, it continues passing the program, if possible, and puts out additional error messages. However, no object file is produced.

Number	Message, Cause/Action
C2000:	**UNKNOWN ERROR** The compiler has detected an unforeseen error condition. Please report this error to an IBM Authorized Dealer.
C2001:	**newline in constant** A newline character in a character or string constant must be preceded by the backslash escape character (\).
C2002:	**out of macro actual parameter space** Arguments to preprocessor macros cannot exceed 256 bytes.
C2003:	**expected 'defined id'** An #if directive has a syntax error.
C2004:	**expected 'defined(id)'** An #if directive has a syntax error.
C2005:	**#line expected a line number, found 'string'** A #line directive lacks the mandatory line number specification.
C2006:	**#include expected a filename, found 'string'** An #include directive lacks the mandatory filename specification.
C2007:	**#define syntax** A #define directive has a syntax error.
C2008:	'*c*' **: unexpected in macro definition** The character *c* is misused in a macro definition.

Number	Message, Cause/Action
C2009:	**reuse of macro formal '*identifier*'** The parameter list in a macro definition contains two occurrences of the same identifier. You should list each unique parameter exactly once.
C2010:	**'*c*' : unexpected in formal list** The character *c* is misused in the list of formal parameters for a macro definition.
C2011:	**'*identifier*' : definition too big** Macro definitions cannot exceed 512 bytes.
C2012:	**missing name following '<'** An #include directive lacks the mandatory filename specification.
C2013:	**missing '>'** The closing greater-than '>' is missing from an #include directive.
C2014:	**preprocessor command must start as first non-whitespace** Non-white-space characters appear before the # sign of a preprocessor directive on the same line.
C2015:	**too many chars in constant** A character constant is limited to a single character or escape sequence. (Multi-character character constants are not supported.)
C2016:	**no closing single quote** Backslash escape character (\) must precede a newline character in a character constant.
C2017:	**illegal escape sequence** The characters after the escape character (\) do not form a valid escape sequence. Use a number corresponding to a valid character as tabulated in Chapter 3 of *IBM C/2 Fundamentals*.
C2018:	**unknown character '0x*n*'** The given hexadecimal number does not correspond to a character. Use a number corresponding to a valid character as tabulated in Chapter 3 of *IBM C/2 Fundamentals*.

Number	Message, Cause/Action
C2019:	**expected preprocessor command, found '*c*'** The character following a # sign is not the first letter of a preprocessor directive.
C2020:	**bad octal number '*n*'.** The character *n* is not a valid octal digit.
C2021:	**expected exponent value, not '*n*'** The exponent of a floating-point constant is not a valid number.
C2022:	**'*n*' : too big for char** The number *n* is too large to be represented as a character.
C2023:	**divide by 0** The second operand in a division operation (/) evaluates to 0, giving undefined results.
C2024:	**mod by 0** The second operand in a remainder operation (%) evaluates to 0, giving undefined results.
C2025:	**'*identifier*' : enum/struct/union type redefinition** The given *identifier* has already been used for an enumeration, structure, or union tag. You should use distinct names for different tags.
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. You should use distinct names for different enumeration constants.
C2028:	**struct/union member needs to be inside a struct/union** Structure and union members must be declared within the structure or union.
C2029:	**'*identifier*' : bit-fields only allowed in structs** Only structure types can contain bit-fields.
C2030:	**'*identifier*' :struct/union member redefinition** The same identifier was used for more than one structure or union member.

Number	Message, Cause/Action
C2031:	'*identifier*' **: function cannot be struct/union member** A function cannot be a member of a structure. Use a pointer to a function instead.
C2032:	'*identifier*' **: base type with near/far/huge not allowed** Declarations of structure and union members cannot use the **near**, **far**, and **huge** keywords.
C2033:	'*identifier*' **: bit-field cannot have indirection** The bit field is declared as 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 exceeds the number of bits in the given unsigned type.
C2035:	**enum/struct/union** '*identifier*' **: unknown size** A member of a structure or union has an undefined size.
C2036:	**left of '->***identifier*' **must have struct/union type** The expression before member selection operator '->' is not a pointer to a structure or union type, or the expression before member selection operator '.' does not evaluate to a structure or union.
C2037:	**left of '->' specifies undefined struct/union** '*identifier*' The expression before member selection operator '->' or '.' identifies a structure or union type that is not defined.
C2038:	'*identifier*' **: not struct/union member** The given *identifier* is used in a context that requires a structure or union member.
C2039:	**'->' requires struct/union pointer** The expression before member selection operator '->' is not a pointer to a structure or union.

Number	Message, Cause/Action
C2040:	**'.' requires struct/union name** The expression before member selection operator '.' is not the name of a structure or union.
C2042:	**signed/unsigned mutually exclusive** You may declare an identifier type as *signed* or *unsigned*, but not both.
C2043:	**illegal break** A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.
C2044:	**illegal continue** A **continue** statement is legal only when it appears within a **do**, **for**, or **while** statement.
C2045:	**'*identifier*' : label redefined** The given *identifier* appears before more than one statement in the same function.
C2046:	**illegal case** The **case** keyword can appear only within a **switch** statement.
C2047:	**illegal default** The **default** keyword can appear only within a **switch** statement.
C2048:	**more than one default** A **switch** statement contains too many **default** labels. Only one is allowed.
C2050:	**non-integral switch expression** Switch expressions must be integers.
C2051:	**case expression not constant** Case expressions must be integer constants.
C2052:	**case expression not integral** Case expressions must be integer constants.
C2053:	**case value '*n*' already used** The decimal equivalent of case value *n* has already been used in this **switch** statement, where *n* is an integer constant.
C2054:	**expected '(' to follow '*identifier*'** The context requires an open parenthesis after the function *identifier*.

Number	Message, Cause/Action
C2055:	**expected formal parameter list, not a type list** An argument type list appears in a function definition where a formal parameter list should appear.
C2056:	**illegal expression** An expression is illegal because of a previous error. The previous error did not produce an error message.
C2057:	**expected constant expression** The context requires a constant expression.
C2058:	**constant expression is not integral** The context requires an integer constant expression.
C2059:	**syntax error : '***token***'** The given *token* caused a syntax error.
C2060:	**syntax error : EOF** The end of the file was found unexpectedly, causing a syntax error.
C2061:	**syntax error : identifier '***identifier***'** The given *identifier* caused a syntax error.
C2062:	type '*identifier*' **unexpected** The given type is misused. Check the syntax of your expression.
C2063:	'*identifier*' **: not a function** The given *identifier* was not declared as a function, but an attempt was made to use it as a function.
C2064:	**term does not evaluate to a function** An attempt is made to call a function through an expression that does not evaluate to a function pointer.
C2065:	'*identifier*' **: undefined** The given *identifier* is not defined.
C2066:	**cast to function returning ... is illegal** An object cannot be cast to a function type.
C2067:	**cast to array type is illegal** An object cannot be cast to an array type.

Number	Message, Cause/Action
C2068:	**illegal cast** A type used in a cast operation is not a legal type.
C2069:	**cast of 'void' term to non-void** The **void** type cannot be cast to any other type.
C2070:	**illegal sizeof operand** The operand of a **sizeof** expression must be an identifier or a type name.
C2071:	'*class*' : **bad storage class** The given storage *class* cannot be used in this context.
C2072:	'*identifier*' : **initialization of a function** Functions cannot be initialized.
C2073:	'*identifier*' : **cannot initialize array in function** Arrays can be initialized only at the external level.
C2074:	'*identifier*' **cannot initialize struct/union in function** Structures and unions can be initialized only at the external level.
C2075:	'*identifier*' : **array initialization needs braces** The braces [{ }] around an array initializer are missing.
C2076:	**struct/union initialization needs braces** The braces [{ }] around a structure or union initializer are missing.
C2077:	**non-integral field initializer** '*identifier*' An attempt is made to initialize a bit-field member of a structure with a non-integer value.
C2078:	**too many initializers** The number of initializers exceeds the number of objects to be initialized.
C2079:	'*variable*' **uses an undefined struct/union** '*identifier*' The given *variable* is declared as a structure or union type identifier that has not been defined.

Number	Message, Cause/Action
C2082:	**redefinition of formal parameter '*identifier*'** A formal parameter to a function is redeclared within the function body.
C2083:	**array '*identifier*' already has a size** The dimensions of the given array have already been declared.
C2084:	**function '*identifier*' already has a body** The given function has already been defined.
C2085:	**'*identifier*' : not in formal parameter list** The given identifier was declared in the list of argument declarations for a function but was not listed in the formal parameter list in the function header.
C2086:	**'*identifier*' : redefinition** The given *identifier* was defined more than once.
C2087:	**'*identifier*' : missing subscript** To refer to an element of an array, you must use a subscript.
C2088:	**use of undefined enum struct/union '*identifier*'** The *identifier* refers to a structure, enumeration, or union type that is 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.
C2093:	**cannot initialize a static or struct with address of automatic vars** You tried to initialize a static pointer to the address of a local variable.
C2094:	**label '*identifier*' was undefined** The function does not contain a statement labeled with the *identifier*.

Number	Message, Cause/Action
C2095:	**function : actual has type void : parameter** *n* Formal parameters and arguments to functions cannot have type **void**; they can, however, have type **void** *, pointer to void.
C2096:	**struct/union comparison illegal** You cannot compare two structures or unions. You can, however, compare individual members of structure and unions.
C2097:	**illegal initialization** An initialization is illegal because of a previous error. The previous error might not have produced an error message.
C2098:	**non-address expression** An attempt was made to initialize an item that is not an *lvalue*.
C2099:	**non-constant offset** An initializer uses a non-constant offset.
C2100:	**illegal indirection** Indirection operator * was applied to a non-pointer value.
C2101:	**'&' on constant** Only variables and functions can have their address taken.
C2102:	**'&' requires lvalue** Address-of operator & can be applied only to lvalue expressions.
C2103:	**'&' on register variable** Register variables cannot have their address taken.
C2104:	**'&' on bit-field** Bit fields cannot have their address taken.
C2105:	**'***operator***' needs lvalue** The *operator* must have an lvalue operand.
C2106:	**'***operator***' : left operand must be lvalue** The left operand of the *operator* must be an lvalue.

Number	Message, Cause/Action
C2107:	**Illegal index, indirection not allowed** A subscript was applied to an expression that does not evaluate to a pointer.
C2108:	**non-integral index** Only integer expressions are allowed in array subscripts.
C2109:	**subscript on non-array** A subscript was used on a variable that is not an array.
C2110:	**'+' : 2 pointers** Two pointers cannot be added.
C2111:	**pointer + non-integral value** Only integer values can be added to pointers.
C2112:	**illegal pointer subtraction** Only pointers that point to the same type can be subtracted.
C2113:	**'–' : right operand pointer** The right-hand operand in a subtraction operation (–) is a pointer, but the left-hand operand is not.
C2114:	**'*operator*' : pointer on left; needs integral right** The left operand of the *operator* is a pointer; the right operand must be an integer value.
C2115:	**'*identifier*' : incompatible types** An expression contains types that are not compatible.
C2116:	**'*operator*' : bad left *or* right operand** The specified operand of the *operator* is an illegal value.
C2117:	**'*operator*' : illegal for struct/union** Structure and union type values are not allowed with the *operator*.
C2118:	**negative subscript** A value defining an array size was negative.

Number	Message, Cause/Action

C2119:	**'typedefs' both define indirection**
	Two **typedef** types are used to declare an item and both **typedef** types have indirection. For example, the declaration of *pshint;* in the following example is illegal:
	`typedef int *P_INT;`
	`typedef short *P_SHORT;`
	`  /* This declaration is illegal */`
	`P_SHORT P_INT pshint;`

C2120:	**'void' illegal with all types** The **void** type cannot be used in declarations with other types.
C2125:	**'*identifier*': allocation exceeds 64KB** The given item exceeds the limit of 64KB. The only items that are allowed to exceed 64KB are **huge** arrays.
C2127:	**parameter allocation exceeds 32KB** The storage space required for the parameters to a function exceeds the limit of 32KB.
C2128:	**'*identifier*' huge array cannot be aligned to segment boundary** The given array violates one of the restrictions imposed on huge arrays. See Chapter 3, "Linking A Program."
C2129:	**static function '*identifier*' not found** A forward reference was made to a missing static procedure.
C2130:	**#line expected a string containing the file name, found 'name'** A #line directive is missing a filename.

C2131:	**attributes specify more than one near/far/huge**
	More than one **near, far,** or **huge** attribute was applied to an item, as in the following example:
	`typedef int near NINT;`
	`NINT far a;  /* Illegal */`

C2132:	**syntax error: unexpected identifier** The given identifier caused a syntax error.
C2133:	**array '*identifier*': unknown size** A negative subscript was used in an array, or there is an improper size designation.

Number	Message, Cause/Action
C2134:	**'*identifier*': struct/union too large** The declared symbol is greater than 2^{32}. If the struct/union did not have a tag name, this message reads "<unnamed> struct/union too large."
C2135:	**missing ')' in macro expansion** A macro reference with arguments is missing a closing parenthesis.
C2137:	**empty character constant** The single quotes delimiting a character constant must contain one character. For example, the declaration char a = ' ' is illegal. To represent a null character constant, use an escape sequence, such as '\0'.
C2138:	**unmatched close comment '*/'** The compiler detected */ without a matching /*. This usually indicates an attempt to use nested comments, which is illegal. Verify that your /* and */ are properly paired and are not nested.
C2139:	**type following *type* is illegal** There is an illegal type combination, such as the following: `long char a;  /* Illegal */`

C2140:	**argument type cannot be function returning...**
	A function is declared as a formal parameter of another function, as in the following example:
	`int funcl(a)`
	`  int a( );  /* Illegal */`

C2141:	**value out of range for enum constant** An enumerated constant has a value outside the range of values allowed for type **int**. The value must be between –32768 and 32767.
C2142:	**ellipsis requires three periods** The compiler has detected the token ".." and assumes "..." was intended. Use "..." to represent a variable-length parameter list.

Number	Message, Cause/Action
C2143:	**syntax error: missing '*token1*' before '*token2*'** The compiler has detected a syntax error that may be a missing token prior to the specified *token2*. The compiler inserts *token1* and attempts to continue parsing. Note that even if the compiler has inserted the correct token, the compile fails until the source file is changed.
C2144:	**syntax error: missing '*token*' before type '*type*'** Same as C2143, except that the second token is known to be a type, such as **int** or **float**.
C2145:	**syntax error: missing '*token*' before identifier** Same as C2143, except that the second token is an identifier whose name is not currently known. This can happen in certain situations involving look-ahead tokens.
C2146:	**syntax error: missing '*token*' before identifier '*identifier*'** Same as C2145, except that the identifier is listed.

C2147:	**unknown size**
	An operation has been done on an unsized array that requires knowledge of the array size, for example:
	`struct foo *p;`
	`...`
	`p[2];`
	where **struct foo** has not been defined at the time the **p[2]** is seen. You may also get this message when attempting to do arithmetic with a pointer to **void.** To correct, cast the pointer to an object of known size.

C2148:	**array too large** You used an array larger than 2^{32} bytes.
C2149:	'*identifier*'**: named bit-field cannot have zero width** Bit fields of zero width must be unnamed.

Number	Message, Cause/Action
C2150:	'*identifier*'**: bit field must have type int, signed int, or unsigned int** You used compile option -Za to force ANSI conformance but declared a bit field with a type other than those permitted.
C2151:	**more than one cdecl/fortran/pascal attribute specified** You gave more than one of the keywords **cdecl**, **fortran**, or **pascal** in a declaration.

C2152:	'*operator*' **: pointers to functions with different attributes**
	The function pointer operands of the specified '*operator*' have differing near or far attributes or different language (**cdecl** or **fortran/pascal**) attributes.
	`int far foo ();  /* far function */`
	`int (near *fp) () = foo();  /* near func ptr - ERROR */`
	`Or:`
	`int pascal foo(int, int);  /* pascal function */`
	`int (*fp) () = foo;/* C function pointer - ERROR */`

C2153:	**hex constants must have at least 1 hex digit** You used the form \x, which is not valid syntax for a hexadecimal constant.
C2154:	**'name' :does not refer to a segment** You used a name that was not a valid segment name.
C2155:	*'name'* **:already in a segment** The *name* has already been defined in the segment. Use a different unique name.
C2156:	**pragma must be at outer level** You used a pragma command interior to a function to which it applies. Place the pragma statement before the function.
C2157:	*'name'* **:must be declared before use in pragma list** You must declare the *name* before using it in a pragma.

Number	Message, Cause/Action
C2158:	*'name'* **:is a function** The given function *name* is incorrect in the context in which it appeared.
C2159:	**more than one storage class specified** You declared a variable with more than one storage class specifier, such as `extern static f;`
C2160:	**## cannot occur at the beginning of a macro definition** You cannot use the token-pasting operator at the beginning of a macro definition. It can appear only between tokens.
C2161:	**## cannot occur at the end of a macro definition** You cannot use the token-pasting operator at the end of a macro definition. It can appear only between tokens.
C2162:	**expected macro formal parameter** A macro formal parameter should be used in this expression.
C2163:	*'functionname'* **:not available as an intrinsic** You gave the *functionname* in an **intrinsic** pragma, but it is not among those listed as intrinsic in "Declaring Functions as Intrinsic /Oi" on page 3-66.
C2164	*'function'* **: intrinsic was not declared** You did not declare the given *function* before using it in an **intrinsic** pragma. This error appears only if you compile with the /Oi option.
C2165:	*'string'* **:cannot modify pointers to data** You attempted to change a pointer to data that cannot be changed.
C2166:	**lval specifies 'const' object** You attempted to modify an item declared with **const** type.
C2167:	*'name'* **:too many actual parameters for intrinsic** You specified too many parameters to an **intrinsic** pragma.

Number	Message, Cause/Action
C2168:	*'name'* **:too few actual parameters for intrinsic** You specified too few parameters to an **intrinsic** pragma.
C2169:	*'name'* **:is an intrinsic, it cannot be defined** You used the given name in a function definition, but it is also an intrinsic function. Use another distinct name for the defined function.
C2170:	*'name'* **:intrinsic not declared as a function** The name you used in an **intrinsic** pragma must be declared with function type.
C2171:	*'string'*:**bad operand** The *string* has incorrect operand syntax.

C2172:	'*functionname*'**: actual is not a pointer: parameter** *n*
	This message is generated when the *n*th parameter (of the *m*th parameter list) of *functionname* is a structure or a union, and the corresponding formal parameter is a pointer to **void**.
	`int function(void *)`
	`struct bar`
	`  {`
	`  int i,j,k;`
	`  } *foo;`
	`main()`
	`{`
	`function(*foo); /*illegal to pass a structure by value*/`
	`  /*to a pointer to void*/`
	`}`

C2173:	'*functionname*'**: actual is not a pointer: parameter** *n*, **parameter list** *m* This message is generated when the *n*th parameter (of the *m*th parameter list) of *functionname* is a structure or a union and the corresponding formal parameter is a pointer to **void**. (See example in C2172.)

Number	Message, Cause/Action
C2174	*function* : **actual has type void: parameter** *n*, **parameter list** *m* You attempted 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 incorrect argument.
C2177:	**constant too big** Information is lost because a constant value is too large to be represented in the type it is assigned to.
C2178	*'name'* **:storage class for same_seg variables must be 'extern'** You specified *name* in a **same_seg** pragma, but it was not declared with **extern** storage class.

C2179	*'name'* **: was used in same_seg, but storage class is no longer 'extern'**
	You specified *name* in a **same_seg** pragma, but it was redeclared with a storage class other than **extern**, as in the following example:
	`extern int i,j;`
	`#pragma same_seg(i,j)`
	`int i;`

C2180	**controlling expression has type 'void'** The controlling expression in an **if**, **while**, **for**, or **do** statement was a function with **void** return type.
C2182	*'name'* **: 'void' on variable** You declared *name* with the **void** keyword. The **void** keyword can be used only in function declarations.
C2183	*name* **: 'interrupt' function must be 'far'** You declared *name* as a **near interrupt** function. You must declare the function without the **near** attribute; and if you compile the program with the default (small) or compact memory model, you must explicitly declare the function with the **far** attribute.

Number	Message, Cause/Action
C2184	*name* : **'interrupt' function cannot be 'pascal/fortran'** The given **interrupt** function was declared with the FORTRAN/Pascal calling convention, either because the **fortran** or **pascal** attribute was used in the declaration or because the program was compiled with the /Gc option. Functions declared with the **interrupt** attribute are required to use the C calling conventions. Therefore, you must either declare the function without the **fortran** or **pascal** attribute if you compile the program without the /Gc option, or declare the function with the **cdecl** attribute if you compile the program with the /Gc option.
C2186	*'name* : **'saveregs/interrupt' modifiers mutually exclusive** You used both **saveregs** and **interrupt** when declaring function *name*. The **saveregs** and **interrupt** modifiers are mutually exclusive.
C2187:	**cast of near function pointer to far function pointer** The compiler does not allow casts on function pointers that change the pointer size. The reason is that the resulting function pointer cannot be used to call a function. You cannot do a near call to a far function or vice versa, so the cast is meaningless and dangerous. The same is true for the inverse, the cast of a far function pointer to near.
C2188	**#error** : *message* The #error directive was used to terminate compilation and display a message.
C2189:	**constant item, -Gm, and data_seg pragma are incompatible** You may not give a new data segment name for **const** items.
C2190	*'seg name'* **: is a text segment** The **data_seg** pragma expects to receive the name of a data segment; you passed the name of a text segment, *segname*.

Number	Message, Cause/Action
C2191	*'seg name'* **: is a data segment** The first argument in an **alloc_text** pragma should be the name of a text segment; you passed the name of a data segment, *segname*.

C2192	*'func name'* **: function has already been defined**
	A function name passed as an argument in an **alloc_text** pragma has already been defined, as in the following example:
	`sample()`
	`{`
	`}`
	`#pragma alloc_text(CODE_SEG, sample)`
	The preceding code causes error message C2192 because the pragma tells the compiler where to allocate the function after it has already been allocated.

C2205:	*'name'* **: cannot initialize 'extern' block scoped variables**
	The ANSI C Standard does not allow the initialization of block-scoped variables declared with the **extern** storage class. For example:
	`int foo()`
	`{`
	`  extern int i=0;  /* illegal */`
	`}`

Warning Error Messages

The messages listed in this section indicate potential problems but do not hinder compiling and linking. The number in brackets at the end of each message gives the minimum warning level that must be set for the message to appear.

Number	Message, Cause/Action
C4001:	**macro '***identifier***': requires parameters [1]** The given *identifier* was defined as a macro taking one or more arguments, but the *identifier* is used in the program without arguments. Specify the correct number of arguments to your macro.

Number	Message, Cause/Action
C4002:	**too many actual parameters for macro '*identifier*' [1]** The number of arguments specified with *identifier* is greater than the number of formal parameters given in the macro definition of the identifier. Specify the correct number of arguments to your macro.
C4003:	**not enough actual parameters for macro '*identifier*' [1]** The number of arguments specified with *identifier* is less than the number of formal parameters given in the macro definition of the identifier. Specify the correct number of arguments to your macro.
C4004:	**missing close parenthesis after 'defined' [1]** The closing parenthesis is missing from an **#if defined** phrase.
C4005:	**'*identifier*' : redefinition [1]** The given *identifier* is redefined.
C4006:	**#undef expected an identifier [1]** The name of the identifier whose definition is to be removed must be given with the **#undef** directive.
C4009:	**string too big, trailing chars truncated [1]** A string exceeds the compiler limit on string size. To correct this problem, you must break the string down into two or more strings.
C4011:	**identifier truncated to '*identifier*' [1]** Only the first 31 characters of an identifier are significant.
C4014:	**'*identifier*' : bit-field type must be unsigned [1]** Bit fields must be declared as **unsigned** integer types. A conversion has been supplied.
C4015:	**'*identifier*' : bit-field type must be integral [1]** Bit fields must be declared as **unsigned** integral types. A conversion has been supplied.

Number	Message, Cause/Action
C4016:	'*name*': **no function return type, using 'int' as default** No function declaration or definition for *name* has been given. The default return type of **int** is assumed.
C4017:	**cast of int expression to far pointer [1]** A **far** pointer represents a full segmented address. On an 8086/8088 processor, casting an **int** value to a **far** pointer produces an address with a meaningless segment value.
C4020:	'*name*' **too many actual parameters [1]** The number of arguments specified in a call to function *name* is greater than the number of parameters specified in the argument type list or in the function definition.
C4021:	'*name*': **too few actual parameters [1]** The number of arguments specified in a call to function *name* is less than the number of parameters specified in the argument type list or in the function definition.
C4022:	'*name*': **pointer mismatch : parameter** *n* **[1]** The given parameter has a different pointer type than is specified in the argument type list or the function definition for the named function.
C4024:	'*name*': **different types : parameter** *n* **[1]** The type of the given parameter in a function call does not agree with the argument type list or the function definition for the named function.
C4025:	**function declaration specified variable argument list [1]** The argument type list in a function declaration ends with a comma, indicating that the function can take a variable number of arguments, but no formal parameters for the function are declared.
C4026:	**function was declared with formal argument list [1]** The function was declared to take arguments, but the function definition does not declare formal parameters.

Number	Message, Cause/Action
C4027:	**function was declared without formal argument list [1]** The argument type list consists of the word **void**. The function was declared to take no argument, but formal parameters are declared in the function definition, or arguments are given in a call to the function.
C4028:	**parameter *n* declaration different [1]** The type of the given parameter does not agree with the corresponding type in the argument type list or with the corresponding formal parameter.
C4029:	**declared parameter list different from definition [1]** The argument type list given in a function declaration does not agree with the types of the formal parameters given in the function definition.
C4030:	**first parameter list is longer than the second [1]** A function is declared more than once, and the argument type lists in the declarations differ.
C4031:	**second parameter list is longer than the first [1]** A function is declared more than once, and the argument type lists in the declarations differ.
C4032:	**unnamed struct/union as parameter [1]** The structure or union type being passed as an argument is not named, so the declaration of the formal parameter cannot use the name and must declare the type.
C4033:	**function must return a value [2]** A function is expected to return a value unless it is declared as **void**.
C4034:	**sizeof returns 0 [1]** The **sizeof** operator is applied to an operand that yields a size of zero.
C4035:	**'*function*' :no return value [2]** A function declared to return a value does not do so.
C4036:	**unexpected formal parameter list [1]** A formal parameter list is given in a function declaration and is ignored.

Number	Message, Cause/Action
C4037:	**'*identifier*' : formal parameters ignored [1]** Formal parameters appeared in a function declaration, for example: `extern int *f(a,b,c);` The formal parameters are ignored.
C4038:	**:'*identifier*' : formal parameter has bad storage class [1]** Formal parameters must have **auto** or **register** storage class.
C4039:	**'*identifier*' : function used as an argument [1]** A formal parameter to a function is declared to be a function, which is illegal. The formal parameter is converted to a function pointer.
C4040:	**near/far/huge on '*identifier*' ignored [1]** The **near**, **far**, or **huge** keyword has no effect in the declaration of the given '*identifier*' and is ignored.
C4041:	**formal parameter on '*identifier*' is redefined [1].** The given formal parameter is redefined in the function body, making the corresponding actual argument unavailable in the function.
C4042:	**'*identifier*' : has bad storage class [1]** 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 is used in place of the illegal class.
C4044:	**huge on '*identifier*' ignored, must be an array [1]** The **huge** keyword can only be used in array declarations.
C4045:	**'*identifier*' : array bounds overflow [1]** Too many initializers are present for the given array. The excess initializers are ignored.
C4046:	**'&' on function/array, ignored [1]** You cannot apply the address-of operator **&** to a function or an array identifier.

Number	Message, Cause/Action

C4047:	'*operator*' **: different levels of indirection [1]**
	An expression involving the specified operator has inconsistent levels of indirection. For example:
	`char **p;  /* Two levels of indirection */`
	`char *q;  /* One level of indirection */`
	`⋮`
	`p=q;  /* Different levels of indirection */`

C4048:	**array's declared subscripts different [1]** An array is declared twice with differing sizes. The larger size is used.
C4049:	'*operator*' **: indirection to different types [1]** The indirection operator * is used in an expression to get access to values of different types.
C4051:	**data conversion [3]** Two data items in an expression had different types, causing the type of one item to be converted.
C4052:	**different enum types [1]** Two different **enum** types are used in an expression.
C4053:	**at least one void operand [1]** An expression with type **void** is used as an operand.
C4054	**insufficient memory may affect optimization** Not enough memory was available to perform all of the requested optimizations. This message appears if available memory is within 64KB of the absolute minimum that will accommodate the executable file.
C4056:	**overflow in constant arithmetic [1]** The result of an operation exceeds 0x7FFFFFFF.
C4057:	**overflow in constant multiplication [1]** The result of an operation exceeds 0x7FFFFFFF.
C4058:	**address of frame variable taken, DS != SS [1]** Program was compiled with the default data segment (DS) not equal to the stack segment (SS), and you tried to point to a frame variable with a **near** pointer.

Number	Message, Cause/Action
C4059:	**segment lost in conversion [1]** The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) results in the loss of the segment address.
C4060:	**conversion of long address to short address [1]** The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) results in the loss of the segment address.
C4061:	**long/short mismatch in argument: conversion supplied [1]** An integral type is assigned to an integer of a different size, causing a conversion to take place. For example, a **long** is given where a **short** was declared.
C4062:	**near/far mismatch in argument: conversion supplied [1]** A pointer is assigned to a pointer with a different size, resulting in the loss of a segment address from a **far** pointer or the addition of a segment address to a **near** pointer.
C4063:	'*identifier*' **: function too large for post-optimizer [0]** The named function was not optimized because not enough space was available. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.
C4064:	**procedure too large, skipping [loop inversion** *or* **branch sequence** *or* **cross jump] optimization and continuing [0]** Some optimizations for a function are skipped because insufficient space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.

Number	Message, Cause/Action
C4065:	**recoverable heap overflow in post optimizer - some optimizations may be missed [0]** Some optimizations are skipped because not enough space is available for optimization. To correct this problem, reduce the size of the function by breaking it down into two or more smaller functions.
C4066:	**local symbol table overflow - some local symbols may be missing in listings [1]** The listing generator ran out of heap space for local variables,so the source listing might not contain symbol-table information for all local variables.

C4067:	**unexpected characters following '*identifier*' directive - newline expected [1]**
	There are extra characters following a preprocessor directive, such as the following :
	`#endif NO_EXT_KEYS`
	This is accepted in the IBM C Compiler, Version 1.00, but not in IBM C/2. IBM C/2 requires comment delimiters, such as the following:
	`#endif /* NO_EXT_KEYS */`

C4068:	**unknown pragma [1]** The compiler does not recognize the pragma you used and ignores this pragma.
C4069:	**conversion of near pointer to long integer [1]** A **near** pointer is being converted to a long integer, which involves extending the high-order word with the current data segment value.
C4070:	**function called as procedure** A function having the **pascal** or **fortran** attribute and a **struct** or floating-point return type was called as a **void** function. The result will never be read or stored.
C4071:	**'*identifier*': no function prototype given [3]** You did not supply an argument-type list for *identifier*.

Number	Message, Cause/Action
C4072 :	**insufficient memory to process debugging information [1]** Your computer lacks enough memory to compile this program using the Zi option.
C4073:	**scoping too deep, deepest scoping merged when debugging [1]** The visibility control of identifiers in deeply-nested blocks exceeds a built-in limit. Variables in all the deepest levels will be visible to CodeView during debugging.
C4074 :	**non-standard extension used -'*description*' [3]** You used a valid construction that is not recognized by the proposed ANSI standard for C. The *description* string may be one of the following: • Trailing ',' used for variable argument list • Cast on lvalue • Extended initializer form • Benign typedef redefinition • Redefined extern to static • Macro formals in strings • Missing ';' following last struct/union member • Bit-field types other than int • Functions given file scope • Cast of data pointer to function pointer • Cast of function pointer to data pointer • Function declaration used ellipsis.
C4075:	**size of switch expression or case constant too large - converted to int [1]** You used a switch expression that evaluated to more than 32767.
C4076:	**'*type*' : may be used on integral types only [1]** You used the given keyword with a non-integral data type.
C4077:	**unknown check_stack option [1]** You gave an incorrect argument to pragma **check_stack**. Use a correct form as given in Chapter 2 of *IBM C/2 Compile, Link, and Run*.

Number	Message, Cause/Action
C4078:	**loss of debugging information caused by optimization** The compiler could not generate debugging information. You can debug by turning off optimization with switch -Od.
C4079	**unexpected token** *'token'* The source line contains a misplaced identifier.
C4080:	**expected 'identifier' for segment name, found** *'token'* **[1]** You used *token* as a segment name, instead of the correct identifier.
C4081:	**expected a comma, found** *token* **[1]** The syntax of the expression requires a comma rather than *token*.
C4082:	**expected an identifier, found** *'token'* **[1]** The syntax of the expression requires another identifier in place of *token*.
C4083:	**expected '(', found** *'token'* **[1]** The syntax of the expression requires a left parenthesis at the place *token* appears.
C4084:	**expected a pragma keyword, found** *'token'* **[1]** You used an unknown identifier in a pragma.
C4085:	**expected [on|off] [1]** The argument in the parenthesized form of the check_stack pragma must be either on or off.
C4086:	**expected [1 | 2 | 4]** The pack pragma requires a parameter for byte alignment. If you do not give it, the compiler assumes 1-byte alignment.
C4087:	*'name'* **: declared with 'void' parameter list [1]** A function declared with a void parameter list was called with actual arguments.

Number	Message, Cause/Action

C4088:	**'*name*' : pointer mismatch: parameter *n*, parameter list *m* [1]**
	The argument in the given function call has a different level of indirection, as in the following example:
	`int (*sample (void *)) (void *);`
	`⋮`
	`main()`
	`{`
	`  sample(10) (10); /* pointer mismatch:`
	`  parameter 1, parameter list 2 */`
	`}`

C4089	***'function'* : different types : parameter *n* parameter list *m***
	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:
	`int (*sample(int,int))(char *);`
	`main()`
	`{`
	`  int i;`
	`  (*sample(10,20))(i);/* pointer`
	`  mismatch : parameter 1,`
	`  parameter list 2. */`
	`}`

C4090:	**different 'const' attributes** You used an incompatible combination of const attributes.
C4091:	**no symbols were declared** Your segment contained no identifiers.
C4092:	**untagged enum/struct/union declared no symbols** An aggregate type had no identifiers.
C4093:	**unescaped newline in character constant in non-active code** The preprocessor found an unmatched single quote or double quote on a single line within an **#if/#ifdef/#elif/#else** block that is being skipped because of a false entry condition.
C4095 :	**expected ')', found *'token'*** This is similar to C4079. The compiler will use *token*.

Number	Message, Cause/Action
C4098:	**void function returning a value [1]** You used a return statement with an expression within a function returning **void**. The value of the expression will be ignored by the calling function.
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]

C4104 :	*'identifier'* **: near data in same_seg pragma, ignored**
	The given **near** identifier was specified in a **same_seg** pragma, as in the following example:
	`extern int near near_var;`
	`extern int far far_var;`
	`#pragma same_seg(near_var, far_var);`
	In this example, the compiler ignores the specification of near_var; consequently, it does not assume that near_var and far_var reside in the same data segment.

C4105	*'name'* **: code modifiers only on function or pointer to function** The **interrupt** attribute was used to declare something other than a function or function pointer.
C4106:	**pragma requires integer between 1 and 127** You used an incorrect number in a **skip** or **page** pragma.
C4107:	**pragma requires integer between 15 and 255** You used an incorrect number in a **pagesize** pragma.

Number	Message, Cause/Action
C4108:	**pragma requires integer between 79 and 132** You used an incorrect number in a **linesize** pragma.
C4109:	**unexpected identifier *'string'*** You supplied an incorrect argument (unknown comment type) in a **comment** pragma.
C4110:	**unexpected token 'int constant'** The compiler encountered an integral constant in an unexpected position in a pragma.
C4111:	**unexpected token *'string'*** The compiler encountered a string literal out of position in a pragma.
C4112:	**macro name *'name'* is reserved, *name* ignored** You attempted to #undef predefined macro names, such as __FILE__ or __DATE__, or to undefine the keyword "defined" which is used in preprocessor directives. This is not allowed by the ANSI C Standard.

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:
	`int (*sample) (int);`
	`int (*example) (char, char);`
	`main()`
	`{`
	`  sample = example;`
	`}`

C4114 :	**same type qualifier used more than once**
	You used the type specifier **const** or **volatile** more than once in a declaration, for example:
	`const const i;`
	`volatile int volatile k;`
	The extra specifier is ignored.

Number	Message, Cause/Action
C4115 :	**'tag' type definition in formal parameter list illegal** This is issued if a function definition or declaration contains the declaration of a struct, union, or enum type. 'Tag' is replaced by the actual tag of the struct/union/enum being declared. For example: `int foo(enum color { red, blue, green } col);` would generate C4116.
C4115 :	**'color' type definition in formal parameter list illegal**

C4116 :	**'no tag' : type definition in formal parameter list illegal**
	Same as C4115 but emitted in the case when no tag is specified, as:
	`int foo(ap)`
	`union {`
	`  int *ip;`
	`  long *lp;`
	`  float *fp;`
	`  } ap;`
	`{`
	`...`
	`}`
	would generate the next description of C4116.

C4116 :	**'no tag' type definition in formal parameter list illegal** These warnings are emitted because you cannot pass a matching argument or define a matching function (in the case of a prototype); the type has prototype scope in the case of a prototype, and function scope in the case of a definition. No reference external to these scopes can match correctly.

Number	Message, Cause/Action

C4185 :	**near call to *function* in different segment**
	You specified the given function in an **alloc_text** pragma without declaring it to be **far**, then called this function from another text segment, as in the following example:
	`int f();`
	`#pragma alloc_text(NEW, f)`
	`main()`
	`{`
	`  f();`
	`}`
	`f()`
	`{`
	`}`
	In this example, the main function (in default text segment) makes a **near** call to the f function (in the text segment NEW). The error generated is:
	near call to _f in different segment
	Although this is a warning message rather than an error message, the resulting code will not work correctly. Note that if you compile with stack checking enabled (the default), you would also get the following error message for the f function:
	near call to __chkstk in different segment

C4186:	**string too long. Truncated to 40 characters.** The string argument for a **title** or **subtitle** pragma exceeded the maximum allowable length and was truncated.

Command Area Error Messages

The following messages indicate errors in the command that you use to call the compiler. In some cases, command errors are fatal and the compiler stops processing. If the error is not fatal, the compiler continues operation but prints a warning message.

Fatal Error Messages

Number	Message, Cause/Action
D1000:	**UNKNOWN COMMAND ENTRY FATAL ERROR** An unforeseen error condition has been detected by the compiler. Please report this error to an IBM Authorized Dealer.
D1001:	**could not execute '***pass***'** The specified compiler file could not be found, or there is not enough space in storage.
D1002:	**too many open files, cannot redirect '***filename***'** No more file handles are available to redirect the output of the -P option to a file. Try editing the CONFIG.SYS file and increasing the value *num* on the line **files** = *num* (if *num* is less than 20.)

Error Messages

Number	Message, Cause/Action
D2000:	**UNKNOWN COMMAND ENTRY ERROR** An unforeseen error condition has been detected by the compiler. Please report this error to an IBM Authorized Dealer.
D2001:	**too many symbols predefined with –D** The limit on command definitions is normally 16; the /U option can increase the limit to 20.
D2002:	**a previously-defined model specification has been overridden.** Two different storage models are specified; the model specified last is used.
D2003:	**missing source file name** You must give the name of the source file to be compiled.
D2004:	**too many commas** Too many commas appear in the command.
D2005:	**comma needed before :'***filename***'** The fields in the command must be set off by commas.
D2006:	**a file name (not a path name) is required** The name of a directory is given where the name of a file is required.

Number	Message, Cause/Action
D2008:	**too many** *option* **flags in '***string***'** Too many letters are given with a specific option (for example, with the /O option).
D2009:	**unknown option '***c***' in '***option***'** One of the letters in the given option is not recognized.
D2010:	**unknown floating-point option** The specified floating-point option (an /FP option) is not one of the five valid options.
D2011:	**only one floating-point model allowed** You can give only one of the five floating-point (/FP) options in the command.
D2012:	**too many linker flags at command prompt** For compile-and-link (CL) only, you attempted to pass more than 128 separate options and object files to the linker.
D2013:	**incomplete model specification** The A*string* option requires all three character (data-pointer size, code-pointer size, and segment setup) in *string*.
D2015:	**assembly files are not handled** You specified a filename with the extension .ASM. The compiler cannot invoke MASM automatically, so it cannot assemble these files.
D2016:	**-Gw and -ND** *name* **are incompatible** You cannot rename the default data segment to *name* when you give the -G2 option because -Gw also requires -Aw.
D2017:	**-Gw and -Au flags are incompatible** You cannot use the -Au option (if the stack segment does not equal the data segment, load the data segment) with -Gw because -Gw also requires -Aw.
D2018:	**cannot open linker cmd file** The compiler cannot open the response file used to pass object filenames and options to the linker. One possible cause of this error is the existence of another file that is a read-only file with the same name as the response file.

Number	Message, Cause/Action
D2019:	**cannot overwrite the source file, '*filename*'** The source file specified an output filename. 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 requires the extended keyword **cdecl** to be enabled if the library functions are to be accessible.
D2021:	**invalid numerical argument '*string*'** You specified a non-numerical string following an option that requires a numerical argument.
D2022:	**cannot open help file '*filename*'** The driver expects the help file to be in the same directory or path it is in.
D2024 :	**-Gm and -ND name are incompatible** You may not change the default data segment name when using ROM code.
D2025 :	**missing argument** You supplied a CL option that requires an argument but did not supply the argument. For instance, the following command generates error D2025: `cl /Tc` The preceding command causes an error because the /Tc option requires a source filename.
D2027:	**cannot link file '*filename*'** You specified a filename with the extension .OBJ. This is not a valid extension as a source filename for the CC command.

Warning Error Messages

D4000:	**UNKNOWN COMMAND ENTRY WARNING** An unforeseen error condition has been detected by the compiler. Please report this error to an IBM Authorized Dealer.

D4001:	**listing has precedence over assembly output** Two different listing options were chosen; the assembly listing is not created.
D4002:	**ignoring unknown flag '*string*'** One of the options given at the command prompt is not recognized and is ignored.
D4003:	**80186/286 selected over 8086 for code generation** Both /G1 and /G2 are selected.
D4004:	**optimizing for time over space** This message confirms that the /Ot option is used for optimizing.
D4005:	**could not execute '*name*',** **please enter new file name (full path) or Ctrl+C to quit:** One of the compiler passes cannot be found on the current disk.
D4006:	**only one of -P/-E/-EP allowed, -P selected** Only one preprocessor option can be specified at one time.
D4007:	**-C ignored (must also specify -P or -E or -EP)** The -C option must be used with one of the preprocessor output flags, -E, -EP, or -P.
D4009:	**threshold only for far/huge data, ignored** The -Gt option cannot be used in memory models that have near data pointers. The -Gt option can be used only with compact-, large-, and huge-memory models.
D4010:	**-Gp not implemented, ignored** The DOS version of the compiler does not allow profiling.
D4011:	**preprocessing overrides source listing** The compiler produces only a preprocessor listing because it cannot produce both a source listing and a preprocessor listing at the same time.
D4012:	**function declarations override source listing** The compiler cannot produce both a source-listing file and the function prototype declarations at the same time.

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:	**incorrect value *n* for '*identifier*'. Default *m* is used** You used an incorrect numerical value for the given switch.
D4017:	**conflicting stack checking options- stack checking disabled** You specified both -Ge (enable stack checking) and -Gs (disable stack checking) at the same command prompt. The compiler will turn off stack checking.
D4019:	**string too long. Truncated to 40 characters** You supplied an overly long string as an argument for the /ND, /NT, /NM, /St, or /Ss option of the CL command; the string was truncated.

Compiler Internal Error Messages

The following messages indicate errors on the part of the compiler. Although the errors are not the fault of the program, you may want to rearrange the code so the program can be compiled. Please report these errors to an IBM Authorized Dealer.

Number	Message, Cause/Action
C1000:	**UNKNOWN FATAL ERROR** An unforeseen error condition has been detected by the compiler.
C2000:	**UNKNOWN ERROR** An unforeseen error condition has been detected by the compiler.
C1001:	**Internal Compiler Error** **(compiler file '*name*',line *n*)** **The compiler performs internal consistency checks during compiling. This message indicates that the consistency check failed and the compiler cannot continue operation.**

Number	Message, Cause/Action
C4000:	**UNKNOWN WARNING** An unforeseen error condition has been detected by the compiler.

Redirecting Compiler Error Messages

Error messages produced by the compiler are sent to the standard output, which is usually your screen. With CL you can redirect the messages to a file or printer by using a DOS redirection symbol, > or >>. This is especially useful in batch file processing.

For example, the following command redirects error messages to the printer device (designated by PRN):

```
CL ALPHA.C > PRN
```

while the following command redirects error messages to the file ALPHA.ERR:

```
CL ALPHA.C > ALPHA.ERR
```

The command redirects only output normally sent to the display. In OS/2 mode, use the stream number when redirecting the output as in:

```
CL ALPHA.C 1>ALPHA.ERR
```

Example

Contents of ALPHA.C:

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char argv[ ];

  {
  register int i;
  char *name;

  for (i = 1; i < arg; ++i)
     if (unlink(name = argv[i])) {
        printf("could not delete %s : ", name);
        perror("");
        }
     }
```

Contents of error message file ALPHA.ERR:

```
alpha.c
alpha.c(11) : error C2065: 'arg' : undefined
alpha.c(12) : warning C4047: '=' : different levels of indirection
```

Corrected version of ALPHA.C:

```
#include <stdio.h>

main(argc, argv)
  int argc;
  char *argv[];

  {
  register int i;
  char *name;

  for (i = 1; i < argc; i++)
     if (unlink(name = argv[i])) {
        printf("could not delete %s : ", name);
        perror("");
        }
  }
```

Only output that ordinarily goes to the screen is redirected. The object file is given the name ALPHA.OBJ and is created in the current working directory.

If you request a source listing, error messages will also appear embedded in the listing, following the line they occur in.

Linker Error Messages

This section lists error messages produced by the IBM Linker.

Fatal errors cause the linker to stop running. Fatal error messages have the following format:

location: fatal error L1*xxx*: *message text*

Non-fatal errors indicate problems in the executable file. LINK produces the executable file (and sets the error bit in the header if for OS/2 mode)

location: error L2 *xxx*: *message text*

Warnings indicate possible problems in the executable file. LINK produces the executable file (it does not set the error bit in the header if for OS/2 mode). Warnings have the following format:

location: error L4*xxx*: *message text*

In these messages, *location* is the input file associated with the error, or LINK if there is no input file. If the input file is a module definitions file, the line number is included, as the following shows:

```
foo.def(3): fatal error L1030:missing internal name
```

If the input file is an .OBJ or .LIB file and has a module name, the module name is enclosed in parentheses, as shown in the following examples:

```
SLIBC.LIB(_file)
MAIN.OBJ(main.c)
TEXT.OBJ
```

Fatal Error Messages

Number	Message, Cause/Action
L1001 :	*option* **: option name ambiguous** A unique option name does not appear after the option indicator (/). For example, the command `LINK /N main;` produces this error, since LINK cannot tell which of the three options beginning with the letter *N* is intended.
L1002 :	*option* **: unrecognized option name** An unrecognized character followed the option indicator (/), as in the following example: `LINK /ABCDEF main;`
L1003 :	*option* **: MAP symbol limit too high** The specified symbol limit value following the MAP option is greater than 32767, or there is not enough memory to increase the limit to the requested value.
L1004 :	*option* **: invalid numeric value** An incorrect value appeared for one of the linker options. For example, a character string is entered for an option that requires a numeric value.
L1005 :	*option* **: packing limit exceeds 65536 bytes** The number following the /PACKCODE or /PACKDATA option is greater than 65536.
L1006 :	*option* **: stack size exceeds 65534 bytes** The size you specified for the stack in the /STACK option of the LINK command is more than 65534 bytes.

Number	Message, Cause/Action
L1007 :	*option* **: interrupt number exceeds 255** You gave a number greater than 255 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 you specified in the /CPARMAXALLOC option is not in the range 1 to 65535.
L1020 :	**no object modules specified** You did not specify any object-file names to the linker.
L1021 :	**cannot nest response files** A response file occurs within a response file.
L1022 :	**response line too long** A line in a response file is longer than 127 characters.
L1023 :	**terminated by user** You entered Ctrl + C.
L1024 :	**nested right parentheses** You typed the contents of an overlay incorrectly at the command prompt.
L1025 :	**nested left parentheses** You typed the contents of an overlay incorrectly at the command prompt.
L1026 :	**unmatched right parenthesis** A right parenthesis is missing from the contents specification of an overlay at the command prompt.
L1027 :	**unmatched left parenthesis** A left parenthesis is missing from the contents specification of an overlay at the command prompt.
L1030 :	**missing internal name** In the module definitions file, when you specify an import by entry number, you must give an internal name, so the linker can identify references to the import.

Number	Message, Cause/Action
L1031 :	**module description redefined** In the module definitions file, a module description specified with the DESCRIPTION keyword is given more than once.
L1032 :	**module name redefined** In the module definitions file, the module name is defined more than once with the NAME or LIBRARY keyword.
L1040 :	**too many exported entries** An attempt is made to export more than 3072 names.
L1041 :	**resident-name table overflow** The total length of all resident names, plus 3 bytes per name, is greater than 65534.
L1042 :	**nonresident-name table overflow** The total length of all nonresident names, plus 3 bytes per name, is greater than 65534.
L1043 :	**relocation table overflow** There are more than 65536 load-time relocations for a single segment.
L1044 :	**imported-name table overflow** The total length of all the imported names, plus 1 byte per name, is greater than 65534 bytes.
L1045 :	**too many TYPDEF records** An object module contains more than 255 TYPDEF records. These records describe communal variables. This error can only appear with programs produced by compilers that support communal variables.
L1046 :	**too many external symbols in one module** An object module specifies 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** The program contains too many group, segment, and class names. Reduce the number of groups, segments, or classes, and recreate the object files.

Number	Message, Cause/Action
L1048 :	**too many segments in one module** An object module has more than 255 segments. Split the module or combine segments.
L1049 :	**too many segments** The program has more than the maximum number of segments. The SEGMENTS option specifies the maximum allowed number; the default is 128. Relink using the /SEGMENTS option with an appropriate number of segments.
L1050 :	**too many groups in one module** The linker found more than 21 group definitions (GRPDEF) in a single module. Reduce the number of group definitions or split the module.
L1051 :	**too many groups** The program defines more than 20 groups, not counting DGROUP. Reduce the number of groups.
L1052 :	**too many libraries** An attempt is made to link with more than 32 libraries. Combine libraries, or use modules that require fewer libraries.
L1053 :	**symbol table overflow** The program had more symbolic information (such as public, external, segment, group, class, and filenames) than the amount that could fit in available real memory. Try freeing memory by linking from the DOS command level instead of from a MAKE file or from an editor. Otherwise, combine modules or segments and try to eliminate as many public symbols as possible.
L1054 :	**out of memory : reduce # in /SEGMENTS:# or /MAP:#** The linker does 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 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.

Number	Message, Cause/Action
L1056 :	**too many overlays** The program defines 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 (compiler or assembler) error. Note which translator (compiler or assembler) produced the incorrect object module and the circumstances, and contact an IBM Authorized Dealer.
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 them. Reduce the number of object files that have debug information.
L1070 :	**segment size exceeds 64K** A single segment contains more than 64KB of code or data. Try compiling, or assembling, 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 is increased to 16 for alignment purposes.
L1072 :	**common area longer than 65536 bytes** The program has more than 64KB of communal variables. This error cannot appear with object files produced by the IBM Macro Assembler/2. It occurs only with programs produced by IBM C/2 or other compilers that support communal variables.
L1073 :	**file-segment limit exceeded** There are more than 255 physical or file segments. You could use the /PACKDATA option, see "Packing Data Segments /PACKDATA" on page 3-31 for more information.

Number	Message, Cause/Action
L1074 :	*name* **: group larger than 64K bytes** A group contained segments that total more than 65536 bytes.
L1075 :	**entry table larger than 65535 bytes** Because of an excessive number of entry names, you have exceeded a linker table size limit. Reduce the number of names in the modules you are linking.
L1080 :	**cannot open list file** The disk or the root directory is full. Delete or move files to make space.
L1081 :	**out of space for run file** The disk the .EXE file is being written on is full. Free more space on the disk and restart the linker.
L1082 :	**stub .EXE file not found** The stub file specified in the module definitions file is not found.
L1083 :	**cannot open run file** The disk or the root directory is full. Delete or move files to make space.
L1084 :	**cannot create temporary file** The disk or root directory is full. Free more space in the directory and restart the linker.
L1085 :	**cannot open temporary file** The disk or the root directory is full. Delete or move files to make space.
L1086 :	**scratch file missing** Internal error. Note the conditions when the error occurs and contact an IBM Authorized Dealer.
L1087 :	**unexpected end-of-file on scratch file** The disk with the temporary linker-output file is removed.
L1088 :	**out of space for list file** The disk the listing file is being written on is full. Free more space on the disk and restart the linker.

Number	Message, Cause/Action
L1089 :	*filename* **: cannot open response file** The linker could not find the specified response file. This usually indicates a typing error.
L1090 :	**cannot reopen list file** The original disk is not replaced at the prompt. Restart the linker.
L1091 :	**unexpected end-of-file on library** The disk containing the library probably was removed. Replace the disk containing the library and run the linker again.
L1092 :	**cannot open module definitions file** The specified module definitions file cannot be opened.
L1093 :	**object not found** LINK could not open the object module you specified.
L1100 :	**stub .EXE file invalid** The stub file specified in the definitions file is not a valid .EXE file.
L1101 :	**invalid object module** One of the object modules is non-valid. If the error persists after recompiling, contact an IBM Authorized Dealer.
L1102 :	**unexpected end-of-file** A non-valid format for a library was found.
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, and contact an IBM Authorized Dealer.
L1104 :	*filename* **: not valid library** The specified file is not a valid library file. This error causes the linker to stop running.
L1113 :	**unresolved COMDEF; internal error** Note the conditions when the error occurs and contact an IBM Authorized Dealer.

Number	Message, Cause/Action
L1114 :	**file not suitable for /EXEPACK; relink without** For the linked program, the size of the packed load image plus the packing overhead is larger than that of the unpacked load image. Relink without the EXEPACK option.
L1115 :	**conflicting iopl-parameters-words value** The number of parameter words from a function declared with the _export attribute does not match the number declared in the .DEF file for that function.

Non-Fatal Error Messages

Number	Message, Cause/Action
L2000 :	**imported entry point** A MODEND, or starting address record, referred to an imported name. Imported program-starting addresses are not supported.
L2001 :	**fixup(s) without data** A FIXUP record occurred without a data record immediately preceding it. This is probably a compiler error. See the technical reference information for DOS for more information on FIXUP.

Number	Message, Cause/Action

L2002 :	**fixup overflow near** *number* **in frame seg** *segname* **target seg** *segname* **target offset** *number*
	The following conditions can cause this error:
	• A group is larger than 64KB.
	• 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 subroutine in a library included in the link.
	• An EXTRN declaration in an assembler-language source file appeared inside the body of a segment.
	For 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 recreate the object file.

L2003 :	**Intersegment self-relative fixup** An intersegment self-relative fix-up is not allowed.
L2004 :	**LOBYTE-type fixup overflow** A LOBYTE fix-up produced an address overflow.
L2005 :	**fixup type unsupported** A fix-up type occurred that is not supported by the linker. This is probably a compiler error. Note the conditions when the error occurs and contact an IBM Authorized Dealer.

Number	Message, Cause/Action
L2010 :	**too many fixups in LIDATA record** There are more fix-ups applying to a LIDATA record than will fit in the linker's 1024-byte buffer. The buffer is divided between the data in the LIDATA record and run-time relocation items, which are 8 bytes apiece, so the maximum varies form 0 to 128. This is probably a compiler error.
L2011 :	*name* **: NEAR/HUGE conflict** Conflicting NEAR and HUGE attributes are given for a communal variable. This error can occur only with programs produced by compilers that support communal variables.
L2012 :	*name* **: array-element size mismatch** A far communal array is declared with two or more different array-element sizes (for example, an array declared once as an array of characters and once as an array of real numbers). This error cannot occur with object files produced by the IBM Macro Assembler/2. It occurs only with IBM C/2 and any other compiler that supports far communal arrays.
L2013 :	**LIDATA record too large** A LIDATA record in an object module contains more than 512 bytes of data. Most likely, an assembly module contains a very complex structure definition or a series of deeply-nested DUP operators. For example, the following structure definition causes this error: `alpha  DB  10DUP(11 DUP(12 DUP(13 DUP(...))))` Simplify the structure definition and reassemble. (LIDATA is a DOS term).
L2022 :	*name* **alias** *internalname*: **export undefined** A name is directed to be exported but is not defined anywhere.
L2023 :	*name* **alias** *internalname*: **export imported** An imported name is directed to be exported.

Number	Message, Cause/Action
L2024 :	*name* **: symbol already defined** Your program defined a symbol name that the linker already uses for one of its low-level symbols. For example, the linker generates special names for overlay support. Choose another name for the symbol in order to avoid conflict.
L2025 :	*name* **: symbol defined more than once** Remove the extra symbol definition from the object file.
L2026 :	**multiple definitions for entry ordinal** *number* More than one entry point name is assigned to the same ordinal.
L2027 :	*name* **: ordinal too large for export** You tried to export more than 3072 names.
L2028 :	**automatic data segment plus heap exceeds 64K** The size of DGROUP near data plus requested heap size is greater than 64KB.

L2029 :	**unresolved externals**
	One or more symbols are declared to be external in one or more modules, but they are not publicly defined in any of the modules or libraries. A list of the unresolved external references appears after the message, as shown in the following example:
	`_exit in file(s)`
	`  main.obj (main.c)`
	`_fopen in files(s)`
	`  fileio.obj(fileio.c) main.obj(main.c)`
	The name that comes before **in file(s)** is the unresolved external symbol. On the next line is a list of object modules that have made references to this symbol. This message and the list are also written to the map file, if one exists.

L2030 :	**starting address not code (use class 'CODE')** You specified a starting address to the linker that is a segment that is not a CODE segment. Reclassify the segment to CODE, or correct the starting point.

Number	Message, Cause/Action
L2041 :	**stack plus data exceed 64KB** The total of near data and requested stack size exceeds 64KB, and the program will not run correctly. Reduce the stack size. The linker checks for this condition only if /DOSSEG is enabled, which is done automatically in the library start-up module.

Warning Error Messages

Number	Message, Cause/Action
L4000 :	**seg disp. included** Generated as a result of using the linker /WARNFIXUP option, described in "Warning of Fix-ups /WARNFIXUP" on page 3-34. The segment value is seg and the location offset is disp.
L4001 :	**frame-relative fixup, frame ignored** A fix-up occurred with a frame segment different from the target segment where either the frame or the target segment is not absolute. Such a fix-up is meaningless in OS/2 mode so the target segment is assumed for the frame segment.
L4002 :	**frame-relative absolute fixup** A fix-up occurred with a frame segment different from the target segment where both frame and target segments were absolute. This fix-up is processed using base-offset arithmetic, but the warning is issued because the fix-up may not be valid in OS/2 mode.
L4003 :	**intersegment self-relative fixup at** *offset* **in segment** *name* **pos:** *offset* **Record type: 9C target external** *'name'* The linker found an intersegment self-relative fix-up. This error may be caused by compiling a small-model program with the /NT option.
L4010 :	**invalid alignment specification** The number following the **/ALIGNMENT** option is not a power of 2, or is not in numerical form.

Number	Message, Cause/Action
L4011 :	**PACKCODE value exceeding 65500 unreliable** Code segments of length 65501-65536 may be unreliable on the 80286 processor.
L4012 :	**load-high disables EXEPACK** The options /HIGH and /EXEPACK are mutually exclusive.
L4013 :	**invalid option for new-format executable file ignored** If an OS/2 mode program is being produced, then the options /CPARMAXALLOC, /DSALLOCATE, /EXEPACK, /NOGROUPASSOCIATION, and /OVERLAYINTERRUPT are meaningless, and the linker ignores them.
L4014 :	**invalid option for old-format executable file ignored** If a DOS format program is produced, the options /ALIGNMENT, /NOFARCALLTRANSLATION, /PACKCODE, and /PACKDATA are meaningless, and the linker ignores them.
L4015 :	**/CODEVIEW disables /EXEPACK** The options /CODEVIEW and /EXEPACK are mutually exclusive.
L4020 :	*name* **: code-segment size exceeds 65500** Code segments of length 65501-65536 may be unreliable on the 80286 processor.
L4021 :	**no stack segment** The program does not contain a stack segment defined with STACK combine type. This message should not appear for modules compiled with IBM C/2, but it could appear for an assembler-language module. Normally, every program should have a stack segment with the combine type specified as STACK. You can ignore this message if you have a specific reason for not defining a stack or for defining one without the STACK combine type.

Number	Message, Cause/Action
L4022 :	*name1, name2* **: groups overlap** Two groups are defined such that one starts in the middle of another. This may occur if you defined segments in a module definitions file or assembly file and did not correctly order the segments by class.
L4023 :	*exportname* **: export internal name conflict** An exported name or its associated internal name conflict with an already-defined public symbol.
L4024 :	*name* **: multiple definitions for export name** The name *name* is exported more than once with different internal names. All internal names except the first are ignored.
L4025 :	*name* **: import internal-name conflict** An imported name, or its associated internal name, is also defined as an exported name. The import name is ignored. The conflict may come from a definition in either the module definition file or an object file.
L4026 :	*modulename* **: self-imported** The module definitions file directed that a name be imported from the module being produced.
L4027 :	*name* **: multiple definitions for import internal-name** An imported name, or its associated internal name, is imported more than once. The imported name is ignored after the first mention.
L4028 :	*name* **: segment already defined** A segment is defined more than once with the same name in the module definitions file. Segments must have unique names for the linker. All definitions with the same name after the first are ignored.
L4029 :	*name* **: DGROUP segment converted to type data** A segment that is a member of DGROUP is defined as type CODE in a module definition file or object file. This probably happened because a CLASS keyword in a SEGMENTS statement is not given.

Number	Message, Cause/Action
L4030 :	*name* : **segment attributes changed to conform with automatic data segment** The segment named *name* is defined in DGROUP, but the *shared* attribute is in conflict with the *instance* attribute. For example, the *shared* attribute is NONSHARED and the *instance* is SINGLE, or the *shared* attribute is SHARED and the *instance* attribute is MULTIPLE. The bad segment is forced to have the right *shared* attribute and the link continues. The image is not marked as having errors.
L4031 :	*name* : **segment declared in more than one group** A segment is declared to be a member of two different groups. Correct the source file and recreate the object files.
L4032 :	*name* : **code-group size exceeds 65500 bytes** Code segments of length 65501-65536 may be unreliable on the 80286 processor.
L4034 :	**more than 239 overlay segments; extra put in root** You specified an overlay structure containing more than 239 segments. The extra segments have been assigned to the root overlay, starting with the 234th segment.
L4036 :	**no automatic data segment** The program did not define a group named DGROUP, which is the DOS and OS/2 convention for the default or automatic data segment. You should give the name DGROUP to your automatic data segment.
L4040 :	**NON-CONFORMING : obsolete** In the module definitions file, NON-CONFORMING is a valid keyword for earlier versions of LINK and is now obsolete.
L4041 :	**HUGE segments not supported** This feature is not implemented in the linker.
L4042 :	**cannot open old version** An old version of the EXE file, specified with the OLD keyword in the module definitions file, could not be opened.

Number	Message, Cause/Action
L4043 :	**old version not segmented-executable format** The old version of the .EXE file, specified with the OLD keyword in the module definitions file, does not conform to segmented-executable format.
L4044 :	**minalloc feature is obsolete; ignored** A line in the SEGMENTS section of the .DEF file contained out-of-date syntax. Refer to "Module Definition Files" on page 3-5.
L4045 :	*name* **: is name of output file** A dynamic link library file was created without specifying an extension. In such cases, the linker supplies an extension of ".DLL." This is to warn you in case you expected a ".EXE" file to be generated.
L4046 :	**module name different from output file name** You specified a module name via the NAME or LIBRARY statement in the definitions file that is different from the output file (.EXE or .DLL) name. This will likely cause problems in BINDING the file or in using it in OS/2 mode. Rename the file to match the module name before it is executed.
L4050 :	**too many public symbols** 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 instead are listed in arbitrary order.
L4051 :	*filename* **: cannot find library** The linker could not find the specified file. Enter a new filename, a new path specification, or both.
L4053 :	**VM.TMP : illegal filename; ignored** VM.TMP appears as an object-file name. Rename the file and rerun the linker.

Number	Message, Cause/Action
L4054 :	*filename* **: cannot find file** The linker could not find the specified file. Enter a new filename, a new path specification, or both.

Library Manager Error Messages

Error messages produced by the IBM Library Manager, LIB, have one of the following formats:

filename|LIB: fatal error U1*xxx* : *messagetext*

or

filename|LIB: warning U4*xxx* : *messagetext*

The message begins with the input filename (*filename*), if one exists, or with the name of the utility.

Fatal Error Messages

Number	Message, Cause/Action
U1150 :	**page size too small** The page size of an input library is too small, which indicates a non-valid input .LIB file.
U1151 :	**syntax error : illegal file specification** You gave a command operator, such as a minus sign (–), without a module name following it.
U1152 :	**syntax error : option name missing** You gave a (/) without a value following it.
U1153 :	**syntax error : option value missing** You gave the /PAGESIZE option without a value following it.
U1154 :	**option unknown** An unknown option is given. Currently, LIB recognizes the /PAGESIZE option only.
U1155 :	**syntax error : illegal input** The given command did not follow correct LIB syntax.

Number	Message, Cause/Action
U1156 :	**syntax error** The given command did not follow correct LIB syntax.

U1157 :	**comma or new line missing**
	A comma or carriage return is expected at the command prompt, 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 of 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 is 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** Note the conditions when the error occurs and contact an IBM Authorized Dealer.
U1173 :	**internal failure** Note the conditions when the error occurs and contact an IBM Authorized Dealer.

Number	Message, Cause/Action
U1174 :	**mark : not allocated** Note the conditions when the error occurs and contact an IBM Authorized Dealer.
U1175 :	**free : not allocated** Note the conditions when the error occurs and contact an IBM Authorized Dealer.
U1180 :	**write to extract file failed** The disk or root directory is full. Delete or move files to make space.
U1181 :	**write to library file failed** The disk or root directory is full. Delete or move files to make space.
U1182 :	*filename* **: cannot create extract file** The disk or root directory is full, or the specified extract file already exists 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 is received prematurely in response to a prompt.
U1185 :	**cannot create new library** The disk or root directory is full, to the library file already exists with read-only protection. Make space on the disk or change the protection of the library file.
U1186 :	**error writing to new library** The disk or root directory is full. Delete or move files to make space.
U1187 :	**cannot open VM.TMP** The disk or root directory is full. Delete or move files to make space.
U1188 :	**cannot write to VM** Note the conditions when the error occurs and contact an IBM Authorized Dealer.
U1189 :	**cannot read from VM** Note the conditions when the error occurs and contact an IBM Authorized Dealer.

Number	Message, Cause/Action
U1190 :	**DOSALLOCHUGE failed** Note the conditions when the error occurs and contact an IBM Authorized Dealer.
U1191 :	**DOSREALLOCHUGE failed** Note the conditions when the error occurs and contact an IBM Authorized Dealer.
U1192 :	**DOSGETHUGESHIFT failed** Note the conditions when the error occurs and contact an IBM Authorized Dealer.
U1200 :	*name* **: invalid library header** The input library file has a non-valid format. It is either not a library file, or it has been corrupted.
U1203 :	*name* **: invalid object module near** *location* The module specified by *name* is not a valid object module.

Error Messages

Number	Message, Cause/Action
U2152 :	*filename* **: cannot create listing** The directory or disk is full, or the cross-reference listing file already exists 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 is not found in the input library.
U2157 :	*filename* **: cannot access file** LIB is unable to open the specified file.
U2158 :	*libraryname* **: invalid library header; file ignored** The input library has an incorrect format.
U2159 :	*filename* **: invalid format** *hexnumber*; **file ignored** The signature byte or word, *hexnumber*, of an input file is not one of the recognized types.

Warning Error Messages

Number	Message, Cause/Action
U4150 :	*modulename* **: module redefinition ignored** A module is specified to be added to a library, but a module with the same name is already in the library. Or, a module with the same name is found more than once in the library.
U4151 :	*'name'*: **symbol defined in module** *name*, **redefinition ignored** The specified symbol is defined in more than one module.
U4153 :	*number* **: page size too small; ignored** The value specified in the /PAGESIZE option is 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 :	*libraryname* **: output-library specification ignored** An output library is 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** For the reason indicated, LIB could not create 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** For the reason indicated, LIB could not create 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.

MAKE Error Messages

Error messages displayed by the IBM Program Maintenance Utility, MAKE, have one of the following formats:

filename (n)|MAKE : fatal error U1*xxx*:*messagetext*

or

MAKE : warning U4*xxx*: *message text*

Fatal Error Messages

Number	Message, Cause/Action
U1000 :	**= syntax error : ')' missing in macro invocation** A left parenthesis appeared without a matching right parenthesis in a macro invocation. The correct form is $(*name*).
U1001 :	**= syntax error : illegal character *'character'* in macro** A non-alphanumeric character other than an underscore appeared in a macro.
U1002 :	**syntax error : bad macro invocation '$'** A single dollar sign ($) appeared without a macro name associated with it. The correct form is $(*name*).
U1003 :	**syntax error : '=' missing in macro** The = 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 quotation mark.
U1017 :	**unknown directive *'directive'*** The *directive* specified is not a recognized directive.

Number	Message, Cause/Action
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 an IBM Authorized Dealer.
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 expected 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 else 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 option 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 = sign, implying a macro definition, was expected.
U1036 :	**syntax error : too many names to left of '='** Only one string is allowed to the left of a macro definition.

Number	Message, Cause/Action
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 an IBM Authorized Dealer.
U1039 :	**internal error : parser** Note the circumstances of the failure and notify an IBM Authorized Dealer.
U1040 :	**internal error : macro-expansion** Note the circumstances of the failure and notify an IBM Authorized Dealer.
U1041 :	**internal error : target building** Note the circumstances of the failure and notify an IBM Authorized Dealer.
U1042 :	**internal error : expression stack overflow** Note the circumstances of the failure and notify an IBM Authorized Dealer.
U1043 :	**internal error:temp file limit exceeded** Note the circumstances of the failure and notify an IBM Authorized Dealer.
U1044 :	**internal error:too many levels of recursion building a target** Note the circumstances of the failure and notify an IBM Authorized Dealer.
U1050 :	*user-specified text* The message specified with the !ERROR directive is displayed.
U1051 :	**usage:[-bcdeinpqrst -f makefile -x stderrfile] [macrodefs] [targets]** An error was made trying to invoke MAKE. 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 an IBM Authorized Dealer.
U1053 :	**file *'filename'* not found** The file was not found. The filename might not be properly specified in the makefile.

Number	Message, Cause/Action
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 script 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 script file failed.
U1059 :	**terminated by user** You pressed Ctrl-Brk to stop MAKE.
U1070 :	**cycle in macro definition '*macroname*'** A cycle was detected in the macro definition specified. This is an invalid definition.
U1071 :	**cycle in dependency tree for target '*targetname*'** A cycle was detected in the dependency tree for the specified target. This is invalid.
U1072 :	**cycle in include files *filenames*** A cycle was detected in the include files specified.
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.

Number	Message, Cause/Action
U1077 :	**'***filename***':return code** *value* The command invocation from MAKE failed. The nonzero return code *value* was returned.
U1078 :	**constant overflow at** *'directive'* A constant in *directive* 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.
U1081 :	**'***filename***':program not found** MAKE could not find the external command or program.
U1085 :	**can't mix implicit and explicit rules** A regular target was specified along with the target for a rule (which has the form *.sufx1.sufx2*). 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.

Number	Message, Cause/Action
U1091 :	**invalid suffixes in inference rule** The suffixes being used in the inference rule are invalid.
U1092 :	**too many names in rule** The rule cannot have more than one pair of extensions (*ext1.ext2*) as a target for the rule.
U1093 :	**can't mix special pseudotargets** It is illegal to list two or more pseudotargets together.

Warning Error Messages

Number	Message, Cause/Action
U4010 :	**nonstandard suffix on command file** *'filename'* The program to be executed has an extension that is not .COM, .EXE, or .BAT.
U4011 :	**command file can only be invoked from command line** A command file cannot be invoked from within another command file. Such an invocation would be ignored.
U4012 :	**resetting value of special macro** *'macroname'* A macro such as $(MAKE) had its value changed from within a makefile.
U4013 :	**Unable to find tool initialization file.** MAKE unable to locate filename TOOLS.INI in current directory or search path.
U4015 :	**no match found for wildcard** *'filename'* There are no file names to match the specified target or dependent file with the wildcard characters: asterisk (*) or 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.

Number	Message, Cause/Action
U4018 :	**special macro undefined** *'macroname'* The special macro *macroname* is undefined.
U4019 :	**Filename '%s' too long; truncated to 8.3.** A target file is greater than eleven characters truncated to DOS format.
U4020 :	**Removed target '%s'** **A processing dependency file was interrupted by the user or the system, creating a corrupt target file. MAKE has removed the target file.**

EXEMOD Error Messages

Error messages from the IBM EXE File Header Utility, EXEMOD, have one of the following formats:

filename|EXEMOD:fatal error U1*xxx* : *messagetext*

or

filename|EXEMOD:warning U4*xxx* : *messagetext*

The message begins with the input filename (*filename*), if one exists, or with the name of the utility.

Fatal Error Messages

Number	Message, Cause/Action
U1050 :	**usage : exemod file [-/h] [-/stack n] [-/max n] [-/min n]** You did not specify the EXEMOD command properly. Try again using the syntax shown. Note that the option indicator can be either a slash or a dash. The single brackets in the error message show your optional choice.
U1051 :	**invalid .EXE file : bad header** The specified input file is not an executable file or it has an incorrect file header.

Number	Message, Cause/Action
U1052 :	**invalid .EXE file : actual length less than reported** The second and third fields in the input-file header indicate a file size greater than the actual size.
U1053 :	**cannot change load-high program** When the minimum allocation value and the maximum allocation value are both zero, you cannot change the file.
U1054 :	**file not .EXE** EXEMOD adds the .EXE extension to any filename without an extension. In this case, no file with the given name and an .EXE extension was found.
U1055 :	*filename* **: cannot find file** The file specified by *filename* was not found.
U1056 :	*filename* **: permission denied** The file specified by *filename* is a read-only file.

Warning Error Messages

Number	Message, Cause/Action
U4050 :	**packed file** The given file is a packed file. This is a warning only.
U4051 :	**minimum allocation less than stack; correcting minimum** If the minimum allocation value is not enough to accommodate the stack (either the original stack request or the changed request), the minimum allocation value is adjusted. This is a warning message only; the change is still performed.
U4052 :	**minimum allocation greater than maximum; correcting maximum** If the minimum allocation value is greater than the maximum allocation value, the maximum allocation value is adjusted. This is a warning message only; the change is still performed.

Errno Value Error Messages

This section lists and describes the values the *errno* variable can be set to when an error occurs in a call to a library routine. Note that only some routines set the *errno* variable. See Chapter 5 in *IBM C/2 Language Reference* for the routines that set *errno*.

An error message is associated with each *errno* value. This message, along with a message that you supply, can be printed by using the **perror** function.

The value of *errno* reflects the error value for the last call that set *errno*. The *errno* value is not automatically cleared by later successful calls. Therefore, test for errors and print error messages immediately after a call to obtain accurate results.

The errno.h include file contains the definitions of the *errno* values. However, not all of the definitions given in errno.h are used under DOS. This section lists only the *errno* values used under DOS. For the complete listing of values, see the errno.h include file.

Errno Values

The following list gives the *errno* values used under DOS, the system error message corresponding to each value, and a brief description of the circumstances that caused the error.

Value	Message, Cause/Action
E2BIG :	**Arg list too long** The argument list exceeds 128 bytes, or the space required for the environment information exceeds 32KB.

Value	Message, Cause/Action
EACCESS :	**Permission denied** The permission setting of the file does not allow the specified access. This error can occur in a variety of circumstances. It signifies that an attempt was made to get access to a file (or, in some cases, a directory) in a way that is incompatible with the attributes of the file. For example, this error can occur when an attempt is made to read from a file that is not open, to open an existing read-only file for writing, or to open a directory instead of a file. Under DOS 3.30 and OS/2, EACCESS can also indicate a locking or sharing violation. This error can also occur in an attempt to rename a file or directory or to remove an existing directory.
EBADF :	**Bad file number** The specified file handle is not a valid file handle value, does not refer to an open file, or an attempt was made to write to a file or device opened for read access (or the reverse).
EDEADLOCK :	**Resource deadlock would occur** Locking violation: the file cannot be locked after 10 attempts.
EDOM :	**Math argument** The argument to a math function is not in the domain of the function.
EEXIST :	**File exists** The O_CREAT and O_EXCL flags are specified when opening a file, but the named file already exists.
EINVAL :	**Non-valid argument** A non-valid value was given for one of the arguments to a function. For example, the value given for the origin when positioning a file pointer is before the beginning of the file.

Value	Message, Cause/Action
EMFILE :	**Too many open files** No more file handles are available, so no more files can be opened.
ENOENT :	**No such file or directory** The specified file or directory does not exist or cannot be found. This message can occur whenever a specified file does not exist or a component of a pathname does not specify an existing directory. It can also occur in OS/2 mode if a filename exceeds 8 characters or if the filename extension exceeds 3 characters.
ENOEXEC :	**Exec format error** An attempt is made to run a file that is not executable or that has a non-valid executable file format.
ENOMEM :	**Not enough core** Not enough storage is available. This message can occur when not enough storage is available to run a child process or when the allocation request in an **sbrk** or **getcwd** call cannot be satisfied.
ENOSPC :	**No space left on the device** No more space for writing is available on the device. For example, the disk is full.
ERANGE :	**Result too large** An argument to a math function is too large, resulting in partial or total loss of significance in the result. This error can also occur in other functions when an argument is larger than expected (for example, when the pathname argument to the **getcwd** function is longer than expected).
EXDEV :	**Cross-device link** An attempt was made to move a file to a different device (using the **rename** function).

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Z

Numerics

Special Characters

Continued from inside front cover.

SUCH WARRANTIES ARE IN LIEU OF ALL OTHER WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Some states do not allow the exclusion of implied warranties, so the above exclusion may not apply to you.

LIMITATION OF REMEDIES

IBM's entire liability and your exclusive remedy shall be as follows:

1) IBM will provide the warranty described in IBM's Statement of Limited Warranty. If IBM does not replace defective media or, if applicable, make the Program operate as warranted or replace the Program with a functionally equivalent Program, all as warranted, you may terminate your license and your money will be refunded upon the return of all of your copies of the Program.
2) For any claim arising out of IBM's limited warranty, or for any other claim whatsoever related to the subject matter of this Agreement, IBM's liability for actual damages, regardless of the form of action, shall be limited to the greater of $5,000 or the money paid to IBM, its Authorized Dealer or its approved supplier for the license for the Program that caused the damages or that is the subject matter of, or is directly related to, the cause of action. This limitation will not apply to claims for personal injury or damages to real or tangible personal property caused by IBM's negligence.
3) In no event will IBM be liable for any lost profits, lost savings, or any incidental damages or other consequential damages, even if IBM, its Authorized Dealer or its approved supplier has been advised of the possibility of such damages, or for any claim by you based on a third party claim.

Some states do not allow the limitation or exclusion of incidental or consequential damages so the above limitation or exclusion may not apply to you.

GENERAL

You may terminate your license at any time by destroying all your copies of the Program or as otherwise described in this Agreement.

IBM may terminate your license if you fail to comply with the terms and conditions of this Agreement. Upon such termination, you agree to destroy all your copies of the Program.

Any attempt to sublicense, rent, lease or assign, or, except as expressly provided herein, to transfer any copy of the Program is void.

You agree that you are responsible for payment of any taxes, including personal property taxes, resulting from this Agreement.

No action, regardless of form, arising out of this Agreement may be brought by either party more than two years after the cause of action has arisen except for breach of the provisions in the Section entitled "License" in which event four years shall apply.

This Agreement will be construed under the Uniform Commercial Code of the State of New York.

Z125-3301-02 4/87

© IBM Corp. 1988
All rights reserved.

International Business
Machines Corporation,
Boca Raton,
Florida 33429-1328

Printed in the
United States of America

15F0383