

⌋

SINIX Open Desktop

Development System

⌋ C Language Guide Part 1

⌋

⌋

1

2

3

4

Order-No. U5754-J-Z95-1-7600
Printed in the Federal Republic of Germany
7110 AG 2900.5 (8890)

SINIX® Open Desktop
Copyright © Siemens AG 1990
All right reserved

Base:
OPEN DESKTOP™
©1983 - 1989 The Santa Cruz Operation, Inc.

Delivery subject to availability;
right of technical modifications reserved.

Published by Bereich
Daten- und Informationstechnik
Postfach 830951, D-8000 München 83
Siemens Aktiengesellschaft

SCO UNIX[®] System V/386

Development System

C User's Guide

The Santa Cruz Operation, Inc.



Portions © 1980, 1981, 1982, 1983, 1984, 1985, 1986, 1987, 1988, 1989 Microsoft Corporation.

All rights reserved.

Portions © 1989 AT&T.

All rights reserved.

Portions © 1983, 1984, 1985, 1986, 1987, 1988, 1989 The Santa Cruz Operation, Inc.

All rights reserved.

No part of this publication may be reproduced, transmitted, stored in a retrieval system, nor translated into any human or computer language, in any form or by any means, electronic, mechanical, magnetic, optical, chemical, manual, or otherwise, without the prior written permission of the copyright owner, The Santa Cruz Operation, Inc., 400 Encinal, Santa Cruz, California, 95062, U.S.A. Copyright infringement is a serious matter under the United States and foreign Copyright Laws.

The copyrighted software that accompanies this manual is licensed to the End User only for use in strict accordance with the End User License Agreement, which should be read carefully before commencing use of the software. Information in this document is subject to change without notice and does not represent a commitment on the part of The Santa Cruz Operation, Inc.

USE, DUPLICATION, OR DISCLOSURE BY THE UNITED STATES GOVERNMENT IS SUBJECT TO RESTRICTIONS AS SET FORTH IN SUBPARAGRAPH (c) (1) OF THE COMMERCIAL COMPUTER SOFTWARE -- RESTRICTED RIGHTS CLAUSE AT FAR 52.227-19 OR SUBPARAGRAPH (c) (1) (ii) OF THE RIGHTS IN TECHNICAL DATA AND COMPUTER SOFTWARE CLAUSE AT DFARS 52.227-7013. "CONTRACTOR/ MANUFACTURER" IS THE SANTA CRUZ OPERATION, INC., 400 ENCINAL STREET, P.O. BOX 1900, SANTA CRUZ, CALIFORNIA, 95061, U.S.A.

Microsoft, MS-DOS, and XENIX are registered trademarks of Microsoft Corporation.

DEC, PDP, VAX, and VT100 are trademarks of Digital Equipment Corporation.

Intel is a registered trademark of Intel Corporation.

UNIX is a registered trademark of AT&T.

1

2

3

4

Contents

1 Introduction

- Overview 1-1
- About This Guide 1-2
- New Features 1-4
- Notational Conventions 1-6
- Books about C 1-9

2 Compiling with the cc Command

- Introduction 2-1
- The Basics: Compiling and Linking C Programs 2-2
- Using cc Options 2-6

3 Linking with the cc Command

- Introduction 3-1
- The Default Linking Process 3-2
- Passing Linker Information: The -link Option 3-3

4 Running C Programs on System V

- Introduction 4-1
- Passing Command-Line Data to a Program 4-2

5 Working with Memory Models

- Introduction 5-1
- Near, Far, and Huge Addressing 5-4
- Using the Standard Memory Models 5-6
- Using the near, far, and huge Keywords 5-14
- Creating Customized Memory Models 5-25
- Setting the Data Threshold 5-30
- Naming Modules and Segments 5-31
- Specifying Text and Data Segments 5-34

6 Improving Program Speed

- Introduction 6-1
- Using Register Variables 6-2
- Optimization Options and Pragmas 6-4
- Choosing the Function-Calling Convention 6-7

Efficiency in Large Data Models 6-8
Efficiency in Large Code Models 6-10

7 Object and Executable File Formats

Introduction 7-1
iAPX.....286 and386 System Architecture 7-2
The Intel Object Module Format 7-4
Definition of Terms 7-6
Module Identification and Attributes 7-9
Segment Definition 7-10
Segment Addressing 7-11
Symbol Definition 7-12
Indices 7-13
Conceptual Framework for Fixups 7-14
Self-Relative Fixups 7-19
Segment-Relative Fixups 7-20
Record Order 7-22
Introduction to the Record Formats 7-24
Numeric List of Record Types 7-50
Type Representations for Communal Variables 7-51
The Segmented x.out Format 7-54

8 C Language Compatibility with Assembly Language

Introduction 8-1
C Calling Sequence for 8086/80286 8-2
Entering an 8086/80286 Assembly Routine 8-3
8086/80286 Return Values 8-4
Exiting an 8086/80286 Routine 8-5
8086/80286 Program Example 8-6
80386 C-Language Calling Sequence 8-7
Entering an 80386 Assembly-Language Routine 8-8
80386 Return Values 8-9
Exiting an 80386 Routine 8-11
80386 Program Example 8-12

9 Error Processing

Introduction 9-1
Using the Standard Error File 9-2
Using the errno Variable 9-3
Printing Error Messages 9-4
Using Error Signals 9-5
Encountering System Errors 9-6

10 Common Object File Format (COFF)

The Common Object File Format (COFF)	10-1
Definitions and Conventions	10-3
File Header	10-5
Optional Header Information	10-7
Section Headers	10-9
Sections	10-12
Relocation Information	10-13
Line Numbers	10-15
Symbol Table	10-17
String Table	10-41
Access Routines	10-42

A Converting from Previous Versions of the Compiler

Introduction	A-1
Differences between Versions 5.1 and 5.0	A-2
Differences between Versions 5.0 and 4.0	A-4
Differences between Versions 4.0 and 3.0	A-8

B Writing Portable Programs

Introduction	B-1
Program Portability	B-3
Machine Hardware	B-4
Compiler Differences	B-11
Environment Differences	B-16
Portability of Data	B-17
Type-Size Summary	B-18
Byte-Ordering Summary	B-20

C Writing Programs for Read-Only Memory

Introduction	C-1
System V Dependent Library Routines	C-2

D C Error Messages and Exit Codes

Introduction	D-1
Command-Line Error Messages	D-2
Compiler Error Messages	D-7



Chapter 1

Introduction

Overview 1-1

About This Guide 1-2

New Features 1-4

Notational Conventions 1-6

Books about C 1-9



Overview

The C language is a powerful general-purpose programming language that can generate efficient, compact, and portable code. The Microsoft® C Optimizing Compiler (cc) for the UNIX System V® operating system is a full implementation of the C language as defined by its authors, Brian W. Kernighan and Dennis M. Ritchie, in *The C Programming Language*. Microsoft is actively involved in the development of the ANSI (American National Standards Institute) standard for the C language; this version of Microsoft C for UNIX System V anticipates and conforms to the forthcoming standard in many areas.

The Microsoft C Compiler offers several important features to help you increase the efficiency of your C programs. You can choose among five standard memory models (small, medium, compact, large, and huge) to set up the combination of data and code storage that best suits your program. For flexibility and even greater efficiency, the C Compiler allows you to “mix” memory models by using special declarations in your program.

The C language itself does not provide such standard features as input and output capabilities and string-manipulation features. These capabilities are provided as part of the run-time library of functions that accompanies the C Compiler.

Compared with other programming languages, Microsoft C is extremely flexible concerning data conversions and nonstandard constructions. The C Compiler offers several levels of warnings to help you control this flexibility; programs in an early stage of development can be processed using the full warning capabilities of the compiler to catch mistakes and unintentional data conversions. An experienced C programmer can use a lower warning level for programs that contain intentionally nonstandard constructions. For more information about this feature, see the “Compiling with the cc Command” chapter in this guide.

About This Guide

This guide explains how to use the C Compiler to compile, link, and run C programs on UNIX System V. The guide assumes that you are familiar with the C language and with UNIX System V, and that you know how to create and edit a C-language source file on your system.

If you have questions about the C language, turn to the *C Language Reference* included in this package. The *C Library Guide* documents the run-time library routines you can use in your C programs.

The remaining chapters of the *C User's Guide* are described below:

Chapter 2, "Compiling with the cc Command," describes how to compile a program using the cc compiler driver. This chapter describes the options most commonly used to control preprocessing, compiling, and output of files.

Chapter 3, "Linking with the cc Command," describes how to link object files using the cc command. This chapter explains how the linker searches for libraries, shows how to specify libraries for linking, and describes the linker options that can be used for C programs.

Chapter 4, "Running C Programs on UNIX System V," explains how to run your executable program file and discusses features specific to the UNIX System V implementation of C. This chapter tells how to pass data from UNIX System V to a program at execution time and how to return an exit code from your program to UNIX System V.

Chapter 5, "Working with Memory Models," describes methods of managing memory models. These methods are useful for writing programs that use more than 64K (kilobytes) of code or data. This chapter also discusses "mixed-model" programming (combining features from the five standard memory models).

Chapter 6, "Improving Program Speed," gives suggestions and hints for maximizing program speed.

Chapter 7, "Object and Executable File Formats," describes the system architecture of the 80x86 microprocessor family, the object module format that the C compiler follows, and the format of the **x.out** file in a segmented environment.

About This Guide

Chapter 8, “C Language Compatibility with Assembly Language,” describes how you can embed assembly-language subroutines within C-language programs.

Chapter 9, “Error Processing,” describes how to process errors detected in calls to the C library routines and explains the functions and variables a program may use to respond to these errors.

Chapter 10, “The Common Object File Format (COFF),” describes the features and contents of *COFF* files.

Appendix A, “Converting from Previous Versions of the Compiler,” summarizes the differences between Version 5.1 of the C Compiler and previous versions. This appendix gives instructions for converting programs written for versions prior to 5.1 to the format accepted by Version 5.1.

Appendix B, “Writing Portable Programs,” lists some of the C-language features that are implementation-dependent, and offers suggestions for increasing program portability.

Appendix C, “Writing Programs for Read-Only Memory,” gives information about modifying start-up code and initializing floating-point support for programs that will be put in read-only memory.

Appendix D, “Error Messages and Exit Codes,” lists and describes the error messages and exit codes generated by the C Compiler and by the `cc` command. It also lists and explains run-time error messages produced by executable programs written in C.

New Features

Several useful features have been added to Version 5.1 of the C Compiler. This section summarizes features added since Version 5.0. For information about differences between Version 5.1 and versions prior to 5.0, see the “Converting from Previous Versions of the Compiler” appendix in this guide.

New features include the following:

Feature	Description								
New cc options	<table border="1"> <thead> <tr> <th>Option</th> <th>Action</th> </tr> </thead> <tbody> <tr> <td>-S</td> <td>Generates an assembly-language source file for the Macro Assembler, <i>masm</i>(CP).</td> </tr> <tr> <td>-xenix</td> <td>Produces object and/or executable files using the Intel Object Module Format (OMF).</td> </tr> <tr> <td>-x2.3</td> <td>Produces object and/or executable files using the Intel Object Module Format (OMF) and the XENIX System V/Release 2.3 run-time library.</td> </tr> </tbody> </table>	Option	Action	-S	Generates an assembly-language source file for the Macro Assembler, <i>masm</i> (CP).	-xenix	Produces object and/or executable files using the Intel Object Module Format (OMF).	-x2.3	Produces object and/or executable files using the Intel Object Module Format (OMF) and the XENIX System V/Release 2.3 run-time library.
	Option	Action							
	-S	Generates an assembly-language source file for the Macro Assembler, <i>masm</i> (CP).							
	-xenix	Produces object and/or executable files using the Intel Object Module Format (OMF).							
-x2.3	Produces object and/or executable files using the Intel Object Module Format (OMF) and the XENIX System V/Release 2.3 run-time library.								
New pragmas	<table border="1"> <thead> <tr> <th>Pragma</th> <th>Action</th> </tr> </thead> <tbody> <tr> <td>comment</td> <td>Places a comment record in the object file.</td> </tr> <tr> <td>Data_seg</td> <td>Specifies the data-segment name used by functions that load their own data segments. The named segment also contains all data that would normally be allocated in the DATA segment.</td> </tr> <tr> <td>linesize</td> <td>Sets the number of characters per line in the source listing.</td> </tr> </tbody> </table>	Pragma	Action	comment	Places a comment record in the object file.	Data_seg	Specifies the data-segment name used by functions that load their own data segments. The named segment also contains all data that would normally be allocated in the DATA segment.	linesize	Sets the number of characters per line in the source listing.
	Pragma	Action							
	comment	Places a comment record in the object file.							
Data_seg	Specifies the data-segment name used by functions that load their own data segments. The named segment also contains all data that would normally be allocated in the DATA segment.								
linesize	Sets the number of characters per line in the source listing.								

New Features

message	Sends a message to the standard output without terminating the compilation.
page	Places a formfeed character(s) in the source listing.
pagesize	Sets the number of lines per page in the source listing.
skip	Skips the specified number of lines in the source listing. Places a comment record in the object file.
subtitle	Specifies a subtitle for the source listing.
title	Specifies a title for the source listing.

Notational Conventions

The following notational conventions are used throughout this guide:

Example of Convention	Description of Convention
Examples	<p>The typeface shown in the left column is used to simulate the appearance of information that would be printed on the screen or by a printer. For example, the following command line is printed in this special typeface:</p> <pre data-bbox="540 707 917 734">cc -Foout.o -DTRUE=1 file.c</pre> <p>When this command line is discussed in text, items appearing on the command line, such as <i>out.o</i>, also appear in the special typeface.</p>
Language elements	<p>Bold type indicates elements of the C language that must appear in source programs as shown. Text that is normally shown in bold type includes operators, keywords, library functions, commands, options, and preprocessor directives.</p> <p>Examples are shown below:</p> <pre data-bbox="470 1111 823 1200">+= #if defined() int if -Fa fopen main sizeof</pre>
ENVIRONMENT, VARIABLES, and MACROS	<p>Bold capital letters are used for environment variables, symbolic constants, and macros.</p>
<i>placeholders</i>	<p>Words in italics are placeholders, representing a variable that you must supply in command-line examples, option specifications, and in the text. Consider the following option:</p> <pre data-bbox="540 1487 682 1514">-H <i>number</i></pre>

Note that *number* is italicized to indicate that it represents a general form for the **-H** option. In an actual command, you would supply a particular number for the placeholder *number*.

Occasionally, italics are also used to emphasize particular words in the text.

Missing code

Vertical ellipses are used in program examples to indicate that a portion of the program is omitted. For instance, in the following excerpt, the ellipses between the statements indicate that intervening program lines occur but are not shown:

```
count = 0;  
.  
.  
.  
*pc++;
```

[*optional items*]

Brackets enclose optional fields in command-line and option specifications. Consider the following option specification:

-D*identifier*[=*string*]

The placeholder *identifier* indicates that you must supply an identifier when you use the **-D** option. The outer brackets indicate that you are not required to supply an equal sign (=) and a string following the identifier. The inner brackets indicate that you are not required to enter a string following the equal sign, but if you do supply a string, you must also supply the equal sign.

Single brackets are used in C-language array declarations and subscript expressions. For instance, *a[10]* is an example of brackets in a C subscript expression.

Notational Conventions

Repeating elements...

Horizontal ellipses are used in syntax examples to indicate that more items having the same form may be entered. For example, in the Bourne shell, several paths can be specified in the **PATH** command, as shown in the following syntax:

```
PATH[=]path[;path]...
```

{*choice1*|*choice2*}

Braces and a vertical bar indicate that you have a choice of two or more items. Braces enclose the choices, and vertical bars separate them. You must choose one of these items unless all of them are also enclosed in square brackets.

For example, the **-W** (warning-level) compiler option has the following syntax:

```
-W {0 | 1 | 2 | 3}
```

You can use **-W1**, **-W2**, or **-W3** to display different levels of warning messages or **-W0** to suppress all warning messages.

“Defined terms”

Quotation marks set off terms defined in the text. For example, the term “far” appears in quotation marks the first time it is defined.

Some C constructs require quotation marks. Quotation marks required by the language have the form " " rather than “ ”. For example, a C string used in an example would be shown in the following form:

```
"abc"
```

KEY+KEY

Small capital letters are used for the names of keys and key sequences, such as ENTER and CTRL+C. Key sequences to be pressed simultaneously are indicated by the key names in small caps separated by a plus sign (CTRL+C).

Books about C

The manuals in this documentation package provide a complete programmer's reference for C. They do not, however, teach you how to program in C. If you are new to C or to programming, you may want to familiarize yourself with the language by reading one or more of the following books:

Hancock, Les, and Morris Krieger. *The C Primer*. New York: McGraw-Hill Book Co., Inc., 1982.

Hansen, Augie. *Proficient C*. Bellevue, Washington: Microsoft Press, 1986.

Harbison, Samuel P., and Greg L. Steele. *C: A Reference Manual*. Englewood Cliffs, New Jersey: Prentice-Hall Software Series, 1987.

Kernighan, Brian W., and Dennis M. Ritchie. *The C Programming Language*. Englewood Cliffs, New Jersey: Prentice-Hall, Inc., 1978.

Kochan, Stephen. *Programming in C*. Hasbrouck Heights, New Jersey: Hayden Book Company, Inc., 1983.

Plum, Thomas. *Learning to Program in C*. Cardiff, New Jersey: Plum Hall, Inc., 1983.

Schildt, Herbert. *C Made Easy*. Berkeley, California: Osborne McGraw Hill, 1985.

Schustack, Steve. *Variations in C*. Bellevue, Washington: Microsoft Press, 1985.

These books are listed for your convenience only.



Chapter 2

Compiling with the cc Command

Introduction 2-1

The Basics: Compiling and Linking C Programs 2-2
The cc Command 2-2

Using cc Options 2-6

Setting Processor and Memory Model (-M) 2-6

Specifying Source Files (-Tc) 2-8

Compiling without Linking (-c) 2-9

Naming the Object File (-Fo) 2-9

Naming the Executable File (-Fe) (-o) 2-10

Creating Listings 2-11

Controlling the Preprocessor 2-28

Checking for Program Errors 2-35

Preparing for Debugging (-Zi, -Od) 2-40

Optimizing 2-41

Enabling/Disabling Language Extensions (-Ze, -Za) 2-53

Packing Structure Members (-Zp) 2-54

Setting the Stack Size (-F) 2-56

Restricting the Length of External Names (-nl) 2-57

Labeling the Object File (-V) 2-57

Changing the Default char Type (-J) 2-58

Controlling the Calling Convention (-Gc) 2-58

Compiling Programs for DOS Environment (-dos, -FP) 2-60

Displaying Compiler Passes (-d, -z) 2-61

Producing OMF Object and Executable Files (-xenix) 2-62

Miscellaneous Pragmas 2-62

Predefined Macro Names 2-65



Introduction

This chapter explains how to compile and link using the `cc` command and discusses commonly used `cc` options. The `cc` command is the only command you need to compile and link your C source files. The `cc` command executes the three compiler passes, then automatically invokes the linker, `ld`, to link your files.

Using the `cc` options described in this chapter, you can control and modify the tasks performed by the command. For example, you can direct `cc` to create an object-listing file or a preprocessed listing. Options also let you give information that applies to the compilation process; you can specify the definitions for manifest (symbolic) constants and macros, and the kinds of warning messages you want to see.

The `cc` command automatically optimizes your program. You never have to give an optimizing instruction unless you want to change the way `cc` optimizes, request more sophisticated optimizations, or disable optimization altogether. For more information on these choices, see the “Optimizing” section in this chapter.

“The Basics: Compiling and Linking C Programs” explains the basic use of the `cc` command to produce an executable program.

“Using `cc` Options,” describes the `cc` options.

For information about linking object files and libraries using the `cc` command, see the “Linking with the `cc` Command” chapter of this guide.

For a discussion of the `cc` options that control memory models, see the “Working with Memory Models” chapter in this guide.

For a summary of the `cc` command and its options, see the *C Language Reference*.

The Basics: Compiling and Linking C Programs

This section explains how to use `cc` to compile and link C programs and discusses the rules and conventions that apply to file names and options used with `cc`.

The `cc` Command

The `cc` command has the following form:

```
cc [option]... file... [option... file...]...[-link[link-libinfo]]
```

Each *option* is one of the command-line options described in the “Using `cc` Options” section, the “Working with Memory Models” chapter, or the “Improving Program Speed” chapter of this guide.

Each *file* names a source or object file to be processed or a library to be searched at link time. See the description on “Specifying Source and Object Files” later in this section for information about specifying source and object files.

The `cc` command automatically specifies the appropriate library to be used during linking. You can use the `-link` option with the optional *link-libinfo* field to specify additional or different libraries, library search paths, and options to be used during linking. You can also specify linker options in the *linkoptions* field. For information about specifying different libraries and linker options, see the “Linking with the `cc` Command” chapter of this guide.

You can give any number of options, file names, and library names on the command line, provided that the command line does not exceed 128 characters.

COFF and OMF

This version of the C Compiler can produce object and/or executable files that use either of two different binary file formats: COFF (Common Object File Format) and OMF (Intel Object Module Format). COFF is the most widely used binary file format. OMF files are produced using the

The Basics: Compiling and Linking C Programs

-xenix option with the compiler. SCO UNIX System V can execute either file format by reading the file header and acting accordingly. Certain system calls behave differently in OMF files because they follow UNIX System V rather than XENIX system conventions. The COFF and OMF formats are described by their corresponding header files: `/usr/include/a.out.h` and `/usr/include/sys/x.out.h` respectively.

Note

The default file name produced by the linker is `a.out` regardless of the actual file format used. Any mention of `x.out` in this guide is referring only to the *format* of OMF executable files.

Table 2.1 shows the tools used with various options to the Microsoft C Compiler, and the type of object/executable file that results.

Table 2.1
Options, Tools, and Resulting Files

Option	MS C Compiler	Assembler	Link Editor	obj format
none	MS	masm	<i>ld</i>	COFF
-c	Creates linkable x.out object file	masm	n/a	COFF
-S	Creates assembly source listing	masm	n/a	COFF
-Fa	Creates assembly source listing	masm	<i>ld</i>	COFF
-xenix	Creates XENIX programs in OMF format	masm	<i>ld</i>	OMF
-S -xenix	Creates XENIX assembly source listing	masm	n/a	OMF
-Fa -xenix	Creates XENIX assembly source listing	masm	<i>ld</i>	OMF

Specifying Source and Object Files

The `cc` command can process source files, object files, library files, or any combination of these. It uses the file-name extension (the period plus any letters that follow it) to determine what kind of processing the file needs, as shown in the following list:

- If the file has a `.c` extension, `cc` compiles the file.

The Basics: Compiling and Linking C Programs

- If the file has a `.o` extension, `cc` processes the file by invoking the linker.
- If the file has a `.a` extension, `cc` assumes the file is a library and passes it to the linker to be searched, unless the `-c` option is given to suppress linking. For a description of the `-c` option, see the section on “Compiling without Linking” under the section “Using cc Options.”
- If the file has the `.asm` extension, it is passed to `masm`.
- If the extension is omitted, `cc` assumes an extension of `.o`. If the extension is anything other than `.c`, `.o`, or `.a`, `cc` assumes the file is an object file unless the file name is specified in association with the `-Tc` option. If the file name is specified with the `-Tc` option, `cc` assumes the file is a C source file. For a description of the `-Tc` option, see the section on “Specifying Source Files” under the section “Using cc Options.”

Examples

```
cc a.c b.c c.o d.o
```

This command line compiles the files `a.c` and `b.c`, creating object files named `a.o` and `b.o`. These object files are then linked with the object files `c.o` and `d.o` to form an executable file named `a.out`.

```
cc a.c b.c c.o -Tcd.src
```

This command performs the same operations as the preceding command line, except that the `-Tc` option indicates that `d.src` is a source file, not an object file. Thus, the files `a.c`, `b.c`, and `d.src` are compiled, creating object files named `a.o`, `b.o`, and `d.o`. These object files are then linked with `c.o` to form an executable file named `a.out`.

Creating Executable Files

When `cc` compiles source files, it creates object files. By default, these object files use the COFF format and have the same base names as the corresponding source files, but with the extension `.o` instead of `.c`. (The base name of a file extension is the portion of the name preceding the period, but excluding the path specification, if any.) After compilation, `cc` runs a conversion program, `cvtomf`, over the object file to convert it into COFF format. For more information about the `cvtomf` conversion program, refer to the manual page `cvtomf (C)`. The converted object file can

The Basics: Compiling and Linking C Programs

now be linked using the AT&T link editor, **ld**. The **-xenix** option suppresses the conversion.

Unless the **-c** option is given, **cc** links these object files, along with any **.o** files you give on the command line, to form an executable file. If only **.o** files are given on the command line, **cc** skips the compilation stage and simply links the files.

Using cc Options

The `cc` command offers a large number of command options to control and modify the compiler's operation. Options begin with a dash (-) and contain one or more letters.

Options can appear anywhere on the `cc` command line. In general, an option applies to all files that follow it on the command line, and it does not affect files preceding it. However, not all options follow this rule; see the discussion of a particular option for information on its behavior. Keep in mind that `cc` options apply only to the compilation process. Unless specifically noted, options do not affect any object files given on the command line. The remainder of this section describes many of the options applicable to `cc`. For a concise list of all possible options, refer to the manual page, `cc(3P)`.

Setting Processor and Memory Model (-M)

The `-M` option sets the program configuration. This configuration defines the program's memory model, word order, and data threshold. It also enables C-language enhancements such as the use of the full 286 instruction set and special keywords.

```
cc -Mstring special.c
```

The *string* contains the argument that defines the configuration. It may be any combination of the following (though `s`, `m`, `c`, `l`, `h`, and `0`, `1`, `2`, `3` are mutually exclusive):

- `s` Create a small model program. This is the default.
- `m` Create a middle model program.
- `c` Create a compact model program.
- `l` Create a large model program.
- `h` Create a huge model program.

- e Enable the keywords: **far**, **near**, **huge**, **pascal** and **fortran**. Also enables certain non-ANSI extensions necessary to ensure compatibility with existing versions of the C compiler.
- 0 Use only 8086 instructions for code generation. This is the default on 8086/80186/80286 systems.
- 1 Use the extended 80186 instruction set.
- 2 Use the extended 80286 instruction set.
- 3 Use the extended 80386 instruction set. This is the default on 80386 systems.
- tnum* Causes all static and global data items whose size is greater than *num* bytes to be allocated to a new data segment. *Num*, the data "threshold," defaults to 32,767. This option can only be used in large model programs (-M1). Its main use is to move data out of the near data segment to allow room for the stack.

```
cc -M1 -Mt12 recursive.c
```

- d Do not assume (during compilation) that the registers **SS** and **DS** will have the same contents at run time. *Warning:* This option has no library or run-time support on UNIX System V. It will **not** cause the stack to be put in a separate segment. It may be of use for DOS cross-development.

-M3 is the default on 80386 systems. Although it is usually advantageous to enable the appropriate instruction set, you are not required to do so. If you have an 80286 processor, for example, but you want your code to be able to run on an 8086, you should not use the 80186/80188 or 80286 instruction set.

Note

The **m**, **c**, **l**, **h**, **b**, **t**, and **d** arguments are not compatible with the -M3 option. The **s** and **e** arguments are compatible with -M0, -M1, -M2, or -M3.

Using cc Options

For a complete description of memory models and segment options, see the “Working with Memory Models” chapter in this guide.

The memory-model option you choose determines the name of the standard libraries that **cc** places in the object file it creates. These libraries are then considered the default libraries, since the linker searches for them by default.

Table 2.2 shows each memory-model option and the corresponding library name that **cc** embeds in the object file.

Table 2.2
cc Options and Default Libraries

Memory-Model Option	Default Libraries
-Ms	Slibc.a Slibcfp.a
-Mm	Mlibc.a Mlibcfp.a
-Mc	Clibc.a Clibcfp.a
-Ml or -Mh	Llibc.a Llibfp.a

Specifying Source Files (-Tc)

Option

-Tc *sourcefile*

The **-Tc** option tells the **cc** command that the given file is a C source file. One or more spaces can appear between **-Tc** and the source-file name.

If this option does not appear, **cc** assumes that files with the extension **.c** are C source files, files with the extension **.a** are libraries, and files with any other extension or with no extension are object files. If you use the **-Tc** option, **cc** treats the given file 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**.

If you have to specify more than one source file with an extension other than `.c`, you must specify each source file in a separate `-Tc` option.

Example

```
cc main.c -Tc test.prg -Tc collate.prg print.prg
```

In this example, the `cc` command compiles the three source files `main.c`, `test.prg`, and `collate.prg`. Since the file `print.prg` is given without a `-Tc` option, `cc` treats it as an object file. Thus, after compiling the three source files, `cc` links the object files `main.o`, `test.o`, `collate.o`, and `print.prg`.

Compiling without Linking (-c)

Option

`-c`

The `-c` (for “compile-only”) option suppresses linking. Source files given on the command line are compiled, but the resulting object files are not linked, no executable file is created, and any object files specified on the command line are ignored. This option is useful when you are compiling individual source files that do not make up a complete program.

The `-c` option applies to the entire `cc` command line, regardless of the option’s position in the command line.

Example

```
cc -c *.c
```

This command line compiles, but does not link, all files with the extension `.c` in the current working directory.

Naming the Object File (-Fo)

Option

`-Foobjfile`

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

Using cc Options

Keep the following rules in mind when using this option:

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

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

If you use the **-Fo** option (that is, if you do not give an object file name with a base and an optional extension), **cc** names the object files according to the following rule:

- If you give a directory specification following the **-Fo** option, **cc** creates the object file in the given directory and uses the default file name (the base name of the source file plus **.o**). Otherwise, *objfile* is created in the current directory. A **.o** extension is added if no extension is given.

To give a directory specification, it must end with a forward slash (/) so that **cc** can distinguish between a directory specification and a file name.

Example

```
cc -Fo/object1/ this.c that.c -Fo/src/newthose those.c
```

In this example, the first **-Fo** option tells the compiler to create, in the */object1* directory, the object file *this.o* (created as a result of compiling *this.c*). The compiler also creates, in the current directory, the object file *that.o* (created as a result of compiling *that.c*). The second **-Fo** option tells the compiler to create the object file named *newthose.o* (created as a result of compiling *those.c*) in the */src* directory.

Naming the Executable File (-Fe) (-o)

Option

```
-Fe exefile  
-o exefile
```

By default, `cc` gives the name `a.out` to the executable file. In UNIX System V, `-Fe` and `-o` are the same, except syntactically. The file name must come immediately after `-Fe`, whereas blanks are permitted between `-o` and the file name. Either option lets you give the executable file a different name or create it in a different directory.

Since `cc` creates only one executable file, you can give the `-Fe` option anywhere on the command line. If more than one `-Fe` option appears, `cc` gives the executable file the name specified in the last `-Fe` option on the command line.

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

Examples

```
cc -Fe/bin/process *.c
cc -o /bin/process *.c
```

These examples compile and link all source files with the extension `.c` in the current working directory. The resulting executable file is named `process.out` and is created in the directory `/bin`.

Creating Listings

A number of options are available with the `cc` command for creating listings. You can create a source listing, a map listing, or one of several kinds of object listings. You can also set the title and subtitle of the source listing from the command line and control the length of source-listing lines and pages.

These options are described in the following sections.

Note

Listings produced by the `cc` command may contain names that begin with more than one underscore (for example, `__chkstk`) or that end with the suffix `QQ`. Names that use these conventions are reserved for internal use by the compiler, and should not be used in your programs, except for those documented in the *C Library Guide*. Moreover, you should avoid creating global names that begin with an underscore. Since the compiler automatically adds another leading underscore, these names will have two leading underscores and might conflict with names reserved by the compiler.

Using cc Options

Types of Listings (-Fs, -Fl, -Fa, -Fc, -Fm -S)

Options

-Fs [<i>listfiles</i>]	Source listing
-Fl [<i>listfile</i>]	Object listing
-Fa [<i>listfile</i>]	Assembly listing
-Fc [<i>listfile</i>]	Combined source and object listing
-Fm [<i>mapfile</i>]	Map file that lists segments, in order
-S	Assembly listing

This section describes how to use command-line options to create listings. For an example of each type of listing and a description of the information it contains, see the section on “Formats for Listings.”

When using an option described in this section, the *listfile* argument, if given, must follow the option immediately, with no intervening spaces. The *listfile* may be a file specification or a path specification. It may also be omitted.

Note

When you give just a path specification as the *listfile* argument, the path specification must end with a forward slash (/) so that **cc** can distinguish it from an ordinary file name.

When you give a path specification as the argument to a listing option, or if you omit the argument altogether, **cc** uses the default file name for the listing type. Table 2.3 gives the default names used for each type of listing. The table also shows the default extensions, which are used when you give a file-name argument that lacks an extension.

Table 2.3
Default File Names and Extensions

Option	Listing Type	Default File Name ¹	Default Extension ²
-Fs	Source	Base name of source file plus .S	.S
-Fl	Object	Base name of source file plus .L	.L
-Fa	Assembly (masm)	Base name of source file plus .asm	.asm
-Fc	Combined source-object	Base name of source file plus .L	.L
-Fm	Map	Prints to standard output.	
-S	Assembly (masm)	Base name of source file plus .asm	.asm

Notes:

- 1 The default file name is used when the option is given with no argument or with a path specification as the argument.
- 2 The default extension is used when a file name lacking an extension is given.
- 3 The assembly-language listing produced by the **-Fa**, **-Fc**, and **-S** options uses **masm** directives.
- 4 The **-Fa** and **-S** options produce the same output, except that you cannot specify the list file with the **-S** option.

Since you can process more than one file at a time with the **cc** command, the order in which you give listing options and the kind of argument you give for each option (file specification or path specification) affect the result. Table 2.4 summarizes the effects of each option with each type of argument.

Using cc Options

Table 2.4
Arguments to Listing Options

Option	File-Name Argument	Path Argument ¹	No Argument
-Fa, -Fc, -Fl, -Fs	Creates a listing for the next source file on the command line; uses default extension if no extension is supplied	Creates listings in the given location for every source file listed after the option on the command line; uses default names	Creates listings in the current directory for every source file listed after the option on the command line; uses default names
-Fm	Uses given file name 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
-S	File name argument is not allowed	Path argument is not allowed	Uses default name

Notes:

- ¹ When you give just a path specification as the argument, the path specification must end with a forward slash (/) so that `cc` can distinguish it from an ordinary file name.

Only one type of object or assembly listing can be produced for each source file. The **-Fc** option overrides the **-Fa** and **-Fl** options and produces a combined listing. If you apply both the **-Fa** and the **-Fl** options to one source file, only the last listing specified on the command line is produced. If you specify both the **-Fa** and the **-Fs** options to one source file, a combined listing is produced. The **-Fs** option may be used with any other option.

Note

The `cc` command optimizes by default, so listing files reflect the optimized code. Since optimization may involve rearrangement of code, the correspondence between your source file and the machine instructions may not be clear, especially when you use the `-Fc` option to mingle the source and assembly codes. To produce a listing without optimizing, use the `-Od` option (discussed in “Preparing for Debugging” later in this section) with the listing option.

The map file is produced during the linking stage. If linking is suppressed with the `-c` option, the `-Fm` option has no effect.

Examples

```
cc -Fshello.src -Fchello.cmb hello.c
```

In this example, `cc` creates a source listing called `hello.src` and a combined source and object listing called `hello.cmb`. The object file has the default name `hello.o`. However, it is removed if the link was successful.

```
cc -Fshello.src -Fshello.lst -Fchello.cod hello.c
```

This command produces a source listing called `hello.lst` rather than `hello.src`, since the last name provided has precedence. This example also produces a combined source and object listing file named `hello.cod`. The object file in both of these examples has the default name `hello.o`.

Setting Titles (-St) and Subtitles (-Ss)**Options**

```
-St "title"
-Ss "subtitle"
```

The `-St` and `-Ss` options set the title and subtitle, respectively, for source listings. The quotation marks (") around the *title* or *subtitle* argument can be omitted if the title or subtitle does not contain space or tab characters. The space between `-St` or `-Ss` and its argument is optional.

The title appears in the upper left corner of each page of the source listing. The subtitle appears below the title.

Using cc Options

The **-St** or **-Ss** option applies to the remainder of the command line or until the next occurrence of **-St** or **-Ss** on the command line. These options do not cause source listings to be created. They take effect only when the **-Fs** option is also used to create a source listing.

Examples

```
cc -St "Income Tax" -Ss 4-14 -Fs tax*.c
```

This command 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.

```
cc -c -Fs -Fa -St"Calc Prog" -Ss"count" ct.c -Ss"sort" srt.c
```

In this command, **cc** compiles two source files and creates two source listings. Each source listing has a unique subtitle, but both listings have the title *Calc Prog*.

Formats for Listings

The rest of this section describes and shows examples of the five types of listings available with the **cc** command. For information on how to create these listings, see “Types of Listings” earlier in this section.

Source Listing

Source listings are helpful for debugging programs as they are being developed. These listings are also useful for documenting the structure of a finished program.

The source listing contains the numbered source-code lines of each procedure in the source file, along with any diagnostic messages that were generated. If the source file compiles with no errors more serious than warning errors, the source listing also includes tables of local symbols, global symbols, and parameter symbols for each function. If the compiler is unable to finish compilation, it does not generate symbol tables.

At the end of the source listing is a summary of the segment sizes in your program. This summary is useful for analyzing the program’s memory requirements.

Any error messages that occurred during compilation appear in the listing after the line that caused the error, as shown in the following example:

```
1 char hexvalue[10];
2
3 main()
4 {
5     long htoi();
6     printf("Please enter the hex value you want to convert:\n");
7     scanf("%s", hexvalue);
8     printf("The integer value of the hex value is %ld\n", htoi(hexvalue));
9 }
10
11 long htoi(hexvalue)
12 char *hexvalue;
13 {
14     register char *ptr=hexvalue;
15     int i=0;
16     long n=0;
17     long expl6();
18     while (*ptr != '\0') {
19         if (*ptr >= 'a' && *ptr <= 'f')
20             *ptr -= 87;
21         else if (*ptr >= 'A' && *ptr <= 'F')
22             *ptr -= 55;
23         else
24             *ptr -= 48;
25     ptr++;
26     bomb.c(25) : error C2059: syntax 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.

Using cc Options

The following example shows the source listing for a simple C program:

```
Hex to ASCII                                     PAGE 1
2/25/87                                         02-25-87
                                                10:44:23

Line# Source Line                               C Optimizing Compiler Version 5.10
1 char hexvalue[10];
2
3 main()
4 {
5     long htoi();
6     printf("Please enter the hex value you want to convert:0);
7     scanf("%s", hexvalue);
8     printf("The integer value of the hex value is %ld0, htoi(hexvalue));
9 }
10
11 long htoi(hexvalue)
12 char *hexvalue;
13 {
14     register char *ptr=hexvalue;
15     int i=0;
16     long n=0;
17     long expl6();
18     while (*ptr != ' ') {
19         if (*ptr >= 'a' && *ptr <= 'f')
20             *ptr -= 87;
21         else if (*ptr >= 'A' && *ptr <= 'F')
22             *ptr -= 55;
23             *ptr -= 48;
24         ptr++;
25     }
26     ptr -= 1;
27     while (ptr>=hexvalue)
28     {
29         n+= (*ptr*expl6(i));
30         i++;
31         ptr--; 33     }
32     return(n);
33 }
34
35 htoi Local Symbols
Name Class Type Size Offset Register
i . . . . . auto -0008
ptr . . . . . auto *** si
n . . . . . auto -0004
hexvalue. . . . . param 0004
36
37 long expl6(exp)
38 int exp;
39 {
40     long result=1;
41     int j;
42     for (j=1; j<=exp; j++)
43         result *= 16;
44     return(result);
45 }
```

```

Hex to A
2/25/87                                02-25-87
                                           10:44:23

                                           C Optimizing Compiler Version 5.10

exp16 Local Symbols

Name          Class  Type      Size  Offset  Register
j . . . . .  auto
result. . . . . auto
exp . . . . . param
                0004

Global Symbols

Name          Class  Type      Size  Offset
exp16 . . . . . global near function  ***  00ae
hexvalue. . . . . common struct/array   10   ***
htoi. . . . . global near function   ***  0038
main. . . . . global near function   ***  0000
printf. . . . . extern near function  ***  ***
scanf. . . . . extern near function  ***  ***

Code size = 00e8 (232)
Data size = 005f (95)
Bss size = 0000 (0)

No errors detected

```

At the end of each function, a table of local symbols is given, as shown in the following example for the function *htoi*:

```

htoi Local Symbols

Name          Class  Type      Size  Offset  Register
i . . . . .  auto
ptr . . . . . auto
n . . . . .  auto
hexvalue. . . . . param
                0004

```


Using cc Options

The following list shows the contents of each column in the symbol table:

Column Contents

- Name* The name of each local symbol in the function.
- Class* Either *auto* if the symbol is a nonstatic local variable, or *param* if the symbol is a formal parameter.
- Offset* The symbol's offset address 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.
- Register* Blank unless the variable is stored in a register, in which case, this column indicates the register (**SI** or **DI**).

At the end of the source code, a table of global symbols is given, as shown in the following example:

Name	Class	Type	Size	Offset
expl6	global	near function	***	00ae
hexvalue.	common	struct/array	10	***
htoi.	global	near function	***	0038
main.	global	near function	***	0000
printf.	extern	near function	***	***
scanf	extern	near function	***	***

The following list shows the contents of each column:

Column Contents

- Name* Each global symbol, external symbol, and statically allocated variable declared in the source file.
- Class* Either *global*, *common*, *extern*, or *static*, depending on how the symbol was defined in the source file.
- Type* A simplified version of the symbol's type as declared in the source file.

For functions, this entry is either *near function* or *far function*, depending on which memory model was used and how the function was declared. For a pointer, this entry is *near pointer*, *far pointer*, or *huge pointer*. For enumeration variables, this entry is *int*. For structures, unions, and arrays, this entry is *struct/array*.

Size Used only for variables. Specifies the number of bytes of storage allocated for the variable. Since the amount of storage allocated for an external array may not be known, its *Size* entry may be undefined.

Offset Used only for symbols with an entry of *global* or *static* in the *Class* column.

For variables, this entry gives the relative offset of the variable's storage in the logical data segment for the program file being compiled. Since the linker usually 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, this entry gives the relative offset of the start of the function in the logical code segment. For small-model programs, the linker combines logical code into a single physical segment, so this entry 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, this entry gives the actual offset of the function in its run-time code segment.

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

```
Code size = 0103 (259)
Data size = 005f (95)
Bss size  = 0000 (0)
```

The number of bytes in each segment is given first in hexadecimal, and then in decimal (in parentheses).

Object Listing

The **-Fl** option produces an object listing. The object listing contains the instruction encoding and assembly code for your program. The line numbers are shown in the listing as comments. The instruction encoding is on the left and the assembly code on the right, as shown in the following example:

Using cc Options

```
; Line 4
PUBLIC _main
_main PROC NEAR
    *** 000000 55                push bp
    *** 000001 8b ec            mov  bp,sp
    *** 000003 33 c0            xor  ax,ax
    *** 000005 e8 00 00        call __chkstk
; Line 6
    *** 000008 b8 00 00        mov  ax,OFFSET DGROUP:$$G12
    *** 00000b 50                push ax
    *** 00000c e8 00 00        call _printf
    *** 00000f 83 c4 02        add  sp,2
```

Assembly Listing

The **-Fa** and **-S** options produce an assembly listing using directives suitable for assembly using the Macro Assembler, **masm**. It contains the assembly code corresponding to your C source file, as shown in the following example:

```
; Line 4
PUBLIC _main
_main PROC NEAR
    push bp
    mov  bp,sp
    xor  ax,ax
    call __chkstk
; Line 6
    mov  ax,OFFSET DGROUP:$$G12
    push ax
    call _printf
    add  sp,2
```

Note that the example shows the same code as in the object listing example, except that the instruction encoding is omitted.

The listing generated by the **-Fa** option in Versions 5.0 and later of the C Compiler can be used as input to **masm**.

Combined Source and Object Listing

The **-Fc** option produces a combined source and object listing. This shows each line of your source program followed by the corresponding line (or lines) of machine instructions, as shown in the following example:

```

_TEXT      SEGMENT
;|*** char hexvalue[10];
;|***
;|*** main()
;|*** {
; Line 4
      PUBLIC _main
_main PROC NEAR
      *** 000000 55                push bp
      *** 000001 8b ec            mov  bp,sp
      *** 000003 33 c0            xor  ax,ax
      *** 000005 e8 00 00        call __chkstk
;|***   long htoi();
;|***   printf("Please enter the hex value you want to convert:");
; Line 6
      *** 000008 b8 00 00        mov  ax,OFFSET DGROUP:$SG12
      *** 00000b 50                push ax
      *** 00000c e8 00 00        call _printf
      *** 00000f 83 c4 02        add  sp,2
;|***   scanf("%s", hexvalue);

```

Note that this sample is like the object-listing sample, except that the source-program line is provided in addition to the line number.

When you examine a listing file, you will notice that the names of globally visible functions and variables begin with an underscore, as shown in the following example. This part of the listing is the same for all three kinds of listings:

```

EXTRN _printf:NEAR
EXTRN _scanf:NEAR
EXTRN __chkstk:NEAR
EXTRN __a1mul:NEAR
EXTRN __aNNalshl:NEAR
EXTRN _hexvalue:TBYTE

```

The C Compiler automatically prefixes an underscore to all global names. If you write assembly-language routines to interface with your C program, this naming convention is important; see the section on “Controlling the Preprocessor” for more information.

The listing may also contain names that begin with more than one underscore (for example, `__chkstk`). Identifiers with more than one leading underscore are reserved for internal use by the compiler, and should not be used in your programs, except for those documented in the *C Library Guide*. Moreover, you should avoid creating global names that begin

Using cc Options

with an underscore. Since the compiler automatically adds another leading underscore, these names will have two leading underscores and might conflict with the names reserved by the compiler.

Listing Pragmas

There are several pragmas that allow you to control the page formatting of the listings produced with the various list options. These pragmas are:

Pragma	Action
linesize	Sets the number of characters per line in the source listing.
page	Places a formfeed character(s) in the source listing.
pagesize	Sets the number of lines per page in the source listing.
skip	Skips the specified number of lines in the source listing.
subtitle	Specifies a subtitle for the source listing.
title	Specifies a title for the source listing.

The remainder of this section discusses each of the preceding pragmas.

The **linesize** Pragma

The **linesize** pragma sets the number of characters per line in the source listing. The syntax of this pragma is:

```
#pragma linesize(characters)
```

In this syntax, the optional parameter *characters* is an integer constant in the range 79-132 that specifies the number of characters you wish each line of the source listing to have. If *characters* is absent, the compiler uses the value specified in the **-SI** option or, if that option is absent, the default value of 79 characters per line. Note that **linesize** takes effect in the line *after* the line in which the pragma itself appears.

The following example uses the pragma to produce a source listing with a 132-character line length:

```
#pragma linesize(132)
```

The page Pragma

The **page** pragma generates a formfeed (page eject) character in the source listing at the place where the pragma appears. The pragma has the following syntax:

```
#pragma page(pages)
```

The optional parameter *pages* is an integer constant in the range 1-127 that specifies the number of pages to eject. If *pages* is absent, the pragma uses a default value of 1, in which case the next line in the source file appears at the top of the next listing page.

The pagesize Pragma

The **pagesize** pragma sets the number of lines per page in the source listing. The pragma has the following syntax:

```
#pragma pagesize(lines)
```

The optional parameter *lines* is an integer constant in the range 15-255 that specifies the number of lines that you wish each page of the source listing to have. If this parameter is absent, the pragma sets the page size to the number of lines specified in the **-Sp** command-line option or, if that option is absent, to a default value of 63 lines.

The following example uses the pagesize pragma to set the number of lines per page of the source listing to 66 lines:

```
#pragma pagesize(66)
```

The skip Pragma

The **skip** pragma generates a newline (carriage return/line feed) in the source listing, at the point where the pragma appears. The pragma has the following syntax:

```
#pragma skip(lines)
```

The optional parameter *lines* is an integer constant in the range 1-127 that specifies the number of lines that you wish to skip. If this parameter is absent, **skip** defaults to one line.

Using cc Options

The subtitle Pragma

The **subtitle** pragma sets a subtitle in the source listing. The pragma has the following syntax:

```
#pragma subtitle(subtitlename)
```

The required parameter *subtitlename* is a string literal containing the subtitle for subsequent pages in the source listing. The subtitle appears below the title on each page of the listing.

If you supply a null string ("") as the *subtitlename* parameter, **subtitle** removes any subtitle that was previously set. The **subtitlename** parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

The following statement sets the subtitle to *Error handler* for subsequent pages in the source listing:

```
#pragma subtitle("Error handler")
```

The title Pragma

The **title** pragma sets a title for the source listing. The pragma has the following syntax:

```
#pragma title(titlename)
```

The required parameter *titlename* is a string literal containing the title for the source listing. The title appears in the upper left corner of each page of the listing.

If you supply a null string ("") as the *titlename* parameter, **title** removes any title that was previously set. The **titlename** parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination.

The following statement sets the title to *File I/O Module* in the source listing:

```
#pragma subtitle("File I/O Module")
```

Map File

The **-Fm** option produces a map file. The map file contains a list of segments in order of their appearance within the load module. As an example, consider the following:

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

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

The starting address and name of each group appear after the list of segments. An example of a group listing follows:

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

In this example, **DGROUP** is the name of the data group. **DGROUP** is the only group used for data segments by programs compiled with the C Compiler, Version 5.1.

The following map file contains two lists of global symbols: the first list is sorted in ASCII-character order by symbol name and the second is by symbol address. A maximum of 2048 symbols can be sorted in each list. (To increase the number of sorted symbols, you must specify the **-MAP** linker option with the *number* argument to create the map file; see the ‘Linking with the cc Command’ chapter of this guide for details.) The notation *Abs* appears next to the names of absolute symbols (symbols containing 16-bit constant values that are not associated with program addresses).

Many of the global symbols that appear in the map file are symbols used internally by the C Compiler. These usually begin with one or more leading underscores or end with *QQ*.

Using cc Options

Address	Publics by Name
003F:0396	STRK22
0047:1D66	__brkotl
003F:04B0	__edata
0047:0910	__end
.	
.	
0047:03EC	__abrkp
0047:0090	__abrktb
0047:03EC	__abrktbe
003F:9876	Abs __acrtmsg
0000:9876	Abs __acrtused
.	
.	
0047:0240	__argv
0047:0242	__argv
Address	Publics by Value
003F:0010	__main
003F:0047	__htci
003F:00DA	__exp16
003F:0113	__chkstk
003F:0129	__astart
003F:0105	__cintDIV
.	
.	

The addresses of the external symbols are in the “*selector:offset*” format, showing the location of the symbol relative to zero (the beginning of the load module).

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

```
Program entry point at 003F:0129
```

Controlling the Preprocessor

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

The C preprocessor recognizes only preprocessor directives. It treats the source file as a text file, processing substitutions and definitions as directed. The preprocessor can be run on a file at any stage of development, whether or not the file is a complete C source file. In fact, the preprocessor is not restricted to processing C files; it can be run on any kind of file. However, input files to the preprocessor must follow the preprocessor rules; therefore, not all arbitrary text files may be suitable for use with the preprocessor. See the *C Language Reference* for a complete discussion of C preprocessor directives and the format expected for preprocessor input.

Defining Constants and Macros (-D)

Option

-D *identifier* [= [*string*]]

The **-D** option lets you define a constant or macro used in your source file. The *identifier* is the name of the constant or macro and *string* is its value or meaning. Note that spaces are permitted (but not required) between **-D** and the identifier.

If you leave out both the equal sign and *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; its definition is the empty string. This definition effectively removes all occurrences of the identifier from the source file. For example, to remove all occurrences of *register*, use the following option:

-Dregister=

Note that the identifier *register* is still considered to be defined.

The effect of using the **-D** option is the same as using a preprocessor **#define** directive at the beginning of your source file. The identifier is defined in the source file being compiled either until an **#undef** directive removes the definition or until the end of the file is reached.

You can supply a command-line definition for an identifier that is also defined within the source file. However, you must use **#undef** to remove the source-file definition, unless the source-file definition is identical to the command-line definition. The command-line definition remains in effect until the identifier is removed with an **#undef** directive.

Using cc Options

The **-D** option is especially useful with the **#if** and **#ifdef** directives because you can control conditional-compilation directives in the source file from the command line.

Examples

```
cc -D NEED=2 main.c
```

This example defines the manifest constant *NEED* in the source file *main.c*. This definition is equivalent to placing the directive at the top of the source file as shown in the following example:

```
#define NEED 2
```

For the next example, suppose a source file named *other.c* contains the following fragment:

```
#if defined(NEED)
.
.
.
#endif
```

Suppose further 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**. For instance, the following command is sufficient to compile all statements following the **#if** directive:

```
cc -DNEED main.c
```

Note that *NEED* does not have to be set to a specific value to be considered defined. The following command, in contrast, causes the statements in the **#if** block to be ignored (not compiled):

```
cc main.c
```

Predefined Identifiers (Manifest Defines)

The compiler defines several identifiers that are useful in writing portable programs. These are known as “manifest defines.” You can use these identifiers to compile code sections conditionally, depending on the processor and operating system being used. They begin with “M_” for “manifest.” The predefined identifiers and their functions are as follows:

Identifier	Function
M_I86	This is an Intel processor.
M_SYS3	This is Unix System III compatible.
M_SYS5	This is Unix System V compatible.
M_BITFIELDS	This compiler supports bitfields.
M_WORDSWAP	The word-within-a-longword order is swapped with respect to the DEC PDP-11.
M_UNIX	Always defined, this identifies the target operating system as an implementation of UNIX System V.
M_In86	Depending on -M0 , -M1 , -M2 , or -M3 , M_I386 is defined with 386 compiler unless -dos is used.
M_I86mM	Always defined, this identifies the memory model, where <i>m</i> is either S (small model), C (compact model), M (medium model), L (large model), or H (huge model). If huge model is used, both M_I86LM and M_I86HM are defined. Small model is the default. Memory models are discussed in the “Working with Memory Models” chapter.
_CHAR_UNSIGNED	This is defined only when the -J option is given to make the char type unsigned by default. For more information, see the section on “Changing the Default char Type.”
M_SDATA or M_LDATA M_STEXT or M_LTEXT	Depending on -M0 , -M1 , or -M2 .

Using cc Options

Removing Definitions of Predefined Identifiers (-U, -u)

Options

-U *identifier*
-u

The **-U** (for “undefine”) option turns off the definition of one of the predefined identifiers discussed in the previous section. One or more spaces may separate the **-U** and *identifier*. You can specify more than one **-U** option on the same command line. The **-u** option turns off all definitions.

Example

```
cc -UM_UNIX -UM_I86 work.c
```

This example removes the definitions of two predefined identifiers. Note that the **-U** option must be given for each removal.

Producing a Preprocessed Listing (-P, -E, -EP)

Options

-P Writes preprocessed output to a file
-E Writes preprocessed output to standard output; includes **#line** directives
-EP Writes preprocessed output to a file and standard output

The **-P**, **-E**, and **-EP** options produce listings of preprocessed files. These options allow you to examine the output of the C preprocessor.

The preprocessed listing file is identical to the original source file except that all preprocessor directives are carried out, macro expansions are performed, and comments are removed. All three options suppress compilation. No object file or listing is produced, even if you specify the **-fo** option or a listing-file option on the **cc** command line.

The **-P** option writes the preprocessed listing to a file with the same base name as the source file, but with a **.i** extension.

The **-E** option copies the preprocessed listing to the standard output (usually your terminal). It places a **#line** directive in the output at the

beginning and end of each included file and around lines removed by preprocessor commands that specify conditional compilation.

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

The **-EP** option combines features of the **-E** and **-P** options; the file is preprocessed and copied both to a new file and to the standard output, but no **#line** directives are added.

Examples

```
cc -P main.c
```

This example creates the preprocessed file *main.i* from the source file *main.c*.

```
cc -E add.c > preadd.c
```

This command creates a preprocessed file with inserted **#line** directives from the source file *add.c*. The output is redirected to the file *preadd.c*.

```
cc -EP add.c
```

The command shown here produces the same preprocessed output as the second example, but without the **#line** directives. The output appears on the screen and is copied to a new file.

Preserving Comments (-C)

Option

-C

Normally, comments are stripped from a source file in the preprocessing stage, since they do not serve any purpose in later stages of compiling. The **-C** (for “comment”) option preserves comments during preprocessing. The **-C** option is valid only when the **-E**, **-P**, or **-EP** option is also used.

Example

```
cc -P -C sample.c
```

Using cc Options

The example produces a listing named *sample.i*. The listing file contains the original source file, including comments, with all preprocessor directives expanded or replaced.

Searching for Include Files (-I, -X)

Options

-I *directory*
-X

The **-I** and **-X** options temporarily override the default search paths for include files. (The default path is */usr/include*.)

You can add to the list of directories searched by using the **-I** (for “include”) option. This option causes the compiler to search the directory or directories you specify before searching the default path */usr/include*. The space between **-I** and *directory* is optional. You can add more than one include directory by giving the **-I** option more than once in the **cc** command. The directories are searched in order of their appearance in the command line.

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

You can prevent the C compiler from searching the default paths for include files by using the **-X** (for “exclude”) option. When **cc** sees the **-X** option, it considers the list of standard places to be empty. This option is often used with the **-I** option to define the location of include files that have the same names as include files found in other directories, but that contain different definitions.

Examples

```
cc -I /include -I/alt/include main.c
```

In this example, **cc** looks for the include files requested by *main.c* in the following order: first in the directory */include*, then in the directory */alt/include*, and finally in the default directory */usr/include*.

```
cc -X -I /alt/include main.c
```

As shown in this example, the compiler looks for include files only in the directory `/alt/include`. First the `-X` option tells `cc` to consider the list of standard places empty; then the `-I` option specifies one directory to be searched.

Checking for Program Errors

You may encounter several different kinds of error messages when you compile, link, and run a C program.

Several `cc` options are available to control the types of warnings generated at compile time, help with syntax checking, and verify compatibility between the actual arguments and formal parameters of a function during the early stages of program development. This section describes these options.

Understanding Error Messages

Error messages can appear at different stages of program development:

- In the compiling stage, the compiler generates a broad range of error and warning messages to help you locate errors and potential problems in your source files.
- During the linking stage, the linker is responsible for generating error messages.
- During program execution, any error messages you see are runtime error messages. This category includes messages about core dumps, segmentation violations, and floating-point exceptions, which are errors generated by an 8087, 80287, or 80387 coprocessor.

Other utilities included in this package, such as the UNIX System V Link Editor (`ld`), and the `make` program-maintenance utility, generate their own error messages.

When you are compiling and linking using the `cc` command, you may see both compiler and linker messages. Compiler messages have numbers preceded by the letter `C`, and linker messages have numbers preceded by the letter `L`.

You can also distinguish the type of a message by its format. See the “Error Messages and Exit Codes” appendix in this guide for a description of compiler error-message formats, a list of actual compiler error messages, and explanations of the circumstances that cause them.

Using cc Options

Compiler error messages are sent to the standard output, which is usually your terminal. If you are using the C-shell, you can redirect the messages to a file by using the standard redirection symbols at the end of your command line:

If you are using the Bourne shell, you can redirect the messages to a file by using the standard redirection syntax:

```
cmd > outputfile 2>&1
```

Example

Assume the following source file is named *rm.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("couldn't delete %s : ", name);
      perror("");
    }
  }
```

The following C-shell command line redirects error messages to a file named *rm.err*:

```
cc rm.c >& rm.err
```

In the previous command, only output that ordinarily goes to the console screen is redirected. The error-message file *rm.err* contains the following information:

```
rm.c(11): error C2069: 'argv' : undefined
rm.c(12): warning C4147: '/' : different levels of indirection
```

Based on the errors generated, you can correct *rm.c* as shown below:

```
#include <stdio.h>

main(argc, argv)
    int argc;
    char *argv[];          /* corrects warning C4047 */

    {
    register int i;
    char *name;

    for (i = 1; i < argc; ++i) /* corrects error C2065 */
        if (unlink(name = argv[i])) {
            printf("couldn't delete %s : ", name);
            perror("");
        }
    }
}
```

2

Setting the Warning Level (-W, -w)

Option

```
-W{0|1|2|3}
-w
```

You can suppress warning messages produced by the compiler by using the **-W** (for “warning”) option. Compiler warning messages are any messages beginning with *C4*; see the “Error Messages and Exit Codes,” appendix for a full listing. Warnings indicate potential problems (rather than actual errors) with statements that may not be compiled as you intend. The **-W** options affect only source files given on the command line, they do not apply to object files.

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

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

Using cc Options

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

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

The **-W3** option displays the highest level of warning messages, including warnings about the uses of non-ANSI features and extended keywords and about function calls before the appearance of function prototypes in the program.

Note that the warning messages in the “Error Messages and Exit Codes” appendix indicate the warning level that must be set (that is, the number for the appropriate **-W** option) for the message to appear.

Example

```
cc -W3 crunch.c print.c
```

This example enables all possible warning messages when the *crunch.c* and *print.c* source files are compiled.

Checking Syntax (-Zs)

Option

-Zs

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

When you give the **-Zs** option, the compiler does not generate code or produce object files, object listings, or executable files. However, the compiler does display error messages if the source file has syntax errors. You can specify the **-Fs** option on the same command line to generate a source listing that shows these error messages. For more information about the **-Fs** option, see the section on “Types of Listings.”

Example

```
cc -Zs test*.c
```

This command causes the compiler to perform a syntax check on all source files in the current working directory that begin with *test* and end with the default extension (*.c*). The compiler displays messages for any errors found.

Generating Function Declarations (-Zg)

Option

-Zg

The **-Zg** option generates a function declaration for each function defined in the source file. You can use the **-Zg** option with multiple source files. 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 encountered are ignored.

The generated list of declarations is written to the standard output. It can be saved in a file using shell redirection.

When the **-Zg** option is used, the source file is not compiled. As a result, no object file or listing is produced.

The list of declarations is helpful for verifying that actual arguments and formal parameters of a function are compatible. You can save the list and include it in your source file to cause the compiler to perform type-checking. The presence of a declared argument-type list for a function “turns on” the compiler’s type-checking between actual arguments to a function (given in the function call) and the formal parameters of a function.

This type-checking can be a helpful feature in writing and debugging C programs, especially when working with older C programs. Argument type-checking is a recent addition to the C language, so many existing C programs will not have argument-type lists. See the *C Language Reference* for more information about function declarations and argument-type lists.

Using cc Options

You can use the **-Zg** option even if your 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 return types are identical.

Note

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

```
struct tagA {  
    .  
    .  
    .  
} A;
```

Example

```
cc -Zg file.c > filedecls.h
```

This command causes the compiler to generate argument-type lists for functions defined in *file.c*. The list of declarations is redirected to *filedecls.h*.

Preparing for Debugging (-Zi, -Od)

Options

- Zi** Creates object file for use with the source-level debugger, **sdb** and **CodeView**
- Od** Disables code optimization to help with debugging

The **-Zi** option produces an object file containing full symbolic-debugging information for use with the source-level debugger. This object file includes full symbol-table information and line numbers. If the **-Zi** option is given with no explicit **-O** options, all optimizations involving code motion and rearrangement are suppressed, although simple optimizations are still performed. If any explicit **-O** options are given, *all* requested optimizations are performed.

The **-Od** option tells the compiler not to perform most optimizations. Some peephole optimizations and other simple optimizations are still performed. (Without the **-Od** option, the default is to optimize.) You may want to use this option when you plan to use a symbolic debugger with your object file, since optimization can involve rearrangement of instructions that make it difficult for you to recognize and correct your code when debugging. However, turning off optimizations may increase the size of the code generated to the point where it might not be possible to link your program.

Other optimization options are discussed in the section on “Optimizing.”

Example

```
cc -zi -Od test.c
```

This command produces an object file named *test.o* that contains line numbers corresponding to the line numbers of *test.c*. A source-listing file *test.lst* is also created. Limited optimization is performed.

Optimizing

The optimizing capabilities available with the C Compiler can reduce the storage space or execution time required for a program. This is achieved by eliminating unnecessary instructions and rearranging code. The compiler performs some optimizations by default. You can use the **-O** options, the **loop_opt** pragma (described in the section on “Loop Optimization”), the **intrinsic** pragma, and the **function** pragma (described in the section “Generating Intrinsic Functions”) to exercise greater control over the optimizations performed. In addition, you can use the **-Gs** option or **check_stack** pragma to reduce program size and speed up execution.

Controlling Optimization (-O Options)

Option

```
-Ostring  
#pragma loop_opt([on|off])  
#pragma intrinsic(function1[,function2]..)  
#pragma function(function1[,function2]..)
```

Using cc Options

The **-O** options give you control over the optimization procedures that the compiler performs. One or more of the letters in *string* following the **-O** let you choose how the compiler performs optimization:

Letter	Optimizing Procedure
none	Performs optimization equivalent to -O0
a	Relaxes alias-checking
c	Eliminates common expressions
d	Disables optimization
i	Expands certain intrinsic functions inline
l	Enables loop optimization
p	Improves consistency of floating-point results
s	Favors code size during optimization
t	Favors execution speed during optimization
x	Maximizes optimization (equivalent to -Oatcli)

The letters can appear in any order; for example, **-Oat** and **-Ota** have the same effect. More than one **-O** option can be given; the compiler uses the last **-O** option given if any conflict arises. Each option applies to all source files following that option on the command line.

The following sections discuss the various optimization options and their effects.

Relaxing Alias Checking (-Oa)

The **a** option letter can be used with the **l**, **s**, or **t** option letter to relax the assumptions the compiler makes about the use of “aliases” in the program. Aliases are multiple names (that is, symbolic references) for the same memory location in a program. Most commonly, aliases occur as a result of code similar to that shown in the following example:

```
func()
{
  int  x, *p;

  p = &x; /* now "x" and "**p" refer to the same */
         /* memory location                      */
  .
  .
  .
}
```

Use of the **-Oa** option can reduce the size of executable files and speed program execution. Its use is especially recommended when you also specify the **-Ol** option, since the compiler can detect a number of loop optimizations when the **-Oa** option is in effect that it cannot detect when **-Oa** is not in effect. However, before you specify **-Oa**, you must make sure that your program does not use aliases either directly or indirectly.

Note

Exercise caution when using the **-Od** option, because responsibility for alias checking is transferred to the programmer from the compiler.

The use of aliases is important only if both names are actually used to reference the memory location. The following example illustrates the use of aliases:

Using cc Options

```
func()
{
    int x, *p;

    p = &x;

    .
    .
    .
    /* ...expressions involving only *p */
    .
    .
    .
}
```

Since all access to the memory location labeled *x* is through the pointer *p*, *x* has no significance in the function. To illustrate, *func* could be rewritten as the following pair of functions:

```
func1()
{
    int x;

    func2(&x);
}

func2(p)
int *p;
{
    .
    .
    .
    /* ...expressions involving *p */
    .
    .
    .
}
```

In this equivalent form, the alias created in *func1* is insignificant, since the memory location is not referenced at all and *func2* does not use aliases since *x* is not even in the scope of the function. The **-Oa** option can be safely specified in compiling either of these equivalent forms.

In addition to the obvious cases discussed above, aliases can be created through the use of pointers in other, more subtle ways. Two such cases involving the use of pointers as function arguments are illustrated in the following example:

```
int x;

func(p)

int *p;
{
    .
    .
    .
    /* ...expressions involving *p and x */
    .
    .
    .
}
```

In this example, *x* is a communal variable, so the function can be called with *func(&x)*. The **-Oa** option can be used safely only if it is known that *func* is never invoked with the address of *x* as an argument.

```
func(p1, p2)

int *p1, *p2;
{
    .
    .
    .
    /* ...expressions involving *p1 and *p2 */
    .
    .
    .
}
```

In this example, the function may be invoked with the same value for both arguments (that is, *func(p,p)* or *func(&x,&x)*). Thus, the **-Oa** option can be safely specified only if it is known that the function is always called with distinct values for the two arguments.

One use of aliases occurs so frequently that a special provision has been made for it. When the compiler encounters a call to a function with address-type arguments, it always assumes that all variables whose addresses are passed to the function are modified. If such function calls appear in a program, the **-Oa** option can be specified safely even though the function call results in an alias for each variable whose address is passed. The following example illustrates how the compiler handles this case:

Using cc Options

```
func1()
{
    int  x, y, a, b;
    .
    .
    x = a + b;

    func2(&a);

    y = a + b;
}
```

As shown, when the compiler encounters the function call *func2(&a)*, it assumes that the function modifies *a*, even if the **-Oa** option has been specified. The compiler generates code to evaluate each instance of the expression *a + b*, rather than eliminating a common subexpression incorrectly.

Although you should convert programs that use aliases if you plan to compile them with the **-Oa** option, it is helpful to know the units of a program where the optimizations affected by the use of **-Oa** are applied. This information indicates where the uses of aliases are most likely to cause incorrect optimizations if **-Oa** is specified. The following list describes the program units where such optimizations are performed:

- All of the C optimizations, except for loop optimizations, that may be affected by the incorrect use of **-Oa** are applied at the level of basic blocks. In the C Compiler, the **-Oa** option can generally be used even if aliases are employed, provided no memory location is referenced by more than one name within any basic block. (A “basic block” is a contiguous sequence of statements, with a unique entry point and exit point and no branching in between. In C programs, basic blocks most often appear as the clauses of statements, **switch** statements, loop bodies, or function bodies, although they may also occur as sequences of statements delimited by user labels.)
- Loop optimizations are applied at the level of whole loop bodies. Thus, if loop optimization is enabled, **-Oa** can generally be used even if aliases are employed, provided that no memory location is referenced by more than one name within any basic block or loop body.

Disabling Optimization (-Od)

The **-Od** option turns off most optimizations. This is useful in the early stages of program development to avoid optimizing code that will later be changed. Because optimization may involve rearrangement of instructions, you may also 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, it can be difficult to recognize and correct your code. However, note that turning off or restricting optimization of a program usually increases the size of the generated code. If your program contains a module that is close to the 64K limit on compiled code, turning off optimization may cause the module to exceed the limit.

Generating Intrinsic Functions (-Oi)

The **-Oi** option tells the compiler to generate intrinsic functions instead of function calls for certain functions. Intrinsic functions may be in-line functions, may use special argument-passing conventions, or (in some cases) may 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 of the additional code that is generated.

The following functions have intrinsic forms:

- **memset**, **memcpy**, and **memcmp**
- **strset**, **strcpy**, **strcmp**, and **strcat**
- **outp**
- **_rotr**, **_rotr**, **_lrotr**, and **_lrotr**
- **min**, **max**, and **abs**

Note

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 with the huge memory model (that is, using the **-Mh** option on the command line).

Using cc Options

You can use the **intrinsic** pragma to generate intrinsic functions only for selected functions. This pragma has the following format:

```
#pragma intrinsic (function1 [,function2]...)
```

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. The **function** pragma has the following format:

```
#pragma function (function1 [,function2]...)
```

Note that you can also use the **function** pragma selectively to generate function calls instead of intrinsic functions when you compile a program with the **-Oi** option.

Loop Optimization (-O1)

The **-O1** option tells the compiler to perform loop optimizations. For best performance, the **-O1** option should be specified along with the **a** option letter (**-Oa1**), since the compiler can detect more loop optimizations when it relaxes its assumptions about the use of aliases.

You can use the **loop_opt** pragma to turn loop optimization on or off for selected functions. When you want to turn off loop optimization, put the following line before the code on which you don't want to perform loop optimization:

```
#pragma loop_opt (off)
```

Note that the preceding line disables loop optimization for all code that follows it in the source file, not just the routines on the same line. To reinstate loop optimization, insert the following line:

```
#pragma loop_opt (on)
```

If no argument is given to the **loop_opt** pragma, loop optimization reverts to the behavior specified on the command line: enabled if the **-Ox** or **-O1** option is in effect, and disabled otherwise. The interaction of the **loop_opt** pragma with the **-O1** and **-Ox** options is explained in greater detail in Table 2.5.

Table 2.5
Using the loop_opt Pragma

Syntax	Compiled with -Ox or -Ol?	Action
#pragma loop_opt()	no	Turns off optimization for loops that follow
#pragma loop_opt()	yes	Turns on optimization for loops that follow
#pragma loop_opt (on)	yes or no	Turns on optimization for loops that follow
#pragma loop_opt (off)	yes or no	Turns off optimization for loops that follow

Achieving Consistent Floating-Point Results (-Op)

The **-Op** option is useful when floating-point results must be consistent within a program. This option changes the way in which the program handles floating-point values.

Ordinarily the compiler stores each floating-point value in an 80-bit register. In subsequent references to that value, the compiler reads the value from the register. When the final value is written to memory, it is truncated, since floating-point types are allocated fewer than 80 bits of storage (32 bits for the **float** type and 64 bits for the **double** type). Thus, the value stored in the register may actually be more precise than the same value stored in a floating-point variable. Since the value is truncated each time it is written to memory, over the course of the program the value stored in the machine register may become quite different from the value that is written to memory.

If you use the **-Op** option, when floating-point values are referenced, the compiler reloads them from floating-point variables rather than from registers. Using **-Op** gives less precise results than using registers, and it may increase the size of the generated code. However, it gives you more control over the truncation (and hence the consistency) of floating-point values.

Using cc Options

Optimizing for Speed and Code Size (-O_t, -O_s)

When you do not give a **-O** option to the **cc** command, it automatically uses **-O_t**, meaning that program-execution speed is favored in the optimization. Wherever the compiler has a choice between producing smaller (but perhaps slower) and larger (but perhaps faster) code, the compiler generates faster code. For example, when the **-O_t** option is in effect, the compiler generates intrinsic functions to perform shift operations on long operands.

To cause the compiler to favor smaller code size instead, use the **-O_s** option. For example, when the **-O_s** option is in effect, the compiler uses function calls to perform shift operations on long operands.

Producing Maximum Optimization (-O_x)

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

```
-Oatcli
```

That is, the **-O_x** option relaxes alias checking, generates all intrinsics for the functions listed in the section “Generating Intrinsic Functions,” performs loop optimizations, favors execution time over code size, and removes stack probes. Note that the interactions between the **-O_x** option and the **loop_opt** pragma are the same as those described in Table 2.5. For more information about stack probes and ways of controlling their use, see the following section, “Removing Stack Probes.”

Examples

```
cc -Oal file.c
```

This command tells the compiler to perform loop optimizations and relax alias-checking when it compiles *file.c*. The compiler favors program speed over program size, since the **-O_t** option is also specified by default.

```
cc -c -Os file.c
```

This command favors code size over execution speed when *file.c* is compiled.

```
cc -Od *.c
```

This command compiles and links all C source files with the default extension (.c) in the current directory and disables optimization. This command is most useful during the early stages of program development, since it improves compilation speed.

Removing Stack Probes (-Gs)

Options

-Gs
#pragma check_stack([on|off])

You can reduce the size of a program and speed up execution slightly by removing stack probes. You can do this either with the **-Gs** option or with the **check_stack** pragma.

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

Use the **-Gs** option when you want to turn off stack-checking for an entire module if you know that the program does not exceed the available stack space. For example, stack probes may not be needed for programs that make very few function calls, or that have only modest local variable requirements. In the absence of the **-Gs** option, stack-checking is on.

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

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


Using cc Options

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

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

Note

For earlier versions of C, the **check_stack** pragma had a different format: **check_stack+** to enable stack-checking and **check_stack-** to disable stack-checking. Although the C Compiler still accepts this format, its use is discouraged, since it may not be supported in future versions.

If no argument is given for the **check_stack** pragma, stack-checking reverts to the behavior specified on the command line: disabled if the **-Gs** option is given, or enabled otherwise. The interaction of the **check_stack** pragma with the **-Gs** option is explained in greater detail in Table 2.6.

Table 2.6
Using the **check_stack** Pragma

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

Note

The **-Gs** option should be used with care. Although it can make programs smaller and faster, it may mean that the program will not be able to detect certain execution errors.

Example

```
cc -Oals -Gs file.c
```

This example optimizes the file *file.c* by removing stack probes with the **-Gs** option. The letters specified with the **-O** option tell the compiler to relax alias-checking (**a**), perform loop optimization (**l**), and favor code size over program speed (**s**). If you want stack-checking for only a few functions in *file.c*, you can use the **check_stack** pragma around the definitions of functions you want to check. Similarly, if you want to perform loop optimization on only a few functions in *file.c*, you can use the **loop_opt** pragma around the definitions of functions on which you want to perform loop optimization.

Enabling/Disabling Language Extensions (-Ze, -Za)**Option**

- Ze** Enables language extensions (default)
- Za** Disables language extensions (strict ansi specifications)

The C Compiler is moving to support the ANSI C standard. In addition, it offers a number of features beyond those specified in the ANSI C standard. These additional features are enabled when the **-Ze** (default) option is in effect and disabled when the **-Za** option is in effect. They include the following:

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

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

Using cc Options

The preceding example could be rewritten to conform with ANSI C as shown here:

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

- Redefinitions of **extern** items as **static**, as follows:

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

- Use of trailing commas (,) without ellipses (...) in function declarations to indicate variable-length argument lists, such as:

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

- Benign **typedef** redefinitions within the same scope, like this:

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

- Use of mixed character and string constants in an initializer, for instance:

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

- Use of bit fields with base types other than **unsigned int** or **signed int**

Use the **-Za** option if you will be porting your program to other environments. The **-Za** option tells the compiler to treat extended keywords as simple identifiers and disable the other extensions listed previously.

Packing Structure Members (-Zp)

Option

```
-Zp[{1|2|4}]  
#pragma pack({1|2|4})
```

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

- Items of type **char** or **unsigned char**, or arrays containing items of these types, are byte-aligned.

- Structures are word-aligned; structures of odd size are padded to an even number of bytes.
- All other types of structure members are word-aligned.

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

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

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

Use the **pack** pragma when you want to specify packing other than that specified on the command line for particular structures. Give the **pack(n)** pragma, where *n* is 1, 2, or 4, before structures that you want to pack differently. To reinstate the packing given on the command line, give the **pack()** pragma with no arguments.

Table 2.7 shows the interaction of the **-Zp** option with the **pack** pragma.

Table 2.7
Using the **pack** Pragma

Syntax	Compiled with -Zp Option?	Action
#pragma pack()	yes	Reverts to packing specified on the command line for structures that follow
#pragma pack()	no	Reverts to default packing for structures that follow

Using cc Options

#pragma pack(<i>n</i>)	yes or no	Packs the following structures to the given byte boundary until changed or disabled
-------------------------------	-----------	---

Example

```
cc -Zp prog.c
```

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

Setting the Stack Size (-F)

Option

-F *hexnum*

The **-F** option sets the size of the program stack. A space must separate the **-F** and *hexnum*. (This option applies only to the 286 compiler.)

The *hexnum* is a hexadecimal value representing the stack size in bytes. The value must be less than 0xFFFF hexadecimal (65,535 decimal).

If you do not specify this option, the start-up routine in the standard C library sets the default stack size to 2K.

If you get a stack-overflow message, you may need to increase the size of the stack. In contrast, if your program uses the stack very little, you may save some space by decreasing the stack size.

The **-F** option is a linking option that affects executable files only; it does not have any effect on source or object files.

Example

```
cc -F C00 *.o
```

This example sets the stack size to C00 hexadecimal (3K decimal) for the program created by linking all of the object files in the current working directory.

Restricting the Length of External Names (-nl)

Option

-nl *number*

The **cc** command allows you to restrict the length of external (public) names by using the **-nl** option. The *number* is an integer specifying the maximum number of significant characters in external names. The space between **-nl** and *number* is optional.

When you use the **-nl** option, the compiler considers only the first *number* characters of external names used in the program. The program may contain external names longer than *number* characters, but the extra characters are simply ignored.

The **-nl** option is typically used to conserve space or to aid in creating portable programs. The C Compiler imposes no restrictions on the length of external names (although it uses only the first 31 characters), but other compilers or linkers may produce errors when they encounter names longer than a predetermined limit.

Labeling the Object File (-V)

Option

-V *string*

Use the **-V** (for “version”) option to embed a text string in an object file. The *string* must be enclosed in double quotation marks (" ") if it contains white-space characters or embedded double quotation marks. A backslash (\) must precede any embedded double quotation marks.

A typical use of the **-V** option is to label an object file with a version number or copyright notice.

Example

```
cc -V "C Compiler Version 5.1" main.c
```

This command places the following string in the object file *main.o*:

```
C Compiler Version 5.1
```

Changing the Default char Type (-J)

Option

-J

In C, the **char** type is signed by default, so if a **char** value is widened to **int** type, the result is sign-extended. You can change this default to **unsigned** with the **-J** option, causing the **char** type to be zero-extended when widened to **int** type. However, if a **char** value is explicitly declared **signed**, the **-J** option does not affect it, and the value is sign-extended when widened to **int** type.

When you specify **-J**, the compiler automatically defines the identifier **_CHAR_UNSIGNED**.

Controlling the Calling Convention (-Gc)

Options

-Gc
fortran
pascal
cdecl

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

Because C, unlike other languages such as Pascal and FORTRAN, allows you to write functions that take variable numbers of arguments, it must handle function calls differently. Languages such as Pascal and FORTRAN normally push actual parameters to a function in left-to-right order, with the last argument in the list being the last one pushed on the stack. In contrast, C functions do not always know the number of actual parameters, so they must push their arguments from right to left, with the first argument in the list being the last one pushed.

Additionally, the calling function must remove the arguments from the stack in C (rather than having the called function do it, as in Pascal and FORTRAN). If the code for removing arguments is in the called function (as in Pascal and FORTRAN), it appears only once; if it is in the calling

function (as in C), it appears every time there is a function call. Since function calls are more numerous than individual functions, the Pascal/FORTRAN method is slightly smaller and more efficient.

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

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

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

The second method for generating the Pascal/FORTRAN calling convention is to use the **-Gc** option. If you do this, the entire module is compiled using the alternative calling convention. You might use this method to make it possible to call all the functions in a C module from another language, or to gain the performance and size improvement provided by this calling convention. When you use **-Gc** to compile a module, the compiler assumes that all functions called from that module use the Pascal/FORTRAN calling convention, even if the functions are defined outside that module. Thus, using **-Gc** would normally mean that you cannot call or define functions that take variable numbers of parameters, and that you cannot call functions such as the C library functions that use the C calling sequence.

To overcome these restrictions, the **cdecl** keyword has been added to C. This keyword is the “inverse” of the **fortran** and **pascal** keywords. When applied to a function or function pointer, it indicates that the associated function is to be called using the normal C calling convention. This allows you to write C programs which take advantage of the more efficient calling convention while still having access to the entire C library, other C objects, and even user-defined functions that can take variable-length argument lists.

If you compile with the **-Gc** option, either you must declare the **main** function in the source program with the **cdecl** keyword, or you must change the start-up routine so that it uses the correct naming and calling conventions when calling **main**.

Run-time library functions all use the C calling convention. Therefore, care must be taken to declare them **cdecl** functions.

Using cc Options

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

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

Examples

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

In this example, *var_print* is allowed to have a variable number of arguments by declaring it as a function using the normal right-to-left C function calling convention and naming conventions. The *cdecl* keyword overrides the left-to-right calling sequence set by use of the **-Gc** option when compiling the source file in which this declaration appears. If this file is compiled without the **-Gc** option, *cdecl* has no effect since it is the same as the default C convention.

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

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

```
long pascal index
```

This example simply changes the naming convention for the data item *index*: it is included in the object file in all capital letters and without a leading underscore.

Compiling Programs for DOS Environment (-dos, -FP)

The C compiler is capable of compiling programs that will execute in the DOS environment.

The **-dos** option instructs the compiler to use the set of libraries in */usr/lib/dos* and to use a different linker. Note that programs compiled with **-dos** will not run in the UNIX System V environment. Also note that many UNIX System V system calls are not supported in DOS.

There are a variety of **-FP** options that can be used along with **-dos** to control floating-point operations. For more information on **-FP** and on DOS cross-development in general, see “The DOS-OS/2 Development Guide” in the *Developer’s Guide*, and the chapter “C Language Portability” in this guide.

Displaying Compiler Passes (-d, -z)

The **cc** command is actually a driver program which executes a series of compiler passes, perhaps an assembler pass, and a linker. It collects the various options and files on its command line and distributes them to the proper pass or to the linker. The C compiler is conceptually a four-pass compiler. The function of the various compiler passes is outlined below:

Pass 0

Pass zero of the compiler is comprised of the preprocessor and parser. The preprocessor handles file inclusion, macro expansion, and text substitution, and allows you to define constructs for conditional compilation. The parser performs two functions: (1) building a context-free grammar tree to pass to Pass 1; and (2) constructing a symbol table.

Pass 1

Pass two generates code. It walks the grammar tree constructed by pass 0, applies semantic rules to each syntactic construct, and produces the binary code indicated by the semantic rules.

Pass 2

The third pass provides post-generation optimization. It analyzes the code generated by pass 1 and applies optimization rules to alter the code for better performance (such as elimination of redundant code, rearrangement, etc.). It creates the object code and outputs listing files (if requested).

The **-d** option displays the various passes and their arguments before they are executed. The **-z** option shows the passes but does not execute them.

Producing OMF Object and Executable Files (-xenix)

By default, `cc` produces object and executable files using the COFF format, which is the same format used by the AT&T development system. The `-xenix` option causes `cc` to produce object and executable files that use the OMF format, which is compatible with the XENIX System V development system tools. When the `-xenix` option is used with any of the options that produce assembly-language output, the warning message normally issued (`masm` directives) is suppressed. Note that UNIX System V can execute programs that use *either* COFF or OMF formats.

Miscellaneous Pragmas

The following pragmas allow you to embed comments in the object or executable file or to send a string to the standard output:

Pragma	Action
comment	Places a comment record in the object file.
Message	Sends a message string to the standard output.

The comment pragma

The `comment` pragma allows you to place a comment record in an object file or executable file. The pragma has the following syntax:

```
#pragma comment (commenttype [, commentstring])
```

The required parameter *commenttype* specifies the type of comment record. The optional *commentstring* parameter is a string literal that provides additional information for some comment types. The following table lists and describes the types of comment records accepted by the `comment` pragma.

Record	Description
compiler	Places the name and version number of the compiler into the object file. This comment record is ignored by the linker. If you supply a <i>commentstring</i> parameter for this record type, the compiler generates a warning message.
exestr	Places the string specified in <i>commentstring</i> into the object file. At link time, this string is placed into the executable file. The string is not loaded into memory when the executable file is loaded; however, it can be found with a program that finds printable strings in files. One use for this comment-record type is to embed a version number or similar information in an executable file.
lib	Places a library-search record into the object file. This comment type must be accompanied by a <i>commentstring</i> containing the name (possibly including the path) of the library that you want the linker to search for. Since the library name precedes the default library-search records in the object file, the linker searches for this library just as if you had named it on the command line. You can place multiple library-search records in the same source file. Each record appears in the object file in the same order it is encountered in the source file.
user	Places a general comment into the object file. The <i>commentstring</i> parameter contains the text of the comment. This comment record is ignored by the linker.

The following examples illustrate some of the uses of the **comment** pragma. The following pragma causes the linker to search for the library **mylibry.a**. The linker searches first in the current working directory and then in the path specified in the **LIB** environment variable:

```
#pragma comment(lib,mylibry)
```

The following pragma causes the compiler to place the name and version number of the compiler in the object file:

Using cc Options

```
#pragma comment (compiler)
```

For comments that take a *commentstring* parameter, you can use a macro in any place where you would use a string literal, provided that the macro expands to a string literal. You can also concatenate any combination of string literals and macros that expand to string literals. For example, the following statement is acceptable:

```
#pragma comment (user, "Compiled on " __DATE__ "at " __TIME__)
```

The message Pragma

The **message** pragma sends a string to the standard output. The pragma has the following syntax:

```
#pragma message (messagestring)
```

The *messagestring* parameter is a string literal that contains the message that you wish to send to the standard output. This pragma does not cause termination of the compilation. A typical use of **message** is to display informational messages at compile time.

The following code fragment uses **message** to display a message during compilation:

```
#if M_I86MM
    #pragma message ("Medium memory model")
#endif
```

The *messagestring* parameter can be a macro that expands to a string literal, and you can concatenate such macros with string literals in any combination. For example, the following statement displays the name of the file being compiled and the date and time when the file was last modified:

```
#pragma message ("Compiling " __FILE__ ". Last modified: " __TIMESTAMP__)
```

Predefined Macro Names

The C Compiler supports all of the predefined macro names found in the ANSI proposed standard for the C language. These provide a convenient means for obtaining the date and time of the compilation and for indicating whether the compiler purports to conform fully to the proposed ANSI standard. The `__TIMESTAMP__` identifier offers a capability not found in the proposed ANSI standard. The following list explains each of these names:

Macro Name	Description
<code>__DATE__</code>	The date of compilation, expressed as a string literal in the form: Mmm [d]d yyyy.
<code>__FILE__</code>	The file name, expressed as a string literal.
<code>__LINE__</code>	The program line number where the <code>__LINE__</code> macro was used.
<code>__STDC__</code>	The integer constant 0. If equal to 1, this macro indicates full conformity with the ANSI proposed standard for the C programming language.
<code>__TIME__</code>	The time of compilation, expressed as a string literal in the form: hh:mm:ss.
<code>__TIMESTAMP__</code>	The date and time of the last modification of the source file, expressed as a string literal in the form: Ddd Mmm [d]d hh:mm:ss yyyy

The `__TIMESTAMP__` macro name is not ANSI standard. Note that its time and date indicate the last modification of the source file, whereas `__DATE__` and `__TIME__` indicate the time of compilation. You can find additional information about the ANSI-compatible predefined identifiers `__DATE__` and `__TIME__` in the *C Language Reference*.

Using cc Options

The following code fragment uses three predefined macros with the **#message pragma** to display informational messages at the time of compilation.

```
#pragma message("Compilation date: "__DATE__")
#pragma message("Compiling: "__FILE__")
#pragma message("Last modification: "__TIMESTAMP__")
```

Here is the output you might see from the preceding code fragment:

```
Compilation date: Dec  2 1987
Compiling: sample.c.
Last modification: Mon Dec  1 12:02:51 1987
```

Chapter 3

Linking with the cc Command

Introduction 3-1

The Default Linking Process 3-2

Passing Linker Information: The -link Option 3-3

 Specifying Libraries 3-3

 Specifying Linker Options 3-5



Introduction

Since the `cc` command controls linking as well as compiling, you can specify linker options and libraries other than the default combined library to be linked with your object files on the `cc` command line.

The Default Linking Process

When the `cc` command compiles a source file, it encodes the name of the appropriate library in the object file. The library name embedded in the library file is determined by the memory-model (`-M`) option you give on the `cc` command line.

If you use the default memory-model option (`-Ms`), `cc` encodes the name of the standard library that corresponds to the defaults.

When an object file is linked, the linker looks for libraries matching the names encoded in the object file.

The result is that you do not ordinarily need to give library names on the `cc` command line. For descriptions of the situations that require you to specify libraries on the `cc` command line, see the “Specifying Libraries” section in this chapter.

The linker used is `/bin/ld`, which by default produces object files in the COFF format. If the object file uses the OMF format, you have two choices:

1. Use `cvtomf` to convert the object file to COFF format. This occurs automatically if you do not use the `-xenix` option.
2. Direct the linker, `/bin/ld`, to link the file and produce an executable file using the `x.out` format. This occurs automatically if you use the `-xenix` option.

The remainder of this section applies only to the XENIX System V linker, `ld(CP)`. The AT&T linker is also described on the manual page, `ld(CP)`.

Passing Linker Information: The **-link** Option

To pass linker options or nondefault library names to the linker, give the following options on the **cc** command line after any source- and object-file names and **cc** options:

-link

Use the rest of the command line to specify linker options, libraries, and library search paths. Note that library names can also be specified with source- and object-file names before the **-link** option on the command line, as long as the library names have the **.a** extension. These library names are searched before library names specified after the **-link** option. Refer to the following sections for more information:

- “Specifying Libraries,” to learn about specifying libraries and library search paths
- “Specifying Linker Options,” for descriptions of the linker options that apply to C.

If you use the **-link** option with the **cc** command, it must be the last option on the command line.

Specifying Libraries

To link object files with libraries other than the default library, give the names of the nondefault libraries on the **cc** command line. Library names appearing before **-link** must have the **.a** extension; library names appearing after **-link** may have blank extensions or no extensions.

Since the object file already contains the names of the correct combined library, you do not need to specify libraries unless you want to do any of the following:

Passing Linker Information: The `-link` Option

- Link with additional libraries
- Look for libraries in different locations
- Override the use of the default library

Linking with Additional Libraries

If you specify additional libraries to `cc`, the linker searches the libraries you specify *before* it searches the default library to resolve external references in the object files. It searches the libraries you specify in their order of appearance on the command line.

If a library name includes a path specification, the linker searches only that path for the library.

If you specify only a library name (without a path specification), the linker searches in the following locations to find the given library file:

- The current working directory
- Any path specifications that you give, in their order of appearance on the command line
- The default location `/lib` or `/lib/386`

If a library name without an extension appears after the `-link` option, the linker automatically supplies the `.a` extension. If you want to link a library file with an extension other than `.a`, you must specify the complete library name.

Looking in Different Locations for Libraries

You can tell the linker to look in different locations for libraries by giving a path specification on the `cc` command line.

The linker looks for the default libraries in the same order as it looks for libraries given on the command line.

Specifying Linker Options

When you use the `cc` command to invoke the linker, any linker options you specify (other than those supported by `cc` options such as `-F` and `-Fm`) must appear after the `-link` option on the command line. All options begin with the dash (`-`).

The following sections outline the rules for specifying linker options on the `cc` command line.

Abbreviations

Since linker options are named according to their functions, some of these options are quite long. You can abbreviate the options to save space and effort. Be sure that your abbreviation is unique, so that the linker can determine which option you want. The minimum legal abbreviation for each option is indicated in the syntax of the option.

Abbreviations must begin with the first letter of the option and must be continuous through the last letter typed. No gaps or transpositions are allowed.

Numerical Arguments

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

- A decimal number from 0 to 65,535
- An octal number from 0 to 0177777. A number is interpreted as octal if it starts with 0. For example, the number `10` is a decimal number, but the number `010` is an octal number, equivalent to 8 in decimal
- A hexadecimal number from 0 to 0xFFFF. A number is interpreted as hexadecimal if it starts with `0x` or `0X`. For example, `0x10` is a hexadecimal number, equivalent to 16 in decimal

Linker Options

This section summarizes some of the linker options that can be used with C programs. Note that this section does not describe all available linker options. For a complete list, refer to the `ld(CP)` manual page in the *Programmer's Reference*.

Passing Linker Information: The `-link` Option

The following linker option is most commonly used with C programs:

`-SE[GMENTS]:number`

Controls the number of segments that the linker allows a program to have. The default is 128, but you can set *number* to any value (decimal, octal, or hexadecimal) in the range 1-1024 (decimal).

For each segment, the linker must allocate some space to keep track of segment information. When you set the segment limit higher than 128, the linker allocates more space for segment information. For programs with fewer than 128 segments, you can keep the storage requirements of the linker at the lowest level possible by setting *number* to reflect the actual number of segments in the program. The linker displays an error message if the number of segments allocated is too high for the amount of memory the linker has available.

The following linker options can be used with C programs, but they perform the same actions as `cc` options. Therefore, you do not need to use them unless you are compiling and linking in separate steps.

`-M[AP][:number]`

Creates a map file. This option is equivalent to using the `-Fm` option with the `cc` command, except that you can give a *number* argument with the `-M` option. The *number* argument is any positive integer (decimal, octal, or hexadecimal) up to 65,535 (decimal) specifying how many symbols are sorted in the map listing. If no *number* argument is given, a maximum of 2048 symbols is sorted. (In practice, the number of sorted symbols is limited by the amount of free heap space.) If a *number* argument is given, the alphabetical list of symbols does not appear in the map file.

`-LI[NENUMBERS]`

Creates a map file and includes the line numbers and associated addresses of the source program. This option is equivalent to using the `-Zd` option with the `cc` command. For more information about the `-Zd` option, see the “Compiling with the `cc` Command” chapter of this guide.

`-ST[ACK]:number`

Specifies the size of the stack for your program, where *number* is any positive value (decimal, octal, or hexadecimal) up to 65,535 (decimal) representing the size, in bytes, of the stack. This option is equivalent to using the `-F` option of the `cc` command. For more information about the `-F` option, see the “Compiling with the `cc` Command” chapter of this guide.

Chapter 4

Running C Programs on System V

Introduction 4-1

Passing Command-Line Data to a Program 4-2



Introduction

After compiling and linking a program with the C Compiler and linking with the linker, you will have an executable file that can be run from the shell prompt.

System V uses the **PATH** environment variable to find executable files. You can execute your program from any directory, as long as the executable program file is in one of the directories on the path set in the **PATH** environment variable.

Your program can also be executed by other programs, and you can write it so that it will be capable of executing other programs. The **exec** and **system** routines provided in the run-time library allow your program to execute other programs. See the *C Library Guide* for a description of these routines.

System V has several other unique capabilities that your program can use if you write the program to take advantage of them. Among these capabilities are the following:

- Receiving arguments from the command line
- Reading information from the environment
- Sending a message to the shell by returning an exit code

This chapter explains how to write programs to take advantage of these features, and how to use them once your program is completed.

Passing Command-Line Data to a Program

Your C program can access data from a command line or from the environment. You can use the Bourne shell (or C-shell) commands to place data in the environment table. Command-line data are arguments that appear on the same line as the program name when you execute the program.

To pass data to your program on the command line, give one or more arguments after the program name when you execute the program. Each argument must be separated from the arguments around it by one or more spaces or tab characters, and may be enclosed in quotation marks (" "). If you want to give a single argument that includes spaces or tab characters, enclose the argument in quotation marks. For example, if your C program is called *try*, you might give it the following command line:

```
try 42 "de f" 16
```

In this case, the program will be executed and three arguments will be passed: *42*, *de f*, and *16*.

For a C program to read the data from the command line, the program should declare two variables as arguments to the **main** function. These variables and their contents are as follows:

Variable	Contents
<i>argc</i>	Number of arguments passed
<i>argv</i>	Array of strings containing arguments

By declaring these variables as arguments to **main**, you make them available as local variables in the **main** function. The following example illustrates how to declare these arguments:

```
main (argc, argv)
int argc;
char *argv[ ];
```

The number of arguments appearing on the command line is passed as the integer variable *argc*, and the command line is passed to the program as the array of strings indicated by *argv*.

Passing Command-Line Data to a Program

The first argument of any command line is the name of the program to be executed. Therefore, the program name is the first string stored in *argv*, at *argv* [0]. Since a program name must be given to run the program, the integer value of *argc* is always at least 1. Therefore, if you pass two arguments to your program, *argc* will have a value of 3 (two arguments and the program name).

The first argument following the program name is stored at *argv* [1], the second is stored at *argv* [2], and so on, to the last argument. There is a third argument passed to the **main** function: *envp*, a pointer to the environment table. This argument is an extension provided by the C Compiler to support code ported from UNIX System V and other UNIX-like systems. When specified, it follows *argv* and is declared as follows:

```
char *envp[ ];
```

Although you can use this pointer to access the value of environment settings, this usage is nonstandard and not recommended. 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 may change the location of the environment table in memory, depending on memory requirements. Therefore, the value given to *envp* at the beginning of the program's execution may not be valid throughout. In contrast, the **putenv** and **getenv** routines access the environment table properly, even when its location changes. These routines use the global variable **environ** (described in the *C Library Guide*), which always points to the correct table location.

Example

```
myprog ABC "abc e" 3 8
```

This command line executes the program named *myprog* and passes the four command-line arguments to the **main** function. The arguments are stored as null-terminated strings, and the number of arguments is stored in *argc*. To access the last argument, for example, you would use an expression like the following:

```
argv[argc - 1]
```

Since 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.



Chapter 5

Working with Memory Models

- Introduction 5-1
 - Memory Model Considerations 5-2
- Near, Far, and Huge Addressing 5-4
- Using the Standard Memory Models 5-6
 - Porting Considerations 5-7
 - Creating Small-Model Programs 5-7
 - Creating Medium-Model Programs 5-9
 - Creating Compact-Model Programs 5-9
 - Creating Large-Model Programs 5-10
 - Creating Huge-Model Programs 5-11
 - Segmentation Errors 5-12
- Using the near, far, and huge Keywords 5-14
 - Library Support for near, far, and huge 5-16
 - Declaring Data with near, far, and huge 5-16
 - Declaring Functions with the near and far Keywords 5-20
 - Pointer Conversions 5-22
- Creating Customized Memory Models 5-25
 - Code Pointers 5-26
 - Data Pointers 5-26
 - Setting Up Segments 5-27
 - Library Support for Customized Memory Models 5-29
- Setting the Data Threshold 5-30
- Naming Modules and Segments 5-31
- Specifying Text and Data Segments 5-34



Introduction

Expanding the computing power of microcomputers often means giving the computer more “space” to work in. The Intel family of microprocessors (8080, 8086, 80286, and 80386) is a good example of such growth. Each new processor was capable of addressing more memory space than its predecessor.

The 8080 processor could address 64 kilobytes (64K) of memory, using 16-bit-wide address registers. For the 8086 processor, the address space was expanded to one-megabyte (1M). However, rather than expand the size of the address registers, a second set of “segment” registers was added. These registers select 64K blocks of memory, known as segments, within the one-megabyte address space. The 16-bit address registers then select an offset from the beginning of a segment through a hardware operation equivalent to shifting the segment register 4 bits (multiplying by 16) and adding that to the offset value. This allows the 8086 to have a larger address space, yet retain the 16-bit registers of the 8080 for backward compatibility.

The same architecture is used for the 80286 processor, except that in the processor’s “protected mode” the 16-bit segment base values are shifted over 8 bits instead of 4 as in the 8086 or in the 80286’s “real mode.” The 80286 thus uses a 24-bit address, capable of addressing up to 16 megabytes of memory.

This segmented architecture can complicate the development of large programs under UNIX System V/86 and UNIX System V/286 Operating Systems. The 80386 processor with its 32-bit registers is not restricted by 64K segments; its segment size is 4096 Mbytes. It is therefore much more like non-segmented architectures such as the Motorola 68000.

However, a substantial amount of software development is done in the UNIX System V/86 and UNIX System V/286 environments. Understanding the potential stumbling blocks in the 80286 world is necessary to develop large programs effectively. Error messages such as “DGROUP allocation exceeds 64K,” “Not enough core,” and “Too big” can be incomprehensible without an understanding of segment usage under UNIX System V.

There are two types of segments under UNIX System V. Text segments (also called code segments) contain the actual machine instructions for the program. Data segments contain all of the program’s data, such as

Introduction

global variables and the stack. Under UNIX System V, the program's stack is included in the first data segment. A program's "memory model" determines how many text and data segments the program is allowed to have.

Memory Model Considerations

If you do not specify a memory model for 286 programs, **cc** uses the small memory model by default. This is sufficient for most programs.

You cannot use the small memory model if your program meets one or more of the following three conditions:

1. Your program has more than 64K of code.
2. Your program has more than 64K of data.
3. Your program contains individual arrays that need to be larger than 64K.

If you decide that the small memory model will not be adequate for your program, you have four options for larger memory models:

1. You can specify one of the other standard memory models (medium, compact, large, or huge) using one of the **-M** options.
2. You can create a mixed-model program using the **near**, **far**, and **huge** keywords.
3. You can create your own customized memory model using the **-Mstring** option.
4. Method 2 can be combined with either method 1 or method 3.

Note

The only memory models supported for 80386 code are pure and impure small model. It is important to note that all other memory models apply to only 8086 and 80286 processors. Large and huge model programs will not run on an 8086, and any program for the 8086 or 80286, of any model, will run on an 80386, although the segment size is still limited to 64K.

When generating code specifically for the 80386 processor under System V, the C compiler supports only “small” model programs, but without the 64K limit, since 80386 registers are all 32 bits wide, and its segments are over four billion bytes long. All models are supported for 86/286 code.

Choosing a memory model for a program is a trade-off between size and speed. Programs of all memory models have one “near” data segment that is addressed through the processor’s DS segment register. References to data in this segment require only a 16-bit address calculation. Large and huge model programs may have one or more additional segments. However, addressing data in these “far” segments requires loading a segment register in addition to calculating the offset within the segment.

Near, Far, and Huge Addressing

Understanding the terms “near,” “far,” and “huge” is crucial to understanding the concept of memory models. These terms indicate how data can be accessed in the pre-386 segmented architecture of the 80x86 family of microprocessors (8086, 80186, 80286).

System V loads the code and data allocated by your program into “segments” of physical memory. Each segment is up to 64K long. With the exception of impure small model programs, separate segments are always allocated for the program code and data. Impure small model programs fit all data and code into one segment. Except for this case, the minimum number of segments allocated for a program is two; these two segments, required for every program, are called “the default segments.” The small memory model uses only the two default segments. The other memory models discussed in this chapter allow more than one code segment per program, more than one data segment per program, or both.

In the 80x86 family of microprocessors, all memory addresses consist of two parts:

1. A 16-bit number that represents the base address of a memory segment
2. Another 16-bit number that gives an offset within that segment

The architecture of the 80x86 microprocessor is such that code can be accessed within the default code or data segment using just the 16-bit offset value. This is possible because the segment addresses for the default segments are always known. This 16-bit offset value is called a “near” address, and can be accessed with a “near” pointer. Since only 16-bit arithmetic is required to access any near item, near references to code or data are smaller and more efficient.

When data or code lies outside the default segments, the address must use both the segment and offset values. Such addresses are called “far” addresses, and can be accessed by using “far” pointers in a C program. Accessing far data or code items is more expensive in terms of program speed and size, but using them allows your programs to address all memory, rather than just a 64K piece.

There is a third type of address in Microsoft C: the “huge” address. A huge address is similar to a far address in that each consists of a segment value and an offset value; but the two differ in the way address arithmetic is performed on pointers. Because items (both code and data) referenced

Near, Far, and Huge Addressing

by far pointers are still assumed to lie completely within the segment in which they start, pointer arithmetic is done only on the offset portion of the address. This gain in pointer arithmetic efficiency is achieved, however, by limiting the size of any single item to 64K. Huge pointers overcome this size limitation by performing pointer arithmetic on all 32 bits of the data item's address, thus allowing data items referenced by huge pointers to span more than one segment, provided they conform to the rules outlined in the section on "Creating Huge-Model Programs."

The rest of this chapter deals with the various methods you can use to control whether your program makes far, near, or huge calls to access code or data.

Using the Standard Memory Models

The standard libraries provided with the UNIX System V Development System support five standard memory models. Using the standard memory models is the simplest way to control how your program accesses code and data in memory.

When you use the standard memory models, the compiler handles library support for you. The library corresponding to the memory model you specify is used automatically. Each memory model has its own library, except for the huge memory model, which uses the large-model library.

The advantage of using standard models for your programs is simplicity. In the standard models, memory management is specified by compiler options; since the standard models do not require the use of extended keywords, they are the best way to write code that can be ported to other systems (particularly systems that do not use segmented architectures).

The disadvantage of using standard memory models exclusively is that they may not produce the most efficient code. For example, if you have an otherwise small-model program containing a large array that pushes the total data size for your program over the 64K limit for small-model, it may be to your advantage to declare the one array with the **far** keyword, while keeping the rest of the program small model, as opposed to using the standard compact-memory model for the entire program. For maximum flexibility and control over how your program uses memory, you can combine the standard-memory-model method with the **near**, **far**, and **huge** keywords described in the “Using the near, far, and huge Keywords” section.

The **-M** option for **cc** is used to specify one of the five standard memory models (small, medium, compact, large, or huge) at compile time. These options are discussed in the next five sections.

Note

In the following sections, which describe in detail the different memory-model addressing conventions, it is important to keep in mind two common features of all five models:

1. No *single* source module can generate 64K or more of code.
 2. No *single* data item can exceed 64K, unless it appears in a huge-model program or it has been declared with the **huge** keyword.
-

Porting Considerations

When porting software to UNIX System V on Intel processors from other operating systems or other processors, it is important to recognize the differences that arise from the Intel-segmented architecture. One common assumption is that an integer occupies the same number of bytes as a pointer. While this is true for small models, it is not true for middle and large models, and can cause many problems. Another common practice is to use the integer 0 to denote a null pointer. For large and huge model programs, 0 must be typecast to an appropriate pointer (typically a pointer to a **char**, such as `(char *)0` to assure that operations with pointers work correctly.

Creating Small-Model Programs

Option

-Ms

The small-model option tells the compiler to create a program that occupies one segment for both code and data. (Impure Small Model)

Using the Standard Memory Models

Impure small-model programs are typically C programs that are short or have a limited purpose. Since code and data for these programs are limited to 64K, the total size of a small-model program can never exceed 64K. Most programs fit easily into this model. Using the `-i` flag, you can create a pure small-model program. A pure small-model program has one segment of code and one segment of data for a total of 128K.

The default in small-model programs is that both code and data items are accessed with near addresses. You can override the default for data by using the `far` or `huge` keyword, and the default for code by using the `far` keyword. The `huge` keyword is relevant only to data items—specifically arrays and pointers to arrays.

The compiler creates small-model programs by default when you do not specify a memory model. The `-Ms` option is provided for completeness; you need never give it explicitly.

Impure Small Model

An “impure” program is one in which both text and data occupy the same physical segment. Impure programs can be created for the 8086, 80186, 80286, or 80386 processor. The maximum program size is 64K for all except the 80386. The `cc` program creates impure small-model programs by default on 8086/80286 systems. They can also be created using the `-Ms` option.

Pure Small Model

A “pure” program is one where text and data are in separate segments. The text is read-only and may be shared by several processes at once. On 8086/80186/80286 processors, the maximum program size is 128K (64K code + 64K data). On the 80386 processor, the maximum program size is 8 gigabytes (4G code plus 4G data). Pure small-model programs are created using the `-i` option. In this context, `-i` stands for “instruction” rather than “impure”. This is the default on 80386 systems.

Creating Medium-Model Programs

Option

-Mm

The medium-model option provides a single segment for program data, and multiple segments for program code. Each source module is given its own code segment.

Medium-model programs are typically C programs that have a large number of program statements (more than 64K of code), but a relatively small amount of data (less than 64K). Program code can occupy any amount of space and is given as many segments as needed; total program data cannot be greater than 64K. The medium model provides a useful trade-off between speed and space, since most programs refer more frequently to data items than to code.

Creating Compact-Model Programs

Option

-Mc

The compact-model option directs the compiler to allow multiple segments for program data but only one segment for the program code.

Compact-model programs are typically C programs that have large amounts of data, but relatively small numbers of program statements. Program data can occupy any amount of space and are given as many segments as needed.

The default in compact-model programs is that code items are accessed with near addresses and data items are accessed with far addresses. You can override the default by using the **near** and **huge** keywords for data, and the **far** keyword for code.

Using the Standard Memory Models

Note

Note that in medium and compact models, **NULL** must be used carefully in certain situations. In memory models where code and data pointers are the same size, it can be used with either. **NULL** only represents a null data pointer in medium and compact models. Consider the following example:

```
void func1(char *dp)
{
.
.
.
}

void func2(char (*fp)(void))
{
.
.
.
}

main()
{
func1(NULL);
func2(NULL);
}
```

This example passes a 16-bit pointer to both *func1* and *func2* if compiled in medium model, and a 32-bit pointer to both *func1* and *func2* if compiled in compact model, unless prototypes are added to the beginning of the program to indicate the types, or an explicit cast is used on the argument to *func1* (compact model) or *func2* (medium model).

Creating Large-Model Programs

Option

-Ml

The large-model option allows the compiler to create multiple segments as needed for both code and data.

Using the Standard Memory Models

Large-model programs are typically very large C programs that use a large amount of data storage during normal processing.

The default in large-model programs is that both code and data items are accessed with far addresses. You can override the default by using the **near** and **huge** keywords for data, and the **near** keyword for code.

Creating Huge-Model Programs

Option

-Mh

The huge-model option is similar to the large-model option, except that the restriction on the size of individual data items is removed for arrays.

However, some size restrictions apply to elements of huge arrays where they are larger than 64K. To provide efficient addressing, array elements are not permitted to cross segment boundaries. This has the following implications:

1. No array element can be larger than 64K.
2. For any array larger than 128K, all elements must have a size in bytes equal to a power of 2 (that is, 2 bytes, 4 bytes, 8 bytes, 16 bytes, and so on). However, if the array is 128K or smaller, its elements may be any size, up to and including 64K.

In huge-model programs, care must be taken when using the **sizeof** operator or when subtracting pointers. The C language defines the value returned by the **sizeof** operator to be an **unsigned int** value, but the size in bytes of a huge array is an **unsigned long** value. To solve this discrepancy, the Microsoft C Compiler produces the correct size of a huge array when a type cast like the following is used:

```
(unsigned long) sizeof(huge_item)
```

Similarly, the C language defines the result of subtracting two pointers as an **int** value. When subtracting two huge pointers, however, the result may be a **long int** value. The Microsoft C Compiler gives the correct result when a type cast like the following is used:

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

Using the Standard Memory Models

Segmentation Errors

When compiling a small- or medium-model program, the compiler places all data in the data segment. However, the compiler cannot know how much total data is allocated in the segment. This is not determined until link time, when data from all the object modules are combined by the linker. If the linker finds that more than 64K have been allocated by the compiler, the linker will return the error message:

```
DGROUP allocation exceeds 64K
```

Errors with Small- and Medium-Model Programs

If this error occurs with a small- or medium-model program, there are three alternatives:

- Reduce the amount of data used by the program.
- Switch to the large-memory model.
- Create a hybrid-model program.

Hybrid models are created by declaring data using the **far** keyword and compiling with the **-Me** flag. The compiler then allocates additional segments for the far data. Care must be taken when referencing data declared in this manner. Since all the library functions will be expecting near data, far data must be transferred into a near data buffer before being passed to any library function, such as *printf()*. The hybrid model is best suited for programs with one or more large, seldom-used arrays or data structures where the rest of the program uses less than 64K of data.

Errors with Large-Model Programs

For large-model programs, the compiler divides different kinds of data into different segments. All initialized data is placed in DATA segments. Uninitialized data is placed in BSS (Blank Storage Space) segments. A large-model program may have as many DATA and BSS segments as needed, but only one near DATA segment (the segment addressed by the CPU DS register). For maximum efficiency, the compiler allocates as much data as possible to the first DATA segment. However, since the total amount of data is not known until all the object modules are linked together, more than 64K of data might be allocated for the first DATA segment. Thus, it is still possible to get the error DGROUP allocation error from the linker even with a large-model program.

Using the Standard Memory Models

One possible solution to this problem is to reduce the amount of initialized data in the program by declaring it uninitialized, then initializing at run time. Another possibility is to use the **-Mt** flag to force the compiler to move some data out of the DATA segment. Normally, the compiler places any initialized data item (single variable, array, or structure) in the first data segment if its size is less than 32767 bytes. The **-Mt** flag will lower this limit. For example, **-Mt1024** tells the compiler to place any data item larger than 1024 bytes in its own segment. The drawback to this solution is that, at run time, a segment register must be loaded for each access to that data. This may affect performance of the program. This method is most appropriate if the program contains a few large arrays or structures.

Another method of reducing the size of the first DATA segment is the use of the **-ND** compiler flag. (See “Setting Up Segments” in the “Creating Customized Memory Models” section in this chapter.) When a module is compiled with this flag, all the data in the module will be placed in its own data segment. Modules compiled using this flag should contain data only, or data and functions that do not use any data items declared in other modules.

80286 programs allocate their maximum stack size at run time; the default size is 4K. Since the stack must also fit in the first data segment, a problem will arise if there is not enough space in the first data segment to fit both the data and the stack. If the size of the data plus the size of the stack exceeds 64K, then, even if the linker will successfully link a program, the program’s first data segment will be too large for the program to run. This problem will be reported by the C shell with the message “Not enough core.” The Bourne shell will report the error with the message “too big.” The two possible solutions to this problem are to reduce the stack size, or to reduce the amount of data in the first data segment. The latter method is recommended, since reducing the stack size may cause the program to run out of stack space.

Determining Segment Size

There are three utilities that are useful for finding and correcting problems related to program segmentation. The size utility **size(CP)** takes one or more executable or object file names as arguments, and prints the size in bytes of the text, DATA, and BSS segments. This information is helpful in determining exactly how much data is used by a program, and how it is divided between the DATA and BSS segments. The **hdr(CP)** utility prints other information about an executable file, such as its memory model and stack size. The **fixhdr(CP)** utility can be used (among other things) to alter the stack size of any executable. This is useful for experimenting with different stack sizes without the need to relink, or for cases where the source code is not available.

Using the near, far, and huge Keywords

One limitation of the predefined memory-model structure is that, when you change memory models, all data and code address sizes are subject to change. However, the Microsoft C Compiler lets you override the default addressing convention for a given memory model and access items with a near, far, or huge pointer. This is done with the **near**, **far**, and **huge** keywords. These special type modifiers can be used with a standard memory model to overcome addressing limitations for particular data or code items, or to optimize access to these items, without changing the addressing conventions for the program as a whole. Table 5.1 explains how the use of these keywords affects the addressing of code or data, or pointers to code or data.

Note

The **near**, **far**, and **huge** keywords are not standard parts of the C language; they are meaningful only for systems that use a segmented architecture similar to that of the 80x86 microprocessors. Keep this in mind if you want your code to be ported to other systems.

Using the near, far, and huge Keywords

Table 5.1
Addressing of Code and Data
Declared with near, far, and huge

Key-word	Data	Pointer or Function	Arithmetic
near	Reside in default data segment; referenced with 16-bit addresses (Pointers to data are 16 bits)	Assumed to be in current code segment; referenced with 16-bit addresses (Pointers to functions are 16 bits)	Uses 16 bits
far	May be anywhere in memory, not assumed to reside in current data segment; referenced with 32-bit addresses (Pointers to data are 32 bits)	Not assumed to be in current code segment; referenced with 32-bit address (Pointers to functions are 32 bits)	Uses 16 bits
huge	May be anywhere in memory, not assumed to reside in current data segment; individual data items (arrays) can exceed 64K in size; referenced with 32-bit addresses (Pointers to data are 32 bits)	Not applicable to code	Uses 32 bits for data

In the Microsoft C Compiler, the **near**, **far**, and **huge** keywords are enabled by default. To treat these keywords as ordinary identifiers, you must give the **-Za** option at compile time. This option is useful if you are concerned with porting C programs from environments in which these are not keywords, especially if you are porting a program in which one of these words is used as a label. For further information about the use and effects of the **-Za** option, see the “Compiling with the cc Command” chapter of this guide.

Using the **near**, **far**, and **huge** Keywords

Library Support for **near**, **far**, and **huge**

When using the **near**, **far**, and **huge** keywords to modify addressing conventions for particular items, you can usually use one of the standard libraries (small, compact, medium, or large) with your program. The large-model libraries are also appropriate for use with huge-model programs. However, you must use care when calling library routines. In general, you cannot pass far pointers, or the addresses of far data items, to a small-model library routine. Of course, you can always pass the *value* of a far item to a small-model 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** keyword, it is strongly recommended that you use function prototypes with argument-type lists to ensure that all pointer arguments are passed to functions correctly. See the section on “Pointer Conversions” for more information.

To learn more about library routines and memory models, see the *C Library Guide*.

Declaring Data with **near**, **far**, and **huge**

The **near**, **far**, and **huge** keywords modify either objects or pointers to objects. When using them to declare data or code (or pointers to data or code), keep the following rules in mind:

- The keyword always modifies the object or pointer immediately to its right. In complex declarations, think of the **far** keyword and the item to its right as being a single unit. For example, in the case of the declaration:

```
char far* *p;
```

p is a pointer (whose size depends on the specified memory model) to a far pointer to **char**. See the *C Language Reference* for complete rules governing the use of special keywords in complex declarations.

- If the item immediately to the right of the keyword is an identifier, the keyword determines whether the item will be allocated in the default data segment (**near**) or a separate data segment (**far** or **huge**). For example:

Using the near, far, and huge Keywords

```
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 will hold a near address (16 bits), a far address (32 bits), or a huge address (also 32 bits). For example,

```
char far *p;
```

allocates *p* as a far pointer (32 bits) to an item of type **char**.

Examples

The examples in this section show data declarations using the **near**, **far**, and **huge** keywords.

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

The first declaration in the example allocates the array *a* in the default data segment. By contrast, the array *b* in the second declaration may be allocated in any far data segment. Since these declarations appear in a small-model program, array *a* probably represents frequently used data that was deliberately placed in the default segment for fast access. Array *b* probably represents seldom used data that might make the default data segment exceed 64K and force the programmer to use a larger memory model if the array were not declared with the **far** keyword. The second declaration uses a large array, because it is more likely that a programmer would want to specify the address allocation size for items of substantial size.

```
char a[3000];           /* large-model program */
char near b[3000];
```

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

```
char huge a[70000];    /* small-model program */
char huge *pa;
```

In this small-model program, *a* must be declared as **huge** because it is larger than 64K. Using the **huge** keyword instead of the standard huge memory model means that the price for using huge data is only paid for this one large item. Other data can be accessed quickly within the default

Using the `near`, `far`, and `huge` Keywords

segment. The pointer *pa* could be used to point to *a*. Any pointer arithmetic for *pa* (such as *pa++*) would be performed using 32-bit arithmetic.

```
char *pa;           /* small-model program */
char far *pb;
```

The pointer *pa* is declared as a near pointer to **char** in the example. The pointer is near by default since the example appears in a small-model program. By contrast, *pb* is allocated as a far pointer to **char**; *pb* could be used to point to, and step through, an array of characters stored in a segment other than the default data segment. For example, *pa* might be used to point to array *a* in the first example, while *pb* might be used to point to array *b*.

```
char far * *pa;     /* small-model program */
char far * *pa;     /* large-model program */
```

The pointer declarations in the example illustrate the interaction between the memory model chosen and the **near** and **far** keywords. Although the declarations for *pa* are identical, in a small-model program, *pa* is declared as a near pointer to an array of far pointers to type **char**, while in a large-model program, *pa* is declared as a far pointer to an array of far pointers to type **char**.

```
char far * near *pb; /* any model */
char far * far *pb;
```

In the first declaration in the example, *pb* is declared as a near pointer to an array of far pointers to type **char**; in the second declaration, *pb* is declared as a far pointer to an array of far pointers to type **char**. Note that, in this example, the **far** and **near** keywords override the model-specific addressing conventions shown in the preceding example. The declarations for *pb* would have the same effect, regardless of the memory model. The examples in the following table illustrate the **far** and **near** keywords as used in declarations in a small-model program. It also gives the size in bits of the address and the value and the type of the value.

Using the near, far, and huge Keywords

Table 5.2
Uses of 8086/80186/80286 near and far Keywords

Declaration	Size of Address	Size of Value	Type of Value
char c;	16	8	data
char far d;	32	8	data
char *p;	16	16	near pointer
char far *q;	16	32	far pointer
char * far r;	32	16	near pointer ¹
char far * far s;	32	32	far pointer ²
int foo();	16	16	integer function
int far foo();	32	16	integer function ³

Notes

- 1 This example of a near 16-bit pointer which may lie in a far data segment is unlikely to be useful; it is shown for syntactic completeness only.
- 2 This is similar to accessing data in a large-model program.
- 3 This example leads to trouble in most environments. The far call changes the CS register, and makes run time support unavailable.

The following example is from a middle-model compilation:

```
int near foo();
```

This allows a near call to the routine *foo* in a program where calls are normally **far**.

If you are using one of the keywords, it would be advisable to check the type of item in separate source files as the compiler does not do this.

If the **-M3e** option is used, the **near** keyword can address items in the program segment itself. The **near** keyword defines an item with a 32-bit address (relative to **DS**).

These keywords override the normal address length generated by the compiler for variables and functions. In pure small-model programs, **far** lets you access data and functions in segments outside the **TEXT** and **DATA** segments.

Using the near, far, and huge Keywords

The examples in the table that follows show **near** and **far** keywords used in declarations of pure small- and mixed-model programs configured with the **-M3e** option:

Table 5.3
Uses of 80386 near and far Keywords

Declaration	Address Size	Allocation Size
char c;	near (32 bits)	8 bits (data)
char far d;	far (48 bits)	8 bits (data)
char *p;	near (32 bits)	32 bits (near pointer)
char far *q;	near (32 bits)	64 bits (far pointer)
char * far r;	far (48 bits)	32 bits (near pointer) ¹
char far * far s;	far (48 bits)	64 bits (far pointer) ²
int foo();	near (32 bits)	function returning 32 bits
int far foo();	far (64/48 bits)	function returning 32 bits ³

Notes

- 1 This example is shown for syntactic completeness only.
- 2 This resembles accessing data in a large-model program.
- 3 This example creates problems in most environments. The far call changes the CS register, and makes run-time support unavailable.

Declaring Functions with the near and far Keywords

The rules for using the **near** and **far** keywords for functions are similar to those for using them with data, as specified in the following list:

- The keyword always modifies the function or pointer immediately to its right. For more information about rules for evaluating complex declarations, see the *C Language Reference*.
- If the item immediately to the right of the keyword is a function, then the keyword determines whether the function will be allocated as near or far. For example:

Using the near, far, and huge Keywords

```
char far fun( );
```

defines *fun* as a function called with a 32-bit address and returning type **char**.

- If the item immediately to the right of the keyword is a pointer to a function, then the keyword determines whether the function will be called using a near (16-bit) or far (32-bit) address. For example:

```
char (far * pfun)( );
```

defines *pfun* as a far pointer (32 bits) to a function returning type **char**.

- Function declarations must match function definitions.
- The **huge** keyword cannot be applied to functions.

Examples

```
void char far fun(void);           /* small model */
void char far fun(void)
{
    .
    .
    .
}
```

In this example, *fun* is declared as a function returning type **char**. The **far** keyword in the declaration means that *fun* must be called with a 32-bit call.

```
static char far * near fun( );    /* large model */
static char far * near fun( )
{
    .
    .
    .
}
```

In the large-model example, *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

Using the near, far, and huge Keywords

call. However, you could not pass a pointer to *fun* as an argument to another function outside the module in which *fun* was declared.

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

The small-model example 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 that if the function indicated by *pfun* has not been declared with the **far** keyword, or if it is not far by default, then calling that function through *pfun* would cause the program to fail.

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

In this final example, *pfun* is declared as a far pointer to a function that returns a far pointer to type **double**, and then assigns the address of *fun* to *pfun*. This might be used in a compact-model program for a function that is not used frequently and thus does not need to be in the default code segment. Both the function and the pointer to the function must be declared with the **far** keyword.

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 memory model used during compilation

For example, in medium-model programs, data-pointer arguments are near by default, and code-pointer arguments are far by default.

- The size of the type of the argument

If a function prototype with argument types is given, the compiler performs type-checking and enforces the conversion of actual arguments to the declared type of the corresponding formal argument. However, if no declaration is present or the argument-type list is empty, the compiler will convert pointer arguments automatically to the default type or the type of the argument, whichever is larger. To avoid mismatched arguments, you should always use a prototype with the argument types.

Examples

```

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

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

    test_fun(x, y, z); /* x will be coerced to far
                       ** pointer in compact, large,
                       ** or huge model
                       */
}

int test_fun(ptr1, ptr2, a)
    int near *ptr1;
    char far *ptr2;
    int a;

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

```

If the preceding example is compiled as a small-model program (with no memory-model options or the **-Ms** option on the **cc** command line) or medium-model program (**-Mm** option), then 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 **-Mc**, **-MI**, or **-Mh** option, both *x* and *y* are automatically converted to far pointers when they are passed to *test_fun*. Since *ptr1*, the first parameter of *test_fun*, is defined as a near-pointer argument, it takes only 16 bits of the 32 bits passed to it. The next parameter, *ptr2*, takes the remaining 16 bits passed to *ptr1*, plus 16 bits of the 32 bits passed to it. Finally, the third parameter, *a*, takes the leftover 16 bits from *ptr2*, instead of the value of *z* in the *main* function. This shifting process does not generate an error message, 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, as shown in the following example:

Using the near, far, and huge Keywords

```
/* First, declare test_fun so the compiler knows in advance
** about the near pointer argument:
*/
int test_fun(int near*, char far *, int);

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

    test_fun(x, y, z); /* now, x will not be coerced
                        ** to a far pointer; it will be
                        ** passed as a near pointer,
                        ** no matter what memory
                        ** model is used
                        */
}

int test_fun(ptr1, ptr2, a)
    int near *ptr1;
    char far *ptr2;
    int a;

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

Note that it would not be sufficient to reverse the definition order for *test_fun* and *main* in the first example to avoid pointer coercions; the pointer arguments must be declared in a forward declaration, as in the second example.

Creating Customized Memory Models

A third method of managing memory models is to combine features of the standard memory models to create your own customized memory model. You should have a thorough understanding of C memory models and the architecture of 8086 and 80286 processors before creating your own non-standard memory models, since there is no library support—other than the C start-up routines—for nonstandard memory models.

The **-Mstring** option lets you change the attributes of the standard memory models to create your own memory models. The three letters in *string* correspond to the code-pointer size, the data-pointer size, and the stack-and data-segment setups. Because the letter allowed in each field is unique to that field, you can give the letters in any order after **-M**. All three letters must be present.

The standard-memory-model options (**-Ms**, **-Mm**, **-Mc**, **-Ml**, and **-Mh**) can be specified in the **-Mstring** form. As an example of how to construct memory models, the standard-memory-model options are listed with their standard equivalents:

Standard	Custom Equivalent
-Ms	-Msnd
-Mm	-Mlnd
-Mc	-Msfd
-Ml	-Mlfd
-Mh	-Mlhd

As an example of the use of customized models, you might want to create a huge-compact model. This model would allow huge data items, but only one code segment. The option for specifying this model would be **-Mshd**.

An even more common use of customized models is to set up segments. (See the section on “Setting Up Segments,” for more information.)

If you use a customized memory model for a program that includes both far and near functions, be aware of the following issues:

Creating Customized Memory Models

- The **chkstk** library function should be called only in functions that are compiled in the same model as the library being used. (For compatibility with UNIX System V, the **chkstk** function name cannot be model-encoded.)
- The interfaces to floating-point function calls are not model encoded, so functions containing floating-point calls must be compiled with the same model as the library being used.

Code Pointers

Options

-Msxx	Near code pointers
-Mlxx	Far code pointers

Note

For the purposes of the descriptions that follow, the letters **l** (for “long”) and **s** (for “short”) are used as code pointers to distinguish them from the letters for data pointers in the memory-model string.

The letter **s** tells the compiler to generate 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 are used to address all code items. Far pointers are the default for medium-, large-, and huge-model programs.

Data Pointers

Options

-Mnxx	Near data pointers
-Mfxx	Far data pointers
-Mhxx	Huge data pointers

Creating Customized Memory Models

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 (64K) 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 (64K) because address arithmetic is performed on the entire 32 bits of the address.

Setting Up Segments

Options

-Mdxx	Sets SS = DS
-Mu[xx]	Sets SS != DS ; DS reloaded on function entry
-Mw[xx]	Sets SS != DS ; DS not reloaded on function entry

The letter **d** tells the compiler that the segment addresses stored in the **SS** and **DS** registers are equal; 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 plus all data must occupy less than 64K; thus, any data item is accessed 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 data segment. The address of this segment is stored in the **DS** and **SS** 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 64K of total data in these models, there can be no more than 64K 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. (For more information about these options, see the sections on “Setting the Data Threshold,” and “Naming Modules and Segments.”)

The letter **u** allocates different segments for the stack and the data segments. Each object file (module) is allocated its own segment for global and static data items. Note that the **-ND** option, described in “Naming

Creating Customized Memory Models

Modules and Segments," must be specified along with the letter **u** to allocate data segments other than the default. When the letter **u** and **-ND** are specified, the address in the **DS** register is saved upon entry to each function, and the new **DS** value for the module in which the function was defined is loaded into the register. The previous **DS** value is restored on exit from the function. Therefore, only one data segment is accessible at any given time. The **-ND** option can be used to combine these segments into a single segment.

If a standard memory-model option precedes it on the command line, the **-Mu** option can be specified without any letters indicating data- or code-pointer sizes. In this case, the program uses the specified memory model, but different segments are set up for the stack and data segments.

A single segment must be allocated for the stack, and its address must be stored in the **SS** register. The stack segment does not change during the execution of the program.

The letter **w**, like the letter **u**, sets up a separate stack segment, but does not automatically load the **DS** register at each module entry point. This option is typically used when writing application programs that interface with an operating system or with a program running at the operating-system level. The operating system or the program running under the operating system actually receives the data intended for the application program and places that data in a segment. Then the operating system or program must load the **DS** register with the segment address for the application program.

As with the **-Mu** option, the **-Mw** option can be specified without data- and code-pointer letters if a standard memory-model option precedes it on the command line. In this case, the program uses the specified memory model, but different segments are set up for the stack and data segments, and the **DS** register is not reloaded at each module entry point.

Even though **u** and **w** set up a separate segment for the stack, the stack's size is still fixed at the default unless this is overridden with the **-F** compiler option.

Library Support for Customized Memory Models

Most C programs make function calls to the routines in the C run-time library. Library support is provided for the five standard memory models (small, medium, compact, large, and huge) through four separate run-time libraries. (Huge and large models both use the large-model library.) 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.

Library support is provided for customized memory models where the stack and default data segments are combined into a single segment (**-Mdxx**), but not for customized memory models where these segments are different (**-Muxx**, **-Mwxx**, **-Mu**, and **-Mw**). In the latter cases, you probably need to create a customized library to be used with your customized memory model. Specify the library files and object files you want to use when linking. Be sure to use the start-up routine from the appropriate library for your memory model. Table 5.4 shows the libraries from which to extract the start-up routine for each customized memory model.

Table 5.4
Start-Up Routines for
Customized Memory Models

<u>Memory-Model Option</u>	<u>Use Start-Up from Library</u>
-Msnx ; -MS plus -Mx¹	/usr/lib/286/Sseg.o
-Msfx ; -Mshx ; -MC¹ plus -Mx	/usr/lib/286/Cseg.o
-Mlnx ; -MM plus -Mx¹	/usr/lib/286/Mseg.o
-Mlfx ; -Mlhx ; -ML plus -Mx ; -MH plus -Mx¹	/usr/lib/286/Lseg.o

¹ in the above table indicates a condition where *x* must be either **u** or **w**.

In general, library functions do not support customized memory models, since a particular run-time routine may in turn call another library routine that conflicts with your customized model.

Setting the Data Threshold

Option

-Gt[*number*]

By default, the compiler allocates all static and global data items within the default data segment in the small and medium memory models. In compact-, large-, and huge-model programs, only *initialized* static and global data items are assigned to the default data segment. The **-Gt** option causes all data items whose sizes are greater than or equal to *number* bytes to be allocated to a new data segment. When *number* is specified, it must follow the **-Gt** option immediately, with no intervening spaces. When *number* is omitted, the default threshold value is 256. When the **-Gt** option is omitted, the default threshold value is 32,767.

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

Naming Modules and Segments

Options

- NM *modulename*
- NT *textsegment*
- ND *datasegment*

“Module” is another name for an object file created by the C compiler. Every module has a name. The compiler uses this name in error messages if problems are encountered during processing. The module name is usually the same as the source-file name. You can change this name using the -NM (for “name module”) option. The new *modulename* can be any combination of letters and digits. The space between -NM and *modulename* is optional.

A “segment” is a contiguous block of binary information (code or data) produced by the C compiler. Every module except impure small 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 the order in which the segments of the program appear in memory when loaded for execution. The segments in the group named **DGROUP** are an exception.

Text and data segment names are normally created by the C compiler. These default names depend on the memory model chosen for the program. For example, in small-model programs, the text segment is named `_TEXT` and the data segment is named `_DATA`. These names are the same for all small-model modules, so all text segments from all modules are loaded as one contiguous block, and all data segments from all modules form another contiguous block.

In medium-model programs, the text from each module is placed in a separate segment with a distinct name, formed by using the module base name along with the suffix `_TEXT`. The data segment is named `_DATA`, as in the small model.

In compact-model programs, the data from each module are placed in a separate segment with a distinct name, formed by using the module base name along with the suffix `_DATA`. The exception to this is initialized global and static data, which are put in the default data segment `_DATA`. The code segment is named `_TEXT`, as in the small model.

In large- and huge-model programs, the text and data from each module are loaded into separate segments with distinct names. Each text segment

Naming Modules and Segments

is given the name of the module plus the suffix `_TEXT`. The data from each segment is placed in a private segment with a unique name (except for initialized global and static data placed in the default data segment). The naming conventions for text and data segments are summarized in Table 5.5.

Table 5.5
Segment-Naming Conventions

Model	Text	Data	Module
Small	<code>_TEXT</code>	<code>_DATA</code>	<i>filename</i>
Medium	<i>module</i> <code>_TEXT</code>	<code>_DATA</code>	<i>filename</i>
Compact	<code>_TEXT</code>	<code>_DATA</code> ¹	<i>filename</i>
Large	<i>module</i> <code>_TEXT</code>	<code>_DATA</code> ¹	<i>filename</i>
Huge	<i>module</i> <code>_TEXT</code>	<code>_DATA</code> ¹	<i>filename</i>

“1” in the above table indicates the name of default data segment; other data segments have unique private names.

You can override the default names used by the C compiler (thus overriding the default loading order) by using the `-NT` (for “name text”) and `-ND` (for “name data”) options. These options set the names of the text and data segments in each module being compiled to the given name. The *textsegment* argument used with the `-NT` option and the *datasegment* argument used with the `-ND` option can be any combination of letters and digits. The space between `-NT` and *textsegment*, like the space between `-ND` and *datasegment*, is optional.

If you use the `-ND` option to change the name of the default data segment, your program can no longer assume that the address contained in the stack segment register (`SS`) is the same as the address in the data segment register (`DS`). You must therefore compile your program either with the `-Mstring` form of the memory-model option and the `u` option for the segment-setup letter, or with the `-M` option for a standard memory model followed by the `-Mu` option, as in the following example:

```
cc -Ms -Mu -ND DATA1 prog1.c
```

Use of the `-Mu` option forces the compiler to generate code to load `DS` with the correct data-segment value on entry to the code. See the section on “Creating Customized Memory Models,” for more information on the

Naming Modules and Segments

options. All modules whose data segments have the same name have these segments combined into a single segment named *DATA1* at link time.

Specifying Text and Data Segments

Pragmas

```
#pragma alloc_text (textsegment, function1[, function2]...)
#pragma same_seg (variable1[, variable2]...)
#pragma data_seg ([[segmentname]])
```

The **alloc_text** pragma gives you source-level control over the segment to which particular functions are allocated. The **same_seg** pragma provides information the compiler can use to generate better code. The **data_seg** pragma allows you to specify the name of the data segment that subsequent load-DS functions use.

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 may appear either before or after the functions are declared or called. Any functions specified in an **alloc_text** pragma must be either 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 is 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).

The **data_seg** pragma specifies the name of the data segment that subsequent load-ds functions should use. A “load-ds” function loads its own data segment upon entry. For more information about load-ds see Appendix A of this manual. In addition, **data_seg** causes the named segment to contain all data that would otherwise be allocated in the **DATA** segment (all subsequent initialized static and global data). If you omit the *segmentname* parameter, the compiler uses the segment name specified in the **-ND** option, or, if that option is absent, the default group **DGROUP**, since **DGROUP** is not a segment.

Chapter 6

Improving Program Speed

Introduction 6-1

Using Register Variables 6-2

Optimization Options and Pragmas 6-4

 Default Optimization 6-4

 Generating Intrinsic Functions 6-4

 Relaxing Alias-Checking 6-5

 Performing Loop Optimizations 6-5

 Removing Stack Probes 6-6

 Maximum Optimization 6-6

Choosing the Function-Calling Convention 6-7

Efficiency in Large Data Models 6-8

 Changing Addressing with near, far, and huge Keywords 6-8

 Setting the Data Threshold 6-9

 Controlling Segments Used for Allocation 6-9

Efficiency in Large Code Models 6-10



Introduction

This chapter describes a number of ways that you can improve the execution speed of programs compiled with the C Compiler. These techniques include:

- Using register variables
- Using optimization options and pragmas
- Choosing function-calling conventions
- Choosing and adjusting memory models

Where applicable, this chapter discusses the interactions between these techniques and the trade-offs involved in using them.

Using Register Variables

One common way to write a program for maximum speed is to declare selected local (**auto**) variables with **register** storage class. The declaration of a register variable requests the compiler to use machine registers when allocating space for the variable, if possible. The **register** storage class can be specified for any variable, but some classes of variables, such as structures, cannot be stored in registers.

Up to two register variables may be allocated per function. In lexical order, the 8086 and 80286 compilers take the first two variables with **register** storage class that meet the size criteria. The 80386 compiler takes the first three variables. Any later requests for **register** storage class are ignored, so be sure to declare the most important register variables first. The compiler deallocates the register when the variable is no longer being used. You may also want to declare register variables in parallel scope to achieve the effect of having more than two register variables per function.

The C Compiler automatically uses registers for variables within loops. Using register declarations for such variables may interfere with optimal loop code; you can experiment with various combinations of register and nonregister declarations to determine which combinations give the best results.

Register declarations can be used effectively for values, especially pointers, that appear outside of loops. Since a certain amount of code is required to save and restore registers, register declarations must be applied to values that are accessed at least three times within a function to cause any improvement in program speed.

Example

```

find_string(arr_of_chars, string)
char *string;
char *arr_of_chars[];
{
    int ix = 0;
    register char *q;
    while (*(q = string)) {      /* string is not null */
        register int i = ix;

        /* search for entry whose first character
         * matches first character of string, if any
         */

        while (i < MAX_ARR_SIZE && *arr_of_chars[i] != *q)
            i++;
        if (i == MAX_ARR_SIZE)
            return(1); /* no matching entry */
        ix = i;
    }

    /* we've found an entry in arr_of_chars which
     * might match string */

    {
        register char *p = arr_of_chars[ix];
        while (*p && *q && *p++ == *q++)
            ;
        if ((*p - *q) == 0)
            return(0) /* they match, return 0 */
        /* otherwise continue checking for possible
         * matches
         */
    }
}
}

```

In this example, the function named *find_string* actually has three register variables: *q*, *i*, and *p*. The function can use all three variables because *i* is through being used by the time *p* is needed. Introducing the *ix* variable to save the pointer from block-to-block speeds execution considerably because most work is being done in register variables.

Optimization Options and Pragmas

The `cc` compiler/linker driver provides a number of optimization options (`-O`, followed by one or more letters) that can improve program speed. In addition, the C Compiler includes several pragmas that allow you to control some of these optimizations on a local basis within a source program. The following sections outline these `cc` options and pragmas and their effects.

Default Optimization

If no `-O` option is given, the compiler uses the `-Ot` option, which optimizes programs for execution speed. However, this option does not enable loop optimizations or intrinsics. Some optimizations, such as long shifts, may be performed in line rather than using helper functions.

Generating Intrinsic Functions

The `-Oi` option generates intrinsic forms of the following functions:

- `memset`, `memcpy`, `memcmp`
- `strset`, `strcpy`, `strcmp`, `strcat`
- `inp`, `outp`
- `_rotl`, `_rotr`, `_lrotl`, `_lrotr`,
- `min`, `max`, `abs`

Intrinsics may be generated as in-line code or with different calling sequences. In general, using intrinsics increases program size but improves program speed. Note that the intrinsic forms of some functions may have slightly different semantics: for example, the intrinsic form of the `memcpy` function in compact- and large-model programs cannot handle huge arrays, but the function form can.

As with `-Ot`, this option may increase program size due to the additional code generated in line for each function. However, program execution is faster because no instructions for calling and returning from functions need to be performed.

The **intrinsic** pragma can be used to specify intrinsic functions on a local basis for any of the functions listed above. For information about the use of the **intrinsic** pragma, see the “Compiling with the **cc** Command” chapter of this guide.

Relaxing Alias-Checking

The **a** option letter can be used with the **l**, **s**, or **t** option letter to relax the assumptions the compiler makes about the use of “aliases” in the program. Use of the **-Oa** option can reduce the size of executable files and speed program execution. This is especially recommended when you also specify the **-Ol** option, since the compiler can detect a number of loop optimizations when the **-Oa** option is in effect that it cannot detect when **-Oa** is not in effect. However, before you specify **-Oa**, you must make sure that your program does not use multiple aliases to refer to the same memory location either directly or indirectly. For example, a program might do this indirectly in functions that operate on a communal variable and a pointer argument, or on multiple pointer arguments.

The **-Oa** option can be specified safely for programs that include calls to functions with address-type arguments. In this case, the compiler assumes that all variables whose addresses are passed to the function are modified, even if **-Oa** is specified.

In the cases noted above, the use of **-Oa** is most likely to cause incorrect optimizations within basic blocks (where most optimizations are applied) and within whole loop bodies (where loop optimizations are applied). In these cases, **-Oa** can still be specified safely even if aliases are used in the program, provided that no memory location is referenced by more than one name within any basic block or (if loop optimization is enabled) any loop body.

For more information and specific examples, see the “Compiling with the **cc** Command” chapter of this guide.

Performing Loop Optimizations

The **-Ol** option tells the compiler to perform loop optimizations. For best performance, use **-Ol** in conjunction with the **a** option letter (**-Oal**), which relaxes the assumptions the compiler makes about the use of aliases in the program. Using **-Oal** instead of just **-Ol** allows the compiler to detect many loop optimizations that it could not otherwise detect. For information about possible restrictions on the uses of the **-Oa** option, see the “Compiling with the **cc** Command” chapter of this guide.

Optimization Options and Pragmas

You can control loop optimization on a local basis by specifying the **loop_opt** pragma. Loop optimization is turned off for any functions following **#pragma loop_opt(off)** and turned on for any functions following **#pragma loop_opt(on)** in a source program. This pragma overrides any loop optimization specified on the **cc** command line.

Removing Stack Probes

The **-Gs** option, described in the “Compiling with the **cc** Command” chapter of this guide, speeds program execution slightly by removing calls to stack-checking routines known as “stack probes.” Stack probes verify that a program has enough stack space to allocate required local variables. The potential disadvantage in removing stack probes is that stack-overflow errors may occur without generating a diagnostic message. However, this technique can be useful for programs that are known not to exceed the available stack space.

You can also control stack checking on a local basis by specifying the **check_stack** pragma. Stack checking is turned off for any functions following a **#pragma check_stack(off)** pragma and turned on for any functions following a **#pragma check_stack(on)** pragma in the source program. This pragma overrides the removal of stack checking specified on the **cc** command line.

Maximum Optimization

The **-Ox** option combines the **-Ot**, **-Oi**, **-Oa** and **-Ol** optimization options described in this section. Provided that the restrictions outlined for each optimization option do not apply, you can use the **-Ox** option to create the fastest possible program.

Choosing the Function-Calling Convention

Because C functions can accept variable numbers of arguments, arguments passed to these functions must be pushed on the stack from right to left, with the first argument in the list being the last one pushed. In addition, the calling function, rather than the called function, is responsible for removing arguments from the stack.

This convention results in somewhat slower programs than the alternative convention used by FORTRAN and Pascal. In the FORTRAN/Pascal convention, arguments are pushed on the stack from left to right, in the order in which they are passed to the function, and the called function removes arguments from the stack. Since the code for removing arguments appears only once (in the called function) for the FORTRAN/Pascal convention, rather than multiple times (every time a function is called) as in the C convention, and since most programs have fewer functions than function calls in a program, the FORTRAN/Pascal calling convention usually results in smaller, faster programs.

You can specify the FORTRAN/Pascal calling convention for all functions in a module by compiling with the `-Gc` option. The trade-off for improved program speed is that you cannot call functions that use the C calling convention or take variable numbers of arguments unless you declare these functions, or pointers to these functions, with the `cdecl` keyword, which specifies the normal C calling conventions for these functions.

If you do not want to specify the FORTRAN/Pascal convention for a whole module, you can declare individual functions or pointers to functions with the `pascal` or `fortran` keyword. Either of these keywords tells the compiler that the function uses the FORTRAN/Pascal calling conventions.

Efficiency in Large Data Models

Programs are most efficient when their data reside in the default data segment, that is, when the data can be accessed with 16-bit (near) addresses. The C Compiler provides two standard memory models in which all data reside in the default data segment: the small (default) model and the medium model. The customized memory models that use near data pointers (`-Mnxx`) also restrict program data to the default data segment. Programs compiled with these models are restricted to 64K of total data.

For programs compiled with the compact, large, and huge memory models, the compiler creates a default data segment containing all initialized global and static data and creates an additional data segment for each program module. Since accessing data outside the default data segment is slower than accessing data within the default data segment, programs will run faster if as many of their variables as possible are declared in such a way that they are allocated in the default data segment. One way to accomplish this is to initialize variables at the time you declare them. This section discusses other ways of controlling the allocation of data for large data models.

Changing Addressing with near, far, and huge Keywords

The **near**, **far**, and **huge** keywords allow you to specify explicitly the addressing used for particular data items and functions. These keywords override the default addressing conventions specified by the program's memory model. Thus, you can use them to improve the speed of access to program data. For example, you can tell the compiler to allocate data items in the default data segment for a compact-, large-, or huge-model program by declaring the items (or pointers to the items) with the **near** keyword. Alternatively, if a program has a small amount of code and data except for one particularly large array, you could compile the program with the small or medium memory model and declare the array with the **far** or **huge** keyword.

The disadvantage of using these keywords is that they are specific to the MS-DOS/UNIX implementation of C and, thus, are not portable to other operating environments.

For more information about **near**, **far**, and **huge** and for examples of their use, see the "Working with Memory Models" chapter in this guide.

Setting the Data Threshold

Another way to control allocation in large data models is to set a data threshold by compiling with the **-Gt** option. This option is especially useful if your program uses more than 64K of initialized static and global data and does not fit in the default data segment. Any data items larger than the value you specify are allocated to their own data segments.

Controlling Segments Used for Allocation

If programs compiled with large data models use external far data items, you can tell the compiler which items reside in the same far data segment by using the **same_seg** pragma. The variables you specify in this pragma help the optimizer recognize common subexpressions involving data loads. Note that you must also compile your program with the **-ND** option to ensure that the variables you specify are allocated in the same segment.

For a description of the **-ND** option and the **same_seg** pragma, see the “Working with Memory Models” chapter of this guide.

Efficiency in Large Code Models

One linker option, **-T**, can result in smaller and faster executable files and improved program-load times for programs that explicitly or implicitly use far function calls.

The **-T** option tells the linker to optimize far calls to procedures that lie in the same segment as the caller. When you specify the **-T** option, the linker optimizes 32-bit calls to procedures in the same segment as the calling procedure. Since the segment addresses of the calling and called procedures are the same, only a 16-bit call is required. If the **-T** option is given, the linker removes the far call and replaces it with code that first places **CS** on the stack, then makes a near call. The called procedure still returns with a far (32-bit) return instruction. However, because both the code segment (stored in **CS**) and the near address are on the stack, the far return is done correctly. The linker also adds a **NOP** instruction so that the five-byte far call is replaced by exactly five bytes of instructions.

Note

You may not want to use the **-T** option if your program includes system-level assembly language routines or if you are linking object files that were compiled with a different C compiler.

Chapter 7

Object and Executable File Formats

Introduction 7-1

iAPX-286 and -386 System Architecture 7-2
 Memory Management 7-2
 Logical Address Space 7-2
 Logical-to-Physical Address Translation 7-2

The Intel Object Module Format 7-4

Definition of Terms 7-6

Module Identification and Attributes 7-9

Segment Definition 7-10

Segment Addressing 7-11

Symbol Definition 7-12

Indices 7-13

Conceptual Framework for Fixups 7-14
 LOCATION Types 7-15

Self-Relative Fixups 7-19

Segment-Relative Fixups 7-20

Record Order 7-22

Introduction to the Record Formats 7-24
 Title and Official Abbreviation 7-24
 The Boxes 7-24
 Rectyp 7-24
 Record Length 7-25
 Name 7-25

Number 7-25
Repeated or Conditional Fields 7-25
Chksum 7-25
Bit Fields 7-26
T-Module Name 7-26
Name 7-27
Seg Attr 7-27
Segment Length 7-29
Segment Name Index 7-30
Class Name Index 7-30
Overlay Name Index 7-30
Group Name Index 7-31
Group Component Descriptor 7-31
Name 7-32
Eight-Leaf Descriptor 7-32
Public Base 7-34
Public Name 7-35
Public Offset 7-35
Type Index 7-36
External Name 7-36
Type Index 7-37
Line-Number Base 7-38
Line-Number 7-38
Line Number Offset 7-38
Segment Index 7-39
Enumerated Data Offset 7-39
Data 7-39
Segment Index 7-39
Iterated Data Offset 7-40
Iterated Data Block 7-40
Repeat Count 7-40
Block Count 7-40
Content 7-41
Thread 7-42
Fixup 7-43
Mod Type 7-46
Comment Type 7-48
Comment 7-49

Numeric List of Record Types 7-50

Type Representations for Communal Variables 7-51

The Segmented x.out Format 7-54
General Description of x.out 7-54
Example of File Layout 7-56

Iterated Segments 7-56
Non-Iterated Segments and Implicit bss 7-57
Large Model 7-58
Special Header Fields 7-58
Symbol Table 7-58
UNIX System V Executable Format 7-59
Selected Portions of Include Files 7-60



Introduction

This chapter describes the object and executable file formats used by UNIX System V/386 release 3.2.

When the **-xenix** option is used with **cc**, the file format used is the Intel Object Module Format, or OMF. When **cc** is used without the **-xenix** option, the file format used is the AT&T Common Object File Format, or COFF.

This chapter is divided into four sections. The first provides you with a brief introduction to the architecture of the iAPX-286 and -386 processors.

The second section discusses OMF which is used by all Microsoft language tools. The implementation of this format makes it possible to compile programs that run in the UNIX System V, XENIX and MS-DOS environments.

The third section provides a brief description of our implementation of the **x.out** format in a segmented environment. For detailed information, see the **x.out** header file (`/usr/include/x.out.h`).

The fourth section describes the Common Object Module Format (COFF) used by the AT&T development system. You can find additional information in the **a.out** header file (`/usr/include/a.out.h`).

iAPX-286 and -386 System Architecture

UNIX System V runs on the 80386 processor in protected mode. This section provides a general introduction to the architecture of protected-mode operation. It does not discuss the various 80386 paging mechanisms. For an in-depth discussion of the iAPX-286 and iAPX-386, refer to the appropriate programmer's reference manual published by Intel.

Memory Management

Memory management provides a mapping from the logical addresses used within a program to physical machine addresses. This serves two purposes:

- Programs are not tied to any particular physical address.
- Access permissions to particular areas of memory can be controlled.

Logical Address Space

The mapping of virtual addresses to physical addresses is achieved by means of descriptor tables which are themselves resident in memory. At any given moment, there are two alternate descriptor tables available: the Global Descriptor Table (GDT) and the Local Descriptor Table (LDT).

The UNIX System V kernel uses the GDT to map the kernel's virtual address space. Each user process has its own LDT as part of its per-process data which maps the logical address space of the process.

Each entry in a descriptor table specifies the base address, length, and access permissions of a particular segment of physical memory.

Logical-to-Physical Address Translation

Logical addresses consist of two parts: a segment selector used to select a particular descriptor table entry, and an offset added to the base address found in the descriptor table to give a physical memory address.

iAPX-286 and -386 System Architecture

The segment selector is a 16-bit number containing three pieces of information:

1. The Request Privilege Level (RPL) is encoded as the low-order two bits of the selector. The RPL is a feature of the system architecture protection scheme. Segment selectors in user processes always have both of these bits set to indicate RPL 3, the lowest privilege level.
2. The Table Indicator (TI) is encoded as the next most significant bit (bit 2). The TI indicates whether address translation will use the GDT (TI = 0) or the LDT (TI = 1). User processes can only access the LDT; therefore the TI for a segment selector in a user process is always 1.
3. The Index field is encoded as the high-order 13 bits of the selector. This is used to index into the appropriate descriptor table and select a particular entry.

When a descriptor table entry has been selected, the offset is added to the base address in physical memory to form a physical address.

Depending on the characteristics of the segment, as defined in the descriptor table, the offset may be a 16- or 32-bit number. The offset will be 16 bits on an 80286 processor or in a 16-bit segment on an 80386 processor. The 32-bit offsets apply only to the 80386.

The Intel Object Module Format

This section presents the object record formats that define the relocatable object language for the iAPX-86 family of microprocessors. The 8086 object language is the output of all language translators that have the 8086 as their target processor and are linked by the link editor. The 8086 object language is input and output for object language processors such as linkers and librarians.

Note

Except where otherwise noted, references to the 8086 in this document refer to the 8086/80286/80386 processors. In general, the 8086/80286 references are made to 16-bit offsets and 64K segment offsets, which do not apply to the 80386.

The 8086 object module formats permit you to specify relocatable memory images that may be linked together. The formats allow efficient use of the memory-mapping facilities of the 8086 microprocessor.

The following record formats, as described in this chapter, are supported. Those formats preceded by an asterisk (*) deviate from the Intel specification.

The Intel Object Module Format

Object Module Record Formats

T-Module Header Record
List of Names Record
*Segment Definition Record
*Group Definition Record
*Type Definition Record
*Public Names Definition Record
*External Names Definition Record
*Line Numbers Record
Logical Enumerated Data Record
Logical Iterated Data Record
Fixup Record
*Module End Record
Comment Record

Definition of Terms

Definition of Terms

The following terms are fundamental to the 8086 relocation and linkage.

OMF

Object Module Formats

MAS

Memory Address Space. Note that the MAS is distinguished from actual memory, which may occupy only a portion of the MAS.

MODULE

An “inseparable” collection of object code and other information.

T-MODULE

A module created by a translator, such as C, Pascal, or FORTRAN.

The following restrictions apply to object modules:

- Every module needs a name. Translators provide names for T-Modules, giving a default name (possibly the filename or a null name) if neither source code nor user specifies otherwise.
- Every T-Module in a collection of linked modules must have a different name so that symbolic debugging systems can distinguish the various line numbers and local symbols. This restriction is not required by **ld**.

FRAME

A contiguous region of MAS that can be addressed using a single segment register. This concept is useful because the content of the four 8086 segment registers defines four (possibly overlapping) FRAMES. No 16-bit address in the 8086 code can access a memory location outside of the current four FRAMES. On an 8086, a FRAME must begin on a paragraph boundary (that is, a multiple of 16 bytes). On 80286 and 80386 processors, this restriction does not apply. On an 80386, a FRAME is a region of up to (2**32) bytes addressed by a single segment register.

LSEG

Logical Segment. A contiguous region of memory whose contents are determined at translation time (except for address-binding). Neither size nor location in MAS is necessarily determined at translation time. Size, although partially fixed, may not be final because the LSEG may be combined at LINK time with other

Definition of Terms

LSEGS, forming a single LSEG. On 8086/80286 processors, an LSEG must not be larger than 64K, so that it can fit in a FRAME. This means that any byte in an LSEG may be addressed by a 16-bit offset from the base of a FRAME covering the LSEG. An 80386 LSEG may be as much as (2^{32}) bytes in size and any byte in it may be addressed by a 32-bit offset from the base of the FRAME containing the LSEG.

PSEG

Physical Segment. This term is equivalent to FRAME. Some people prefer PSEG to FRAME because the terms PSEG and LSEG reflect the physical and logical nature of the underlying segments.

FRAME NUMBER

This term is only used in reference to 8086 processors, or 80286/80386 processors operating in real address mode. Every FRAME begins on a paragraph boundary. The paragraphs in MAS can be numbered from 0 through 65535. These numbers, each of which defines a FRAME, are called FRAME NUMBERS.

PARAGRAPH NUMBER

This term is equivalent to FRAME NUMBER.

PSEG NUMBER

This term is equivalent to FRAME NUMBER.

GROUP

A collection of LSEGS defined at translation time, whose final locations in MAS are constrained such that there is at least one FRAME that covers (contains) every LSEG in the collection.

The notation Gr A(X,Y,Z) means that LSEGS X, Y, and Z form a group whose name is A. The fact that X, Y, and Z are all LSEGS in the same group does not imply any ordering of X, Y, and Z in MAS, nor does it imply any contiguity between X, Y, and Z.

The link editor does not currently allow an LSEG to be a member of more than one group. The link editor ignores all attempts to place an LSEG in more than one group.

CANONIC

Any location in the 8086 MAS is contained in exactly 4096 distinct FRAMES, but one of these FRAMES can be distinguished because it has a higher FRAME NUMBER. This distinguished FRAME is called "the canonic FRAME" of the location. The canonic FRAME of a given byte is the FRAME so chosen that the byte's offset from that FRAME lies in the range 0 to 15 (decimal). Thus, if FOO is a symbol defining a memory location, one may speak of the "canonic FRAME of FOO," or of "FOO's canonic

Definition of Terms

FRAME.” By extension, if S is any set of memory locations, then there exists a unique FRAME that has the lowest FRAME NUMBER in the set of canonic FRAMES of the locations in S. This unique FRAME is called the canonic FRAME of the set S. Thus, we may speak of the canonic FRAME of an LSEG, or of a group of LSEGs.

SEGMENT NAME

LSEGs are assigned segment names at translation time. These names serve two purposes:

- They play a role at LINK time in determining which LSEGs are combined with other LSEGs.
- They are used in assembly source code to specify groups.

CLASS NAME

LSEGs may optionally be assigned class names at translation time. Classes define a partition on LSEGs: two LSEGs are in the same class if they have the same class name.

The link editor applies the following semantics to class names. The class name “CODE” or any class name whose suffix is “CODE” implies that all segments of that class contain only code and may be considered read-only. Such segments may be overlaid if the user specifies the module containing the segment as part of an overlay.

OVERLAY NAME

LSEGs may optionally be assigned overlay names. The overlay name of an LSEG is ignored by `ld` (version 2.40 and later versions), but it is used by Intel relocation and linkage products.

COMPLETE NAME

The complete name of an LSEG consists of the segment name, class name, and overlay name. LSEGs from different modules are combined if their complete names are identical.

Module Identification and Attributes

A module header record is always the first record in a module and provides the module name.

In addition to a name, a module may have the attribute of being a main program and may have a specified starting address. When you are linking multiple modules together, only one module with the main attribute should be given.

In summary, modules may or may not be main and may or may not have a starting address.

Segment Definition

A module is a collection of object code defined by a sequence of records produced by a translator. The object code represents contiguous regions of memory whose contents are determined at translation time. These regions are called LOGICAL SEGMENTS (LSEGS). A module defines the attributes of each LSEG. The SEGMENT DEFINITION RECORD (SEGDEF) is the vehicle by which all LSEG information (name, length, memory alignment, and so on) is maintained. The LSEG information is required when multiple LSEGS are combined and when segment addressability is established. (See “Segment Addressing”.) The SEGDEF records must follow the first header record.

Segment Addressing

The 8086/80286 addressing mechanism provides segment base registers from which a 64-Kbyte region of memory, called a FRAME, may be addressed. There are one code-segment base register (CS), two data-segment base registers (DS, ES), and one stack-segment base register (SS). The 80386 has two additional segment registers: FS and GS, and can address up to $(2^{**}32)$ bytes of memory from each segment register.

The greatest possible number of LSEGs that may make up a memory image far exceeds the number of available base registers. Thus, base registers may require frequent loading. This would occur in a modular program with many small data and/or code LSEGs.

Since such frequent loading of base registers is undesirable, it is a good strategy to collect many small LSEGs together into a single unit that fits in one memory frame so that all the LSEGs may be addressed using the same base register value. This addressable unit is a GROUP. See the section "Definition of Terms" in this chapter.

To have addressability of objects within a GROUP, each GROUP must be explicitly defined in the module. The GROUP DEFINITION RECORD (GRPDEF) provides a list of constituent segments either by segment name or by segment attribute such as "the segment defining symbol FOO" or "the segments with class name ROM."

The GRPDEF records within a module must follow all SEGDEF records because GRPDEF records can reference SEGDEF records when defining a GROUP. The GRPDEF records must also precede all other records except header records, as the linker must process them first.

Symbol Definition

The **ld** command supports three different types of records that fall into the class of symbol definition records. The two most important types are PUBLIC NAMES DEFINITION RECORDs (PUBDEFs) and EXTERNAL NAMES DEFINITION RECORDs (EXTDEFs). These types are used to define globally visible procedures and data items and to resolve external references. In addition, TYPDEF records are used by **ld** for the allocation of communal variables. (See the section “Type Representations for Communal Variables” later in this chapter.)

Indices

“Index” fields appear throughout this document. An index is an integer that selects some particular item from a collection of such items. Some examples are NAME INDEX, SEGMENT INDEX, GROUP INDEX, EXTERNAL INDEX, and TYPE INDEX.

In general, indices must assume values quite large (that is, much larger than 255). Nevertheless, a great number of object files will contain no indices with values greater than 50 or 100. Therefore, indices will be encoded in one or two bytes, as required.

The high-order (left-most) bit of the first, and possibly the only, byte determines whether the index occupies one byte or two. If the bit is 0, then the index is a number between 0 and 127, occupying one byte. If the bit is 1, then the index is a number between 0 and 32K-1, occupying two bytes, and is determined as follows: the low-order 8 bits are in the second byte, and the high-order 7 bits are in the first byte.

Conceptual Framework for Fixups

A “fixup” is some modification to object code, requested by a translator, performed by **ld**, achieving address-binding.

Note

This definition of “fixup” accurately represents the viewpoint maintained by **ld**. Nevertheless, the link editor can be used to achieve modifications of object code (that is, “fixups”) that do not conform to this definition. For example, the binding of code to either hardware floating-point or software floating-point subroutines is a modification to an operation code, where the operation code is treated as if it were an address. The previous definition of “fixup” is not intended to disallow or disparage object code modifications.

8086 translators specify a fixup with four data items:

- The place and type of a LOCATION to be fixed up.
- One of two possible fixup MODES.
- A TARGET, which is a memory address to which LOCATION must refer.
- A FRAME defining a context within which the reference takes place.

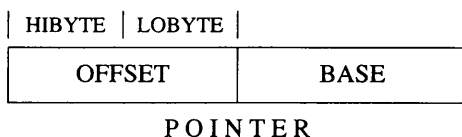
There are 5 types of LOCATION: a POINTER, a BASE, an OFFSET, a HIBYTE, and a LOBYTE.

The vertical alignment of the following figure illustrates four points. (Remember that the high-order byte of a word in 8086 memory is the byte with the higher address.) The **ld** command does not require the presence of the high- or low-order complement of these items. (For instance, in the case of HIBYTE, a high-order word, it doesn’t matter if the low-order word is present.)

- A BASE is the high-order word of a pointer.
- An OFFSET is the low-order word of a pointer.

Conceptual Framework for Fixups

- A HIBYTE is the high-order half of an OFFSET.
- A LOBYTE is the low-order half of an OFFSET.



LOCATION Types

A LOCATION is specified by two data: (1) the LOCATION type, and (2) where the LOCATION is. The first is specified by the LOC subfield of the LOCAT field of the FIXUP record; the second is specified by the DATA RECORD OFFSET subfield of the LOCAT field of the FIXUP record.

The link editor supports two fixup MODES: “self-relative” and “segment-relative.”

Self-Relative fixups support the 8- and 16-bit offsets that are used in the CALL, JUMP and SHORT-JUMP instructions. Segment-Relative fixups support all other addressing modes of the 8086.

The TARGET is the location in MAS being referenced. (More explicitly, the TARGET may be considered the lowest byte in the object being referenced.) A TARGET is specified in one of eight ways. There are four “primary” ways, and four “secondary” ways. Each primary way of specifying a TARGET uses two kinds of data: an INDEX-or-FRAME-NUMBER ‘X’, and a displacement ‘D’.

- (T0) X is a SEGMENT INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.
- (T1) X is a GROUP INDEX. The TARGET is the Dth byte in the LSEG identified by the INDEX.
- (T2) X is an EXTERNAL INDEX. The TARGET is the Dth byte following the byte whose address is (eventually) given by the External Name identified by the INDEX.
- (T3) X is a FRAME NUMBER. The TARGET is the Dth byte in the FRAME identified by the FRAME NUMBER (that is, the address of TARGET is $(X*16)+D$).

Conceptual Framework for Fixups

Each secondary way of specifying a TARGET uses only one data item: the INDEX-or-FRAME-NUMBER X. An implicit displacement equal to zero is assumed.

- (T4) X is a SEGMENT INDEX. The TARGET is the 0th (first) byte in the LSEG identified by the INDEX.
- (T5) X is a GROUP INDEX. The TARGET is the 0th (first) byte in the LSEG in the specified group that is eventually LOCATED lowest in MAS.
- (T6) X is an EXTERNAL INDEX. The TARGET is the byte whose address is the External Name identified by the INDEX.
- (T7) X is a FRAME NUMBER. The TARGET is the byte whose 20-bit address is $(X*16)$.

Note

The link editor does not support methods T3 and T7.

The following nomenclature is used to describe a TARGET:

TARGET:	SI (<segment name>), <displacement>	[T0]
TARGET:	GI (<group name>), <displacement>	[T1]
TARGET:	EI (<symbol name>), <displacement>	[T2]
TARGET:	SI (<segment name>)	[T4]
TARGET:	GI (<group name>)	[T5]
TARGET:	EI (<symbol name>)	[T6]

The following examples illustrate how this notation is used:

TARGET: SI(CODE),1024	The 1025th byte in the segment "CODE".
TARGET: GI(DATAAREA)	The location in MAS of a group called "DATAAREA".
TARGET: EI(SIN)	The address of the external subroutine "SIN".

Conceptual Framework for Fixups

TARGET: EI(PAYSCHEDULE), 24 The 24th byte following the location of an EXTERNAL data structure called "PAYSCHEDULE".

Every 8086 memory reference is to a location contained within some FRAME, where the FRAME is designated by the content of some segment register. For **ld** to form a correct, usable memory reference, it must know what the TARGET is, and to which FRAME the reference is being made. Thus, every fixup specifies such a FRAME in one of six ways. Some use data X, which is in INDEX-or-FRAME-NUMBER, as above. Others require no data.

The six methods of specifying frames are:

1. (F0) X is a SEGMENT INDEX. The FRAME is the canonic FRAME of the LSEG defined by the INDEX.
2. (F1) X is a GROUP INDEX. The FRAME is the canonic FRAME defined by the group (that is, the canonic FRAME defined by the LSEG in the group that is eventually LOCATED lowest in MAS).
3. (F2) X is an EXTERNAL INDEX. The FRAME is determined when the External Name's public definition is found. There are three cases:
 - (F2a) The symbol is defined relative to some LSEG, and there is no associated GROUP. The LSEGs canonic FRAME is specified.
 - (F2b) The symbol is defined absolutely, without reference to an LSEG, and there is no associated GROUP. The FRAME is specified by the FRAME NUMBER subfield of the PUBDEF record that gives the symbol's definition.
 - (F2c) Regardless of how the symbol is defined, there is an associated GROUP. The canonic FRAME of the GROUP is specified. (The group is specified by the GROUP INDEX subfield of the PUBDEF Record.)
4. (F3) X is a FRAME NUMBER (specifying the obvious FRAME).
5. (F4) No X. The FRAME is the canonic FRAME of the LSEG containing LOCATION.
6. (F5) No X. The FRAME is determined by the TARGET. There are four cases:

Conceptual Framework for Fixups

- (F5a) The TARGET specifies a SEGMENT INDEX: in this case, the FRAME is determined as in (F0).
- (F5b) The TARGET specifies a GROUP INDEX: in this case, the FRAME is determined as in (F1).
- (F5c) The TARGET specifies an EXTERNAL INDEX: in this case, the FRAME is determined as in (F2).
- (F5d) The TARGET is specified with an explicit FRAME NUMBER: in this case the FRAME is determined as in (F3).

Note

The link editor does not support frame methods F2b, F3, or F5d.

Nomenclature describing FRAMEs is similar to the above nomenclature for TARGETs.

FRAME:	SI (<segment name>)	[F0]
FRAME:	GI (<group name>)	[F1]
FRAME:	EI (<symbol name>)	[F2]
FRAME:	LOCATION	[F4]
FRAME:	TARGET	[F5]
FRAME:	NONE	[F6]

For an 8086 memory reference, the FRAME specified by a self-relative reference is usually the canonic FRAME of the LSEG containing the LOCATION, and the FRAME specified by a segment relative reference is the canonic FRAME of the LSEG containing the TARGET.

Self-Relative Fixups

Self-relative fixups can be applied to LOCATIONS which are either 16- or 32-bit OFFSETS or they are LOBYTES. The result of applying a self-relative fixup to any other type of LOCATION is undefined.

Both the LOCATION and the TARGET must lie within the FRAME specified for the fixup.

The value to be used in the fixup is defined as the displacement from the byte in memory following the LOCATION to the TARGET.

If the LOCATION to be fixed-up is a LOBYTE, the fixup value must lie in the range -128 to 127.

If the LOCATION to be fixed up is a 16-bit OFFSET, the fixup value must lie in the range -32768 to 32767.

The fixup value is added to the existing contents of the LOCATION, ignoring any overflow.

Self-relative fixups are typically applied to the relative displacement values used in instructions such as conditional jumps.

Segment-Relative Fixups

Segment-relative fixups can be applied to any type of LOCATION.

The way in which a LOCATION containing a BASE component (that is, a BASE or a POINTER) is fixed up depends on whether the code is to run in real or virtual address mode. The contents of the BASE portion of a LOCATION must ultimately be capable of being loaded into a segment register; therefore, in real address mode this will be a paragraph number and in virtual address mode this will be a selector value.

Fixup values for the BASE and OFFSET components of a LOCATION are calculated as follows:

1. In real address mode:

The base fixup value (FBVAL) is defined as the FRAME NUMBER of the FRAME specified in the fixup.

The offset fixup value (FOVAL) is defined as the offset of the TARGET from the start of the FRAME specified in the fixup. This offset must be ≥ 0 and \leq FFFF.

2. In protected mode:

The base fixup value (FBVAL) is defined as the segment selector of the FRAME specified in the fixup.

The offset fixup value (FOVAL) is defined as the offset of the TARGET from the start of the FRAME specified in the fixup. This offset must be ≥ 0 and \leq the maximum segment size implied by the segment selector for the FRAME; that is, $(2^{**}16)-1$ for 80286 segments and 16-bit 80386 segments, or $(2^{**}32)-1$ for 32-bit 80386 segments.

The fixup values for BASE and OFFSET are applied to the LOCATION as follows:

1. If the LOCATION is a BASE or a POINTER, then FBVAL is stored in the BASE component of the LOCATION.
2. If the LOCATION is a POINTER, or a 16- or 32-bit OFFSET, or a LOBYTE, then the offset fixup value (FOVAL) is added to the existing contents of the OFFSET component of the LOCATION ignoring any overflow.

Segment-Relative Fixups

3. If the LOCATION is a HIBYTE, then FOVAL is divided by 256 and the result is added to the LOCATION, ignoring overflow.

Record Order

Record Order

An object code file must contain a sequence of one or more modules or a library containing zero or more modules. A module is defined as a collection of object code defined by a sequence of object records. The following syntax shows the valid orderings of records to form a module. In addition, the given semantic rules provide information about how to interpret the record sequence.

Note

The syntactic description language used below is defined in WIRTH: CACM, November 1977, vol.#20, no.#11, pp.#822-823. The character strings represented by capital letters above are not literals, but are identifiers that are further defined in the section describing the record formats.

object file	= tmodule
tmodule	= THEADR seg-grp {component} modtail
seg_grp	= {LNAMES} {SEGDEF} {TYPDEF EXTDEF GRPDEF}
component	= data debug_record
data	= content_def thread_def TYPDEF PUBDEF EXTDEF
debug_record	= LINNUM
content_def	= data_record {FIXUPP}
thread_def	= FIXUPP (containing only thread fields)
data_record	= LIDATA LEDATA
modtail	= MODEND

Record Order

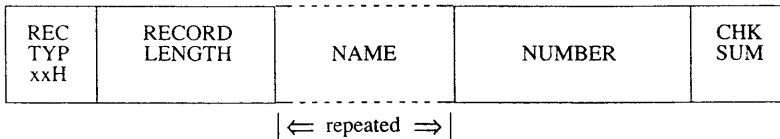
The following rules apply:

- A FIXUPP record always refers to the previous DATA record.
- All LNames, SEGDEF, GRPDEF, TYPDEF, and EXTDEF records must precede all records that refer to them.
- COMENT records may appear anywhere in a file, except as the first or last record in a file or module, or within a content_def.

Introduction to the Record Formats

The following pages present diagrams of record formats in schematic form. Here is a sample record format, to illustrate the various conventions.

SAMPLE RECORD FORMAT (SAMREC)



Title and Official Abbreviation

At the top is the name of the record format described, with an official abbreviation. To promote uniformity among various programs, including translators and debuggers, the abbreviation should be used in both code and documentation. The record format abbreviation is always six letters.

The Boxes

Each format is drawn with boxes of two sizes. The narrow boxes represent single bytes. The wide boxes represent two bytes each. The wide dashed boxes represent a variable number of bytes, one or more, depending upon content. The wide solid boxes represent 4-byte fields.

Rectyp

The first byte in each record contains a value between 0 and 255, indicating the record type. For records that have both 16- and 32-bit versions, the low-order bit of the record type indicates the type: 0=16-bit, 1=32 bit.

Record Length

The second field in each record contains the number of bytes in the record, exclusive of the first two fields.

Name

Any field that indicates a “NAME” has the following internal structure: the first byte contains a number between 0 and 127, inclusive, that indicates the number of remaining bytes in the field. The remaining bytes are interpreted as a byte string.

Most translators constrain the character set to be a subset of the ASCII character set.

Number

A 4-byte NUMBER field represents a 32-bit unsigned integer, where the first 8 bits (least-significant) are stored in the first byte (lowest address), the next 8 bits are stored in the second byte, and so on.

Repeated or Conditional Fields

Some portions of a record format contain a field or a series of fields that may be repeated one or more times. Such portions are indicated by the “repeated” or “rpt” brackets below the boxes.

Similarly, some portions of a record format are present only if some given condition is true; these fields are indicated by similar “conditional” or “cond” brackets below the boxes.

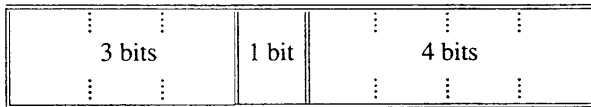
Checksum

The last field in each record is a check sum, which contains the 2’s complement of the sum (modulo 256) of all other bytes in the record. Therefore, the sum (modulo 256) of all bytes in the record equals 0.

Introduction to the Record Formats

Bit Fields

Descriptions of contents of fields will sometimes be at the bit level. Boxes with complete vertical lines drawn through them represent bytes or words; the partial vertical lines indicate bit boundaries; thus the byte represented below, has three bit-fields of 3-, 1-, and 4-bits.



T-MODULE HEADER RECORD
(THEADR)

REC TYP 80H	RECORD LENGTH	T-MODULE NAME	CHK SUM
-------------------	------------------	------------------	------------

Every module output from a translator must have a T-MODULE HEADER RECORD.

T-Module Name

The T-MODULE NAME provides a name for the T-MODULE.

LIST OF NAMES RECORD
(LNAMEs)

REC TYP 96H	RECORD LENGTH	NAME	CHK SUM
-------------------	------------------	------	------------

| ← repeated ⇒ |

This record provides a list of names that may be used in following SEG-DEF and GRPDEF records as the names of Segments, Classes, and/or Groups.

Introduction to the Record Formats

The ordering of L NAMES records within a module, together with the ordering of names within each L NAMES Record, induces an ordering on the names. Thus, these names are considered to be numbered: 1, 2, 3, 4, ... These numbers are used as "Name Indices" in the Segment Name Index, Class Name Index, and Group Name Index fields of the SEGDEF and GRPDEF Records.

Name

NAME is a repeatable field which provides a name and which may have zero length.

SEGMENT DEFINITION RECORD (SEGDEF)

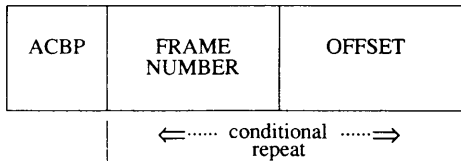
REC TYP 98H 99H	RECORD LENGTH	SEG ATTR	SEGMENT LENGTH	SEGMENT NAME INDEX	CLASS NAME INDEX	OVER- LAY NAME INDEX	CHK SUM
--------------------------	------------------	-------------	-------------------	--------------------------	------------------------	-------------------------------	------------

SEGMENT INDEX values 1 through 32767, which are used in other record types to refer to specific LSEGS, are defined implicitly by the sequence in which SEGDEF Records appear in the object file.

In the REC TYP field, 98H and 99H describe 16- and 32-bit segments, respectively.

Seg Attr

The SEG ATTR field provides information on various attributes of a segment, and has the following format:



Introduction to the Record Formats

The ACBP byte contains four numbers which are the A, C, B, and P attribute specifications. This byte has the following format:



"A" (Alignment) is a 3-bit subfield that specifies the alignment attribute of the LSEG. The semantics are defined as follows:

- A=0 SEGDEF describes an absolute LSEG.
- A=1 SEGDEF describes a relocatable, byte-aligned LSEG.
- A=2 SEGDEF describes a relocatable, word-aligned LSEG.
- A=3 SEGDEF describes a relocatable, paragraph-aligned LSEG.
- A=4 SEGDEF describes a relocatable, page-aligned LSEG.
- A=5 SEGDEF describes a relocatable, double-word-aligned LSEG.
(386 OMF only)

If A=0, the FRAME NUMBER and OFFSET fields will be present. Using **ld**, absolute segments may be used for addressing purposes only. For example, the starting address of a ROM and the symbolic names for addresses within the ROM may be defined in this way. **ld** will ignore any data specified as belonging to an absolute LSEG.

"C" (Combination) is a 3-bit subfield that specifies the combination attribute of the LSEG. Absolute segments (A=0) must have combination zero (C=0). For relocatable segments, the C field encodes a number (0,1,2,4,5,6 or 7) that indicates how the segment can be combined. The interpretation of this attribute is best given by considering how two LSEGS are combined:

- Let X,Y be LSEGS, and let Z be the LSEG resulting from the combination of X,Y.
- Let LX and LY be the lengths of X and Y, and let MXY denote the maximum of LX, LY.
- Let G be the length of any gap required between the X- and Y-components of Z to accommodate the alignment attribute of Y.
- Let LZ denote the length of the (combined) LSEG Z; let dx ($0 \leq dx < LX$) be the offset in X of the (combined) LSEG Z; let dx ($0 \leq dx < LX$) be the offset in X of a byte, and let dy similarly be the offset in Y of a byte.

Introduction to the Record Formats

The following table gives the length LZ of the combined LSEG Z, and the offsets dx' and dy' in Z for the bytes corresponding to dx in X and dy in Y. Intel additionally defines alignment types 5 and 6 and also processes code and data placed in segment with align-type.

Combination Attribute Example

C	LZ	dx'	dy'	
2	LX+LY+G	dx	dy+LX+G	Public
5	LX+LY+G	dx	dy+LX+G	Stack
6	MX Y	dx	dy	Common

The table has no lines for C=0, C=1, C=3, C=4, and C=7. C=0 indicates that the relocatable LSEG may not be combined; C=1 and C=3 are undefined. C=4 and C=7 are treated like C=2. C1, C4, and C7 all have different meanings according to the Intel standard.

“B” (Big) is a 1-bit subfield which, if 1, indicates that the Segment Length is exactly $2^{*}16$ ($2^{*}32$ in the case of 32-bit segments). In this case the SEGMENT LENGTH field must contain zero.

The “P” field must always be zero. The “P” field is the “Page resident” field according to the Intel standard.

The FRAME NUMBER and OFFSET fields (present only for absolute segments, A=0) specify the placement in MAS of the absolute segment. The range of OFFSET is constrained to be between 0 and 15 inclusive. If a value larger than 15 is desired for OFFSET, then an adjustment of the FRAME NUMBER should be done.

Segment Length

The SEGMENT LENGTH field gives the length of the segment in bytes. The length may be zero; if so, **ld** will *not* delete the segment from the module. The SEGMENT LENGTH field is two bytes for a 16-bit segment (Rectyp 98) and four bytes for a 32-bit segment (Rectyp 99). This is large enough for numbers up to $(2^{*}16)-1$ and $(2^{*}32)-1$, respectively. The B attribute bit in the ACBP field (see SEG ATTR section) must be used to indicate a length of $(2^{*}16)$ or $(2^{*}32)$.

Introduction to the Record Formats

Segment Name Index

The Segment Name is a name the programmer or translator assigns to the segment. Examples: CODE, DATA, STACK, TAXDATA, MODULENAME_CODE. This field provides the Segment Name, by indexing into the list of names provided by the L NAMES Record(s).

Class Name Index

The Class Name is a name the programmer or translator can assign to a segment. If none is assigned, the name is null, and has length 0. The purpose of Class Names is to allow the programmer to define a “handle” used in the ordering of the LSEGS in MAS. Examples: RED, WHITE, BLUE; ROM FASTRAM, DISPLAYRAM. This field provides the Class Name, by indexing into the list of names provided by the L NAMES Record(s).

Overlay Name Index

Note

This is ignored in **ld** versions 2.40 and later, but supported in all earlier versions. However, semantics differ from Intel semantics.

The Overlay Name is a name the translator and/or **ld**, at the programmer’s request, applies to a segment. The Overlay Name, like the Class Name, may be null. This field provides the Overlay Name, by indexing into the list of names provided by the L NAMES Record(s).

Note

The “Complete Name” of a segment is a 3-component entity comprising a Segment Name, a Class Name, and an Overlay Name. (The latter two components may be null.)

GROUP DEFINITION RECORD
(GRPDEF)



Group Name Index

The Group Name is a name by which a collection of LSEGs may be referenced. The important property of such a group is that, when the LSEGs are eventually fixed in MAS, there must exist some FRAME which “covers” every LSEG of the group.

The GROUP NAME INDEX field provides the Group Name, by indexing into the list of names provided by the L NAMES Record(s).

Group Component Descriptor

Each GROUP COMPONENT DESCRIPTOR has the following format:



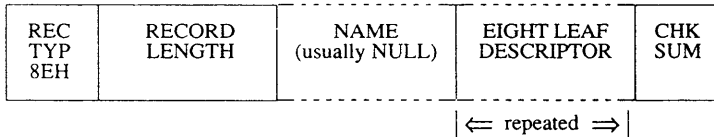
The first byte of the DESCRIPTOR contains 0FFH; the DESCRIPTOR contains one field, which is a SEGMENT INDEX that selects the LSEG described by a preceding SEGDEF record.

Intel defines 4 other group descriptor types, each with its own meaning. They are 0FEH, 0FDH, 0fBH, and 0fAH. The link editor will treat all of these values the same as 0FFH (i.e., it always expects 0FFH followed by a segment index, and it does not check to see if the value is actually 0FF).

Introduction to the Record Formats

TYPE DEFINITION RECORD

(TYPDEF)



The link editor uses TYPDEF records only for communal variable allocation. This is *not* Intel's intended purpose. See "Type Representations for Communal Variables."

As many "EIGHT LEAF DESCRIPTOR" fields as necessary are used to describe a branch. (Every such field except the last in the record describes eight leaves; the last such field describes from one to eight leaves.)

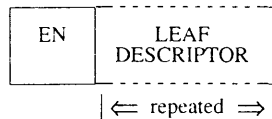
TYPE INDEX values 1 through 32767, which are contained in other record types to associate object types with object names, are defined implicitly by the sequence in which TYPDEF records appear in the object file.

Name

Use of this field is reserved. Translators should place a single byte containing 0 in it (the representation of a name of length zero).

Eight-Leaf Descriptor

This field can describe up to eight Leaves.



The EN field is a byte: the 8 bits, left to right, indicate if the following 8 Leaves (left to right) are Easy (bit=0) or Nice (bit=1).

Introduction to the Record Formats

The LEAF DESCRIPTOR field, which occurs between 1 and 8 times, has one of the following formats:

0 to 127	
129	0 to 64K-1
132	0 to 16M-1
136	-2G-1 to 2G-1

The first format (single byte), containing a value between 0 and 127, represents a Numeric Leaf whose value is the number given.

The second format, with a leading byte containing 129, represents a Numeric Leaf. The number is contained in the following two bytes.

The third format, with a leading byte containing 132, represents a Numeric Leaf. The number is contained in the following three bytes.

The fourth format, with a leading byte containing 136, represents a Signed Numeric Leaf. The number is contained in the following four bytes, sign extended if necessary.

PUBLIC NAMES DEFINITION RECORD (PUBDEF)

REC TYP 90H 91H	RECORD LENGTH	PUBLIC BASE	PUBLIC NAME	PUBLIC OFFSET	TYPE INDEX	CHK SUM
			←..... repeated→			

This record provides a list of one or more PUBLIC NAMES; for each one, three data are provided: (1) a base value for the name, (2) the offset value of the name, and (3) the type of entity represented by the name.

Introduction to the Record Formats

In the RECORD TYPE field, 90H and 91H describe 16- and 32-bit public definition records, respectively.

Public Base

The PUBLIC BASE has the following format:



The GROUP INDEX field has a format given earlier, and provides a number between 0 and 32767 inclusive. A non-zero GROUP INDEX associates a group with the public symbol, and is used as described in the section of this chapter titled “Conceptual Framework for Fixups,” case (F2c). A zero GROUP INDEX indicates that there is no associated group.

The SEGMENT INDEX field has a format given earlier, and provides a number between 0 and 32767, inclusive.

A non-zero SEGMENT INDEX selects an LSEG. In this case, the location of each public symbol defined in the record is taken as a non-negative displacement (given by a PUBLIC OFFSET field) from the first byte of the selected LSEG, and the FRAME NUMBER field must be absent.

A SEGMENT INDEX of 0 (legal only if GROUP INDEX is also 0) means that the location of each public symbol defined in the record is taken as a displacement from the base of the FRAME defined by the value in the FRAME NUMBER field.

The FRAME NUMBER is present if both the SEGMENT INDEX and GROUP INDEX are zero.

A non-zero GROUP INDEX selects some group; this group is taken as the “frame of reference” for references to all public symbols defined in this record; that is, **ld** will perform the following:

1. Any fixup of the form:
 TARGET: EI(P)
 FRAME: TARGET

Introduction to the Record Formats

(where “P” is a public symbol in this PUBDEF record) will be converted by **ld** to a fixup of the form:

```
TARGET: SI(L),d  
FRAME: GI(G)
```

where “SI(L)” and “d” are provided by the SEGMENT INDEX and PUBLIC OFFSET fields. (The “normal” action would have the frame specifier in the new fixup be the same as in the old fixup: FRAME: TARGET.)

2. When the value of a public symbol, as defined by the SEGMENT INDEX, PUBLIC OFFSET, and (optional) FRAME NUMBER fields, is converted to a {base,offset} pair, the base part will be taken as the base of the indicated group. If a non-negative 16-bit offset cannot then complete the definition of the public symbol’s value, an error occurs.

A GROUP INDEX of zero selects no group. **ld** will not alter the FRAME specification of fixups referencing the symbol, and will take, as the base part of the absolute value of the public symbol, the canonic frame of the segment (either LSEG or PSEG) determined by the SEGMENT INDEX field.

Public Name

The PUBLIC NAME field gives the name of the object whose location in MAS is made available to other modules. The name must contain one or more characters.

Public Offset

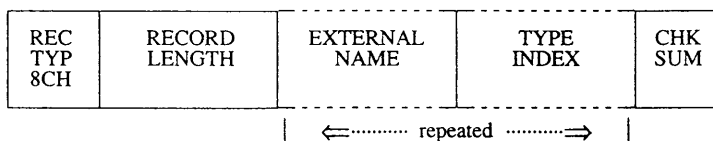
The PUBLIC OFFSET field is a 16-bit value (Rectyp=90H), or a 32-bit value (Rectyp=91H), which is either the offset of the Public Symbol with respect to an LSEG (if SEGMENT INDEX > 0), or the offset of the Public Symbol with respect to the specified FRAME (if SEGMENT INDEX = 0).

Introduction to the Record Formats

Type Index

The TYPE INDEX field identifies a single preceding TYPDEF (Type Definition) Record containing a descriptor for the type of entity represented by the Public Symbol. This field is ignored by **ld**.

EXTERNAL NAMES DEFINITION RECORD (EXTDEF)



This record provides a list of external names, and for each name, the type of object it represents. **ld** will assign to each External Name the value provided by an identical Public Name (if such a name is found).

External Name

This field provides the name, which must have non-zero length, of an external object.

Inclusion of a Name in an External Names Record is an implicit request that the object file be linked to a module containing the same name declared as a Public Symbol. This request obtains whether or not the External Name is referenced within some FIXUPP Record in the module.

The ordering of EXTDEF Records within a module, together with the ordering of External Names within each EXTDEF Record, induces an ordering on the set of all External Names requested by the module. Thus, External Names are considered to be numbered 1, 2, 3, 4, These numbers are used as "External Indices" in the TARGET DATUM and/or FRAME DATUM fields of FIXUPP Records to refer to a particular External Name.

Note

8086 External Names are numbered positively: 1,2,3,... This is a change from 8080 External Names, which were numbered starting from zero: 0,1,2,... This conforms with other 8086 Indices (Segment Index, Type Index, etc.) which use 0 as a default value with special meaning.

External indices may not reference forward. For example, an external definition record defining the kth object must precede any record referring to that object with index k.

Type Index

This field identifies a single preceding TYPDEF (Type Definition) record containing a descriptor for the type of object named by the External Symbol.

The TYPE INDEX is used only in communal variable allocation by the link editor.

LINE NUMBERS RECORD
(LINNUM)

REC TYP 94H 95H	RECORD LENGTH	LINE NUMBER BASE	LINE NUMBER	LINE NUMBER OFFSET	CHK SUM
--------------------------	------------------	------------------------	----------------	--------------------------	------------

| ←..... repeated⇒ |

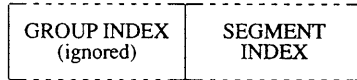
This record provides the means by which a translator may pass the correspondence between a line number in source code and the corresponding translated code.

In the RECORD TYPE field, 94H and 95H describe 16- and 32-bit line number records, respectively.

Introduction to the Record Formats

Line-Number Base

The LINE-NUMBER BASE has the following format:



The SEGMENT INDEX determines the location of the first byte of code corresponding to some source line number.

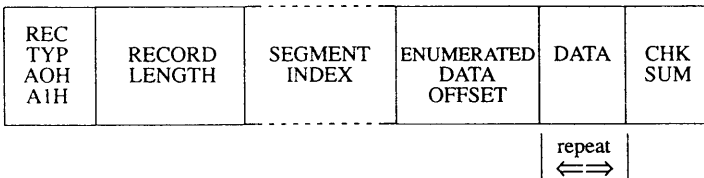
Line-Number

A line number between 0 and 32767, inclusive, is provided in binary by this field. The high-order bit is reserved for future use and must be zero.

Line Number Offset

The LINE-NUMBER OFFSET field is either a 16-bit value (Rectyp=94H) or a 32-bit value (Rectyp=95H) that is the offset of the line number with respect to an LSEG (if SEGMENT INDEX > 0).

LOGICAL ENUMERATED DATA RECORD (LEDATA)



This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

In the RECORD TYPE field, A0H and A1H describe 16- and 32-bit LEDATA records, respectively.

Segment Index

This field must be non-zero and specifies an index relative to the SEGMENT DEFINITION RECORDS found previous to the LEDATA RECORD.

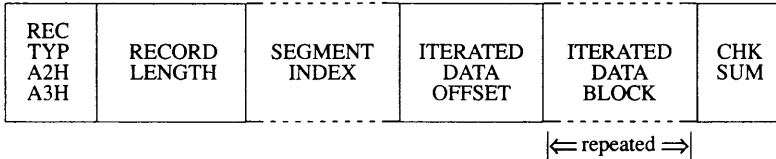
Enumerated Data Offset

This field specifies either a 16-bit offset (Rectyp=A0H) or a 32-bit offset (Rectyp=A1H) that is relative to the base of the LSEG specified by the SEGMENT INDEX and defines the relative location of the first byte of the DAT field. Successive data bytes in the DAT field occupy successively higher locations of memory.

Data

This field provides up to 1024 consecutive bytes of relocatable or absolute data.

LOGICAL ITERATED DATA RECORD
(LIDATA)



This record provides contiguous data from which a portion of an 8086 memory image may be constructed.

In the RECORD TYPE field, A2H and A3H describe 16- and 32-bit LIDATA records, respectively.

Segment Index

This field must be non-zero and specifies an index relative to the SEG-DEF records found previous to the LIDATA RECORD.

Iterated Data Offset

This field specifies either a 16-bit offset (Rectype=A2H) or a 32-bit offset (Rectyp=A3H) that is relative to the base of the LSEG specified by the SEGMENT INDEX and defines the relative location of the first byte in the ITERATED DATA BLOCK. Successive data bytes in the ITERATED DATA BLOCK occupy successively higher locations of memory.

Iterated Data Block

This repeated field is a structure specifying the repeated data bytes. The structure has the following format:



Note

The link editor cannot handle LIDATA records whose ITERATED DATA BLOCK is larger than 512 bytes.

Repeat Count

This field specifies the number of times that the CONTENT portion of this ITERATED DATA BLOCK is to be repeated. REPEAT COUNT must be non-zero.

Block Count

This field specifies the number of ITERATED DATA BLOCKS that are to be found in the CONTENT portion of this ITERATED DATA BLOCK. If this field has value zero, then the CONTENT portion of this ITERATED DATA BLOCK is interpreted as data bytes. If non-zero, then the CONTENT portion is interpreted as that number of ITERATED DATA BLOCKS.

Content

This field may be interpreted in one of two ways, depending on the value of the previous BLOCK COUNT field.

If BLOCK COUNT is zero, then this field is a 1-byte count followed by the indicated number of data bytes.

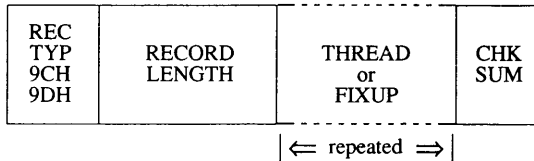
If BLOCK COUNT is non-zero, then this field is interpreted as the first byte of another ITERATED DATA BLOCK.

Note

From the outermost level, the number of nested ITERATED DATA BLOCKS is limited to 17, i.e., the number of levels of recursion is limited to 17.

FIXUP RECORD

(FIXUPP)



This record specifies 0 or more fixups. Each fixup requests a modification (fixup) to a LOCATION within the previous DATA record. A data record may be followed by more than one fixup record that refers back to it. Each fixup is specified by a FIXUP field that specifies four data: a location, a mode, a target, and a frame. The frame and the target may be specified totally within the FIXUP field, or may be specified by reference to a preceding THREAD field.

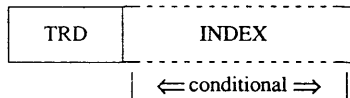
A THREAD field specifies a default target or frame that may subsequently be referred to in identifying a target or a frame. Eight threads are provided: four for frame specification and four for target specification. Once a target or frame has been specified by a THREAD, it may be referred to by following FIXUP fields (in the same or following FIXUPP records), until another THREAD field with the same type (TARGET or FRAME) and Thread Number (0 - 3) appears (in the same or another FIXUPP record).

Introduction to the Record Formats

In the RECORD TYPE field, 9CH and 9DH describe 16- and 32-bit FIX-UPP records, respectively.

Thread

THREAD is a field with the following format:



The TRD DAT (ThReaD DATa) subfield is a byte with this internal structure:



The “Z” is a 1-bit subfield, currently without any defined function, that is required to contain 0.

The “D” subfield is one bit that identifies what type of thread is being specified. If D=0, then a target thread is being defined; if D=1, then a frame thread is being defined.

METHOD is a 3-bit subfield containing a number between 0 and 3 (D=0) or a number between 0 and 6 (D=1).

If D=0, then $METHOD = (0, 1, 2, 3, 4, 5, 6, 7) \bmod 4$, where the 0, ..., 7 indicate methods T0, ..., T7 of specifying a target. Thus, METHOD indicates what kind of INDEX or FRAME NUMBER is required to specify the target, without indicating if the target will be specified in a primary or secondary way. Note that methods 2b, 3, and 7 are not supported by **ld**.

If D=1, then $METHOD = 0, 1, 2, 4, 5$, corresponding to methods F0, ..., of specifying a frame. Here, METHOD indicates what kind (if any) of Index is required to specify the frame. Note that methods 3 and 5d are not supported by **ld**.

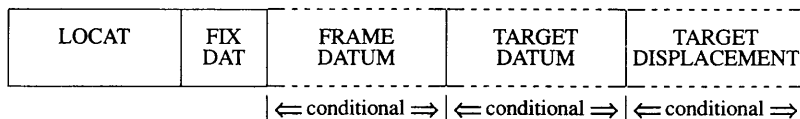
THRED is a number between 0 and 3, and associates a Thread Number to the frame or target defined by the THREAD field.

Introduction to the Record Formats

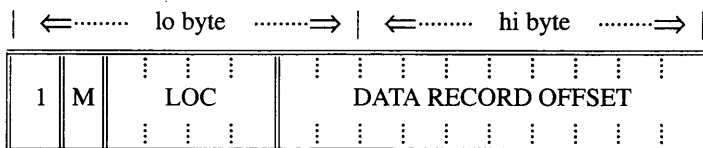
INDEX contains a Segment Index, Group Index, or External Index depending on the specification in the METHOD subfield of the TRD field. This subfield will not be present if F4 or F5 are specified by METHOD.

Fixup

FIXUP is a field with the following format:



LOCAT is a byte pair with the following format:



M is a 1-bit subfield that specifies the mode of the fixups: self-relative (M=0) or segment-relative (M=1).

Note

Self-Relative fixups may not be applied to LIDATA records.

LOC is a four-bit subfield indicating the type of location that is to be fixed up:

Introduction to the Record Formats

0	-	8 bit lobyte
1	-	16 bit offset
2	-	16 bit base
3	-	32 bit pointer
4	-	8 bit hibyte
5	-	16 bit offset (linker resolved)
9	-	32 bit offset
11	-	48 bit pointer
13	-	32 bit offset (linker resolved)

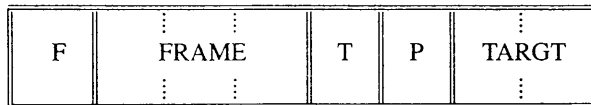
LOC values 9, 11, and 13 are only valid in 32-bit FIXUPP records (record type 9D). All values not mentioned are invalid.

The DATA RECORD OFFSET is a number between 0 and 1023, inclusive, that gives the relative position of the lowest order byte of LOCATION (the actual bytes being fixed up) within the preceding DATA record. The DATA RECORD OFFSET is relative to the first byte in the data fields in the DATA RECORDs.

Note

It is possible for the value of DATA RECORD OFFSET to designate a "location" within a REPEAT COUNT subfield or a BLOCK COUNT subfield of the ITERATED DATA field. Such a reference is an error. The action of **ld** on such a malformed record is undefined.

FIX DAT is a byte with the following format:



Note

Frame method 2b, F3, and F5d are not supported. Target method T3 and T7 are not supported.

Introduction to the Record Formats

F is a 1-bit subfield that specifies whether the frame for this FIXUP is specified by a thread (F=1) or explicitly (F=0).

FRAME is a number interpreted in one of two ways as indicated by the F bit. If F is zero, FRAME is a number between 0 and 5 and corresponds to methods F0, ..., F5 of specifying a FRAME. If F=1, then FRAME is a thread number (0-3). It specifies the frame most recently defined by a THREAD field that defined a frame thread with the same thread number. (Note that the THREAD field may appear in the same, or in an earlier FIXUPP record.)

“T” is a 1-bit subfield that specifies whether the target specified for this fixup is defined by reference to a thread (T=1), or is given explicitly in the FIXUP field (T=0).

“P” is a 1-bit subfield that indicates whether the target is specified in a primary way (requires a TARGET DISPLACEMENT, P=0) or specified in a secondary way (requires no TARGET DISPLACEMENT, P=1). Since a target thread does not have a primary/secondary attribute, the P bit is the only field that specifies the primary/secondary attribute of the target specification.

TARGET is interpreted as a 2-bit subfield. When T=0, it provides a number between 0 and 3, corresponding to methods T0, ..., T3 or T4, ..., T7, depending on the value of P (P can be interpreted as the high-order bit of T0, ..., T7). When the target is specified by a thread (T=1), then TARGET specifies a thread number (0-3).

FRAME DATUM is the “referent” portion of a frame specification, and is a Segment Index, a Group Index, or an External Index. The FRAME DATUM subfield is present only when the frame is specified neither by a thread (F=0) nor explicitly by methods F4 or F5 or F6.

TARGET DATUM is the “referent” portion of a target specification, and is a Segment Index, a Group Index, an External Index, or a Frame Number. The TARGET DATUM subfield is present only when the target is not specified by a thread (T=0).

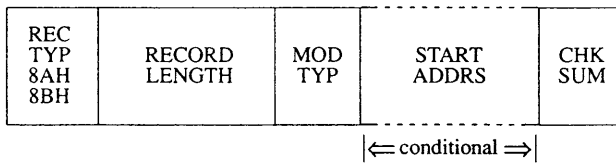
TARGET DISPLACEMENT is the displacement required by “primary” methods of specifying TARGETS. This field is 2 bytes long in 16-bit FIXUPP records (Rectyp=9CH) and 4 bytes long in 32-bit FIXUPP records (Rectyp=9DH). This subfield is present if P=0.

Introduction to the Record Formats

Note

All these methods are described in the section of this chapter titled "Conceptual Framework for Fixups."

MODULE END RECORD (MODEND)

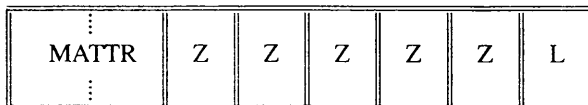


This record serves two purposes. It denotes the end of a module and indicates whether the module just terminated has a specified entry point for initiation of execution. If the latter is true, the execution address is specified.

In the RECORD TYPE field, 8AH and 8BH describe 16- and 32-bit MODEND records, respectively.

Mod Type

This field specifies the attributes of the module. The bit allocation and associated meanings are as follows:



Introduction to the Record Formats

MATTR is a 2-bit subfield that specifies the following module attributes:

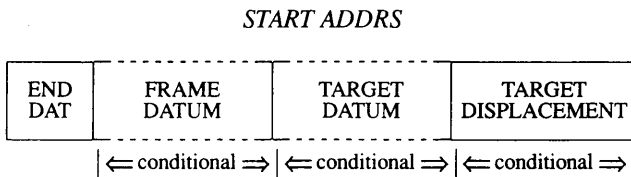
MATTR	MODULE ATTRIBUTE
0	Non-main module with no START ADDRS
1	Non-main module with START ADDRS
2	Main module with no START ADDRS
3	Main module with START ADDRS

“L” indicates whether the START ADDRS field is interpreted as a logical address that requires fixing up by *ld* (L=1). Note that with *ld*, L must always equal 1.

“Z” indicates that this bit has not currently been assigned a function. These bits are required to be zero.

Physical start addresses (L=0) are not supported.

The START ADDRS field (present only if MATTR is 1 or 3) has the following format:



The starting address of a module has all the attributes of any other logical reference found in a module. The mapping of a logical starting address to a physical starting address is done in exactly the same manner as mapping any other logical address to a physical address as specified in the discussion of fixups and the FIXUPP record. The above subfields of the START ADDRS field have the same semantics as the FIX DAT, FRAME DATUM, TARGET DATUM, and TARGET DISPLACEMENT fields in the FIXUPP record. Only “primary” fixups are allowed. Frame method F4 is not allowed.

The TARGET DISPLACEMENT field is 2 bytes in a 16-bit MODEND record (Rectyp=8AH) and 4 bytes in a 32-bit MODEND record (Rectyp=8BH).

Introduction to the Record Formats

COMMENT RECORD (COMENT)

REC TYP 88H	RECORD LENGTH	COMMENT TYPE	COMMENT	CHK SUM
-------------------	------------------	-----------------	---------	------------

This record allows translators to include comments in object text.

Comment Type

This field indicates the type of comment carried by this record. This allows comments to be structured for those processes that wish to selectively act on comments. The format of this field is as follows:

NP	NL	Z	Z	Z	Z	Z	Z	COMMENT CLASS
----	----	---	---	---	---	---	---	------------------

The NP (NOPURGE) bit, if 1, indicates that it is not able to be purged by object file utility programs which implement the capability of deleting COMMENT record.

The NL (NOLIST) bit, if 1, indicates that the text in the COMMENT field is not to be listed in the listing file of object file utility programs which implement the capability of listing object COMMENT records. In the above diagram, "Z" indicates no value and must equal zero.

The COMMENT CLASS field is defined as follows:

- 0 Language translator comment.
- 1 Intel copyright comment. The NP bit must be set.
- 2-155 Reserved for Intel use. (See Note 1 below.)
- 156-255 Reserved for users. Intel products will apply no semantics to these values. (See Note 2 below.)

Introduction to the Record Formats

NOTES:

1. Class value 159 is used to specify a library to add to the link editor's library search list. The comment field will contain the name of the library. Note that unlike all other name specifications, the library name is not prefixed with its length. Its length is determined by the record length.
2. Class value 156 is used to specify a DOS level number. When the class value is 156, the comment field will contain a two-byte integer specifying a DOS level number.
3. Class value 161 is used to indicate that the module contains UNIX System V extensions to OMF, such as the various 32-bit record types.

Comment

This field provides the commentary information.

Numeric List of Record Types

Numeric List of Record Types

*6E	RHEADR	*92	LOCSYM
*70	REGINT	*93	MLOC386
*72	REDATA	94	LINNUM
*74	RIDATA	95	MLIN386
*76	OVLDEF	96	LNAMES
*78	ENDREC	98	SEGDEF
*7A	BLKDEF	99	MSEG386
*7C	BLKEND	9A	GRPDEF
*7E	DEBSYM	9C	FIXUPP
80	THEADR	9D	MFIX386
*82	LHEADR	*9E	(none)
*84	PEDATA	A0	LEDATA
*86	PIDATA	A1	MLED386
88	COMENT	A2	LIDATA
8A	MODEND	A3	MLID386
8B	H386END	*A4	LIBHED
8C	EXTDEF	*A6	LIBNAM
8E	TYPDEF	*A8	LIBLOC
90	PUBDEF	*AA	LIBDIC
91	MPUB386		

Note

The record types marked with an asterisk are not supported by the link editor. They will be ignored if they are found in an object module.

Type Representations for Communal Variables

This section defines the UNIX System V standard for communal variable allocation on the 8086 and 80286.

A communal variable is an uninitialized public variable whose final size and location are not fixed at compile time. Communal variables are similar to FORTRAN common blocks in that if a communal variable is declared in more than one object module being linked together, then its actual size will be the largest size specified in the several declarations. In the C language, all uninitialized public variables are communal. The following example shows three different declarations of the same C communal variable:

```
char foo[4];           /* In file a.c */
char foo[1];          /* In file b.c */
char foo[1024];       /* In file c.c */
```

If the objects produced from a.c, b.c, and c.c are linked together, then the linker will allocate 1024 bytes for the char array "foo."

A communal variable is defined in the object text by an external definition record (EXTDEF) and the type definition record (TYPDEF) to which it refers.

The TYPDEF for a communal variable has the following format:

REC TYP 8EH	RECORD LENGTH	0	EIGHT LEAF DESCRIPTOR	CHK SUM
-------------------	------------------	---	--------------------------	------------

The EIGHT LEAF DESCRIPTOR field has the following format:

EN	LEAF DESCRIPTOR
----	--------------------

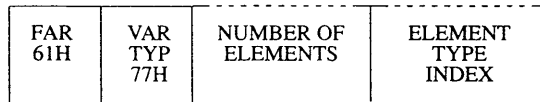
Type Representations for Communal Variables

The EN field specifies whether the next 8 leaves in the LEAF DESCRIPTOR field are EASY (bit = 0) or NICE (bit = 1). This byte is always zero for TYPDEFS for communal variables.

The LEAF DESCRIPTOR field has one of the following two formats. The format for communal variables in the default data segment (near variables) is as follows:



The VAR TYP field may be either SCALAR (7BH), STRUCT (79H), or ARRAY (77H). The VAR SUBTYP field (if any) is ignored by ld. The format for communal variables not in the default data segment (far variables) is as follows:



The VARIABLE TYPE field must be ARRAY (77H). The length field specifies the NUMBER OF ELEMENTS, and the ELEMENT TYPE INDEX is an index to a previously defined TYPDEF whose format is that of a near communal variable.

The format for the LENGTH IN BITS or NUMBER OF ELEMENTS fields is the same as the format for the LEAF DESCRIPTOR field, described in the TYPDEF record format section of this chapter.

Link Time Semantics

All EXTDEFs referencing a TYPDEF of the previously described formats are treated as communal variables. All others are treated as externally defined symbols for which a matching public symbol definition (PUBDEF) is expected. A PUBDEF matching a communal variable definition will override the communal variable definition. Two communal variable definitions are said to match if the names given in the definitions match. If two matching definitions disagree as to whether a communal variable is near or far, the linker will assume the variable is near.

Type Representations for Communal Variables

If the variable is near, then its size is the largest specified for it. If the variable is far, then the link editor issues a warning if there are conflicting array element size specifications; if there are no such conflicts, then the variable's size is the element size times the largest number of elements specified. The sum of the sizes of all near variables must not exceed 64K bytes. The sum of the sizes of all far variables must not exceed the size of the machine's addressable memory space.

“Huge” Communal Variables

A far communal variable whose size is larger than 64K bytes will reside in segments that are contiguous (8086) or have consecutive selectors (80286). No other data items will reside in the segments occupied by a huge communal variable.

If the linker finds matching huge and near communal variable definitions, it issues a warning message, since it is impossible for a near variable to be larger than 64K bytes.

The Segmented x.out Format

This section describes the executable object file format used in XENIX System V and in UNIX System V when the `-xenix` option is used. The format used is an extension to the existing *x.out* format, specifically enhanced for the segmented architecture of the 286 CPU. Note that *x.out* is a name for the format of the file, the actual executable file will be called *a.out* by default.

The UNIX System V linker (*/bin/ld*, see the “Linking with the `cc` Command” chapter) will link the Intel 86 Relocatable Object Module Format into the executable format described in this section.

The UNIX System V product supports a subset of segmented OMF. Other parts are specified here for use by other vendors, and to reserve their meaning for possible future use. Those parts supported in this release of UNIX System V are:

- The *x.out* header
- The *x.out* extended header
- The file segment table
- Multiple non-iterated text segments
- Multiple non-iterated data segments
- Symbol table segments in the format described herein

Note specifically that the machine-dependent table is *not* supported. The iterated text/data feature is supported by the kernel, but the UNIX System V linker will expand iterated records.

General Description of x.out

The following is a general description of the *x.out* object file format, extended to handle segmentation. It implements iterated text and data segments, huge, large, middle, and small model, as well as block alignment to improve the efficiency of loading executable files.

Note

The default file name produced by the linker is *a.out* regardless of the actual file format used. Any mention of *x.out* in this guide is referring only to the *format* of OMF executable files.

The extensions to the existing format consist of adding a file segment table that describes and points to various (possibly block-aligned) file segments. A file segment may contain a memory image, may indicate how to construct a memory image (iterated text or data), or may contain symbols or other non-executable information. In addition to the file segment table, there is an optional machine-dependent table.

The header must be first item in the object file, and the extended header must immediately follow the header. The extended header indicates the segment and (optional) machine-dependent tables' sizes and positions. Although the segment table is not block aligned, individual entries will line up on a multiple of 32 bytes (the size of a segment table entry). The segment table indicates the sizes and positions of the remaining file segments. The file segments may be aligned on a boundary that is a multiple of 512 bytes, with that multiple stored in the extended header, or at location zero if the file segments are not block aligned.

The segment table is an array of records describing the file segments, each containing:

- A segment type: text, data, symbols, etc.
- Segment attributes, specific to the type of segment.
- A file pointer to the (possibly iterated) text/data for this segment.
- A physical size, the size of the segment in the file.
- A virtual size, the size the segment will occupy in memory.
- A location counter, this segment's current base address, usually 0.

A sample of a segment table entry is shown below. The *xs* fields in this data structure are referred to throughout the remaining discussion in this section.

The Segmented x.out Format

```


Segment table entry


struct xseg {
    unsigned short xs_type;           /* x.out segment table entry */
    unsigned short xs_attr;          /* segment attributes */
    unsigned short xs_seg;           /* segment number */
    unsigned short xs_sres;          /* unused */
    long xs_filpos;                  /* file position */
    long xs_psize;                   /* physical size (in file) */
    long xs_vsize;                   /* virtual size (in core) */
    long xs_rbase;                   /* relocation base address */
    long xs_lres;                    /* unused */
    long xs_lres2;                   /* unused */
};
```

The segment table is a contiguous array of the above structures. Each file segment has a corresponding segment table entry that describes the segment's position *xs_filpos* and physical size *xs_psize* in the file. If there is no associated file segment, both fields must be set to zero.

The kernel's local descriptor table (LDT) can be built from the virtual size, the segment type, and segment attribute fields.

Example of File Layout

This section provides an example of the layout of an *x.out* file where:

- The segment table has two entries (segments).
- The file page size is 512 bytes.
- Both file segments are smaller than 512 bytes.
- The second file segment contains iterated data.

Accessing the machine-dependent table and the file segment table must always be done through the absolute file pointers in the extended header. The ordering of the two tables and file segments shown above is not required to be consistent with the *x.out* UNIX System V specification.

Iterated Segments

The data structure for an iterated segment is shown below:

The Segmented x.out Format

```
struct xiter {
    long    xi_size;      /* byte count */
    long    xi_rep; /* replication count */
    long    xi_offset;   /* destination offset in segment */
};
```

If the segment contains iterated text/data (indicated by a bit in the *xs_attr* field), the *xs_filpos* field is the file position of some number of iteration records mixed with the text/data to be iterated. If any part of a segment is iterated, then all of that segment is represented as iterated; non-iterated portions may be represented by an iteration record with a replication count of one.

The format of the text/data to be iterated is:

<iteration record> <text/data> <iteration record> <text/data> ...

where each <iteration record> is of the above “struct xiter” data structure. Each iteration record is followed by *xi_size* bytes of text/data that are to be placed in the current segment at the specified offset *xi_offset* *xi_rep* times. When *xs_psize* bytes of iteration records and text/data have been expanded, the iteration is complete.

Under UNIX System V, areas of memory that are initialized by more than one iteration record will have the contents of those memory areas undefined. Areas of memory that are not initialized by any iteration records will be zeroed out. An iteration byte count *xi_size* of zero will not result in any iteration. Portions of a segment that are to be **bss** sections should use an iteration record with a non-zero byte count and replicate one or more zeroed data bytes. For more information on **bss** sections, see the chapter on the Common Object File Format in this book.

This representation of iterated text/data will handle iterations that contain very large replication counts and/or very large non-iterated sizes.

Non-Iterated Segments and Implicit **bss**

If the iteration bit in *xs_attr* is **not** set, no iterations are required to initialize the segment. If the implicit **bss** bit in the *xs_attr* field is set and the virtual size is greater than the physical size, then the rest of the segment (up to *xs_vsize* bytes) is filled with zeros by the kernel loader. This implicit **bss** definition means that small and middle model executables’ single data segments may still contain unexpanded **bss** without the use of explicit iteration records.

The Segmented *x.out* Format

Segments made up entirely of implicit “C” bss need only set the physical size to zero, and set the implicit bss bit. This eliminates the need for any file segment containing data or iteration records. If there are no iterations and no implicit bss, the virtual size of the segment *xs_vsize* must be the same as the physical size *xs_psize*, and a single copy of the text/data located at *xs_filpos* is all that is required to initialize the segment.

Large Model

With *x.out* format, large model is supported by allowing multiple logical text and/or data segments. Middle and small models are simpler cases, with perhaps single logical segments for data (or both text and data). Iterated segments are independent of memory model.

Special Header Fields

The model bits in the *x_renv* field of the main header, XE_LDATA and XE_LTEXT, usually indicate the default size of data and text pointers used in the executable code. The kernel depends on these two bits to indicate the size of data and text pointers passed in system calls. However, since multiple segments are allowed in small and middle model, there can be little other meaning attached to these bits. Passing near data and/or text pointers implies use of the first data and text segments, respectively.

Also in the *x_renv* field, the absolute bit, XE_ABS, identifies a standalone executable file. When this bit is set, the extended header stack size field is used as the default physical load address. The UNIX System V kernel loader will not load a binary if the XE_ABS bit is set. The UNIX System V boot loader will not load a binary unless the XE_ABS bit is set. See the manual page on **ld(CP)** for information about how to set the XE_ABS bit and the physical load address.

Symbol Table

The data structure for the *x.out* symbol table is shown below:

The Segmented x.out Format

```
struct sym {                                /* x.out symbol table entry */
    unsigned short s_type;
    unsigned short s_seg;
    long           s_value;
};
```

The symbol table differs from the previous *x.out* format only in that the *s_seg* field now holds the selector of the segment that defines the symbol. If the symbol is absolute, the value field holds the symbol's value; otherwise, it holds the offset in the indicated segment to which the symbol refers.

The symbol name trails the above "struct sym" data structure in the form of a null terminated string. The type field values are defined in */usr/include/sys/reldsym.h*.

The use of the *xs_seg* field in the segment table is undefined for symbol table segments. Its use may be defined by the particular symbol table format used.

UNIX System V Executable Format

UNIX System V does not execute binaries that make use of selectors below 0x3f or selectors that do not have the low 3 bits set (LDT, ring 3). UNIX System V also requires that the first data selector be after the last text selector. Binaries are allowed to have zero length segments or "holes" (unused selectors) in text or data, but holes in text may not contain data selectors, and holes in data may not contain text selectors.

The fields, *xext.xe_eseg:xexec.x_entry*, must contain the initial **cs:ip** addresses of the user process. **cs:ip** are the addresses of the starting segments of the program to be run.

Small-model impure binaries (text and data combined into a single segment) must have a single file segment, of type data, with a selector of at least 0x47. It must contain all text, followed by all data, followed by bss. The sizes of each must be stored in the *x_text*, *x_data*, and *x_bss* fields of the main header. UNIX System V uses the value stored in the *xext.xe_eseg* field as the text selector, which must be at least 0x3f and less than the data selector. All text/data/bss binaries are executable through the text selector, and all text/data/bss binaries are readable and writable through the data selector. UNIX System V maps the text selector to the same memory as the data selector.

The Segmented x.out Format

In addition to the above, the linker, **ld**, generates binaries that conform to the following:

- Text selectors start at 0x3f.
- Data selectors start at the first free selector past text.
- All text selectors are contiguous.
- All data selectors are contiguous.
- Small-model impure binaries conform to the above specification, with 0x47 as the data selector. In the symbol table, the selector 0x47 is associated with data symbols, and the selector 0x3f is associated with text symbols, to allow **adb** and **nm** to present consistent data to the user.

Selected Portions of Include Files

The following are selected portions of the *usr/include/sys/x.out.h* and *usr/include/sys/relsym.h* include files:

```
struct xexec {
    /* x.out header */
    unsigned short x_magic;
        /* magic number */
    unsigned short x_ext;
        /* size of header extension */
    long          x_text;
        /* size of text segment */
    long          x_data;
        /* size of initialized data */
    long          x_bss;
        /* size of uninitialized data */
    long          x_syms;
        /* size of symbol table */
    long          x_reloc;
        /* relocation table length */
    long          x_entry;
        /* entry offset, see xe_eseg */
    char          x_cpu;
        /* cpu type & byte/word order */
    char          x_relsym;
        /* relocation & symbol format */
    unsigned short x_renv;
        /* run-time environment */
};
```

The Segmented x.out Format

```
struct xext {
    /* x.out header extension */
    long    xe_trsize;
           /* size of text relocation */
    long    xe_drsize;
           /* size of data relocation */
    long    xe_dbase;
           /* data relocation base */
    long    xe_stksize;
           /* stack size (if XE_FS set) */
    long    xe_segpos;
           /* segment table position */
    long    xe_segsize;
           /* segment table size */
    long    xe_mdtpos;
           /* machine dependent table position */
    long    xe_mdtsize;
           /* machine dependent table size */
    char    xe_mdtype;
           /* machine dependent table type */
    char    xe_pagesize;
           /* file pagesize, in multiples of 512 */
    char    xe_ostype;
           /* operating system type */
    char    xe_osvers;
           /* operating system version */
    unsigned short xe_eseg;
           /* entry segment (hardware dependent) */
    unsigned short xe_sres;
           /* reserved */
};

/*
 *   Definitions for xexec.x_renv (short).
 *
 *   vv    version compiled for
 *   xx    extra (zero)
 *   s     set if segmented x.out
 *   a     set if absolute (set up for physical address)
 *   i     set if segment table contains iterated text/data
 *   h     set if huge model data
 *   f     set if floating point hardware required
 *   t     set if large model text
 *   d     set if large model data
 *   o     set if text overlay
 *   f     set if fixed stack
 *   p     set if text pure
 *   s     set if separate I & D
 *   e     set if executable
 */
```

The Segmented x.out Format

```
#define XE_V2      0x4000
/* up to and including 2.3 */
#define XE_V3      0x8000
/* after version 2.3 */
#define XE_VERS    0xc000
/* version mask */

#define XE_SEG0x0800
/* segment table present */
#define XE_ABS0x0400
/* absolute memory image (standalone) */
#define XE_ITER    0x0200
/* iterated text/data present */
#define XE_HDATA   0x0100
/* huge model data */
#define XE_FPH0x0080
/* floating point hardware required */
#define XE_LTEXT   0x0040
/* large model text */
#define XE_LDATA   0x0020
/* large model data */
#define XE_OVER    0x0010
/* text overlay */
#define XE_FS      0x0008
/* fixed stack */
#define XE_PURE    0x0004
/* pure text */
#define XE_SEP0x0002
/* separate I & D */
#define XE_EXEC    0x0001
/* executable */

struct xseg {
/* x.out segment table entry */
  unsigned short xs_type;
/* segment type */
  unsigned short xs_attr;
/* segment attributes */
  unsigned short xs_seg;
/* segment number */
  unsigned short xs_sres;
/* unused */
  long xs_filpos;
/* file position */
  long xs_psize;
/* physical size (in file) */
  long xs_vsize;
/* virtual size (in core) */
  long xs_rbase;
/* relocation base address */
  long xs_lres;
/* unused */
  long xs_lres2;
/* unused */
};
```

The Segmented x.out Format

```
struct xiter {
    /* x.out iteration record */
    long    xi_size;
           /* byte count */
    long    xi_rep;
           /* # of repetitions */
    long    xi_offset;
           /* destination offset in segment */
};

struct sym {
    /* x.out symbol table entry */
    unsigned short s_type;
    unsigned short s_seg;
    long          s_value;
};

/*
 * Definitions for xe_mdttype
 */
#define XE_MDTNONE    0
           /* no machine dependent table */
#define XE_MDT286    1
           /* iAPX286 LDT */

/*
 * Definitions for xe_ostype
 */
#define XE_OSNONE    0
#define XE_OSUNIX System V    1
           /* UNIX System V */
#define XE_OSRMX    2
           /* iRMX */

/*
 * Definitions for xe_osvers
 */
#define XE_OSXV3    1
           /* UNIX System V */

/*
 * Definitions for xs_type:
 * Values from 64 to 127 are reserved.
 */
#define XS_TNULL    0    /* unused segment */
#define XS_TTEXT    1    /* text segment */
#define XS_TDATA    2    /* data segment */
#define XS_TSYMS    3    /* symbol table segment */
#define XS_TREL4    4    /* relocation segment */
```


The Segmented x.out Format

```
/*
 *   Definitions for xs_attr:
 *   The top bit is set if the file segment represents
 *   a memory image. The other 15 bits' definitions
 *   depend on the type of file segment.
 */
#define XS_AMEM      0x8000
/* segment represents a memory image */
#define XS_AMASK     0x7fff
/* type specific field mask */

/*
 *   Definitions for xs_attr, built by or'ing the following
 *   bit patterns: these values are valid for XS_TTEXT and
 *   XS_TDATA file segments only.
 */
#define XS_AITER     0x0001
/* contains iteration records */
#define XS_AHUGE     0x0002
/* contains huge element */
#define XS_ABSS      0x0004
/* contains implicit bss */
#define XS_APURE     0x0008
/* is read-only, may be shared */
#define XS_AEDOWN    0x0010
/* segment expands downward */

/*
 *   Definitions for xs_attr.
 *   These values are valid for XS_TSYMS file segments only.
 */
#define XS_SXSEG     0x0001
/* x.out segmented format */
```

When using the *xs_seg* field, note that if the *XS_AMEM* bit is set in the *xs_attr* field, the file segment represents a memory image, and the value placed in this field should be the segment number as used by the hardware to reference the segment. This is the actual value placed in the segment register. For the 286, it is simply an LDT selector (under UNIX System V, if the privilege level is not 3, the file will not be executed). Otherwise the segment is not a memory image, and the contents of the field are not defined. File segments other than memory images may define and use this field as needed.

There are two bits in the *xexec_cpu* field that are used to indicate the CURRENT byte and word ordering of the non-character data fields of the header, extended header, segment table, and symbol table. These bits, *XC_BSWAP* and *XC_WSWAP*, do not indicate the byte and word ordering of the target CPU, *XC_CPU*.

The segment table is not block aligned. No individual segment table entry may straddle a block boundary.

Chapter 8

C Language Compatibility with Assembly Language

Introduction 8-1

C Calling Sequence for 8086/80286 8-2

Entering an 8086/80286 Assembly Routine 8-3

8086/80286 Return Values 8-4

Exiting an 8086/80286 Routine 8-5

8086/80286 Program Example 8-6

80386 C-Language Calling Sequence 8-7

Entering an 80386 Assembly-Language Routine 8-8

80386 Return Values 8-9

Exiting an 80386 Routine 8-11

80386 Program Example 8-12



Introduction

This chapter explains how to use 8086/286/386 assembly-language routines with C-language programs and functions. In particular, it explains how to call assembly-language routines from C-language programs and how to call C-language functions from an assembly-language routine.

This assembly-language interface is especially useful for those assembly-language programmers who wish to use the functions of the standard C library and other C libraries.

Note

Two different calling conventions are available. The 8086/80286 calling convention is established by configuring C-language programs with the **-M0**, **-M1**, or **-M2** option. The 80386 calling convention is established by configuring C-language programs with the **-M3** option.

C Calling Sequence for 8086/80286

To receive values from C-language function calls or to pass values to C functions, assembly-language routines must follow the C argument passing conventions. 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 expression's value before pushing it onto the stack.

Arguments with **char**, **int**, or **unsigned** type occupy a single word (16 bits) on the stack. Arguments with **long** type occupy a double word (32 bits) with the value's high-order word occupying the first word pushed onto the stack. Arguments with **float** type are converted to **double** type (64 bits). Note that **char** type arguments are zero-extended to **int** type before being pushed on the stack.

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.

After a function returns control to a routine, the calling routine is responsible for removing arguments from the stack.

Entering an 8086/80286 Assembly Routine

Assembly-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. The following example illustrates the recommended instruction sequence for entry to an assembly-language routine:

```
entry:
    push    bp
    mov     bp, sp
    push    di
    push    si
```

This is the same sequence used by the C compiler.

If this sequence is used, the last argument passed by the function call (which is also the first argument given in the call's argument list) is at address “[bp+4]”. Subsequent arguments begin at address “[bp+6]” or “[bp+8]” depending on the size of the first argument.

This sequence is strongly recommended even if the **SI** and **DI** registers are not modified, since it allows backtracing with the **adb** program during program debugging.

8086/80286 Return Values

Assembly-language routines that wish to return values to a C-language program or receive return values from C functions must follow the C return value conventions. C functions place return values that have **int**, **char**, or **unsigned** type in the **AX** register. They place values with **long** type in the **AX** and **DX** registers, with the high order word in **DX**.

To return a structure or a floating point value, C functions place the address of the given value in the **AX** register. The structure or floating point value must be in a static area in memory. Long addresses are returned in the **AX** and **DX** registers with the segment selector in **DX**.

Exiting an 8086/80286 Routine

Assembly-language routines that return control to C programs should restore the values of the **BP**, **SI**, and **DI** registers before returning control. The following example illustrates the recommended instruction sequence for exiting a routine:

```
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. It also does not remove arguments from the stack. This is the responsibility of the calling routine.

8086/80286 Program Example

To illustrate the assembly-language interface, consider the following example of a C function:

```
add(i, j)
int i, j;
{
    return(i+j);
}
```

If written as an assembly-language routine, this function must save the proper registers, retrieve the arguments from the stack, add the arguments, place the return value in the **AX** register, then restore registers and return control. The following is an example of how the routine can be written:

```
_add:
    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
```

If, on the other hand, the C function is to be called by an assembly-language routine, the routine must contain instructions that push the arguments on the stack in the proper order, call the function, and clear the stack. It may then use the return value in the **AX** register. The following is an example of the instructions that can do this:

```
push <j value>
push <i value>
call _add
add  sp, *4
```

Note that the C compiler does not preserve **ES** over calls. Assembly-language routines need not preserve **ES** and should not assume that it will be preserved if they make calls to routines written in C.

80386 C-Language Calling Sequence

To receive values from 80386 C-language function calls, or to pass values to 80386 C-language functions, assembly-language routines must follow the 80386 C-language argument-passing conventions.

C-language function calls pass arguments to the function by pushing each argument onto the stack. The call pushes the last function argument first and the first function argument last onto the stack. If an argument is an expression, the call computes the expression's value before pushing it onto the stack.

Arguments with **char**, **int**, **unsigned**, **short**, or **long** type occupy a doubleword (32 bits or 4 bytes) on the stack. Arguments with **float** type are converted to **double** type (64 bits or 8 bytes). Note that **char**, **unsigned char**, **short**, and **unsigned short** type arguments are sign extended or zero extended, respectively, to **int** type before being pushed onto the stack.

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 of the structure is pushed onto the stack.

After a function returns control to the calling routine, the calling routine is responsible for removing all function arguments from the stack.

Entering an 80386 Assembly-Language Routine

Assembly-language routines that receive control from 80386 C function calls should preserve the contents of the **EBP**, **ESI**, **EDI**, and **EBX** registers. In addition, the routines should set the **EBP** register to the current **ESP** register value before proceeding with their tasks. The following example illustrates a recommended instruction sequence for entry to an assembly-language routine:

```
entry:
    push    ebp
    mov     ebp, esp
    push    edi
    push    esi
    push    ebx
```

Note that this is the same routine that the compiler uses after pushing the function arguments onto the stack.

If this sequence is used, the last function argument pushed by the function call (which is also the first argument in the function's argument list) is at address "[ebp+8]". Subsequent arguments are at address "[ebp+12]" or "[ebp+16]", depending on the size of the argument pushed onto the stack at "8[ebp]".

80386 Return Values

Assembly-language routines that return values to an 80386 C-language program or receive return values from 80386 C-language functions must follow the 80386 C-language return-value conventions. C-language functions place return values that have **int**, **char**, **unsigned**, **short**, and **long** types in the **EAX** register.

Floating-point values are returned to the top of the `ndp` 80287 stack. The following example shows the recommended instruction sequence for passing floating-point values:

```
float func(), f;
f = func(f)
    fld  DWORD PTR f
    sub  esp, 8
    fstp QWORD PTR [esp]
    call func      ; result in ST(0)
    add  esp, 8
    fstp DWORD PTR f
```

The following example shows the recommended instruction sequence for returning floating-point values:

```
float fvalue;
return (fvalue);
    fld  fvalue      ; result in ST(0)
    pop  edx
    pop  esi
    pop  edi
    leave
    ret
```

C-language structure returns are returned to a buffer whose address is passed as a hidden first parameter.

The following example shows the recommended instruction:

80386 Return Values

```
struct shape
{
    int stuff, to, fill, it, with;
} in, out, them();
out = them(in);

    sub    esp,20
    mov    edi,esp
    lea   edi,in          ; structure copy input
    mov   ecx,5          ; struct onto stack
    rep   movsd
    lea   eax,out        ; pass address of
    push  eax            ; assignment as extra "hidden"
    call  them           ;parameter
    add   esp,24
```

The following example shows the recommended instruction sequence for returning C-language structure returns:

```
struct shape source;
return shape;
    mov    edi,[ebp+8]
    mov    esi,source
    mov    ecx,5
    rep   movsd
    pop   ebx
    pop   esi
    pop   edi
    leave
    ret
```

Exiting an 80386 Routine

Before returning control from an assembly-language routine to an 80386 C-language program, restore the **EBP**, **ESI**, **EDI**, and **EBX** registers. The following example illustrates the recommended instruction sequence for exiting a routine:

```
pop ebx
pop esi
pop edi
leave
ret
```

This sequence does not save the **EAX**, **ECX**, or **EDX** register. These registers are scratch registers for use by the compiler. If the routine modifies segment register **ES**, **SS**, or **DS**, the routine must preserve the modified segment registers. The sequence does not remove arguments from the stack. This is the responsibility of the calling routine.

80386 Program Example

The following example illustrates an 80386 C-language function that can be written as an assembly-language routine. The function takes two integer arguments and adds them together, returning the resultant value.

```
int add(i, j)
int i, j;
{
return(i + j);
}
```

If written as an assembly-language routine, this function must save the proper registers, retrieve the arguments from the stack, add the arguments, place the return value in the **EAX** register, then restore the proper registers and return control to the calling routine. The following is an example of how the routine can be written:

```
_add:
    push ebp
    mov  ebp, esp
    push edi
    push esi
    push ebx

    mov  eax, [ebp+8]
    add  eax, [ebp+12]

    pop  ebx
    pop  esi
    pop  edi
    mov  esp, ebp
    pop  ebp
    ret
```

Note

In the above assembly-language routine, it is not necessary to save the contents of the **ESI**, **EDI**, and **EBX** registers because the routine does not modify their contents. If the **ESI**, **EDI**, or **EBX** register was modified by the routine, its contents must be saved.

80386 Program Example

If the C-language function is to be called by an assembly-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 can then use the return value in the **EAX** register. The following is an example of the instructions that perform this task:

```
push <j value>
push <i value>
call _add
add esp,8
```




Chapter 9

Error Processing

Introduction 9-1

Using the Standard Error File 9-2

Using the errno Variable 9-3

Printing Error Messages 9-4

Using Error Signals 9-5

Encountering System Errors 9-6



Introduction

System V automatically detects and reports errors that occur when using standard C library functions. Errors range from problems with accessing files to allocating memory. In most cases, the system simply reports the error and lets the program decide how to respond. System V terminates a program only if a serious error has occurred, such as a violation of memory space.

This chapter explains how to process errors, and describes the functions and variables a program may use to respond to errors.

Using the Standard Error File

The standard error file is a special output file that can be used by a program to display error messages. The standard error file is one of three standard files (standard input, output, and error) automatically created for the program when it is invoked. Note that this feature is only available under the Bourne shell (*/bin/sh*).

The standard error file, like the standard output, is normally assigned to the user's terminal screen. Thus, error messages written to the file are displayed on the screen. The file can also be redirected by using the shell's redirection symbol (>). For example, the following command redirects the standard error file to the file *errorlist* under Bourne shell:

```
dc 2>errorlist
```

The standard error file, like the standard input and standard output, has predefined file pointer and file descriptor values. The file pointer **stderr** may be used in stream functions to copy data to the error file. The file descriptor **2** may be used in low-level functions to copy data to the file. For example, in the following program fragment, **stderr** is used to write the message "Unexpected end of file." to the standard error file.

```
if ( (c=getchar()) == EOF)
    fprintf(stderr, "Unexpected end of file.\n");
```

The standard error file is not affected by the shell's pipe symbol (*|*). This means that even if the standard output of a program is piped to another program, errors generated by the original program will still appear at the terminal screen (or in the appropriate file if the standard error is redirected).

Using the `errno` Variable

The `errno` variable is a predefined external variable which contains the error number of the most recent System V system function error. Errors detected by system functions, such as access permission errors and lack of space, cause the system to set the `errno` variable to a number and return control to the program. The error number identifies the error condition. The variable may be used in subsequent statements to process the error.

The file `errno.h` contains manifest constant definitions for each error number, and the external declaration of `errno`. These constants may be used in any program in which the line:

```
#include <errno.h>
```

is placed at the beginning of the program. The meaning of each manifest constant is described in the manual page, `intro(S)`.

The `errno` variable is typically used immediately after a system function has returned an error. In the following program fragment, `errno` is used to determine the course of action after an unsuccessful call to the `open` function:

```
if ( (fd=open("accounts", O_RDONLY)) == -1 )
    switch (errno) {
        case(EACCES):
            fd = open("/usr/tmp/accounts",O_RDONLY);
            break;
        default:
            exit(errno);
    }
```

In this example, if `errno` is equal to `EACCES` (a manifest constant), permission to open the file `accounts` in the current directory is denied, so the file is opened in the directory `/usr/tmp` instead. If the variable is any other value, the program terminates.

Printing Error Messages

The **perror** function copies a short error message describing the most recent system function error to the standard error file. The function call has the form:

```
perror (s);
```

where *s* is a pointer to a string containing additional information about the error.

The **perror** function places the given string before the error message and separates the two with a colon (:). Each error message corresponds to the current value of the **errno** variable. For example, in the following program fragment, **perror** displays the message:

```
accounts: Permission denied.
```

if **errno** is equal to the constant **EACCES**:

```
if ( errno == EACCES ) {  
    perror("accounts");  
    fd = open ("/usr/tmp/accounts", O_RDONLY);  
}
```

All error messages displayed by **perror** are stored in an array named **sys_errno**, an external array of character strings. The **perror** function uses the variable **errno** as the index to the array element containing the desired message. For more information on the **perror** function, see the **perror(S)** manual page.

Using Error Signals

Some program errors cause System V to generate error signals. These signals are passed back to the program that caused the error and normally terminate the program. The most common error signals are SIGBUS, the bus error signal; SIGFPE, the floating point exception signal; SIGSEGV, the segment violation signal; SIGSYS, the system call error signal; and SIGPIPE, the pipe error signal. Other signals are described in the **signal(S)** manual page.

A program can, if necessary, catch an error signal and perform its own error processing by using the **signal** function. This function, as described in the “Using Signals” chapter of the *Programmer’s Guide*, can set the action of a signal to a user-defined action. For example, the function call:

```
signal(SIGBUS, fixbus);
```

sets the action of the bus error signal to the action defined by the user-supplied function *fixbus*. Such a function usually attempts to remedy the problem, or at least display detailed information about the problem before terminating the program.

For details about how to catch, redefine, and restore these signals, see “Signals and Interrupts” in the *Programmer’s Guide*.

Encountering System Errors

Programs that encounter serious errors, such as hardware failures or internal errors, generally do not receive detailed reports on the cause of the errors. Instead, UNIX operating systems treat these errors as “system errors,” and report them by displaying a system error message on the system console. This section briefly describes some aspects of System V system errors and how they relate to user programs.

Most system errors occur during calls to system functions. If the system error is recoverable, the system will return an error value to the program and set the **errno** variable to an appropriate value. No other information about the error is available.

Although the system lets two or more programs share a given resource, it does not keep close track of which program is using the resource at any given time. When an error occurs, the system returns an error value to all programs regardless of which caused the error. No information about which program caused the error is available.

System errors that occur during routine I/O operations initiated by the System V system itself generally do not affect user programs. Such errors cause the system to display appropriate system error messages on the system console.

Some system errors are not detected by the system until after the corresponding function has returned successfully. Such errors occur when data written to a file by a program has been queued for writing to disk at a more convenient time, or when a portion of data to be read from disk is found to already be in memory and the remaining portion is not read until later. In such cases, the system assumes that the subsequent read or write operation will be carried out successfully and passes control back to the program along with a successful return value. If operation is not carried out successfully, it causes a delayed error.

When a delayed error occurs, the system usually attempts to return an error on the next call to a system function that accesses the same file or resource. If the program has already terminated or does not make a suitable call, then the error is not reported.

Chapter 10

Common Object File Format (COFF)

- The Common Object File Format (COFF) 10-1
- Definitions and Conventions 10-3
 - Sections 10-3
 - Physical and Virtual Addresses 10-3
 - Target Machine 10-4
- File Header 10-5
 - Magic Numbers 10-5
 - Flags 10-6
 - File Header Declaration 10-6
- Optional Header Information 10-7
 - Standard UNIX System **a.out** Header 10-7
 - Optional Header Declaration 10-8
- Section Headers 10-9
 - Flags 10-10
 - Section Header Declaration 10-11
 - .bss** Section Header 10-11
- Sections 10-12
- Relocation Information 10-13
 - Relocation Entry Declaration 10-14
- Line Numbers 10-15
 - Line Number Declaration 10-16
- Symbol Table 10-17
 - Special Symbols 10-18
 - Inner Blocks 10-19
 - Symbols and Functions 10-21
 - Symbol Table Entries 10-21
 - Symbol Names 10-22
 - Storage Classes 10-24
 - Storage Classes for Special Symbols 10-25
 - Symbol Value Field 10-26

Section Number Field	10-27
Section Numbers and Storage Classes	10-28
Type Entry	10-28
Type Entries and Storage Classes	10-30
Structure for Symbol Table Entries	10-32
Auxiliary Table Entries	10-33
File Names	10-34
Sections	10-34
Tag Names	10-35
End of Structures	10-35
Functions	10-36
Arrays	10-37
End of Blocks and Functions	10-37
Beginning of Blocks and Functions	10-38
Names Related to Structures, Unions, and Enumerations	10-38
Auxiliary Entry Declaration	10-39
String Table	10-41
Access Routines	10-42

The Common Object File Format (COFF)

This chapter describes the Common Object File Format (COFF) used on your computer with UNIX System V. *COFF* is the format of the output file produced by the UNIX System assembler, *as*, and link editor, *ld*. This is the default format used by *cc* without the *-xenix* option.

The following list describes key features of *COFF*:

- Applications can add system-dependent information to the object file without causing access utilities to become obsolete.
- Space is provided for symbolic information used by debuggers and other applications.
- Programmers can modify the way the object file is constructed by providing directives at compile time.

The object file supports user-defined sections and contains extensive information for symbolic software testing. An object file contains

- a file header
- optional header information
- a table of section headers
- data corresponding to the section headers
- relocation information
- line numbers
- a symbol table
- a string table

The Common Object File Format (COFF)

Figure 10-1 shows the overall structure of a COFF object file.

Figure 10-1 Object File Format

FILE HEADER
Optional Information
Section 1 Header
...
Section <i>n</i> Header
Raw Data for Section 1
...
Raw Data for Section <i>n</i>
Relocation Info for Sect. 1
...
Relocation Info for Sect. <i>n</i>
Line Numbers for Sect. 1
...
Line Numbers for Sect. <i>n</i>
SYMBOL TABLE
STRING TABLE

The last four sections (relocation, line numbers, symbol table, and the string table) may be missing if the program is linked with the `-s` option of the `ld` command, or if the line number information, symbol table, and string table are removed by the `strip` command. The line number information does not appear unless the program is compiled with the `-g` option of the `cc` command. Also, if there are no unresolved external references after linking, the relocation information is no longer needed and is absent. The string table is also absent if the source file does not contain any symbols with names longer than eight characters.

An object file that contains no errors or unresolved references is considered executable.

Definitions and Conventions

Before proceeding further, you should become familiar with the following terms and conventions.

Sections

A section is the smallest portion of an object file that is relocated and treated as one separate and distinct entity. In the most common case, there are three sections named **.text**, **.data**, and **.bss**. Additional sections accommodate comments, multiple text or data segments, shared data segments, or user-specified sections. However, the operating system loads only **.text**, **.data**, and **.bss** into memory when the file is executed.

Note

It is a mistake to assume that every COFF file will have a certain number of sections, or to assume characteristics of sections such as their order, their location in the object file, or the address at which they are to be loaded. This information is available only after the object file has been created. Programs manipulating COFF files should obtain it from file and section headers in the file.

Physical and Virtual Addresses

The physical address of a section or symbol is the offset of that section or symbol from address zero of the address space. The term physical address as used in COFF does not correspond to general usage. The physical address of an object is not necessarily the address at which the object is placed when the process is executed. For example, on a system with paging, the address is located with respect to address zero of virtual memory and the system performs another address translation. The section header contains two address fields, a physical address, and a virtual address; but in all versions of COFF on UNIX Systems, the physical address is equivalent to the virtual address.

Definitions and Conventions

Target Machine

Compilers and link editors produce executable object files that are intended to be run on a particular computer. In the case of cross-compilers, the compilation and link editing are done on one computer, with the intent of creating an object file that can be executed on another computer. The term, target machine, refers to the computer on which the object file is destined to run. In the majority of cases, the target machine is the same computer on which the object file is being created.

File Header

The file header contains the 20 bytes of information shown in Table 10.1. The last 2 bytes are flags that are used by `ld` and object file utilities.

Table 10.1
File Header Contents

Bytes	Declaration	Name	Description
0-1	unsigned short	<code>f_magic</code>	Magic number
2-3	unsigned short	<code>f_nscns</code>	Number of sections
4-7	long int	<code>f_timdat</code>	Time and date stamp indicating when the file was created, expressed as the number of elapsed seconds since 00:00:00 GMT, January 1, 1970
8-11	long int	<code>f_symptr</code>	File pointer containing the starting address of the symbol table
12-15	long int	<code>f_nsyms</code>	Number of entries in the symbol table
16-17	unsigned short	<code>f_opthdr</code>	Number of bytes in the optional header
18-19	unsigned short	<code>f_flags</code>	Flags (see Table 10.2.)

Magic Numbers

The magic number specifies the target machine on which the object file is executable.

File Header

Flags

The last 2 bytes of the file header are flags that describe the type of the object file. Currently defined flags are found in the header file `filehdr.h` and are shown in Table 10.2.

Table 10.2
File Header Flags

Mnemonic	Flag	Meaning
F_RELFLG	00001	Relocation information stripped from the file
F_EXEC	00002	File is executable (i.e., no unresolved external references)
F_LNNO	00004	Line numbers stripped from the file
F_LSYMS	00010	Local symbols stripped from the file
F_AR16WR	0000200	16-bit byte reversed word
F_AR32WR	0000400	32-bit byte reversed word

File Header Declaration

The C structure declaration for the file header is given in Figure 10-2. This declaration may be found in the header file `filehdr.h`.

Figure 10-2 File Header Declaration

```
struct filehdr
{
    unsigned short    f_magic;    /* magic number */
    unsigned short    f_nscns;    /* number of section */
    long              f_timdat;   /* time and date stamp */
    long              f_symptr;   /* file ptr to symbol table */
    long              f_nsyms;    /* number entries in the symbol t
    unsigned short    f_opthdr;   /* size of optional header */
    unsigned short    f_flags;    /* flags */
};
#define FILHDR struct filehdr
#define FILHSZ sizeof(FILHDR)
```

Optional Header Information

The template for optional information varies among different systems that use COFF. Applications place all system-dependent information into this record. This allows different operating systems access to information that only that operating system uses without forcing all COFF files to save space for that information. General utility programs (for example, the symbol table access library functions, the disassembler, etc.) are made to work properly on any common object file. This is done by seeking past this record using the size of optional header information in the file header field `f_opthdr`.

Standard UNIX System a.out Header

By default, files produced by the link editor for a UNIX System always have a standard UNIX System `a.out` header in the optional header field. The UNIX System `a.out` header is 28 bytes. The fields of the optional header are described in Table 10.3.

Table 10.3
Optional Header Contents

Bytes	Declaration	Name	Description
0-1	<code>short</code>	<code>magic</code>	Magic number
2-3	<code>short</code>	<code>vstamp</code>	Version stamp
4-7	<code>long int</code>	<code>tsize</code>	Size of text in bytes
8-11	<code>long int</code>	<code>dsize</code>	Size of initialized data in bytes
12-15	<code>long int</code>	<code>bsize</code>	Size of uninitialized data in bytes
16-19	<code>long int</code>	<code>entry</code>	Entry point
20-23	<code>long int</code>	<code>text_start</code>	Base address of text
24-27	<code>long int</code>	<code>data_start</code>	Base address of data

Whereas the magic number in the file header specifies the machine on which the object file runs, the magic number in the optional header supplies information telling the operating system on that machine how that file should be executed. The magic numbers recognized by UNIX System V are given in Table 10.4.

Optional Header Information

Table 10.4
UNIX System V Magic Numbers

Value	Meaning
0407	Text segment is not write-protected or sharable; data segment is contiguous with the text segment.
0410	Data segment starts at the next segment following the text segment and the text segment is write-protected.
0413	Text and data segments are aligned within a.out so it can be directly paged.
0443	Defines a.out to be a target shared library.

Optional Header Declaration

The C language structure declaration currently used for UNIX System's **a.out** file headers is given in Figure 10-3. This declaration may be found in the header file **aouthdr.h**.

Figure 10-3 **aouthdr** Declaration

```
typedef struct aouthdr
{
    short    magic;        /* magic number */
    short    vstamp;      /* version stamp */
    long     tsize;       /* text size in bytes, padded
                          * to full word boundary
                          */
    long     dsize;       /* initialized data size */
    long     bsize;       /* uninitialized data size */
    long     entry;       /* entry point */
    long     text_start;  /* base of text for this file */
    long     data_start   /* base of data for this file */
} AOUTHDR;
```

Section Headers

Every object file has a table of section headers to specify the layout of data within the file. The section header table consists of one entry for every section in the file. The information in the section header is described in Table 10.5.

Table 10.5
Section Header Contents

Bytes	Declaration	Name	Description
0-7	char	s_name	8-character null padded section name
8-11	long int	s_paddr	Physical address of section
12-15	long int	s_vaddr	Virtual address of section
16-19	long int	s_size	Section size in bytes
20-23	long int	s_scnptr	File pointer to raw data
24-27	long int	s_relptr	File pointer to relocation entries
28-31	long int	s_lnnoptr	File pointer to line number entries
32-33	unsigned short	s_nreloc	Number of relocation entries
34-35	unsigned short	s_nlnno	Number of line number entries
36-39	long int	s_flags	Flags (see Table 10.6)

The size of a section is padded to a multiple of 4 bytes. File pointers are byte offsets that can be used to locate the start of data, relocation, or line number entries for the section. They can be readily used with the `fseek(S)` system call.

Section Headers

Flags

The lower 2 bytes of the flag field indicate a section type. The flags are described in Table 10.6.

Table 10.6
Section Header Flags

Mnemonic	Flag	Meaning
STYP_REG	0x00	Regular section (allocated, relocated, loaded)
STYP_DSECT	0x01	Dummy section (not allocated, relocated, not loaded)
STYP_NOLOAD	0x02	Noload section (allocated, relocated, not loaded)
STYP_GROUP	0x04	Grouped section (formed from input sections)
STYP_PAD	0x08	Padding section (not allocated, not relocated, loaded)
STYP_COPY	0x10	Copy section (for a decision function used in updating fields; not allocated, not relocated, loaded, relocation and line number entries processed normally)
STYP_TEXT	0x20	Section contains executable text
STYP_DATA	0x40	Section contains initialized data
STYP_BSS	0x80	Section contains only uninitialized data
STYP_INFO	0x200	Comment section (not allocated, not relocated, not loaded)
STYP_OVER	0x400	Overlay section (relocated, not allocated, not loaded)
STYP_LIB	0x800	For <code>.lib</code> section (treated like STYP_INFO)

Section Header Declaration

The C structure declaration for the section headers is described in Figure 10-4. This declaration may be found in the header file `scnhdr.h`.

Figure 10-4 Section Header Declaration

```

struct scnhdr
{
    char        s_name[8];           /* section name */
    long        s_paddr;            /* physical address */
    long        s_vaddr;            /* virtual address */
    long        s_size;             /* section size */
    long        s_scnptr;           /* file ptr to section raw data */
    long        s_relptr;           /* file ptr to relocation */
    long        s_lnnoptr;          /* file ptr to line number */
    unsigned short s_nreloc;        /* number of relocation entries */
    unsigned short s_nlnno;        /* number of line number entries */
    long        s_flags;            /* flags */
};

#define SCNHDR  struct scnhdr
#define SCNHSZ  sizeof(SCNHDR)

```

.bss Section Header

The one deviation from the normal rule in the section header table is the entry for uninitialized data in a `.bss` section. A `.bss` section has a size and symbols that refer to it, and symbols that are defined in it. At the same time, a `.bss` section has no relocation entries, no line number entries, and no data. Therefore, a `.bss` section has an entry in the section header table but occupies no space elsewhere in the file. In this case, the number of relocation and line number entries, as well as all file pointers in a `.bss` section header, are 0. The same is true of the `STYP_NOLOAD` and `STYP_DSECT` sections.

Sections

Figure 10-1 shows that section headers are followed by the appropriate number of bytes of text or data. The raw data for each section begins on a 4-byte boundary in the file.

Link editor `SECTIONS` directives allow users to do the following, among other things:

- describe how input sections are to be combined
- direct the placement of output sections
- rename output sections.

If no `SECTIONS` directives are given, each input section appears in an output section of the same name. For example, if a number of object files, each with a `.text` section, are linked together, the output object file contains a single `.text` section made up of the combined input `.text` sections.

Relocation Information

Object files have one relocation entry for each relocatable reference in the text or data. The relocation information consists of entries with the format described in Table 10.7.

Table 10.7
Relocation Section Contents

Bytes	Declaration	Name	Description
0-3	long int	r_vaddr	(Virtual) address of reference
4-7	long int	r_symndx	Symbol table index
8-9	unsigned short	r_type	Relocation type

The first 4 bytes of the entry are the virtual address of the text or data to which this entry applies. The next field is the index, counted from 0, of the symbol table entry that is being referenced. The type field indicates the type of relocation to be applied.

As the link editor reads each input section and performs relocation, the relocation entries are read. They direct how references found within the input section are treated. The currently recognized relocation types are given in Table 10.8.

Relocation Information

Table 10.8
Relocation Types

Mnemonic	Flag	Meaning
R_ABS	0	Reference is absolute; no relocation is necessary. The entry will be ignored.
R_DIR16 *	01	Direct, 16-bit reference to a symbol's virtual address.
R_REL16 *	02	"PC-relative", 16-bit reference to a symbol's virtual address. Relative references occur in instructions such as jumps and calls.
R_DIR32	06	Direct 32-bit reference to the symbol's virtual address.
R_SEG12 *	011	Direct, 16-bit reference to the segment-selector bits of a 32-bit virtual address.
R_PCRLONG †	024	"PC_relative", 32-bit reference to a symbol's virtual address.

* 80286 Computer only.

† 80386 Computer only.

Relocation Entry Declaration

The structure declaration for relocation entries is given in Figure 10-5. This declaration may be found in the header file **reloc.h**.

Figure 10-5 Relocation Entry Declaration

```
struct reloc
{
    long          r_vaddr; /* virtual address of reference */
    long          r_symndx; /* index into symbol table */
    unsigned short r_type; /* relocation type */
};

#define RELOC    struct reloc
#define RELSZ    10
```

Line Numbers

When invoked with the `-g` option, the `cc` and `f77` commands cause an entry in the object file for every source line where a breakpoint can be inserted. You can then reference line numbers when using a software debugger like `sdb`. All line numbers in a section are grouped by function as shown in Table 10.9.

Table 10.9
Line Number Grouping

symbol index	0
physical address	line number
physical address	line number
.	.
.	.
.	.
symbol index	0
physical address	line number
physical address	line number

The first entry in a function grouping has line number 0 and has, in place of the physical address, an index into the symbol table for the entry containing the function name. Subsequent entries have actual line numbers and addresses of the text corresponding to the line numbers. The line number entries are relative to the beginning of the function and appear in increasing order of address.

Line Numbers

Line Number Declaration

The structure declaration currently used for line number entries is given in Figure 10-6.

Figure 10-6 Line Number Entry Declaration

```
struct lineno
{
    union
    {
        long    l_symndx;    /* sytbl index of func name */
        long    l_paddr;    /* paddr of line number */
    } l_addr;
    unsigned short  l_lno;    /* line number */
};

#define LINENO    struct lineno
#define LINESZ    6
```

Symbol Table

Because of symbolic debugging requirements, the order of symbols in the symbol table is very important. Symbols appear in the sequence shown in Figure 10-7.

Figure 10-7 COFF Symbol Table

filename 1
function 1
local symbols for function 1
function 2
local symbols for function 2
...
statics
...
filename 2
function 1
local symbols for function 1
...
statics
...
defined global symbols
undefined global symbols

The word *statics* in Figure 10-7 means symbols defined with the C language storage class *static* outside any function. The symbol table consists of at least one fixed-length entry per symbol with some symbols followed by auxiliary entries of the same size. The entry for each symbol is a structure that holds the value, the type, and other information.

Symbol Table

Special Symbols

The symbol table contains some special symbols that are generated by `as` and other tools. These symbols are given in Table 10.10.

Table 10.10
Special Symbols in the Symbol Table

Symbol	Meaning
<code>.file</code>	filename
<code>.text</code>	address of <code>.text</code> section
<code>.data</code>	address of <code>.data</code> section
<code>.bss</code>	address of <code>.bss</code> section
<code>.bb</code>	address of start of inner block
<code>.eb</code>	address of end of inner block
<code>.bf</code>	address of start of function
<code>.ef</code>	address of end of function
<code>.target</code>	pointer to the structure or union returned by a function
<code>xfake</code>	dummy tag name for structure, union, or enumeration
<code>.eos</code>	end of members of structure, union, or enumeration
<code>etext</code>	next available address after the end of the output section <code>.text</code>
<code>edata</code>	next available address after the end of the output section <code>.data</code>
<code>end</code>	next available address after the end of the output section <code>.bss</code>

Six of these special symbols occur in pairs. The `.bb` and `.eb` symbols indicate the boundaries of inner blocks; a `.bf` and `.ef` pair brackets each function. An `xfake` and `.eos` pair names and defines the limit of structures, unions, and enumerations that were not named. The `.eos` symbol also appears after named structures, unions, and enumerations.

When a structure, union, or enumeration has no tag name, the compiler invents a name to be used in the symbol table. The name chosen for the symbol table is `xfake`, where `x` is an integer. If there are three unnamed structures, unions, or enumerations in the source, their tag names are `.0fake`, `.1fake`, and `.2fake`. Each of the special symbols has different information stored in the symbol table entry as well as the auxiliary entries.

Inner Blocks

The C language defines a block as a compound statement that begins and ends with braces, { and }. An inner block is a block that occurs within a function (which is also a block).

For each inner block that has local symbols defined, a special symbol **.bb** is put in the symbol table immediately before the first local symbol of that block. Also a special symbol **.eb** is put in the symbol table immediately after the last local symbol of that block. The sequence is shown in Figure 10-8.

Figure 10-8 Special Symbols (**.bb** and **.eb**)

.bb
local symbols
for that block
.eb

Symbol Table

Because inner blocks can be nested by several levels, the **.bb-.eb** pairs and associated symbols may also be nested (see Figure 10-9).

Figure 10-9 Nested blocks

```
{          /*block 1*/
  inti;
  charc;
  ...
  {          /*block 2*/
    longa;
    ...
    {          /*block 3*/
      intx;
      ....
    }          /*block 3*/
  }          /*block 2*/
}          /*block 1*/

{          /*block 4*/
  longi;
  ...
}          /*block 4*/
}          /*block 1*/
```

The symbol table would look like Figure 10-10.

Figure 10-10 Example of the Symbol Table

.bb for block 1
i
c
.bb for block 2
a
.bb for block 3
x
.eb for block 3
.eb for block 2
.bb for block 4
i
.eb for block 4
.eb for block 1

Symbols and Functions

For each function, a special symbol **.bf** is put between the function name and the first local symbol of the function in the symbol table. Also, a special symbol **.ef** is put immediately after the last local symbol of the function in the symbol table. The sequence is shown in Figure 10-11.

Figure 10-11 Symbols for Functions

function name
.bf
local symbol
.ef

Symbol Table Entries

All symbols, regardless of storage class and type, have the same format for their entries in the symbol table. The symbol table entries each contain 18 bytes of information. The meaning of each of the fields in the symbol table entry is described in Table 10.11. It should be noted that indices for symbol table entries begin at 0 and count upward. Each auxiliary entry also counts as one symbol.

Symbol Table

Table 10.11
Symbol Table Entry Format

Bytes	Declaration	Name	Description
0-7	(see text below)	_n	These 8 bytes contain either a symbol name or an index to a symbol
8-11	long int	n_value	Symbol value; storage class dependent
12-13	short	n_snum	Section number of symbol
14-15	unsigned short	n_type	Basic and derived type specification
16	char	n_sclass	Storage class of symbol
17	char	n_numaux	Number of auxiliary entries

Symbol Names

The first 8 bytes in the symbol table entry are a union of a character array and two longs. If the symbol name is eight characters or less, the (null-padded) symbol name is stored there. If the symbol name is longer than eight characters, then the entire symbol name is stored in the string table. In this case, the 8 bytes contain two long integers, the first is zero, and the second is the offset (relative to the beginning of the string table) of the name in the string table. Since there can be no symbols with a null name, the zeroes on the first 4 bytes serve to distinguish a symbol table entry with an offset from one with a name in the first 8 bytes as shown in Table 10.12.

Table 10.12
Name Field

Bytes	Declaration	Name	Description
0-7	char	n_name	8-character null-padded symbol name
0-3	long	n_zeroes	Zero in this field indicates the name is in the string table
4-7	long	n_offset	Offset of the name in the string table

Special symbols generated by the C Compilation System are discussed earlier in the section “Special Symbols” in this chapter.

Symbol Table

Storage Classes

The storage class field has one of the values described in Table 10.13. These `#define`'s may be found in the header file `storclass.h`.

Table 10.13
Storage Classes

Mnemonic	Value	Storage Class
C_EFCN	-1	physical end of a function
C_NULL	0	-
C_AUTO	1	automatic variable
C_EXT	2	external symbol
C_STAT	3	static
C_REG	4	register variable
C_EXTDEF	5	external definition
C_LABEL	6	label
C_ULABEL	7	undefined label
C_MOS	8	member of structure
C_ARG	9	function argument
C_STRTAG	10	structure tag
C_MOU	11	member of union
C_UNTAG	12	union tag
C_TPDEF	13	type definition
C_USTATIC	14	uninitialized static
C_ENTAG	15	enumeration tag
C_MOE	16	member of enumeration
C_REGPARAM	17	register parameter
C_FIELD	18	bit field
C_BLOCK	100	beginning and end of block
C_FCN	101	beginning and end of function
C_EOS	102	end of structure
C_FILE	103	file name
C_LINE	104	used only by utility programs
C_ALIAS	105	duplicated tag
C_HIDDEN	106	like static, used to avoid name conflicts

All of these storage classes except for `C_ALIAS` and `C_HIDDEN` are generated by the `cc` or `as` commands. The compress utility, `cprs`, generates the `C_ALIAS` mnemonic. This utility (described in the *Programmer's Reference Manual*) removes duplicated structure, union, and enumeration definitions and puts alias entries in their places. The storage class `C_HIDDEN` is not used by any UNIX System V tools.

Symbol Table

Some of these storage classes are used only internally by the C Compilation Systems. These storage classes are C_EFCN, C_EXTDEF, C_ULABEL, C_USTATIC, and C_LINE.

Storage Classes for Special Symbols

Some special symbols are restricted to certain storage classes. They are given in Table 10.14.

Table 10.14
Storage Class by Special Symbols

Special Symbol	Storage Class
.file	C_FILE
.bb	C_BLOCK
.eb	C_BLOCK
.bf	C_FCN
.ef	C_FCN
.target	C_AUTO
.xfake	C_STRTAG, C_UNTAG, C_ENTAG
.eos	C_EOS
.text	C_STAT
.data	C_STAT
.bss	C_STAT

Also some storage classes are used only for certain special symbols. They are summarized in Table 10.15.

Table 10.15
Restricted Storage Classes

Storage Class	Special Symbol
C_BLOCK	.bb, .eb
C_FCN	.bf, .ef
C_EOS	.eos
C_FILE	.file

Symbol Table

Symbol Value Field

The meaning of the value of a symbol depends on its storage class. This relationship is summarized in Table 10.16.

Table 10.16
Storage Class and Value

Storage Class	Meaning of Value
<code>C_AUTO</code>	stack offset in bytes
<code>C_EXT</code>	relocatable address
<code>C_STAT</code>	relocatable address
<code>C_REG</code>	register number
<code>C_LABEL</code>	relocatable address
<code>C_MOS</code>	offset in bytes
<code>C_ARG</code>	stack offset in bytes
<code>C_STRTAG</code>	0
<code>C_MOU</code>	0
<code>C_UNTAG</code>	0
<code>C_TPDEF</code>	0
<code>C_ENTAG</code>	0
<code>C_MOE</code>	enumeration value
<code>C_REGPARM</code>	register number
<code>C_FIELD</code>	bit displacement
<code>C_BLOCK</code>	relocatable address
<code>C_FCN</code>	relocatable address
<code>C_EOS</code>	size
<code>C_FILE</code>	(see text below)
<code>C_ALIAS</code>	tag index
<code>C_HIDDEN</code>	relocatable address

If a symbol has storage class `C_FILE`, the value of that symbol equals the symbol table entry index of the next `.file` symbol. That is, the `.file` entries form a one-way linked list in the symbol table. If there are no more `.file` entries in the symbol table, the value of the symbol is the index of the first global symbol.

Relocatable symbols have a value equal to the virtual address of that symbol. When the section is relocated by the link editor, the value of these symbols changes.

Section Number Field

Section numbers are listed in Table 10.17

Table 10.17
Section Number

Mnemonic	Section Number	Meaning
N_DEBUG	-2	Special symbolic debugging symbol
N_ABS	-1	Absolute symbol
N_UNDEF	0	Undefined external symbol
N_SCNUM	1-077777	Section number where symbol is defined

A special section number (-2) marks symbolic debugging symbols, including structure/union/enumeration tag names, typedefs, and the name of the file. A section number of -1 indicates that the symbol has a value but is not relocatable. Examples of absolute-valued symbols include automatic and register variables, function arguments, and `.eos` symbols.

With one exception, a section number of 0 indicates a relocatable external symbol that is not defined in the current file. The one exception is a multiply-defined external symbol (i.e., FORTRAN common or an uninitialized variable-defined external to a function in C). In the symbol table of each file where the symbol is defined, the section number of the symbol is 0, and the value of the symbol is a positive number giving the size of the symbol. When the files are combined to form an executable object file, the link editor combines all the input symbols of the same name into one symbol with the section number of the `.bss` section. The maximum size of all the input symbols with the same name is used to allocate space for the symbol and the value becomes the address of the symbol. This is the only case where a symbol has a section number of 0 and a non-zero value.

Symbol Table

Section Numbers and Storage Classes

Symbols having certain storage classes are also restricted to certain section numbers. They are summarized in Table 10.18.

Table 10.18
Section Number and Storage Class

Storage Class	Section Number
C_AUTO	N_ABS
C_EXT	N_ABS, N_UNDEF, N_SCNUM
C_STAT	N_SCNUM
C_REG	N_ABS
C_LABEL	N_UNDEF, N_SCNUM
C_MOS	N_ABS
C_ARG	N_ABS
C_STRTAG	N_DEBUG
C_MOU	N_ABS
C_UNTAG	N_DEBUG
C_TPDEF	N_DEBUG
C_ENTAG	N_DEBUG
C_MOE	N_ABS
C_REGPARM	N_ABS
C_FIELD	N_ABS
C_BLOCK	N_SCNUM
C_FCN	N_SCNUM
C_EOS	N_ABS
C_FILE	N_DEBUG
C_ALIAS	N_DEBUG

Type Entry

The type field in the symbol table entry contains information about the basic and derived type for the symbol. This information is generated by the C Compilation System only if the **-g** option is used. Each symbol has exactly one basic or fundamental type but can have more than one derived type. The format of the 16-bit type entry is:

Symbol Table

d6	d5	d4	d3	d2	d1	typ
-----------	-----------	-----------	-----------	-----------	-----------	------------

Bits 0 through 3, called **typ**, indicate one of the fundamental types given in Table 10.19.

Table 10.19
Fundamental Types

Mnemonic	Value	Type
T_NULL	0	type not assigned
T_ARG	1	Function argument (used only by compiler)
T_CHAR	2	character
T_SHORT	3	short integer
T_INT	4	integer
T_LONG	5	long integer
T_FLOAT	6	floating point
T_DOUBLE	7	double word
T_STRUCT	8	structure
T_UNION	9	union
T_ENUM	10	enumeration
T_MOE	11	member of enumeration
T_UCHAR	12	unsigned character
T_USHORT	13	unsigned short
T_UINT	14	unsigned integer
T_ULONG	15	unsigned long

Symbol Table

Bits 4 through 15 are arranged as six 2-bit fields marked **d1** through **d6**. These **d** fields represent levels of the derived types given in Table 10.20.

Table 10.20
Derived Types

Mnemonic	Value	Type
DT_NON	0	no derived type
DT_PTR	1	pointer
DT_FCN	2	function
DT_ARY	3	array

The following examples demonstrate the interpretation of the symbol table entry representing type.

```
char *func();
```

Here **func** is the name of a function that returns a pointer to a character. The fundamental type of **func** is 2 (character), the **d1** field is 2 (function), and the **d2** field is 1 (pointer). Therefore, the type word in the symbol table for **func** contains the hexadecimal number 0x62, which is interpreted to mean a function that returns a pointer to a character.

```
short *tabptr[10][25][3];
```

Here **tabptr** is a three-dimensional array of pointers to short integers. The fundamental type of **tabptr** is 3 (short integer); the **d1**, **d2**, and **d3** fields each contains a 3 (array), and the **d4** field is 1 (pointer). Therefore, the type entry in the symbol table contains the hexadecimal number 0x7f3 indicating a three-dimensional array of pointers to short integers.

Type Entries and Storage Classes

Table 10.21 shows the type entries that are legal for each storage class.

Table 10.21
Type Entries by Storage Class

Storage Class	d Entry			typ Entry Basic Type
	Function?	Array?	Pointer?	
C_AUTO	no	yes	yes	Any except T_MOE
C_EXT	yes	yes	yes	Any except T_MOE
C_STAT	yes	yes	yes	Any except T_MOE
C_REG	no	no	yes	Any except T_MOE
C_LABEL	no	no	no	T_NULL
C_MOS	no	yes	yes	Any except T_MOE
C_ARG	yes	no	yes	Any except T_MOE
C_STRTAG	no	no	no	T_STRUCT
C_MOU	no	yes	yes	Any except T_MOE
C_UNTAG	no	no	no	T_UNION
C_TPDEF	no	yes	yes	Any except T_MOE
C_ENTAG	no	no	no	T_ENUM
C_MOE	no	no	no	T_MOE
C_REGPARAM	no	no	yes	Any except T_MOE
C_FIELD	no	no	no	T_ENUM, T_UCHAR, T_USHORT, T_UNIT, T_ULONG
C_BLOCK	no	no	no	T_NULL
C_FCN	no	no	no	T_NULL
C_EOS	no	no	no	T_NULL
C_FILE	no	no	no	T_NULL
C_ALIAS	no	no	no	T_STRUCT, T_UNION, T_ENUM

Conditions for the **d** entries apply to **d1** through **d6**, except that it is impossible to have two consecutive derived types of function.

Symbol Table

Although function arguments can be declared as arrays, they are changed to pointers by default. Therefore, no function argument can have array as its first derived type.

Structure for Symbol Table Entries

The C language structure declaration for the symbol table entry is given in Figure 10-12. This declaration may be found in the header file `syms.h`.

Figure 10-12 Symbol Table Entry Declaration

```
struct syment
{
    union
    {
        {
            char _n_name[SYMMLEN]; /* symbol name*/
            struct
            {
                long _n_zeroes; /* symbol name */
                long _n_offset; /* location in string table */
            } _n_n;
            char *_n_nptr[2]; /* allows overlaying */
        } _n;
        unsigned long n_value; /* value of symbol */
        short n_snum; /* section number */
        unsigned short n_type; /* type and derived */
        char n_sclass; /* storage class */
        char n_nmaux; /* number of aux entries */
    };

#define n_name _n._n_name
#define n_zeroes _n._n_n._n_zeroes
#define n_offset _n._n_n._n_offset
#define n_nptr _n._n_nptr[1]

#define SYMMLEN 8
#define SYMESZ 18 /* size of a symbol table entry */
```

Auxiliary Table Entries

An auxiliary table entry of a symbol contains the same number of bytes as the symbol table entry. However, unlike symbol table entries, the format of an auxiliary table entry of a symbol depends on its type and storage class. They are summarized in Table 10.22.

Table 10.22
Auxiliary Symbol Table Entries

Name	Storage Class	Type Entry		Auxiliary Entry Format
		d1	typ	
.file	C_FILE	DT_NON	T_NULL	file name
.text,.data, .bss	C_STAT	DT_NON	T_NULL	section
<i>tagname</i>	C_STRTAG C_UNTAG C_ENTAG	DT_NON	T_NULL	tag name
.eos	C_EOS	DT_NON	T_NULL	end of structure
<i>fname</i>	C_EXT C_STAT	DT_FCN	(Note 1)	function
<i>arrname</i>	(Note 2)	DT_ARY	(Note 1)	array
.bb,.eb	C_BLOCK	DT_NON	T_NULL	beginning and end of block
.bf,.ef	C_FCN	DT_NON	T_NULL	beginning and end of function
name related to structure, union, enumeration	(Note 2)	DT_PTR, DT_ARR, DT_NON	T_STRUCT, T_UNION, T_ENUM	name related to structure, union, enumeration

Notes to Table 10.22:

1. Any except T_MOE.
2. C_AUTO, C_STAT, C_MOS, C_MOU, C_TPDEF.

Symbol Table

In Table 10.22, *tagname* means any symbol name including the special symbol *.xfake*, and *fcname* and *arrname* represent any symbol name for a function or an array respectively. Any symbol that satisfies more than one condition in Table 10.22 should have a union format in its auxiliary entry. It is a mistake to assume how many auxiliary entries are associated with any given symbol table entry. This information is available and should be obtained from the **n_numaux** field in the symbol table.

File Names

Each of the auxiliary table entries for a file name contains a 14-character file name in bytes 0 through 13. The remaining bytes are 0.

Sections

The auxiliary table entries for sections have the format as shown in Table 10.23.

Table 10.23
Format for Auxiliary Table Entries for Sections

Bytes	Declaration	Name	Description
0-3	long int	x_scnlen	section length
4-5	unsigned short	x_nreloc	number of relocation entries
6-7	unsigned short	x_nlinno	number of line numbers
8-17	-	-	unused (filled with zeroes)

Tag Names

The auxiliary table entries for tag names have the format shown in Table 10.24.

Table 10.24
Tag Names Table Entries

Bytes	Declaration	Name	Description
0-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, and enumeration
8-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry beyond this structure, union, or enumeration
16-17	-	-	unused (filled with zeroes)

End of Structures

The auxiliary table entries for the end of structures have the format shown in Table 10.25.

Symbol Table

Table 10.25
Table Entries for End of Structures

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Functions

The auxiliary table entries for functions have the format shown in Table 10.26.

Table 10.26
Table Entries for Functions

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-7	long int	x_fsize	size of function (in bytes)
8-11	long int	x_innoptr	file pointer to line number
12-15	long int	x_endndx	index of next entry beyond this point
16-17	unsigned short	x_tvndx	index of function's address in the transfer vector table (not used in UNIX System V.)

Arrays

The auxiliary table entries for arrays have the format shown in Table 10.27. Defining arrays with more than four dimensions produces a warning message.

Table 10.27
Table Entries for Arrays

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	unsigned short	x_inno	line number of declaration
6-7	unsigned short	x_size	size of array
8-9	unsigned short	x_dimen[0]	first dimension
10-11	unsigned short	x_dimen[1]	second dimension
12-13	unsigned short	x_dimen[2]	third dimension
14-15	unsigned short	x_dimen[3]	fourth dimension
16-17	-	-	unused (filled with zeroes)

End of Blocks and Functions

The auxiliary table entries for the end of blocks and functions have the format shown in Table 10.28.

Table 10.28
End of Block and Function Entries

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_inno	C-source line number
6-17	-	-	unused (filled with zeroes)

Symbol Table

Beginning of Blocks and Functions

The auxiliary table entries for the beginning of blocks and functions have the format shown in Table 10.29.

Table 10.29
Format for Beginning of Block and Function

Bytes	Declaration	Name	Description
0-3	-	-	unused (filled with zeroes)
4-5	unsigned short	x_lno	C-source line number
6-11	-	-	unused (filled with zeroes)
12-15	long int	x_endndx	index of next entry past this block
16-17	-	-	unused (filled with zeroes)

Names Related to Structures, Unions, and Enumerations

The auxiliary table entries for structure, union, and enumeration symbols have the format shown in Table 10.30.

Table 10.30
Entries for Structures, Unions, and Enumerations

Bytes	Declaration	Name	Description
0-3	long int	x_tagndx	tag index
4-5	-	-	unused (filled with zeroes)
6-7	unsigned short	x_size	size of the structure, union, or enumeration
8-17	-	-	unused (filled with zeroes)

Aggregates defined by **typedef** may or may not have auxiliary table entries. For example,

```
typedef struct people STUDENT;
struct people
{
    char name[20];
    long id;
};
typedef struct people EMPLOYEE;
```

The symbol **EMPLOYEE** has an auxiliary table entry in the symbol table, but the symbol **STUDENT** will not because it is a forward reference to a structure.

Auxiliary Entry Declaration

The C language structure declaration for an auxiliary symbol table entry is given in Figure 10-13. This declaration may be found in the header file **syms.h**.

Figure 10-13 Auxiliary Symbol Table Entry (1 of 2)

```
union auxent
{
    struct
    {
        long x_tagndx;
        union
        {
            struct
            {
                unsigned short x_lno;
                unsigned short x_size;
            } x_lnsz;
            long x_fsize;
        } x_misc;
        union
        {
            struct
            {
                long x_lnoptr;
                long x_endndx;
            } x_fcn;
            struct
            .
            .
            .
        }
    }
};
```

Symbol Table

Figure 10-13 Auxiliary Symbol Table Entry (Sheet 2 of 2)

```
.
.
.
    (
        unsigned short x_dimen[DIMNUM];
    ) x_ary;
} x_fcarray;
unsigned short x_tvndx;
} x_sym;
struct
{
    char x_fname[FILNMLEN];
} x_file;
struct
{
    long x_scrlen;
    unsigned short x_nreloc;
    unsigned short x_nlinno;
} x_scn;
struct
{
    long x_tvfill;
    unsigned short x_tvlen;
    unsigned short x_tvran[2];
} x_tv;
}
#define FILNMLEN 14
#define DIMNUM 4
#define AUXENT union auxent
#define AUXESZ 18
```

String Table

Symbol table names longer than eight characters are stored contiguously in the string table with each symbol name delimited by a null byte. The first four bytes of the string table are the size of the string table in bytes; offsets into the string table, therefore, are greater than or equal to 4. For example, given a file containing two symbols (with names longer than eight characters, **long_name** and **another_one**) the string table has the format as shown in Figure 10-14:

Figure 10-14 String Table

'l'	'o'	'n'	'g'
'_'	'n'	'a'	'm'
'e'	'\0'	'a'	'n'
'o'	't'	'h'	'e'
'r'	'_'	'o'	'n'
'e'	'\0'	'...'	'...'

The index of **long_name** in the string table is 4 and the index of **another_one** is 14.

Access Routines

UNIX System V contains a set of access routines that are used for reading the various parts of a common object file. Although the calling program must know the detailed structure of the parts of the object file it processes, the routines effectively insulate the calling program from the knowledge of the overall structure of the object file.

The access routines can be divided into four categories:

1. functions that open or close an object file
2. functions that read header or symbol table information
3. functions that position an object file at the start of a particular section of the object file
4. a function that returns the symbol table index for a particular symbol

These routines can be found in the library **libld.a** and are listed in the System Services section of the *Programmer's Reference Manual*.

Appendix A

Converting from Previous Versions of the Compiler

Introduction A-1

Differences between Versions 5.1 and 5.0 A-2
 New Features for the System V Release of C A-2
 New Pragmas A-3

Differences between Versions 5.0 and 4.0 A-4
 Enhancements and Additions A-4
 Changes to the Language Syntax A-4
 New Features for the Microsoft Implementation of C A-6

Differences between Versions 4.0 and 3.0 A-8
 Enhancements and Additions A-8
 Changes in the Language Syntax A-8
 New Features for This Implementation of C A-11



Introduction

This appendix describes differences between Version 5.1 and Version 5.0, between Version 5.0 and Version 4.0, and between Version 4.0 and Version 3.0, of the System V Microsoft C Compiler. If you have an earlier version of the compiler, or if you have written programs for an earlier version, this chapter can help you convert your previous source code. The actions necessary to convert source code depend on which of the earlier versions you have used.

Version 5.1 is an update of Version 5.0. Code written for Version 5.0 should compile without change on the Version 5.1 compiler. The primary changes are new pragmas, new keywords, and new command-line changes. Version 5.0 is an update of Version 4.0. Generally, the two versions are compatible: most C source code written for Version 4.0 should compile without change on Version 5.0, although there are erroneous C constructs allowed in Version 4.0 that are not allowed in Version 5.0, and changes in the emerging ANSI C standard may force changes in source programs. For more information, see the *C Language Reference*. In some cases you may be able to enhance your programs by revising them to take advantage of new library functions and other features available with Version 5.0.

Differences between Versions 5.1 and 5.0

Changes in Version 5.1 since Version 5.0 fall into the following categories:

- New compiler options
- New pragmas
- New keywords

New Features for the System V Release of C

The following new options have been added to the System V implementation of the Microsoft C Compiler:

Option	Effect
-S	Generates assembly-language output. The resulting file is intended for the Macro Assembler, masm(CP) .
-xenix	Produces OMF-formatted object files using the language development tools and (if applicable) an <i>x.out</i> format executable file. It also suppresses warning messages about masm directives in any assembly language output files.

New Pragmas

The following new pragmas have been added to Version 5.1 of the C compiler:

Pragma	Effect
comment	Places a comment record in the object file.
linesize	Sets the number of characters per line in the source listing.
message	Sends a message to the standard output without terminating the compilation.
page	Skips the specified number of pages in the source listing.
pagesize	Sets the number of lines per page in the source listing.
skip	Skips the specified number of lines in the source listing.
subtitle	Specifies a subtitle for the source listing.
title	Specifies a title for the source listing.

The pragmas are described in Chapter 2.

Differences between Versions 5.0 and 4.0

Changes in Version 5.0 since Version 4.0 fall into the following categories:

- Enhancements and additions to the compiler software to allow for more flexible programming, improved code generation, and increased support for the developing ANSI standard
- Changes in the language syntax
- Changes in function operations, primarily to conform to the specifications for these functions in the ANSI standard

These features and the changes required to take advantage of them are discussed in the following sections.

Enhancements and Additions

Enhancements for Version 5.0 include the following:

- Improved code generation, including loop optimization; improved large-model code generation; and intrinsic functions
- Faster compilation speed
- Support for code that will be loaded into read-only memory (ROM)
- New error-message numbering

Changes to the Language Syntax

Some Version 5.0 changes were made to the C-language syntax to make it conform more closely to the new ANSI standard. Most of these changes do not affect source code written for the Version 4.0 compiler. The changes are summarized as follows:

- Full function prototyping is supported in Version 5.0. A function prototype is a forward declaration containing the types and, optionally, names of the parameters (if any) expected in the function call.

Differences between Versions 5.0 and 4.0

It can also include identifiers for the arguments, though they go out of scope at the end of the prototype. Prototypes allow the compiler to perform type checking on the actual arguments passed when the function is called. If the compiler does not find a prototype, the first occurrence of the function (definition or call) is used as the basis of a prototype for that function. That prototype is used to perform type checking against subsequent calls, subsequent declarations, or the definition. For more information about function prototyping, see the *C Language Reference*.

- The **const** and **volatile** type specifiers have been implemented for Version 5.0. The **const** type specifier declares an object as an unmodifiable value. It can be used for objects of any fundamental or aggregate type or for pointers to objects of any type. The **volatile** type specifier is implemented syntactically, but not semantically. For more information, see the *C Language Reference*.

Note

Programs that currently use **const** or **volatile** as identifiers must be recoded to use other names.

- In Version 5.0, variables of **enum** type are treated as if they are of **int** type in all cases. Therefore, **enum** variables can be used in indexing expressions and as operands of all relational and arithmetic operators.
- String concatenation is supported in Version 5.0. This feature causes adjacent string literals to be concatenated into a single string literal. This means, for example, that instead of using a backslash before a new-line character to indicate continuation of a long string literal, the literal can simply be broken into two or more quoted string literals on separate lines. For more information, see the *C Language Reference*.
- New preprocessor features in Version 5.0 include the stringizing operator (**#**), which allows arguments in macro expansions to be expanded into a string literal containing the expanded argument; and the concatenation operator (**##**), which concatenates the tokens on either side of the operator into a new token in macro expansions. For more information, see the *C Language Reference*.

Differences between Versions 5.0 and 4.0

Note

Previous versions of C allowed expansion of macro formal arguments appearing in string literals and character constants. Programs that rely on this feature *must* be recoded to use the stringizing operator. For information, see the discussion of string literals in the *C Language Reference*.

- The **long double** data type is now supported; the **long float** data type is no longer supported.
- The three-digit forms of hex escape sequences (`\xddd`) and octal escape sequences (`\ddd`) are now supported.
- The unary plus (+) operator is allowed, but ignored semantically.

New Features for the Microsoft Implementation of C

The following new **cc** command options have been added to the Microsoft C Compiler for Version 5.0:

Option	Effect
-Oi	Enables intrinsic code generation for all available functions
-Ol	Enables loop optimizations for an entire program
-Op	Forces consistent precision in floating-point math operations
-Sl	Specifies the line width for source listings
-Sp	Specifies the number of lines per page for source listings
-Ss	Specifies subtitles for source listings
-St	Specifies titles for source listings

Differences between Versions 5.0 and 4.0

- Tc** Tells the compiler that the following file is a C source file
- Zp** Packs structures on one-, two-, or four-byte boundaries

The following new pragmas have been added to the Microsoft C Compiler for Version 5.0 to control the specified features on a local basis:

Pragma	Effect
loop_opt	Turns loop optimizations on and off
pack	Specifies packing alignment for structures
intrinsic	Specifies which functions are compiled as intrinsic functions
function	Specifies which functions are compiled as standard function calls
same_seg	Tells the compiler to assume that specified variables are allocated in the same far data segment
alloc_text	Specifies modules to be grouped into a specified far text segment

Note that the existing **check_stack** pragma uses the following new format for specifying arguments:

```
#pragma check_stack([[on|off]])
```

Differences between Versions 4.0 and 3.0

Changes between Versions 4.0 and 3.0 fall into the same categories as those between Versions 5.0 and 4.0.

- Enhancements and additions to the compiler software to allow for more flexible programming, improved code generation, and increased support for the developing ANSI standard
- Changes in the language syntax

These features and the changes required to take advantage of them are discussed in the following sections.

Enhancements and Additions

Enhancements for Version 4.0 include the following:

- New options for `cc` and `xld`
- Improved code optimization
- New memory models (**compact** and **huge**)
- Source listings
- Numbered error messages
- Huge arrays, allowing a single array to be larger than 64K

These changes should have no effect on Version 3.0 source code.

For information on changes to the syntax of the `cc` command line, see the “Compiling with the `cc` Command” chapter of this guide.

Changes in the Language Syntax

Some Version 4.0 changes were made to the C-language syntax to make it conform more closely to the developing ANSI standard. Most of these changes do not affect source code written for the Version 3.0 compiler. The changes are summarized as follows:

Differences between Versions 4.0 and 3.0

- The `\a` escape sequence represents the bell (or alert) character in Version 4.0.

You can make your source code more portable by using `\a` instead of `\x7`. For more information, see the *C Language Reference*.

- The **signed** keyword was added to improve portability.

The **signed** keyword can be used to specify signed items. This keyword is particularly useful for declaring signed **char** types in programs compiled with the **-J** option. (**-J** changes the default mode for the **char** type to unsigned.) For more information on signed types, see the *C Language Reference*.

- The syntax was changed for making function calls with a variable number of arguments.

The following two declarations contrast the Version 3.0 form and the Version 4.0 form:

```
int func (int,);          /* Forward declaration in
                        ** Version 3.0 syntax
                        */

int func (int,...);      /* Forward declaration in
                        ** Version 4.0 syntax
                        */
```

This change was made to conform to changes in the ANSI standard for the C language. Both forms are supported in Version 4.0 of the C Compiler. Microsoft recommends the use of the Version 4.0 form in all programs.

- Prior to Version 4.0, the compiler allowed arbitrary strings of characters after a syntactically correct preprocessor command. To conform to the developing ANSI standard, this was disallowed in Version 4.0.

Beginning with Version 4.0, the following usage, for example, causes the compiler to generate a warning message:

```
#endif    Block ends here
```

In Versions 4.0 and later, such strings must be enclosed in comment delimiters, as in the following example:

```
#endif    /* Block ends here */
```


Differences between Versions 4.0 and 3.0

- Names of types defined with **typedef** are not keywords in Version 4.0, as they were in Version 3.0. In Version 4.0, these names are in the same naming class as names of functions and variables, and can be redefined in a nested block.

For more information, see the *C Language Reference*.

- Beginning with Version 4.0, the **#pragma** directive is supported.

A “pragma” is an instruction to the compiler. Its syntax is similar to the syntax of preprocessor directives, but its purpose is different. The syntax is as follows:

```
#pragma charstring
```

The only pragma instruction supported in the C Compiler, Version 4.0, is the **check_stack** pragma. This pragma is specific to System V, and is discussed in greater detail in the “Compiling with the cc Command” chapter of this guide.

- Hexadecimal and octal integer constants are handled differently in Version 4.0 than they are in Version 3.0.

For more information, see the *C Language Reference*.

- The extended keywords **fortran**, **pascal**, **cdecl**, **near**, and **huge** are enabled by default in Version 4.0. They can be disabled by giving the **-Za** option on the command line.
- Two new reserved words, **const** and **volatile**, were added but not implemented for Version 4.0.
- In Version 3.0, when a near pointer is converted to type **long int**, it is first converted to type **short int**, then to **long int**; as a result, in Version 3.0 the expression in the **if** statement evaluates as true in the following fragment:

```
char *ptr = NULL;
long i;

i = (long) ptr;
if (i == 0L) {
    .
    .
}
```

In Version 4.0, the conversion order of near pointers to long integers was changed so that it conforms to the order in which the

Differences between Versions 4.0 and 3.0

compiler does all other conversions that increase the length of a variable: first the size, then the mode. (For example, the compiler converts a variable with type **char** to type **unsigned long** by first converting it to **signed long**, then to **unsigned long**.) Because of this change, the preceding code now converts *ptr* to a far pointer by loading the appropriate segment register value, then changing that to a long integer. The expression following the *if* statement would most likely be false in Version 4.0, since the segment registers do not usually contain 0.

A

New Features for This Implementation of C

The following features were added to the C compiler for Version 4.0:

- Two new memory models: huge and compact
- The **huge**, **signed**, and **cdecl** keywords
- A pragma (**check_stack**) to control stack checking
- The **-J** option to change the default mode for the **char** type to unsigned
- The **-Gc** option to specify the alternative calling sequence and naming conventions used in Pascal and FORTRAN

These features are discussed in the “Working with Memory Models” chapter. In most cases, they will not affect existing Version 3.0 source code. However, you may be able to improve your existing programs by modifying them to take advantage of the new memory models or the **huge** keyword.



Appendix B

Writing Portable Programs

Introduction B-1

Program Portability B-3

Machine Hardware B-4

Byte Length B-4

Word Length B-4

Storage Alignment B-5

Byte Order in a Word B-6

Bit Fields B-7

Pointers B-7

Address Space B-9

Character Set B-9

Compiler Differences B-11

Signed/Unsigned char and Sign Extension B-11

Shift Operations B-11

Identifier Length B-12

Register Variables B-12

Type Conversion B-12

Functions with a Variable Number of Arguments B-14

Side Effects and Evaluation Order B-14

Environment Differences B-16

Portability of Data B-17

Type-Size Summary B-18

Byte-Ordering Summary B-20



Introduction

The standard definition of the C programming language leaves many details to be decided in specific implementations of the language. These unspecified features of the language detract from its portability and must be studied when attempting to write portable C code.

Most of the issues affecting C portability arise from differences either in target-machine hardware or in compilers. C was designed to compile efficient code for the target machine (initially a Digital Equipment Corporation PDP-11®), so many of the language features not precisely defined are those that reflect a particular machine's hardware characteristics.

This appendix highlights the various aspects of C that may not be portable across different machines and compilers. It also briefly discusses the portability of a C program in terms of its environment. The environment is determined by the system calls and library routines a program uses during execution, file path names it requires, and other items not guaranteed to be constant across different systems.

The C language has been implemented on many different computers with widely different hardware characteristics, from small eight-bit microprocessors to large mainframes. This appendix is concerned with the portability of C code in the MS-DOS, XENIX, and System V programming environments. This is a more restricted problem to consider, since all MS-DOS, System V, and XENIX operating systems to date run on hardware with the following basic characteristics:

- ASCII character set
- Eight-bit bytes
- Two-byte or four-byte integers
- Two's-complement arithmetic

These features are not formally defined for the language and may not be found in all implementations of C. However, the remainder of this appendix is devoted to those systems where these basic assumptions hold.

The C-language definition contains no specification of how input and output are performed. These specifications are left to system calls and library routines on individual systems. Within the System V and XENIX systems there are system calls and library routines that can be considered

Introduction

portable. This version of the C Compiler includes system calls and library routines that can be considered portable across System V, XENIX, and MS-DOS systems. The run-time library for the System V C Compiler for MS-DOS is composed primarily of System V and XENIX compatible routines. By restricting the use of XENIX and System V routines to those included in the MS-DOS library, the System V programmer can develop MS-DOS programs in the System V environment; C programs written on MS-DOS are easily portable to XENIX or System V.

Program Portability

A program is “portable” if it can be compiled and run successfully on different machines without alteration. There are many ways to write portable programs. One way is to avoid using inherently nonportable language features. Another is to isolate any nonportable interactions with the environment, such as I/O to nonstandard devices. For example, programs should avoid hard-coded path names unless a path name is common to all systems.

Files required at compile time (such as include files) may also introduce nonportability if the path names used are not the same on all machines. In some cases, include files containing machine-specific definitions can be used to make the source code itself portable.

Machine Hardware

Differences in the hardware of the various target machines and differences in the corresponding C compilers cause the greatest number of portability problems. This section lists problems commonly encountered.

Byte Length

By definition, the **char** data type in C must be large enough to hold as positive integers all members of a machine's character set. For the machines described in this appendix, the **char** size is an eight-bit byte.

Word Length

The size of the basic data types for a given implementation are not formally defined in the C language. Therefore, they often follow the most natural size for the underlying machine. It is safe to assume that **short** is no longer than **long**. Beyond that, no assumptions are portable. For example, on some machines **short** is the same length as **int**, whereas on others **long** is the same length as **int**.

Two areas where different **int** sizes affect program portability are the following:

1. Array indexing. For very large arrays, a variable of type **int** may not be long enough to store the indices of the highest-numbered array elements.
2. Pointer subtraction. On some machines, an **int** variable may not be long enough to store the results of pointer subtraction. See the section on "Pointers" in this appendix for more information about this problem.

Programs that need to assume the size of a particular data type should avoid hard-coded constants where possible. Such information can usually be written in a fairly portable way. For example, the maximum positive integer (on a two's-complement machine) can be obtained with the following directive:

```
#define MAXPOS ((int)((unsigned)-1) >> 1)
```

This is preferable to the following code:

```

#ifdef PDP11
#define MAXPOS 32767
#else
.
.
.
#endif

```

To find the number of bytes in an **int**, use **sizeof(int)** rather than 2, 4, or some other nonportable constant.

Storage Alignment

The C language defines no particular layout for storage of data items relative to each other. The layout for storage of structure elements, or unions within the structure or union, is also left undefined by the language.

Some processors require that data types longer than one byte be aligned on even-byte address boundaries. Others, such as the 8086/8088, have no such hardware restriction. However, even with these machines, most compilers generate code that aligns words, structures, arrays, and long words on even addresses or on even long-word addresses. Therefore, the following code sequence may give different results, depending on specific alignment requirements on different machines:

```

struct s_tag {
    char c;
    int i;
};
printf("%d\n", sizeof(struct s_tag));

```

This variation in data storage has two major implications: data accessed as nonprimitive data types are not portable, and code that makes assumptions about the layout on a particular machine is not portable.

Therefore, unions containing structures are nonportable if the union is used to access the same data in different ways. Unions are only likely to be portable if they are used exclusively to store different data in the same space at different times. For example, if the following union were used to obtain four bytes from a long word, the code would not be portable to some machines:

Machine Hardware

```
union {
    char c[4];
    long lw;
} u;
```

The **sizeof** operator should always be used when reading and writing structures, as follows:

```
struct s_tag st;
.
.
.
write(fd, &st, sizeof(st));
```

Using the **sizeof** operator ensures portability of the source code, but does not produce a portable data file. Portability of data is discussed in the “Portability of Data” section in this appendix.

Byte Order in a Word

The variation in byte order in a word affects the portability of data more than the portability of source code. However, any program that makes use of knowledge of the internal byte order in a word is not portable. For example, on some XENIX or System V systems there is an include file, **misc.h**, that contains the following structure declaration:

```
/*
 * structure to access an
 * integer in bytes
 */
struct {
    char  lobyte;
    char  hibyte;
};
```

With certain less-restrictive compilers, this declaration could be used to access the high- and low-order bytes of an integer separately and in a completely nonportable way. The correct way to do this is to use mask and shift operations to extract the required byte, as shown in the following example:

```
#define LOBYTE(i) (i & 0xff)
#define HIBYTE(i) ((i >> 8) & 0xff)
```

These definitions provide a portable way to extract the least-significant and the next-least-significant bytes of an integer. Since the **int** type can be either two or four bytes, depending on the machine, even these definitions do not provide a completely portable way to access the bytes of an **int**.

One result of the byte-ordering problem is that the following code sequence will not always perform as intended:

```
int c = 0;
read(fd, &c, 1);
```

On machines where the low-order byte is stored first, the value of *c* is the byte value read. On other machines, the byte is read into some byte other than the low-order one, so the value of *c* is different.

B

Bit Fields

Bit fields are not implemented in all C compilers. The C Compiler implements bit fields and allows them to have any length up to the size of a **long**. However, in many implementations no bit field may be larger than an **int**, and no bit field can overlap an **int** boundary. If necessary, the compiler will leave gaps and move to the next **int** boundary. To ensure portability no individual field should exceed 16 bits.

The C language makes no guarantees about whether bit fields are assigned left to right or right to left. Therefore, although bit fields may be useful for storing flags and other small data items, their use in unions to dissect bits from other data is definitely nonportable.

Pointers

The C language is fairly generous in allowing manipulation of pointers, to the extent that most compilers do not generate warnings for nonportable pointer operations. A common nonportable use of pointers is the use of casts to assign one pointer to another pointer of a different data type. This practice usually makes some assumption about the internal byte ordering and layout of the data type, and is therefore nonportable. In the following code, the byte order in the array *c* is not portable:

```
char c[4];
long *lp;

lp = (long *)&c[0];
*lp = 0x12345678L;
```

Code like this is usually unnecessary or invalid. It is acceptable, however, when the **malloc** function is used to allocate space for variables that do not have **char** type. The routine is declared as type **char ***, and the return value is cast to the type to be stored in the allocated memory. If this type is not **char ***, then a compiler may issue a warning concerning illegal

Machine Hardware

type conversion. In addition, the **malloc** function is designed always to return a starting address suitable for storing all types of data. A compiler may not know this, so it may give an additional warning about possible data-alignment problems. In the following example, **malloc** is used to obtain memory for an array of 50 integers:

```
extern void *malloc( );
int *ip;

ip = (int *)malloc(50 * sizeof(int));
```

This example will elicit a warning message from some compilers.

The *C Language Reference* states that a pointer can be assigned (or cast) to an integer large enough to hold it. Note that the size of the **int** type depends on the given machine and implementation. This type is **long** on some machines and **short** on others. The size may also be modified by **near** and **far** declarations. In general, do not assume that the following statement is always true:

```
sizeof(char *) == sizeof(int)
```

For example, the following construction is nonportable, assuming that the function identifier *func* is not previously declared:

```
int p;
p = (char *)func( );
```

This example assumes that a **char** pointer has the same length as an **int**.

Another consequence of different-sized **int** types on different machines is that pointer subtraction may not give the expected results. As an example of this case, subtracting pointers to the beginning and end of a very large array may give a result that is too large to store in an **int** variable, as shown in the following example:

```
int arr[20000], *b = arr, *e = &arr[20000];
int diff;
diff = e - b; /* result too large to store in
              int variable diff */
```

To correct this problem, coerce the result of the pointer subtraction **long** type, then assign the result to a variable of **unsigned int** type, as shown in the following example:

```
unsigned int udiff;
udiff = (long) ((int huge *)e - (int huge *)b);
```

In most implementations, the null pointer value **NULL** is defined to be the **int** value 0. The length of the 0 value can lead to problems for functions that expect pointer arguments longer than an **int**. For portable code, always use the following form to pass a **NULL** value of the correct size:

```
func( (char *)NULL );
```

Address Space

The address space available to a program varies considerably from system to system. Some small processors allow only 64K for program text and data combined. Others allow up to 64K of data and 64K of program text. Larger machines may allow considerably more text and possibly more data as well.

Large programs, or programs that require large data areas, may have portability problems on small machines.

Character Set

The C language does not require the use of the ASCII character set. In fact, the only character-set requirements are that all characters must fit in the **char** data type, and all characters must have positive values.

In the ASCII character set, all characters have values between 0 and 127 and therefore can be represented in seven bits. On an eight-bits-per-byte machine they are all positive, regardless of whether **char** is treated as signed or unsigned.

A set of character-classification macros is included as part of the run-time library for the C Compiler. These macros should be used for most tests on character quantities. The macros are defined in the include file **ctype.h**, and are described in the *C Library Guide*.

The character-classification macros provide insulation from the internal structure of the character set. In addition, the names of the macros are often more meaningful than the equivalent line of code. Compare the following two lines:

```
if(isupper(c))  
    if((c >= 'A') && (c <= 'Z'))
```

Machine Hardware

With some of the other macros, such as **isxdigit** to test for a hexadecimal digit, the advantage is even greater. Also, the internal implementation of the macros makes them more efficient than an explicit test with an if statement.

Compiler Differences

There are a number of C compilers running under various operating systems. The main areas of differences between compilers are outlined in this section.

B

Signed/Unsigned char and Sign Extension

The current state of the signed versus unsigned **char** problem is best described as unsatisfactory. The sign-extension problem is a serious barrier to writing portable C, and the best solution at present is to write defensive code that does not rely on particular implementation features.

Shift Operations

The left-shift operator (`<<`) shifts its operand a number of bits left, filling vacated bits with zeros. This is called a logical shift. When the right-shift operator (`>>`) is applied to an unsigned quantity, it performs a logical-shift operation; when it is applied to a signed quantity, the vacated bits may be filled with zeros (logical shift) or with sign bits (arithmetic shift). The decision is implementation dependent, and code that assumes a particular implementation is nonportable.

With compilers that use arithmetic right shift, it is necessary to shift and mask the appropriate number of high-order bits to avoid sign extension, as follows:

```
char c;  
c = (c >> 3) & 0x1f;
```

You can also avoid sign extension by using the divide operator (`/`) as follows:

```
char c;  
c = c / 8;
```


Compiler Differences

Identifier Length

The use of long symbols and identifier names will cause portability problems with some compilers. To avoid these problems, a program should keep the following symbols as short as possible:

- C preprocessor symbols
- C local symbols
- C external symbols

Some loaders also place restrictions on the number of unique characters in C external symbols. Symbols unique in the first six characters are unique to most C-language processors.

In some C implementations, the case of letters in identifiers is not significant.

Register Variables

The number and type of register variables in a function depend on the machine hardware and the compiler. Excess and invalid register declarations are treated as nonregister declarations and should not cause a portability problem. On an 8086, 8088 or 80286 processor, up to two register declarations are significant, and they must be applied to types of size **int** or smaller. On the 80386 processor, three register declarations are significant.

Since the compiler ignores excess variables of **register** type, the most important **register**-type variables should be declared first. In this way, register variables that the compiler ignores will be those that are the least important.

Type Conversion

The C language has some rules for implicit type conversion; it also allows explicit type conversions by type casting. The most common portability problem in implicit type conversion is unexpected sign extension. This is a potential problem whenever something of type **char** is compared with an **int**.

Compiler Differences

The following example will never evaluate true on a machine that sign-extends **char** types but treats hexadecimal numbers as unsigned:

```
char c;

if(c == 0x80) {
    .
    .
    .
}
```

The following construction is also nonportable:

```
char c;
unsigned int u;

if (u == (unsigned)c) {
    .
    .
    .
}
```

Two problems can arise in the preceding example:

1. The **char** type may be considered either signed or unsigned, depending on the implementation.
2. For implementations that consider the **char** type to be signed, two different methods of carrying out the conversion are possible: the **char** value may be sign extended to **int** type first, then converted to **unsigned** type; or the **char** type may be converted to an unsigned type of the same size, then zero extended to **int** length.

The only safe comparison between **char** type and **int** is the following:

```
int c;

if(c == 'x') {
    .
    .
    .
}
```

This comparison is reliable because C guarantees all character constants to be positive.

Compiler Differences

Type conversion also occurs when arguments are passed to functions. Types **char** and **short** become **int**. Extending the **char** type can produce unexpected results. For example, the following program yields a result of -128 on some machines:

```
char c = 128;
printf("%d\n", c);
```

The unexpected negative value is produced because *c* is converted to **int** when it is passed to the **printf** function. The function itself has no knowledge of the original type of the argument and is expecting an **int**. The correct way to handle this situation is to code defensively and allow for the possibility of sign extension, as in the following example:

```
char c = 128;
printf("%d\n", c & 0xff);
```

Functions with a Variable Number of Arguments

Functions with a variable number of arguments present a particular portability problem if the type of the arguments is also variable. In such cases the code is dependent on the size of various data types. For portability, these cases should be avoided.

Side Effects and Evaluation Order

The C language makes few guarantees about the order of evaluation of operands in an expression or arguments to a function call. Therefore, the following statement is not portable:

```
func(i++, i++);
```

Even the following statement is unwise if **func** is ever likely to be replaced by a macro, since the macro may use *i* more than once:

```
func(i++);
```

Certain System V- and XENIX-compatible macros commonly appear in user programs; some of these use their argument only once, and therefore can safely be called with a side-effect argument. To determine whether a macro handles side effects correctly, examine the code for that macro to see whether or not the argument is evaluated more than once.

Operands to the following operators are guaranteed to be evaluated left to right:

```
,      &&   ||   ?:
```

Compiler Differences

Note that the comma operator here is a separator for two C statements. A list of items separated by commas in a declaration list is not guaranteed to be processed left to right. Therefore, the following declaration on an 8086 or 8088 processor, where only two register variables may be declared, could give any two of the four variables **register** type, depending on the compiler:

```
register int a, b, c, d;
```

To give register storage to the most important variables, use separate declaration statements and declare the most important variables first. The order of processing of individual declaration statements is guaranteed to be sequential in the following statements:

```
register int a;  
register int b;  
register int c;  
register int d;
```

B

Environment Differences

Most programs make system calls and use library routines for various services. This section indicates some of those routines that are not always portable and those that particularly aid portability.

System calls specific to an operating system are not portable if they are not present on all other operating-system implementations of C. Most of the system calls defined in the UNIX System V run-time library are compatible with DOS system calls and are therefore portable to a DOS environment.

Any program is nonportable that contains hard-coded path names to files or directories, or that contains user identifier numbers, log-in names, terminal lines, or other system-dependent parameters. These types of constants should be in header files, passed as command-line arguments, or obtained from the environment.

Note that the members of the **printf** and **scanf** families of functions, including **fprintf**, **fscanf**, **printf**, **sprintf**, **scanf**, **vfprintf**, **vprintf**, **vsprintf**, and **sscanf**, have evolved in several ways, and some features are not completely portable. Some of the format-conversion characters have changed their meanings, in particular those relating to uppercase and lowercase in the output of hexadecimal numbers and the specification of **long** integers on 16-bit word machines. The C specifications for these routines are given in the *C Library Guide*.

Users should be wary of porting object files that reference the **setjmp** or **longjmp** functions from System V or XENIX to MS-DOS, unless these object files were compiled with the **-dos** option. The MS-DOS versions of these functions use a larger buffer size and may cause memory to be overwritten. Such object files can be ported from MS-DOS to System V without problems, and the corresponding source files can be ported in either direction.

Portability of Data

Data files are almost always nonportable across different central-processing-unit (CPU) architectures. As mentioned above, structures, unions, and arrays have varying internal layout and padding requirements on different machines. In addition, byte ordering within words and actual word length may differ.

The only way to achieve data-file portability is to write and read data files as one-dimensional character arrays. This procedure prevents alignment and padding problems if the data are written and read as characters, and interpreted that way. Thus ASCII text files can usually be moved between different machine types without significant problems.

Type-Size Summary

Table B.1 summarizes the sizes of the various data types as defined in the C Compiler, Version 5.1.

Table B.1
C Type Sizes

Type Name (Alternate Names)	Storage	Range of Values
char (signed char)	1 byte	-128 to 127
int (signed) (signed int)	Implementation dependent (2 bytes in UNIX C 5.1)	(-32,768 to 32,767 for UNIX C Version 5.1) (-2,147,483,648 to 2,147,483,647 for 386)
short (short int) (signed short) (signed short int)	2 bytes	-32,768 to 32,767
long (long int) (signed long) (signed long int)	4 bytes	-2,147,483,648 to 2,147,483,647
unsigned char	1 byte	0 to 255
unsigned (unsigned int)	Implementation dependent (2 bytes in C 5.1)	(0 to 65,535 for C 5.1) (0 to 4,294,967,295 for 386)
unsigned short (unsigned short int)	2 bytes	0 to 65,535

Type-Size Summary

Table B.1
C Type Sizes

Type Name (Alternate Names)	Storage	Range of Values
unsigned long (unsigned long int)	4 bytes	0 to 4,294,967,295
enum	Implementation dependent (2 bytes in C 5.1)	(0 to 65,535 for C 5.1) (0 to 4,294,967,295 for 386)
float	4 bytes	Approximately 3.4E-38 to 3.4E+38 (7-digit precision)
double	8 bytes	Approximately 1.7E-308 to 1.7E+308 (15-digit precision)
long double	Implementation dependent (8 bytes in C 5.1)	Approximately 1.7E-308 to 1.7E+308 (15-digit precision)

- 1 Any type size modified by the **unsigned** keyword can be modified by the **signed** keyword instead. The **signed** keyword is useful if the **-J** option has been used to change the default sign of the **char** type.

B

Byte-Ordering Summary

Tables B.2 and B.3 summarize byte ordering for **short** and **long** types, respectively. The following conventions are used in these tables:

1. The lowest physically addressed byte of the data item is **a0**; **a1** has the byte address **a0 + 1**, and so on.
2. The least-significant byte of the data item is **b0**; **b1** is the next least significant, and so on.

Since byte ordering is machine specific, any program that actually makes use of the following information is guaranteed to be nonportable:

Table B.2
Byte Ordering for Short Types

CPU	Byte Order
8086	b0 b1
80286	b0 b1
PDP-11®	b0 b1
VAX-11®	b0 b1
M68000	b1 b0
Z8000®	b1 b0

Byte-Ordering Summary

Table B.3
Byte Ordering for Long Types

CPU	Byte Order
8086	b0 b1 b2 b3
80286	b0 b1 b2 b3
PDP-11®	b2 b3 b0 b1
VAX-11®	b0 b1 b2 b3
M68000	b3 b2 b1 b0
Z8000®	b3 b2 b1 b0

B



Appendix C

Writing Programs for Read-Only Memory

Introduction C-1

System V Dependent Library Routines C-2



Introduction

This appendix presents information for developers who will be downloading code written with the C Compiler into read-only memory (ROM). Code of this type is more commonly known as “ROMable” code. Information is given about the run-time library routines that directly interface with System V.

System V Dependent Library Routines

Because ROMable programs are often run outside a System V environment, they cannot include calls to run-time library routines that perform their operations through calls to System V functions. Table C.1 lists the library routines that call System V functions.

Table C.1
System V Dependent Library Routines

abort	_exit	fwrite	read
access	ifclose	getch	rmdir
chdir	fgetc	getcwd	scanf
chmod	fgetchar	getpid	sopen
chsize	fgets	gets	sprintf
close	flush	getw	sscanf
creat	fopen	labs	stat
dup	fprintf	localtime	system
dup2	fputc	locking	tell
eof	fputchar	lseek	time
execl	fputs	mkdir	tmpfile
execle	fread	mktemp	unlink
execlp	freopen	open	utime
execlp	fscanf	perror	vfprintf
execv	fseek	printf	vprintf
execve	fstat	putch	vsprintf
execvp	ftell	puts	write
execvpe	ftime	putw	

A program containing calls to any of these routines cannot run in a non-System V environment unless you do one of the following:

- Write replacements for these System V-dependent routines as needed.
- Edit the program to remove the calls to the listed routines.
- Obtain the library source files from System V and edit them so that they do not include System V function calls, and write functional equivalents of the System V functions that can be called from your program.

System V Dependent Library Routines

Note that certain functions that are not listed above may call System V functions indirectly: that is, they may be part of a series of nested calls that call routines in the list.



Appendix D

C Error Messages and Exit Codes

Introduction D-1

Command-Line Error Messages D-2
 Command-Line Fatal-Error Messages D-2
 Command-Line Error Messages D-2
 Command-Line Warning Messages D-4

Compiler Error Messages D-7
 Fatal-Error Messages D-8
 Compilation-Error Messages D-14
 Warning Messages D-29
 Compiler Limits D-39

Compiler Exit Codes D-41



Introduction

This appendix lists error messages you may encounter as you develop a program, and gives a brief description of actions you can take to correct the errors. It also describes the exit codes returned by the compiler.

Command-Line Error Messages

Messages that indicate errors on the command line used to invoke the compiler have one of the following formats:

command line fatal error D1xxx: messagetext (fatal error)
command line error D2xxx: messagetext (error)
command line warning D4xxx: messagetext (warning error)

If possible, the compiler continues operation, printing a warning message. In some cases, command-line errors are fatal and the compiler terminates processing.

Command-Line Fatal-Error Messages

The following messages identify fatal errors. The compiler driver cannot recover from a fatal error; it terminates after printing the error message.

D1000 UNKNOWN COMMAND LINE FATAL ERROR
The compiler detected an unknown fatal-error condition.

D1001 could not execute '*filename*'
The compiler could not find the given file in the current working directory or any of the other directories named in the **PATH** variable.

D1002 too many open files, cannot redirect '*filename*'
No more file descriptors were available to redirect the output of the **-P** option to a file.

Command-Line Error Messages

When the compiler driver encounters any of the errors listed in this section, it continues compiling the program (if possible) and outputs additional error messages. However, no object file is produced.

D2000 UNKNOWN COMMAND LINE ERROR
The compiler detected an unknown error condition.

Command-Line Error Messages

D2001 too many symbols predefined with -D
Too many symbolic constants were defined using the **-D** option on the command line.

The limit on command-line definitions is normally 16; you can use the **-U** or **-u** option to increase the limit to 200.

D2002 a previously defined model specification has been overridden
Two different memory models were specified; the model specified later on the command line was used.

D2003 missing source file name
You did not give the name of the source file to be compiled.

D2007 bad *option* flag, would overwrite '*string1*' with '*string2*'
The specified option was given more than once, with conflicting arguments *string1* and *string2*.

D2008 too many *option* flags, '*string*'
Too many letters were given with the specified option (for example, with the **-O** option).

D2009 unknown *option character* in '*optionstring*'
One of the letters in the given option was not recognized.

D2012 too many linker flags on command line
You tried to pass more than 128 separate options and object files to the linker.

D2013 incomplete model specification
Not enough characters were given for the **-Astring** option. The option requires all three letters (to specify the data-pointer size, code-pointer size, and segment setup).

D2014 **-ND** not allowed with **-Ad**
You cannot rename the default data segment unless you give the **-Auxx** option (**SS != DS**, load **DS**) on the command line.

D2016 **-Gw** and **-ND name** are incompatible
You tried to rename the default data segment to the given name when you specified the **-Gw** option.

Renaming the default data segment is illegal in this case because the **-Gw** option requires the **-Auxx option**.

Command-Line Error Messages

D2017 `-Gw` and `-Au` flags are incompatible
You tried to specify the `-Auxx` option (`SS != DS`, load `DS`) with the `-Gw` option.

Specifying `-Auxx` with `-Gw` is illegal because the `-Gw` option requires the `-Auxx` option.

D2019 cannot overwrite the source file, *'name'*
You specified the source file as an output-file name.

The compiler does not allow the source file to be overwritten by one of the compiler output files.

D2020 `-Gc` option requires extended keywords to be enabled (`-Ze`)
The `-Gc` option and the `-Za` option were specified on the same command line.

The `-Gc` option requires the extended keyword `cdecl` to be enabled if library functions are to be accessible.

D2021 invalid numerical argument *'string'*
A non-numerical string was specified following an option that required a numerical argument.

D2023 invalid model specification - small model only

D2024 : `-Gm` and `-ND` are incompatible options
You compiled with both the `-Gm` and `-ND` compiler options. These options are incompatible because `-Gm` indicates that string literals and near const data items should be allocated in the `CONST` segment, while the `-ND` option attempts to allocate the same items in a different, named segment.

Command-Line Warning Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking.

D4000 UNKNOWN COMMAND LINE WARNING
An unknown fatal condition has been detected by the compiler.

D4001 listing has precedence over assembly output
Two different listing options were chosen; the assembly listing is not created.

Command-Line Error Messages

D4002 ignoring unknown flag *'string'*

One of the options given on the command line was not recognized and is ignored.

D4003 80186/286 selected over 8086 for code generation

Both the **-G0** option and either the **-G1** or **-G2** option were given; **-G1** or **-G2** takes precedence.

D4004 optimizing for time over space

This message confirms that the **-Ot** option is used for optimizing.

D4006 only one of -P/-E/-EP allowed, -P selected

Only one preprocessor output option can be specified at one time.

D4007 -C ignored (must also specify -P or -E or -EP)

The **-C** option must be used in conjunction with one of the preprocessor output flags, **-E**, **-EP**, or **-P**.

D4008 non-standard model -- defaulting to small model libraries

A nonstandard memory model was specified with the option. The library search records in the object model were set to use the small-model libraries.

D4009 threshold only for far/huge data, ignored

The **-Gt** option cannot be used in memory models that have near data pointers. It can be used only in compact, large, and huge models.

D4011 preprocessing overrides source listing

Only a preprocessor listing was generated, since the compiler cannot generate both a source listing and a preprocessor listing at the same time.

D4012 function declarations override source listing

The compiler cannot generate 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.

D

Command-Line Error Messages

D4014 invalid value *number* for '*string*'. Default *number* is used

An invalid value was given in a context where a particular numerical value was expected.

D4017 conflicting stack checking options - stack checking disabled

Both the **-Ge** and the **-Gs** flags are given in one compile command (**-Ge** enables stack checking, **-Gs** disables it).

Compiler Error Messages

The error messages produced by the C compiler fall into three categories:

1. Fatal-error messages
2. Compilation-error messages
3. Warning messages

The messages for each category are listed below in numerical order, with a brief explanation of each error. To look up an error message, first determine the message category, then find the error number. All messages give the file name and line number where the error occurs.

D

Fatal-Error Messages

Fatal-error messages indicate a severe problem, one that prevents the compiler from processing your program any further. These messages have the following format:

filename(line) : fatal error C1xxx: messagetext

After the compiler displays a fatal-error message, it terminates without producing an object file or checking for further errors.

Compilation-Error Messages

Compilation-error messages identify actual program errors. These messages appear in the following format:

filename(line) : error C2xxx: messagetext

The compiler does not produce an object file for a source file that has compilation errors in the program. When the compiler encounters such errors, it attempts to recover from the error. If possible, it continues to process the source file and produce error messages. If errors are too numerous or too severe, the compiler stops processing.

Compiler Error Messages

Warning Messages

Warning messages are informational only; they do not prevent compilation and linking. These messages appear in the following format:

filename(line) : warning C4xx: messagetext

You can use the **-W** option to control the level of warnings that the compiler generates. This option is described in the “Compiling with the cc Command” chapter of this guide.

Fatal-Error Messages

The following messages identify fatal errors. The compiler cannot recover from a fatal error; it terminates after printing the error message.

C1000 UNKNOWN FATAL ERROR

An unknown error condition has been detected by the compiler.

C1001 Internal Compiler Error

The compiler detected an internal inconsistency.

Note that the file name refers to an internal compiler file, *not* your source file.

C1002 out of heap space

The compiler has run out of dynamic memory space. This usually means that your program has many symbols and/or complex expressions.

To correct the problem, divide the file into several smaller source files, or break expressions into subexpressions.

C1003 error count exceeds *n*; stopping compilation

Errors in the program were too numerous or too severe to allow recovery, and the compiler must terminate.

Compiler Error Messages

C1004 unexpected EOF

This message appears when you have insufficient disk space for the compiler to create the temporary files it needs. The space required is approximately two times the size of the source file. This message can also occur when a comment does not have a closing delimiter (**/*), or when an **#if** directive occurs without a corresponding closing **#endif** directive.

C1005 string too big for buffer

A string in a compiler intermediate file overflowed a buffer.

C1006 write error on compiler intermediate file

The compiler was unable to create the intermediate files used in the compilation process.

The following conditions commonly cause this error:

1. A system file or the inode table is full at time of compilation
2. Not enough space on a device containing a compiler intermediate file

C1007 unrecognized flag '*string*' in '*option*'

The *string* in the command-line *option* was not a valid option.

C1009 compiler limit possibly a recursively defined macro

The expansion of a macro exceeds the available space.

Check to see if the macro is recursively defined, or if the expanded text is too large.

C1010 compiler limit : macro expansion too big

The expansion of a macro exceeds the available space.

C1012 bad parenthesis nesting - missing '*character*'

The parentheses in a preprocessor directive were not matched; *character* is either a left or right parenthesis.

C1013 cannot open source file '*filename*'

The given file either did not exist, could not be opened, or was not found. Make sure your environment settings are valid and that you have given the correct path name for the file.

Compiler Error Messages

C1014 too many include files
Nesting of **#include** directives exceeds 10 levels.

C1016 **#if[n]def** expected an identifier
You must specify an identifier with the **#ifdef** and **#ifndef** directives.

C1017 invalid integer constant expression
The expression in an **#if** directive must evaluate to a constant.

C1018 unexpected **'#elif'**
The **#elif** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

C1019 unexpected **'#else'**
The **#else** directive is legal only when it appears within an **#if**, **#ifdef**, or **#ifndef** directive.

C1020 unexpected **'#endif'**
An **#endif** directive appears without a matching **#if**, **#ifdef**, or **#ifndef** directive.

C1021 bad preprocessor command **'string'**
The characters following the number sign (**#**) do not form a valid preprocessor directive.

C1022 expected **'#endif'**
An **#if**, **#ifdef**, or **#ifndef** directive was not terminated with an **#endif** directive.

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, try to simplify your program.

C1027 DGROUP data allocation exceeds 64K
More than 64K of variables was allocated to the default data segment.

For compact-, medium-, large-, or huge-model programs, use the **-Gt** option to move items into separate segments.

Compiler Error Messages

C1032 cannot open object listing file *'filename'*
One of the following statements about the file name or path name given (*filename*) is true:

1. The given name is not valid.
2. The file with the given name cannot be opened for lack of space.
3. A read-only file with the given name already exists.

C1033 cannot open assembly-language output file *'filename'*
One of the conditions listed under error message C1032 prevents the given file from being opened.

C1034 cannot open source file *'filename'*
One of the conditions listed under error message C1032 prevents the given file from being opened.

C1035 expression too complex, please simplify
The compiler cannot generate the 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 error message C1032 prevents the given file from being opened.

C1037 cannot open object file *'filename'*
One of the conditions listed under error message C1032 prevents the given file from being opened.

C1039 unrecoverable heap overflow in Pass 3
The post-optimizer compiler pass overflowed the heap and could not continue.

Try recompiling with the **-Od** option (see “Compiling with the cc Command”) or try rewriting the function containing the line that caused the error.

C1040 unexpected EOF in source file *'filename'*
The compiler detected an unexpected end-of-file condition while creating a source listing or mingled source/object listing.

This error probably occurred because the source file was edited during compilation.

Compiler Error Messages

C1041 cannot open compiler intermediate file - no more files

The compiler could not create intermediate files used in the compilation process because no more file descriptors were available.

C1042 cannot open compiler intermediate file - no such file or directory

The compiler could not create intermediate files used in the compilation process because the /tmp directory did not exist.

C1043 cannot open compiler intermediate file

The compiler could not create intermediate files used in the compilation process. The exact reason is unknown.

C1044 out of disk space for compiler intermediate file

The compiler could not create intermediate files used in the compilation process because no more space was available.

To correct the problem, make more space available on the disk and recompile.

C1045 floating point overflow

The compiler generated a floating-point exception while doing constant arithmetic on floating-point items at compile time, as in the following example:

```
float fp_val = 1.0e100;
```

In this example, the double-precision constant *1.0e100* exceeds the maximum allowable value for a floating-point data item.

C1047 too many *option* flags, '*string*'

The *option* appeared too many times. The *string* contains the occurrence of the option that caused the error.

C1048 Unknown option '*character*' in '*optionstring*'

The *character* was not a valid letter for *optionstring*.

C1049 invalid numerical argument '*string*'

A numerical argument was expected instead of *string*.

C1050 code segment '*segmentname*' too large

A code segment grew to within 36 bytes of 64K during compilation.

A 36-byte pad is used because of a bug in some 80286 chips that can cause programs to exhibit strange behavior when, among other conditions, the size of a code segment is within 36 bytes of 64K.

Compiler Error Messages

C1052 too many `#if/#ifdef`'s

You have exceeded the maximum nesting level for `#if/#ifdef` directives.

C1053 compiler limit : struct/union nesting

Structure and union definitions were nested to more than 10 levels.

C1054 compiler limit : initializers too deeply nested

The compiler limit on nesting of initializers was exceeded. The limit ranges from 10 to 15 levels, depending on the combination of types being initialized.

To correct this problem, simplify the data type being initialized to reduce the levels of nesting, or assign initial values in separate statements after the declaration.

C1056 compiler limit : out of macro expansion space

The compiler has overflowed an internal buffer during the expansion of a macro; reduce the complexity of the macro.

C1057 unexpected EOF in macro expansion;

(missing `'`'?)

The compiler has encountered the end of the source file while gathering the arguments of a macro invocation. Usually this is the result of a missing closing parenthesis (`)` on the macro invocation.

C1059 out of near heap space

The compiler has run out of storage for items that it stores in the "near" (default data segment) heap. This usually means that your program has too many symbols or complex expressions. To correct the problem, divide the file into several smaller source files, or break expressions into smaller subexpressions.

C1060 out of far heap space

The compiler has run out of storage for items that it stores in the "far" heap. Usually this is the result of too many symbols in the symbol table.

C1064 : too many text segments

You defined more than 10 distinct text segments with the `alloc_text` pragma.

Compiler Error Messages

Compilation-Error Messages

The messages listed below indicate that your program has errors. When the compiler encounters any of the errors listed in this section, it continues parsing the program (if possible) and outputs additional error messages. However, no object file is produced.

C2000 UNKNOWN ERROR

The compiler detected an unknown error condition.

C2001 newline in constant

A new-line character in a character or string constant was not in the correct escape-sequence format (`\n`).

C2002 out of macro actual parameter space

Arguments to preprocessor macros exceeded 256 bytes.

C2003 expected 'defined id'

The identifier to be checked in an `#if` directive was not enclosed in parentheses.

C2004 expected 'defined(id)'

An `#if` directive caused a syntax error.

C2005 `#line` expected a line number, found 'token'

A `#line` directive lacked the required line-number specification.

C2006 `#include` expected a file name, found 'token'

An `#include` directive lacked the required file-name specification.

C2007 `#define` syntax

A `#define` directive caused a syntax error.

C2008 '*character*' : unexpected in macro definition

The given character was used incorrectly in a macro definition.

C2009 reuse of macro formal '*identifier*'

The given identifier was used twice in the formal-parameter list of a macro definition.

C2010 '*character*' : unexpected in formal list

The given character was used incorrectly in the formal-parameter list of a macro definition.

C2011 '*identifier*' : definition too big

The given macro definitions exceeded 256 bytes.

Compiler Error Messages

C2012 missing name following '<'

An **#include** directive lacked the required file-name specification.

C2013 missing '>'

The closing angle bracket (>) was missing from an **#include** directive.

C2014 preprocessor command must start as first non whitespace

Non-white-space characters appear before the number sign (#) of a preprocessor directive on the same line.

C2015 too many chars in constant

A character constant containing more than one character or escape sequence was used.

C2016 no closing single quote

A character constant was not enclosed in single quotation marks.

C2017 illegal escape sequence

The character or characters after the escape character (\) did not form a valid escape sequence.

C2018 unknown character '0xcharacter'

The given hexadecimal number does not correspond to a character.

C2019 expected preprocessor command, found 'character'

The given character followed a number sign (#), but it was not the first letter of a preprocessor directive.

C2020 bad octal number 'character'

The given character was not a valid octal digit.

C2021 expected exponent value, not 'character'

The given character was used as the exponent of a floating-point constant but was not a valid number.

C2022 'number' : too big for char

The *number* was too large to be represented as a character.

C2023 divide by 0

The second operand in a division operation (/) evaluated to zero, giving undefined results.

C2024 mod by 0

The second operand in a remainder operation (%) evaluated to zero, giving undefined results.

Compiler Error Messages

C2025 '*identifier*' : enum/struct/union type redefinition
The given identifier had already been used for an enumeration, structure, or union tag.

C2026 '*identifier*' : member of enum redefinition
The given identifier had already been used for an enumeration constant, either within the same enumeration type or within another enumeration type with the same visibility.

C2028 struct/union member needs to be inside a struct/union
Structure and union members must be declared within the structure or union.

This error may be caused by an enumeration declaration that contains a declaration of a structure member, as in the following example:

```
enum a {
    january,
    february,
    int march; /* structure declaration:
                ** illegal
                */
};
```

C2029 '*identifier*' : bit-fields allowed only in structs
Only structure types may contain bit fields.

C2030 '*identifier*' : struct/union member redefinition
The *identifier* was used for more than one member of the same structure or union.

C2031 '*identifier*' : function cannot be
struct/union member
The given function was declared to be a member of a structure.

To correct this error, use a pointer to the function instead.

C2032 '*identifier*' : base type with near/far/huge
not allowed
The given structure or union member was declared with the **near**, **far**, or **huge** keyword.

C2033 '*identifier*' : bit-field cannot have indirection
The given bit field was declared as a pointer (*), which is not allowed.

Compiler Error Messages

C2034 *'identifier'* : bit-field type too small for number of bits

The number of bits specified in the bit-field declaration exceeded the number of bits in the given base type.

C2035 enum/struct/union *'identifier'* : unknown size

The given structure or union had an undefined size.

C2036 left of *'member'* must have struct/union type

The expression before the member-selection operator (->) was not a pointer to a structure or union type, or the expression before the member-selection operator (.) did not evaluate to a structure or union. In this message, *member* is a member designator in one of the following forms:

->*identifier*
.*identifier*

C2037 left of *'->'* or *'.'* specifies undefined struct/union *'identifier'*

The expression before the member-selection operator (-> or .) identified a structure or union type that was not defined.

C2038 *'identifier'* : not struct/union member

The given identifier was used in a context that required a structure or union member.

C2039 *'->'* requires struct/union pointer

The expression before the member-selection operator (->) was a structure or union name, not a pointer to a structure or union as expected.

C2040 *'.'* requires struct/union name

The expression before the member-selection operator (.) was a pointer to a structure or union, not a structure or union name as expected.

C2041 keyword *'enum'* illegal

The **enum** keyword appeared in a structure or union declaration, or an **enum** type definition was not formed correctly.

C2042 signed/unsigned keywords mutually exclusive

The signed and unsigned keywords may not appear in the same declaration.

C2043 illegal break

A **break** statement is legal only when it appears within a **do**, **for**, **while**, or **switch** statement.

Compiler Error Messages

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 label appeared before more than one statement in the same function.

C2046 illegal case

The **case** keyword may appear only within a **switch** statement.

C2047 illegal default

The **default** keyword may appear only within a **switch** statement.

C2048 more than one default

A **switch** statement contained more than one **default** label.

C2049 cast has illegal formal parameter list

A formal parameter list was given in a type-cast expression.

C2050 non-integral switch expression

A switch expression was not integral.

C2051 case expression not constant

Case expressions must be integral constants.

C2052 case expression not integral

Case expressions must be integral constants.

C2053 case value *number* already used

The given case value was already used in this **switch** statement.

C2054 expected '(' to follow '*identifier*'

The context requires parentheses after the function *identifier*.

C2055 expected formal parameter list, not a type list

An argument-type list appeared in a function definition instead of a formal parameter list.

C2056 illegal expression

An expression was illegal because of a previous error. (The previous error may not have produced an error message.)

C2057 expected constant expression

The context requires a constant expression.

C2058 constant expression is not integral

The context requires an integral constant expression.

Compiler Error Messages

C2059 syntax error : *'token'*
The given token caused a syntax error.

C2060 syntax error : EOF
The end of the file was encountered unexpectedly, causing a syntax error. This error can be caused by a missing closing curly brace (}) at the end of your program.

C2061 syntax error : identifier *'identifier'*
The given identifier caused a syntax error.

C2062 type *'type'* unexpected
The given type was misused.

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 was made to call a function through an expression that did not evaluate to a function pointer.

C2065 *'identifier'* : undefined
The given identifier was not defined.

C2066 cast to function returning . . . is illegal
An object was cast to a function type.

C2067 cast to array type is illegal
An object was cast to an array type.

C2068 illegal cast
A type used in a cast operation was not a legal type.

C2069 cast of *'void'* term to non-void
The **void** type was cast to a different type.

C2070 illegal sizeof operand
The operand of a **sizeof** expression was not an identifier or a type name.

C2071 *'class'* : bad storage class
The given storage class cannot be used in this context.

C2072 *'identifier'* : initialization of a function
An attempt was made to initialize a function.

Compiler Error Messages

C2073 *'identifier'* : cannot initialize array in function
An attempt was made to initialize the given array within a function.
Arrays can be initialized only at the external level.

C2074 cannot initialize struct/union in function
An attempt was made to initialize the given structure or union within
a function. Structures and unions can be initialized only at the exter-
nal level.

C2075 *'identifier'* : array initialization needs
curly braces
The braces ({ }) around the given array initializer were missing.

C2076 *'identifier'* : struct/union initialization needs
curly braces
The braces ({ }) around the given structure or union initializer were
missing.

C2077 non-integral field initializer *'identifier'*
An attempt was made to initialize a bit-field member of a structure
with a nonintegral value.

C2078 too many initializers
The number of initializers exceeded the number of objects to be ini-
tialized.

C2079 *'expression'* uses undefined struct/union
The given identifier was declared as a structure or union type that
had not been defined.

C2082 redefinition of formal parameter *'identifier'*
A formal parameter to a function was redeclared within the function
body.

C2083 array *'identifier'* already has a size
The dimensions of the given array had already been declared.

C2084 function *'identifier'* already has a body
The given function had already been defined.

C2085 *'identifier'* : not in formal parameter list
The given parameter was declared in a function definition for a
nonexistent formal parameter.

C2086 *'identifier'* : redefinition
The given identifier was defined more than once.

Compiler Error Messages

C2087 '*identifier*' : missing subscript

The definition of an array with multiple subscripts was missing a subscript value for a dimension other than the first dimension, as in the following example:

```
int func(a)
    char a[10][];          /* Illegal */
    {
        .
        .
        .
    }

int func(a)
    char a[][5];          /* Legal */
    {
        .
        .
        .
    }
```

C2088 use of undefined enum/struct/union '*identifier*'

The given identifier referred to a structure or union type that was not defined.

C2089 typedef specifies a near/far function

The use of the **near** or **far** keyword in a **typedef** declaration conflicted with the use of **near** or **far** for the declared item, as in the following example:

```
typedef int far FARFUNC( );
FARFUNC near *fp;
```

C2090 function returns array

A function cannot return an array. (It can return a pointer to an array.)

C2091 function returns function

A function cannot return a function. (It can return a pointer to a function.)

C2092 array element type cannot be function

Arrays of functions are not allowed; however, arrays of *pointers* to functions are allowed.

Compiler Error Messages

C2093 cannot initialize a static or struct with address of automatic vars

You cannot use the address of an auto variable in the initializer of a static item.

C2094 label '*identifier*' was undefined

The function did not contain a statement labeled with the given identifier.

C2095 *function*: actual has type void: parameter *number*

An attempt was made to pass a **void** argument to a function. Formal parameters and arguments to functions cannot have type **void**; they can, however, have type **void *** (pointer to **void**).

C2096 struct/union comparison illegal

You cannot compare two structures or unions. (You can, however, compare individual members within structures and unions.)

C2097 illegal initialization

An attempt was made to initialize a variable using a nonconstant value.

C2098 non-address expression

An attempt was made to initialize an item that was not an lvalue.

C2099 non-constant offset

An initializer used a nonconstant offset.

C2100 illegal indirection

The indirection operator (*) was applied to a nonpointer value.

C2101 '&' on constant

The address-of operator (&) did not have an lvalue as its operand.

C2102 '&' requires lvalue

The address-of operator must be applied to an lvalue expression.

C2103 '&' on register variable

An attempt was made to take the address of a register variable.

C2104 '&' on bit-field

An attempt was made to take the address of a bit field.

Compiler Error Messages

C2105 *'operator'* needs lvalue
The given operator did not have an lvalue operand.

C2106 *'operator'* : left operand must be lvalue
The left operand of the given operator was not an lvalue.

C2107 illegal index, indirection not allowed
A subscript was applied to an expression that did not evaluate to a pointer.

C2108 non-integral index
A nonintegral expression was used in an array subscript.

C2109 subscript on non-array
A subscript was used on a variable that was not an array.

C2110 '+' : 2 pointers
An attempt was made to add one pointer to another.

C2111 pointer + non-integral value
An attempt was made to add a nonintegral value to a pointer.

C2112 illegal pointer subtraction
An attempt was made to subtract pointers that did not point to the same type.

C2113 '-' : right operand pointer
The right operand in a subtraction operation (-) was a pointer, but the left operand was not.

C2114 *'operator'* : pointer on left; needs integral right
The left operand of the given operator was a pointer; the right operand must be an integral value.

C2115 *'identifier'* : incompatible types
An expression contained incompatible types.

C2116 *'operator'* : bad *left* (or *right*) operand
The specified operand of the given operator was illegal for that operator.

C2117 *'operator'* : illegal for struct/union
Structure and union type values are not allowed with the given operator.

C2118 negative subscript
A value defining an array size was negative.

Compiler Error Messages

C2119 'typedefs' both define indirection
Two **typedef** types were used to declare an item and both **typedef** types had indirection. For example, the declaration of *p* in the following example is illegal:

```
typedef int *P_INT;
typedef short *P_SHORT;
/* this declaration is illegal */
P_SHORT P_INT p;
```

C2120 'void' illegal with all types
The **void** type was used in a declaration with another type.

C2121 typedef specifies different enum
An attempt was made to use a type declared in a **typedef** statement to specify both an enumeration type and another type.

C2122 typedef specifies different struct
An attempt was made to use a type declared in a **typedef** statement to specify both a structure type and another type.

C2123 typedef specifies different union
An attempt was made to use a type declared in a **typedef** statement to specify both a union type and another type.

C2125 *identifier* : allocation exceeds 64K
The given item exceeds the size limit of 64K.

The only items that are allowed to exceed 64K are huge arrays.

C2126 *identifier* : automatic allocation exceeds 32K
The space allocated for the local variables of a function exceeded the limit of 32K.

C2127 parameter allocation exceeds 32K
The storage space required for the parameters to a function exceeded the limit of 32K.

C2128 *identifier* : huge array cannot be aligned to segment boundary
The given array violated one of the restrictions imposed on huge arrays; see the "Working with Memory Models" chapter for more information on these restrictions.

Compiler Error Messages

C2129 static function '*identifier*' not found
A forward reference was made to a static function that was never defined.

C2130 #line expected a string containing the file name, found '*token*'
A file name was missing from a **#line** directive.

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
An identifier appeared in a syntactically illegal context.

C2133 array '*identifier*' : unknown size
An attempt was made to declare an unsized array as local variable, as in the following example:

```
int mat_add(array1)  
  int array1[];      /* Legal */  
  {  
    int array2[];    /* Illegal */  
    .  
    .  
    .  
  }
```

C2134 *identifier* : struct/union too large
The size of a structure or union exceeded the compiler limit (2³² bytes). This limit is 64K on 80286 systems.

C2135 missing ')' in macro expansion
A macro reference with arguments was missing a closing parenthesis ()).

C2137 empty character constant
The illegal character constant '' was used.

C2138 unmatched close comment '/*'
The compiler detected an open-comment delimiter (/*) without a matching close-comment delimiter (*/).

This error usually indicates an attempt to use illegal nested comments.

Compiler Error Messages

C2139 type following *'type'* is illegal
An illegal type combination such as the following was used:

```
long char a;
```

C2140 argument type cannot be function
returning ...
A function was declared as a formal parameter of another function,
as in the following example:

```
int func1(a)
    int a( );    /* Illegal */
```

C2141 value out of range for enum constant
An enumeration constant had a value outside the range of values
allowed for type **int**.

C2142 ellipsis requires three periods
The compiler detected the token *".."* and assumed that *"..."* was
intended.

C2143 syntax error : missing *'token1'* before *'token2'*
The compiler expected *token1* to appear before *token2*. This message
may appear if a required closing curly brace {}, right parenthesis ()),
or semicolon (;) is missing.

C2144 syntax error : missing *'token'* before type *'type'*
The compiler expected the given token to appear before the given
type name. This message may appear if a required closing curly
brace {}, right parenthesis ()), or semicolon (;) is missing.

C2145 syntax error : missing *'token'* before
identifier
The compiler expected the given token to appear before an identifier.
This message may appear if a semicolon (;) does not appear after the
last declaration of a block.

C2146 syntax error : missing *'token'* before identif-
ier *'identifier'*
The compiler expected the given token to appear before the given
identifier.

C2147 array : unknown size
An attempt was made to increment an index or pointer to an array
whose base type has not yet been declared.

Compiler Error Messages

C2148 array too large
An array exceeded the maximum legal size (2^{32} bytes).

C2149 *identifier* : named bit-field cannot have 0 width
The given named bit field had a zero width. Only unnamed bit fields are allowed to have zero width.

C2150 *identifier* : bit-field must have type int, signed int, or unsigned int
The ANSI C standard requires bit fields to have types of **int**, **signed int**, or **unsigned int**. This message appears only if you compiled your program with the **-Za** option.

C2151 more than one cdecl/fortran/pascal attribute specified
More than one keyword specifying a function-calling convention was given.

C2152 *identifier* : pointers to functions with different attributes
An attempt was made to assign a pointer to a function declared with one calling convention (**cdecl**, **fortran**, or **pascal**) to a pointer to a function declared with a different calling convention.

C2153 hex constants must have at least 1 hex digit
At least one hexadecimal digit must follow the "x". The hexadecimal constants 0x and 0X are illegal.

C2154 '*name*' : does not refer to a segment
The *name* was the first identifier given in an **alloc_text** pragma argument list and it is already defined as something other than a segment name.

C2155 '*name*' : already in a segment
The function *name* appears in more than one **alloc_text** pragma.

C2156 pragma must be at outer level
Certain pragmas must be specified at a global level, outside a function body, and there is an occurrence of one of these pragmas within a function.

C2157 '*name*' : must be declared before use in pragma list
The function *name* in the list of functions for an **alloc_text** pragma has not been declared prior to being referenced in the list.

Compiler Error Messages

C2158 '*name*' : is a function
Name was specified in the list of variables in a **same_segment** pragma, but was previously declared as a function.

C2159 more than one storage class specified
Illegal declaration—only one storage class is allowed.

C2160 ## cannot occur at the beginning of a macro definition
A macro definition cannot begin with a token-pasting (##) operator.

C2161 ## cannot occur at the end of a macro definition
A macro definition cannot end with a token-pasting (##) operator.

2162 expected macro formal parameter
The token following a stringizing operator (#) must be a formal parameter name.

2163 '*string*' : not available as an intrinsic
A function specified in the list of functions for an intrinsic or function pragma is not one of the functions available in intrinsic form.

C2165 '*keyword*' : cannot modify pointers to data
Bad use of **fortran**, **pascal** or **cdecl** keyword to modify pointer to data.

C2167 '*name*' : too many actual parameters for intrinsic
A reference to the intrinsic function *name* contains too many actual parameters.

C2168 '*name*' : too few actual parameters for intrinsic
A reference to the intrinsic function *name* contains too few actual parameters.

C2169 '*name*' : is an intrinsic, it cannot be defined
An attempt was made to provide a function definition for a function already declared as an intrinsic.

C2170 *identifier* : intrinsic not declared as a function
You tried to use the **intrinsic** pragma for an item other than a function, or for a function that does not have an intrinsic form.

C2177 constant too big
Information was lost because a constant value was too large to be represented in the type to which it was assigned. (1)

Compiler Error Messages

C2171 '*operator*' : bad operand
Illegal operand type for the specified unary operator.

C2187 : cast of near function pointer to far function pointer
You attempted to cast a near function pointer as a far function pointer.

C2189 : constant item, `-Gm` and `data_seg pragma` are incompatible
You compiled with the `-Gm` option and allocated a string literal or **near const** data item within the scope of a `data_seg pragma`. The `-Gm` option indicates that the data item should be allocated in the **CONST** data segment, while the `data_seg pragma` indicates that the same item should be allocated in a different, named segment.

Warning Messages

The messages listed in this section indicate potential problems but do not hinder compilation and linking. The number in parentheses at the end of each warning-message description (if any) gives the minimum warning level that must be set for the message to appear.

C4000 UNKNOWN WARNING
The compiler detected an unknown error condition.

C4001 macro '*identifier*' requires parameters
The given identifier was defined as a macro taking one or more arguments, but it was used in the program without arguments. (1)

C4002 too many actual parameters for macro '*identifier*'
The number of actual arguments specified with the given identifier was greater than the number of formal parameters given in the macro definition of the identifier. (1)

C4003 not enough actual parameters for macro '*identifier*'
The number of actual arguments specified with the given identifier was less than the number of formal parameters given in the macro definition of the identifier. (1)

Compiler Error Messages

C4004 missing close parenthesis after 'defined'
The closing parenthesis was missing from an **#if defined** phrase. (1)

C4005 '*identifier*' : redefinition
The given identifier was redefined. (1)

C4006 #undef expected an identifier
The name of the identifier whose definition was to be removed was not given with the **#undef** directive. (1)

C4009 string too big, trailing chars truncated
A string exceeded the compiler limit on string size. To correct this problem, break the string into two or more strings. (1)

C4011 identifier truncated to '*identifier*'
Only the identifier's first 31 characters are significant. (1)

C4014 '*identifier*' : bit-field type must be unsigned
The given bit field was not declared as an **unsigned** type.

Bit fields must be declared as **unsigned** integral types. A conversion has been supplied. (1)

C4015 '*identifier*' : bit-field type must be integral
The given bit field was not declared as an **integral** type.

Bit fields must be declared as **unsigned** integral types. A conversion has been supplied. (1)

C4016 '*identifier*' : no function return type
The given function had not yet been declared or defined, so the return type was unknown.

The default return type (**int**) is assumed. (2)

C4017 cast of int expression to far pointer
A **far** pointer represents a full segmented address. On an 8086/8088 processor, casting an **int** value to a **far** pointer may produce an address with a meaningless segment value. (1)

C4020 too many actual parameters
The number of arguments specified in a function call was greater than the number of parameters specified in the argument-type list or function definition. (1)

Compiler Error Messages

C4021 too few actual parameters

The number of arguments specified in a function call was less than the number of parameters specified in the argument-type list or function definition. (1)

C4022 pointer mismatch : parameter *n*

The pointer type of the given parameter was different from the pointer type specified in the argument-type list or function definition. (1)

C4024 different types : parameter *n*

The type of the given parameter in a function call did not agree with the type given in the argument-type list or function definition. (1)

C4025 function declaration specified variable argument list

The argument-type list in a function declaration ended with a comma or a comma followed by ellipsis dots (*,...*), indicating that the function could take a variable number of arguments, but no formal parameters were declared for the function. (1)

C4026 function was declared with formal argument list

The function was declared to take arguments, but the function definition did not declare formal parameters. (1)

C4027 function was declared without formal argument list

The function was declared to take no arguments (the argument-type list consisted of the word **void**), but formal parameters were declared in the function definition or arguments were given in a call to the function. (1)

C4028 parameter *n* declaration different

The type of the given parameter did not agree with the corresponding type in the argument-type list or with the corresponding formal parameter. (1)

C4029 declared parameter list different from definition

The argument-type list given in a function declaration did not agree with the types of the formal parameters given in the function definition. (1)

D

Compiler Error Messages

C4030 first parameter list is longer than the second

A function was declared more than once with different argument-type lists in the declarations. (1)

C4031 second parameter list is longer than the first

A function was declared more than once with different argument-type lists. (1)

C4032 unnamed struct/union as parameter

The structure or union type being passed as an argument was not named, so the declaration of the formal parameter cannot use the name and must declare the type. (1)

C4033 function must return a value

A function is expected to return a value unless it is declared as `void`. (2)

C4034 `sizeof` returns 0

The `sizeof` operator was applied to an operand that yielded a size of zero. (1)

C4035 *identifier* : no return value

A function declared to return a value did not do so. (2)

C4036 unexpected formal parameter list

A formal parameter list was given in a function declaration. The formal parameter list is ignored. (1)

C4037 '*identifier*' : formal parameters ignored

No storage class or type name appeared before the declarators of formal parameters in a function declaration, as in the following example:

```
int *f(a,b,c);
```

The formal parameters are ignored. (1)

C4038 '*identifier*' : formal parameter has bad storage class

The given formal parameter was declared with a storage class other than `auto` or `register`. (1)

C4039 '*identifier*' : function used as an argument

A formal parameter to a function was declared to be a function, which is illegal. The formal parameter is converted to a function pointer. (1)

Compiler Error Messages

C4040 *near/far/huge* on *'identifier'* ignored

The **near** or **far** keyword has no effect in the declaration of the given identifier and is ignored. (1)

C4041 formal parameter *'identifier'* is redefined

The given formal parameter was redefined in the function body, making the corresponding actual argument unavailable in the function. (1)

C4042 *'identifier'* : has bad storage class

The specified storage class cannot be used in this context (for example, function parameters cannot be given **extern** class). The default storage class for that context was used in place of the illegal class. (1)

C4043 *'identifier'* : void type changed to int

An item other than a function was declared to have **void** type. (1)

C4044 *huge* on *'identifier'* ignored, must be an array

The **huge** keyword was used to declare the given nonarray item. (1)

C4045 *'identifier'* : array bounds overflow

Too many initializers were present for the given array. The excess initializers are ignored. (1)

C4046 *'&'* on function/array, ignored

An attempt was made to apply the address-of operator (**&**) to a function or array identifier. (1)

C4047 *'operator'* : different levels of indirection

An expression involving the specified operator had inconsistent levels of indirection. (1)

The following example illustrates this condition:

```
char **p;  
char *q;  
.  
.  
.  
p = q;
```

Compiler Error Messages

C4048 array's declared subscripts different
An array was declared twice with different sizes. The larger size is used. (1)

C4049 '*operator*' : indirection to different types
The indirection operator (*) was used in an expression to access values of different types. (1)

C4051 data conversion
Two data items in an expression had different types, causing the type of one item to be converted. (2)

C4052 different enum types
Two different **enum** types were used in an expression. (1)

C4053 at least one void operand
An expression with type **void** was used as an operand. (1)

C4056 overflow in constant arithmetic
The result of an operation exceeded 0x7FFFFFFF. (1)

C4057 overflow in constant multiplication
The result of an operation exceeded 0x7FFFFFFF. (1)

C4058 address of frame variable taken, DS != SS
The program was compiled with the default data segment (**DS**) not equal to the stack segment (**SS**), and the program tried to point to a frame variable with a near pointer. (1)

C4059 segment lost in conversion
The conversion of a **far** pointer (a full segmented address) to a **near** pointer (a segment offset) resulted in the loss of the segment address. (1)

C4060 conversion of long address to short address
The conversion of a long address (a 32-bit pointer) to a short address (a 16-bit pointer) resulted in the loss of the segment address. (1)

C4061 long/short mismatch in argument:
conversion supplied
The base types of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)

Compiler Error Messages

C4062 near/far mismatch in argument: conversion supplied

The pointer sizes of the actual and formal arguments of a function were different. The actual argument is converted to the type of the formal parameter. (1)

C4063 '*identifier*' : function too large for post-optimizer

The given function was not optimized because not enough space was available. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)

C4064 procedure too large, skipping *description* optimization and continuing

Some optimizations for a function were skipped because not enough space was available for optimization. (0)

To correct this problem, reduce the size of the function by dividing it into two or more smaller functions.

The *description* in this message may appear as any of the following:

```
loop inversion
branch sequence
cross jump
```

C4065 recoverable heap overflow in post-optimizer - some optimizations may be missed

Some optimizations were skipped because not enough space was available for optimization. To correct this problem, reduce the size of the function by dividing it into two or more smaller functions. (0)

C4066 local symbol table overflow - some local symbols may be missing in listings

The listing generator ran out of heap space for local variables, so the source listing may not contain symbol-table information for all local variables.

Compiler Error Messages

C4067 unexpected characters following '*directive*'
directive - newline expected
Extra characters followed a preprocessor directive, as in the following example (1):

```
#endif NO_EXT_KEYS
```

This is accepted in Version 3.0, but not in Versions 4.0 and 5.0. Versions 4.0 and 5.0 require comment delimiters, such as the following:

```
#endif /* NO_EXT_KEYS */
```

C4068 unknown pragma

The compiler did not recognize a pragma and ignored it. (1)

C4069 conversion of near pointer to long integer

A near pointer was converted to a long integer, which involves first extending the high-order word with the current data-segment value, *not* 0 as in Version 3.0. (1)

C4071 '*identifier*' : no function prototype given

The given function was called before the compiler found the corresponding function prototype. (3)

C4072 Insufficient memory to process debugging information

You compiled the program with the **-Zi** option, but not enough memory was available to create the required debugging information. (1)

C4073 scoping too deep, deepest scoping merged when debugging

Declarations appeared at a static nesting level greater than 13. As a result, all declarations will seem to appear at the same level. (1)

C4074 non standard extension used - '*extension*'

The given nonstandard language extension was used when the **-Ze** option was in effect. These extensions are given in the "Compiling with the cc Command" chapter of this guide. (If the **-Za** option is in effect, this condition generates an error.) (3)

Compiler Error Messages

C4075 size of switch expression or case constant too large - converted to int
A value appearing in a **switch** or **case** statement was larger than an **int**. The compiler converts the illegal value to an **int**. (1)

C4076 '*type*' : may be used on integral types only
The type modifiers **signed** and **unsigned** can be combined only with other integral types.

C4077 unknown check_stack option
Unknown option given when using the old form of the **check_stack** pragma. The option must be empty, +, or -.

C4079 unexpected char '*character*'
Unexpected separator *character* found in argument list of a pragma.

C4080 missing segment name
The first argument in the argument list for the **alloc text** pragma is missing a segment name. This happens if the first token in the argument list is not an identifier.

C4081 expected a comma
There is a missing comma (,) between two arguments of a pragma.

C4082 expected an identifier
There is a missing identifier in list of arguments to a pragma.

C4083 missing '('
There is a missing opening parenthesis (() in the argument list for a pragma.

C4084 expected a pragma keyword
The token following the **pragma** keyword is not an identifier.

C4085 expected [onloff]
Bad argument given for new form of **check_stack** pragma.

C4086 expected [1|2|4]
Bad argument given for **pack** pragma.

C4087 '*name*' : declared with *void parameter list*
The function *name* was declared as taking no parameters, but a call to the function specifies actual parameters.

C4090 different '*const*' attributes
The program passed a pointer to a *const* item to a function where the corresponding formal parameter is a pointer to a non-*const* item, which means the item could be modified by the function undetected.

Compiler Error Messages

C4091 no symbols were declared
An empty declaration was detected. (2)

C4092 untagged enum/struct/union declared no symbols
An empty declaration was detected that used an untagged
enum/struct/union. (2)

C4093 unescaped newline in character constant in
non-active code

The constant expression of an **#if**, **#elif**, **#ifdef**, or **#ifndef** preprocessor directive evaluated to 0, making the following code inactive, and a new-line character appeared between a single or double quotation mark and the matching single or double quotation mark in that inactive code.

C4094 unexpected newline

A new-line character appeared in a pragma where a comma, right parenthesis, or identifier was expected, as in the following examples:

```
#pragma intrinsic (memset  
#pragma intrinsic (memset,
```

C4095 too many arguments for pragma

More than one argument was given for a pragma that can take only one argument.

C4106 : pragma requires integer between 1 and 127

You must supply an integer constant in the range 1-127, inclusive, for the given pragma.

C4107 : pragma requires integer between 15 and 255

You must supply an integer constant in the range 15-255, inclusive, for the given pragma.

C4108 : pragma requires integer between 79 and 132

You must supply an integer constant in the range 79-132, inclusive, for the given pragma.

C4109 : unexpected identifier '*token*'

The designated line contains an unexpected token.

C4110 : unexpected token '*int constant*'

The designated line contains an unexpected integer constant.

C4111 : unexpected token string

The designated line contains an unexpected string.

Compiler Error Messages

C4112 : macro name '*name*' is reserved, '*command*' ignored
You attempted to define a predefined macro name or the preprocessor operator **defined**. This warning error also occurs if you attempt to undefine a predefined macro name. If you attempt to define or undefine a predefined macro name using command-line options, '*command*' will still be either **#define** or **#undef**.

Compiler Limits

To operate the C Compiler, you must have sufficient disk space available for the compiler to create temporary files used in processing. The space required is approximately two times the size of the source file.

Table D.1 summarizes the limits imposed by the C compiler. If your program exceeds one of these limits, an error message will inform you of the problem.

Table D.1
Limits Imposed by the C Compiler

<u>Program Item</u>	<u>Description</u>	<u>Limit</u>
String literals	Maximum length of a string, including the terminating null character (<code>\0</code>)	4k bytes
Constants	Maximum size of a constant is determined by its type; see the <i>C Language Reference</i> for a discussion of constants	
Identifiers	Maximum length of an identifier	31 bytes (additional characters are discarded)
Declarations	Maximum level of nesting for structure/union definitions	10 levels

Compiler Error Messages

Preprocessor directives	Maximum size of a macro definition	512 bytes
	Maximum number of actual arguments to a macro definition	8 arguments
	Maximum length of an actual preprocessor argument	256 bytes
	Maximum level of nesting for #if , #ifdef , and #ifndef directives	32 levels
	Maximum level of nesting for include files	10 levels

The compiler does not set explicit limits on the number and complexity of declarations, definitions, and statements in an individual function or in a program. If the compiler encounters a function or program that is too large or too complex to be processed, it produces an error message to that effect.

Compiler Exit Codes

All the programs in the C Compiler package return an exit code (sometimes called an “errorlevel” code) that can be used by other programs such as **make**. If the program finishes without errors, it returns a code of 0. The code returned varies depending on the error encountered.

Code	Meaning
------	---------

- | | |
|---|---|
| 0 | No fatal error |
| 2 | Program error (such as compiler error) |
| 4 | System level error (such as out of disk space or compiler internal error) |



Index

Special Characters

- { } (braces) 1-6
- [] (brackets) 1-6
- | (bar) 1-6
- (dash) option character
 - linker 3-5
- (hyphen) option character, cc 2-6
- _ (underscore), in names 2-12, 2-23

A

- Address space B-9
- Addresses
 - components 5-4
 - far 5-4
 - huge 5-4
 - near 5-4
- Alignment *See* Storage alignment
- alloc_text pragma 5-33
- argc variable 4-2
- Argument type list 2-39
- Arguments
 - linker options 3-5
 - listing options 2-13
 - main function *See* main function
 - variable number of 6-6, B-14
- argv variable 4-2
- Assembly language
 - interface 8-12
 - return values 8-9
 - routines
 - entry 8-8
 - exit 8-11
- Assembly-language interface, described 8-1
- Assembly-listing files
 - creating 2-11, 2-12
 - extensions 2-13
 - format 2-22

B

- Bar (|) 1-6
- BASE 7-14
- Bibliography 1-9
- Bit fields B-7
- Bold font 1-6
- Braces ({}) 1-6
- Brackets ([]) 1-6
- Byte length B-4
- Byte order B-20, B-6

C

- C calling conventions
 - described 8-1
- C compiler
 - impure small model 5-8
 - M option 2-6
 - manifest defines 2-30
 - model and segment options 2-6
 - pure small model 5-8
- C language
 - calling sequence 8-7
 - interface with assembly language 8-12
 - return values 8-9
- c option 2-9
- C option 2-33
- Call sequence 8-2
- Calling conventions
 - C 6-6
 - FORTRAN/Pascal 6-6
- Calling sequence
 - assembly language 8-7
 - C language 8-7
- Canonic Frame 7-8
- Capital letters
 - small 1-6
 - use of 1-6
- cc command
 - file processing 2-3
 - format 2-2
- cc options
 - assembly listing 2-11, 2-12
 - c 2-9
 - C 2-33
 - command line, order 2-6
 - comments, preserving 2-33
 - constants and macros, defining 2-29

Index

cc options (*continued*)

- D 2-29
- data segments, naming 5-31, 5-34, 6-9
- data threshold, setting 5-30
- default libraries 2-8
- differences from linker options 3-5
- E 2-32
- EP 2-32
- F 3-6
- Fa 2-12, 2-22
- Fc 2-12
- Fe 2-11
- Fl 2-12
- Fm 2-12
- Fo 2-9
- format 2-6
- Fs 2-12
- function declarations, generating 2-39
- Gs 6-6
- Gt 5-30
- I 2-34
- include files, searching for 2-34
- line numbers 2-40
- link 2-2, 3-2
- linker information, passing 3-2
- listing 2-6
- M 5-25, 5-26, 5-27
- Mc 5-9
- memory models
 - code-pointer size 5-26
 - compact 5-9
 - data-pointer size 5-26
 - default libraries 2-6
 - huge 5-11
 - large 5-10
 - medium 5-9
 - mixed 5-25, 5-26, 5-27
 - segments, setting up 5-27
 - small 5-7
- Mh 5-11
- Ml 5-10
- Mm 5-9
- Ms 5-7
- naming
 - executable files 2-11
 - modules 5-31
 - object files 2-9
- ND 5-31, 5-34, 6-9
- NM 5-31
- NT 5-31
- o 2-11
- Oa 6-5
- object files
 - naming 2-9

cc options (*continued*)

- object files (*continued*)
 - specifying 2-3
- object listing 2-11, 2-12
- Od 2-40
- Oi 6-4
- Ol 6-5
- optimization
 - alias checking, relaxing 6-5
 - disabling 2-40
 - execution time 6-4
 - intrinsic functions 6-4
 - loops 6-5
 - Oi 6-4
 - program speed 6-3
- option character
 - hyphen (-) 2-6
- Ot 6-4
- P 2-32
- predefined identifiers, removing
 - definitions of 2-32
- preprocessed listing 2-32
- preprocessor
 - C 2-33
 - D 2-28
 - U and -u 2-32
- S 2-12, 2-22
- source files, specifying 2-3, 2-8
- source listing 2-12
- source/object listing 2-12
- special keywords, disabling 5-15
- Ss 2-15
- St 2-15
- stack probes, removing 6-6
- standard places, ignoring 2-34
- subtitle 2-15
- suppressing
 - linking 2-9
- syntax checking 2-38
- Tc 2-4, 2-8
- text segments, naming 5-31
- titles 2-15
- U and -u 2-32
- Version 4.0, new for A-11
- Version 5.0, new for A-6
- W0, -W1, -W2, and -W3 2-37
- warning level 2-37
- X 2-34
- Za 5-15
- Zd 3-6
- Zg 2-39
- Zi 2-40
- Zs 2-38

cdecl keyword

cdecl keyword (*continued*)
 -Gc option, used with 6-7

Character
 classification, macros B-9
 set B-9
 types
 signed B-11
 unsigned B-11

check_stack pragma 6-6

Class name, LSEG 7-8

Code pointers, mixed memory models 5-26

COFF 2-2

Combination Attribute 7-29

Command line
 arguments
 executable file 4-2
 cc 2-2
 error messages D-2
 length, maximum 2-2

Commands
 notational conventions 1-6

COMMENT 7-48
 RECORD 7-48

Comments, preserving 2-33

Common Object File Format 2-2

Compact memory models *See* Memory models

Compilation
 error messages D-7

Compiler
 differences, other compilers
 portability problems B-11
 differences, Version 4.0
 cc options A-11
 enhancements and additions A-8
 language changes A-8
 differences, Version 5.0
 enhancements and additions A-4
 language changes A-4
 new cc options A-6
 pragmas, new A-7
 documentation 1-2
 error messages *See* Error messages,
 compiler
 naming conventions 2-23
 stopping 2-2

Compiler, converting from previous versions
See Compiler differences

Compiler guide, organization 1-2

Compiler options *See* cc options

Complete name, LSEG 7-8

Conditional compilation 2-29

Constants
 defining 2-29
 manifest *See* Constants, symbolic

Constants (*continued*)
 symbolic 2-29

Controlling
 linker 3-5
 preprocessor 2-32
 segments 3-6
 stack size 3-6

Conventions, notational 1-6

Conversion
 near pointers to long integers A-10
 pointer arguments 5-22

Correctable error messages D-7

ctype.h macros B-9

Customized memory models *See* Mixed
 memory models

D

-D option 2-29

Dash (-)
 linker option character 3-5

Data
 passing to programs 4-2
 portability B-17
 segment
 data threshold, setting 5-30
 default, contents 5-30
 default name 5-31
 mixed memory models 5-27
 naming 5-31
 types, size of B-4

Data pointers, mixed memory
 models 5-26

_DATA segment 5-31

Data Structures
 x.out symbol table 7-58

Data threshold, setting 5-30

data_seg pragma 5-33

Debugging, preparing for
 -Zi and -Od options 2-40

Default libraries
 object files, used in 3-3

DGROUP group 5-31

Differences from previous versions *See*
 Compiler differences

Directory names, notational
 conventions 1-6

Documentation, compiler 1-2

DS register 5-27

Index

E

- E option 2-32
- EAX register 8-9, 8-11
- EBP register 8-8, 8-11
- EBX register 8-11
- ECX register 8-11
- EDI register 8-8, 8-11
- EDX register 8-9, 8-11
- EIGHT
 - LEAF
 - DESCRIPTOR 7-32
 - EIGHT LEAF DESCRIPTOR 7-32
- Ellipses, use of 1-6
- environ variable 4-3
- Environment
 - portability problems B-16
 - table
 - pointer to 4-3
 - variable names, notational conventions 1-6
 - variables
 - INCLUDE 2-34
 - LIB 3-4
 - PATH 4-1
 - SET 4-2
- envp variable 4-3
- EP option 2-32
- errno variable
 - defined 9-3
 - described 9-3
- Error messages
 - compiler
 - command line D-2
 - compilation D-7
 - correctable D-7
 - fatal D-7, D-8
 - identifying 2-35
 - redirecting 2-35
 - warning D-29, D-8
 - format *See* Error messages, compiler
 - source listings 2-16
 - warning messages, setting level of 2-37
- Errors
 - catching signals 9-5
 - delayed 9-6
 - errno variable 9-3
 - error constants 9-3
 - error numbers 9-3
 - printing error messages 9-4
 - processing 9-1
 - routine system I/O 9-6
 - sharing resources 9-6
 - signals 9-5

Errors (*continued*)

- standard error file 9-2
- system 9-6
- ESI register 8-8, 8-11
- ESP register 8-8
- Evaluation order B-14
- exec function 4-1
- Executable files
 - cc command and 2-5
 - command-line arguments 4-2
 - extensions 2-11
 - naming, default 2-11
 - naming with cc 2-11
 - passing data to 4-2
 - running 4-1
- Executable Format 7-59
- Execution-time optimization 6-4
- Extensions
 - executable files 2-11
 - listing files, defaults for 2-12
 - map files 2-13
 - object files 2-10
 - object-listing files 2-13
 - source-listing files 2-13
 - source/object-listing files 2-13

F

- F option 3-6
- Fa option 2-12, 2-22
- Far keyword 5-20
- far keyword
 - default addressing conventions 5-14
 - effects
 - data declarations 5-16, 6-8
 - function declarations 5-20
 - library routines, used with 5-16
 - small-model programs, used in 5-8
- Far pointers 5-14
- Fatal-error messages D-7, D-8
- Fc option 2-12
- Fe option 2-11
- File names
 - notational conventions 1-6
- Files
 - assembly listing 2-12, 2-22
 - executable *See* Executable files
 - listing, preprocessed 2-32
 - map
 - creating 2-12, 2-15, 3-6
 - default names 2-13
 - listing formats 2-27

Files (*continued*)map (*continued*)

-MAP linker option 3-6

object

cc command, used with 2-3, 2-4

listing 2-12, 2-13, 2-21

source 2-3

source listing *See* Source-listing filessource/object listing *See* Source/object-listing files

FIXUP

RECORD 7-41

FIXUPP 7-41

Fixups

definition 7-14

segment-relative 7-15, 7-20

self-relative 7-15, 7-19

-Fl option 2-12

-Fm option 2-12

-Fo option 2-9

fortran keyword 6-7

FRAME

definition 7-6

specifying 7-17

FRAME NUMBER 7-7

-Fs option 2-12

Functions

arguments, variable number of 6-6, B-14

calling conventions

C 6-6

FORTRAN/Pascal 6-6

declarations

generating 2-39

near and far keywords 5-20

G

getenv function 4-3

Global symbols *See* Public symbols

GROUP 7-7

Group Definition Record 7-31

GRPDEF 7-31

-Gs option 6-6

-Gt option 5-30

H

Hardware Reference Numbers 7-64

HIBYTE 7-15

Huge arrays 5-11

huge keyword

data declarations, effects in 5-16, 6-8

default addressing conventions 5-14

library routines, used with 5-16

small-model programs, used in 5-8

Huge memory model *See* Memory models

Huge pointers 5-14

Hyphen (-), cc option character 2-6

I

-I option 2-34

iAPX-286, -386

address translation

logical to physical 7-2

descriptor tables 7-2

GDT 7-2

LDT 7-2

logical address space 7-2

memory management 7-2

pointers

to logical addresses 7-2

protected mode 7-2

segment selector 7-2

INDEX field 7-2

RPL field 7-2

TI field 7-2

system architecture 7-2

Identifier length *See* Names, length

Identifiers

predefined

listed 2-30

M_I86 2-30

M_I86xM 2-30

M_XENIX 2-30

removing definitions of 2-32

Implicit bss 7-57

Include files

directory specification 2-34

portability problems B-3

search path 2-34

INCLUDE variable

overriding 2-34

Index fields 7-13

Indices 7-13

Intel Object Module Format 2-2

Index

Italics 1-6
Iterated Segments 7-56

K

Key sequences, notational conventions 1-6

Keywords

- cdecl 6-7
- far 5-20
- fortran 6-7
- near 5-20
- pascal 6-7
- Version 4.0, new for A-11

L

Large memory model *See* Memory models.
large

Large Model 7-58

LIB variable 3-3, 3-4

Libraries

- default
 - A options 2-8
 - M options 3-2
 - overriding 3-4
- mixed-model programs 5-29
- names in object files 3-2
- search
 - path 3-3, 3-4
 - specifying 3-3
 - standard places 3-4

Library

- routines
 - exec 4-1
 - getenv 4-3
 - putenv 4-3
 - system 4-1
 - system dependent C-2

LIDATA 7-39

LINE

NUMBERS

RECORD 7-37

-LINENUMBERS (-LI) linker option 3-6

linesize pragma 2-24

-link option 2-2, 3-2

Linker

- error messages 2-35

Linker options

- abbreviations 3-5

Linker options (*continued*)

- cc options, differences from 3-5

- line numbers, displaying 3-6

- LINENUMBERS (-LI) 3-6

- map file 3-6

- MAP (-M) 3-6

- numerical arguments 3-5

- rules 3-5

- segments

- number of 3-6

- SEGMENTS (-SE) 3-6

- stack size, setting 3-6

- STACK (-ST) 3-6

- T 6-9

- translating far calls 6-9

LINNUM 7-37

List of Names Record 7-26

Listing files

- assembly 2-11, 2-12, 2-22

- map 2-12

- object 2-11, 2-12, 2-21

- preprocessed 2-32

- source 2-11, 2-12, 2-16

- source/object 2-12, 2-22

listing pragmas 2-24

LNames 7-26

LOBYTE 7-15

LOCATION, types 7-14

LOGICAL

ITERATED

DATA

RECORD 7-39

Logical Segment 7-7

Long pointers *See* Far pointers

Loop optimization 6-5

loop_opt pragma 2-41, 6-5

LSEG 7-7

M

-M option 5-25, 5-26, 5-27

M option

- cc 2-6

Macros

- character classification B-9

- defined 2-29

- notational conventions 1-6

main function

- arguments to 4-2

Manifest constants, notational

- conventions 1-6

Manifest defines

Manifest defines (*continued*)
 C compiler 2-30

Map files
 creating 2-12, 2-15, 3-6
 extensions 2-13, 3-6
 -Fm option 2-15
 format 2-27
 -MAP linker option 3-6
 program entry point 2-28
 segment lists 2-27
 symbol tables 2-27

-MAP linker option 3-6

MAS 7-6

-Mc option 5-9

Medium memory model *See* Memory models

Memory Address Space 7-6

Memory addresses *See* Addresses

Memory models
 compact 5-9
 default 5-2, 5-8
 huge 5-11
 large 5-10
 medium 5-9
 mixed *See* Mixed memory models
 options
 code-pointer size 5-26
 compact model 5-9
 data-pointer size 5-26
 default libraries 2-8
 huge model 5-11
 large model 5-10
 medium model 5-9
 segment setup 5-27
 small model 5-7
 small 5-2, 5-7, 5-19
 standard
 advantages 5-6
 common features 5-6
 disadvantages 5-6
 Version 4.0, new for A-11

Memory models, customized *See* Mixed memory models

-Mh option 5-11

M_I86 identifier 2-30

M_I86xM identifier 2-30

Mixed memory models
 code pointers 5-26
 creating 5-25
 data pointers 5-26
 library support 5-29
 near, far, huge keywords 5-14
 segment setup options 5-27

-Ml option 5-10

-Mm option 5-9

MODE 7-15

MODEND 7-46

MODULE 7-6
 END
 RECORD 7-46

Module header record 7-9

Modules, naming 5-31

-Ms option 5-7

M_XENIX identifier 2-30

N

Names
 executable files 2-11
 global 2-12, 2-23
 length B-12
 modules, changing 5-31
 object files 2-9
 segments, changing 5-31
 underscores (_), using in 2-12, 2-23

Naming conventions
 compiler 2-23
 segments 5-32

-ND option 5-31, 5-34, 6-9

Near keyword 5-20

near keyword
 data declarations, effects in 5-16, 6-8
 default addressing conventions 5-14
 function declarations, effects in 5-20
 library routines, used with 5-16

Near pointer 5-14

-NM option 5-31

Non-Iterated Segments 7-57

Notational conventions 1-6

-NT option 5-31

Numeric record types 7-50

O

-O (optimization) options 2-41

-o option 2-11

-Oa option, cc 6-5

object file format 2-2

Object File Format
 Executable 7-54

Object files
 cc command 2-3, 2-4
 default extension 2-3, 2-8
 extensions 2-10

Index

Object files (*continued*)
 library names in 3-2
 naming 2-9
 specifying to cc 2-3
Object listing *See* Object-listing files
Object Module Format 2-2
Object Module Formats 7-5, 7-6
Object-listing files
 creating 2-12
 extensions 2-13
 format 2-21
-Od option 2-40
OFFSET 7-14
-Oi option 6-4
-Oj option 6-5
OMF 2-2, 7-6
omf Subset 7-54
Optimization
 alias checking, relaxing 6-5
 default 2-1
 disabling 2-40
 execution time 6-4
 intrinsic pragmas 6-4
 listing files 2-14
 loops 6-5
 options 2-41
 stack probes, removing 6-6
Optimizing *See* Optimization
Optional fields, notational conventions 1-6
Options, cc *See* cc options
Options, linker *See* Linker options
-Ot option 6-4
Overlay Name, LSEG 7-8
Overview 1-1

P

-P option 2-32
page pragma 2-25
pagesize pragma 2-25
PARAGRAPH NUMBER 7-7
pascal keyword 6-7
Path names
 notational conventions 1-6
 portability problems B-3
PATH variable 4-1, 4-2
perror function 9-4
Physical Segment 7-7
Placeholders 1-6
Pointers
 arguments, size conversion 5-22
 code 5-26

Pointers (*continued*)
 far 5-14, 5-26
 huge 5-14
 manipulation B-7
 near
 conversion to long integers A-10
 customized memory models 5-26
 near keywords, used with 5-14
 subtracting in huge-model
 programs 5-11
Portability
 address space B-9
 bit fields B-7
 byte length B-4
 byte order B-20, B-6
 case distinctions B-12
 character set B-9
 data B-17
 data types, size of B-4
 environment differences B-16
 evaluation order B-14
 functions with variable number of
 arguments B-14
 guidelines B-2
 hardware B-4
 identifier length B-12
 include files B-3
 path names B-3
 pointer manipulation B-7
 register variables B-12
 shift operations B-11
 side effects B-14
 sign extension B-11
 signed and unsigned char types B-11
 storage alignment B-5
 type conversion B-12
 word length B-4
Pragmas
 alloc_text 5-33
 check_stack 6-6
 data_seg 5-33
pragmas
 linesize 2-24
 listing 2-24
Pragmas
 loop_opt 2-41, 6-5
pragmas
 page 2-24
 pagesize 2-24
Pragmas
 same_seg 5-33, 5-34, 6-9
pragmas
 skip 2-24
 skip page 2-24

pragmas (*continued*)
 subtitle 2-24
 title 2-24
 Pragmas
 Version 4.0, new for A-11
 Version 5.0, new for A-7
 Preprocessor
 options
 comments, preserving 2-33
 -D 2-29
 predefined identifiers, removing definitions
 of 2-32
 use 2-28
 Product names, notational conventions 1-6
 Prompts 1-6
 PSEG
 definition 7-7
 NUMBER 7-7
 PUBDEF 7-33
 PUBLIC
 NAMES
 DEFINITION
 RECORD 7-33
 Public names *See* Public symbols
 Public symbols, listing 2-15, 3-6
 putenv function 4-3

Q

Quotation marks, use of 1-6

R

Record format, sample 7-24
 Record formats 7-4
 Record order 7-22
 Record types 7-51
 numeric 7-50
 Register variables 6-1, B-12
 Registers
 EAX 8-9, 8-11
 EBP 8-8, 8-11
 EBX 8-11
 ECX 8-11
 EDI 8-8, 8-11
 EDX 8-9, 8-11
 ESI 8-8, 8-11
 ESP 8-8
 Relocatable memory images 7-4
 Return values 8-4
 assembly language 8-9
 Routine entry sequence 8-3
 Routine exit sequence 8-5
 Routines
 assembly language
 entry 8-8
 exit 8-11
 Run file *See* Executable file

S

-S option 2-12, 2-22
 same_seg pragma 5-33, 5-34, 6-9
 Sample x.out File 7-56
 Search paths
 changing
 include files 2-34
 libraries 3-4
 include files 2-34
 libraries 3-3, 3-4
 SEGDEF 7-27
 Segment addressing 7-11
 Segment definition 7-10
 Segment definition record 7-27
 Segment lists
 map files 2-27
 source listings 2-21
 Segment Name, LSEG 7-8
 Segment Numbers 7-64
 Segment registers 8-11
 Segment-Relative fixups 7-15
 Segment-Relative Fixups 7-20
 Segments
 data
 default name 5-31
 mixed memory models 5-27
 names 5-31
 naming 5-31
 threshold, effect of 5-30
 default 5-4
 defined 5-4
 names, changing 5-31
 naming conventions 5-32
 number allowed 3-6
 setting up 5-27
 source listing 2-21
 stack 5-27
 text
 default name 5-31
 naming 5-31
 -SEGMENTS (-SE) linker option 3-6

Index

- Self-Relative fixups 7-15, 7-19
 - SET variable 4-2
 - Shift operations B-11
 - Short pointers *See* Near pointers
 - Side effects B-14
 - Sign extension B-11
 - Signals
 - catching 9-5
 - on program errors 9-5
 - Signed char type B-11
 - sizeof operator 5-11
 - skip pragma 2-25
 - Small capitals, use of 1-6
 - Small memory model *See* Memory models
 - Small model 5-19
 - impure 5-8
 - pure 5-8
 - Source files
 - default extension 2-3, 2-8
 - specifying to cc 2-3
 - Source listing *See* Source-listing files
 - Source-listing files
 - creating 2-12
 - described 2-11
 - error messages 2-16
 - extensions 2-13
 - format 2-16, 2-17
 - segment lists 2-21
 - subtitles 2-15
 - symbol tables 2-19
 - titles 2-15
 - Source/object-listing files
 - creating 2-12
 - extensions 2-13
 - format 2-22
 - Special Header Fields 7-58
 - Special keywords, disabling 5-15
 - Ss option 2-15
 - SS register 5-27
 - St option 2-15
 - Stack
 - probes 6-6
 - segments, mixed memory models 5-27
 - size
 - setting 3-6
 - Stack order 8-2
 - STACK (-ST) linker option 3-6
 - Standard files
 - redirecting 9-2
 - Standard places
 - changing 2-34
 - ignoring 2-34
 - libraries 3-4
 - stderr, the standard error file 9-2
 - Storage alignment B-5
 - Strings
 - notational conventions 1-6
 - subtitle pragma 2-26
 - Subtitles, source listings 2-15
 - Switches *See* Options
 - Symbol definition 7-12
 - Symbol Table 7-58
 - Symbol tables
 - map files, used in 2-27
 - object files, used in (-Zi option) 2-40
 - source listings, used in 2-19
 - Syntax conventions *See*
 - Notational conventions
 - sys_erro array, described 9-4
 - System errors
 - described 9-6
 - reporting 9-6
 - system function 4-1
- ## T
- T linker option 6-9
 - TARGET 7-15
 - Tc option 2-4, 2-8
 - _TEXT segment 5-31
 - Text segments
 - default name 5-31
 - naming 5-31
 - THEADR 7-26
 - title pragma 2-26
 - Titles, source listings 2-15
 - T-MODULE 7-6
 - T-Module Header Record (THEADR) 7-26
 - TYPDEF 7-32
 - Types
 - checking 2-39
 - conversion B-12
- ## U
- U and -u options 2-32
 - Underscore (_) in names 2-12, 2-23
 - Unsigned char type B-11
 - Uppercase letters, use of 1-6

V

Variables, register *See* Register variables
Vertical bar (|) 1-6

W

-W0, -W1, -W2, and -W3 options 2-37
Warning error messages 2-37, D-29, D-8
Wild card
 characters 2-9

X

-X option 2-34
x.out
 file layout 7-56
 general description 7-54
 implicit bss 7-57
 iterated segments 7-56
 large model 7-58
 non-iterated segments 7-57
 special fields 7-58
 symbol table 7-58
x.out Examples 7-60
x.out Executable Format 7-59
x.out Format 7-54
x.out Include Files 7-60
x.out Segmented OMF Specification 7-54

Z

-Za option 5-15
-Zd option 3-6
-Zg option 2-39
-Zi option 2-40
-Zs option 2-38

.

1

2

3

Suggestions – Criticisms – Corrections

Are you happy with this manual? If so, let us know.

If not, help us improve it by informing us

- where you have noticed mistakes
- where the content is unclear.

From:

Name

Company/department

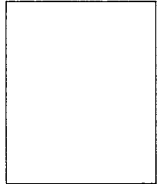
Address

Postal Code

Telephone ()

Local Siemens office

Contact person



Siemens AG
DI ST QM2
Manualredaktion
Otto-Hahn-Ring 6
Postfach 830951

D-8000 München 83

From:

Name

Company/department

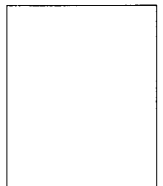
Address

Postal Code

Telephone ()

Local Siemens office

Contact person



Siemens AG
DI ST QM2
Manualredaktion
Otto-Hahn-Ring 6
Postfach 830951

D-8000 München 83

Hit a "stumbling block"? Tell us where.



Manual title: SINIX Open Desktop V1.0, U5754-J-Z95-1-7600

Page	Problem:

I am a programmer a system administrator an ordinary user _____

I use the manual frequently occasionally for reference _____

Manual title: SINIX Open Desktop V1.0, U5754-J-Z95-1-7600

Page	Problem:

I am a programmer a system administrator an ordinary user _____

I use the manual frequently occasionally for reference _____