iC-86/286/386
COMPILER USER'S GUIDE
FOR DOS SYSTEMS

Order Number: 483327-002

REV.	REVISION HISTORY	DATE
-001	Original Issue. Incorporates and updates the Version 4.1 iC-86/286 User's Guide and the Version 4.2 iC-386 User's Guide.	11/90
-002	Describes Version 4.3, including 80C187 coprocessor support.	11/91

Intel Corporation, 3065 Bowers Avenue, Santa Clara, California 95052-8126

In the United States, additional copies of this manual or other Intel literature may be obtained by writing:

Literature Distribution Center
Intel Corporation
P.O. Box 7641
Mt. Prospect, IL 60056-7641

Or you can call the following toll-free number:

1-800-548-4725

In locations outside the United States, obtain additional copies of Intel documentation by contacting your local Intel sales office. For your convenience, international sales office addresses are printed on the last page of this document.

Intel Corporation makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. Intel Corporation assumes no responsibility for any errors that may appear in this document. Intel Corporation makes no commitment to update nor to keep current the information contained in this document.

Intel Corporation assumes no responsibility for the use of any circuitry other than circuitry embodied in an Intel product. No other circuit patent licenses are implied.

Intel software products are copyrighted by and shall remain the property of Intel Corporation. Use, duplication or disclosure is subject to restrictions stated in Intel's Software License Agreement, or in the case of software delivered to the government, in accordance with the software license agreement as defined in FAR 52.227-7013.

No part of this document may be copied or reproduced in any form or by any means without prior written consent of Intel Corporation.

Intel Corporation retains the right to make changes to these specifications at any time, without notice.

Contact your local sales office to obtain the latest specifications before placing your order.

The following are trademarks of Intel Corporation and its affiliates and may be used only to identify Intel products. (Registered trademarks are followed by a superscripted ® .)

376	i287	Intel®	LANPrint®	PRO750
Above	i386	Intel287	LANSelect®	ProSolver
ActionMedia	i387	Intel386	LANShell®	READY-LAN
BITBUS	i486	Intel387	LANSight	Reference Point®
Code Builder	i487	Intel486	LANSpace®	RMX/80
DeskWare	i750®	Intel487	LANSpool®	SatisFAXtion®
Digital Studio	i860	intel inside	MAPNET	SnapIn 386
DVI®	i960	Intellec®	Matched	StorageBroker
EtherExpress	¹	iPSC®	MCS®	SugarCube
ETOX	i®	iRMX®	Media Mail	The Computer Inside
ExCA	ICE	iSBC®	NetPort	Token Express
FaxBACK	iLBX	iSBX	NetSentry	Visual Edge
Grand Challenge	Inboard	iWARP	OpenNET	WYPIWYF

IBM and PC AT are registered trademarks, and PC XT is a trademark of International Business Machines Corporation.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

VAX and VMS are trademarks of Digital Equipment Corporation.

Contents

Chapter 3 Compiler Controls and Qualifiers

Chapter 4 Segmentation Memory Models

Chapter 5 Listing Files

Chapter 6 Processor-specific Facilities

Chapter 7 Assembler Header File

Chapter 8 Function-calling Conventions

Chapter 9 Subsystems

Chapter 10 Language Implementation

Chapter 11 Messages

Figures

Tables

Getting Started

This section of the *iC-86/286/386 Compiler User's Guide* tells you what is in this manual and where to find the information you need to install and use the software.

Installing the Software

Insert Disk 1 into the `A:` drive and type `a:install`. The tabbed Installation section near the end of this manual contains detailed instructions for installing the iC-86/286/386 compilers and libraries.

Learning About and Using the iC-86/286/386 Compiler

Chapter 1 contains an overview describing the compiler and its compatibility with other Intel C compilers, how to use the iC-86/286/386 compiler to develop applications, and information on related manuals. Chapter 2 shows you how to use the iC-86/286/386 compiler, and Chapter 5 explains the listing files.

Exploring Advanced Features

Chapter 4 discusses memory segmentation models, and Chapter 9 describes how to extend the segmentation models with subsystems. Chapter 6 contains information on header files that provide access to processor architectural features, and Chapter 7 explains how to use an assembler header file to aid interfacing iC-86/286/386 code with ASM code.

Finding Reference Information

Chapter 3 contains reference information about controls. Chapters 8 and 10 contain reference information about calling conventions, data types, keywords, and language implementation (including conformance to the 1989 ANSI C standard). Chapter 11 lists the messages that appear in print files and on screen. Following the Installation section, the Glossary lists definitions for terms used in this manual.

Contents

1

Overview

Overview

1

This chapter provides an overview of the features of the iC-86, iC-286, and iC-386 compilers (referred to as iC-86/286/386) and their role in developing applications. References throughout the chapter direct you to more detailed information. This chapter contains information on the following topics:

- development of an application using an iC86/286/386 compiler and related Intel development tools
- compiler capabilities
- compatibility with other translators and utilities
- this manual and related publications
- trademarks

1.1 Software Development With iC-86/286/386

The iC-86/286/386 compilers supports modular, structured development of applications. Figure 1-1 shows the development path using the iC-86/286/386 compilers. Some of the tasks in developing a modular, structured iC-86/286/386 application are as follows:

- Compile and debug application modules separately.
- Select appropriate optimizations for the code.
- Use LINK86, BND286, or BND386 to link or bind the compiled modules and libraries to create a loadable file. See Chapter 2 in this manual for examples of linking and binding. Use LOC86 or a system builder (BLD286 or BLD386), to create a bootloadable file.
- Use OH86 or OH386 to prepare the code for programming into ROM.

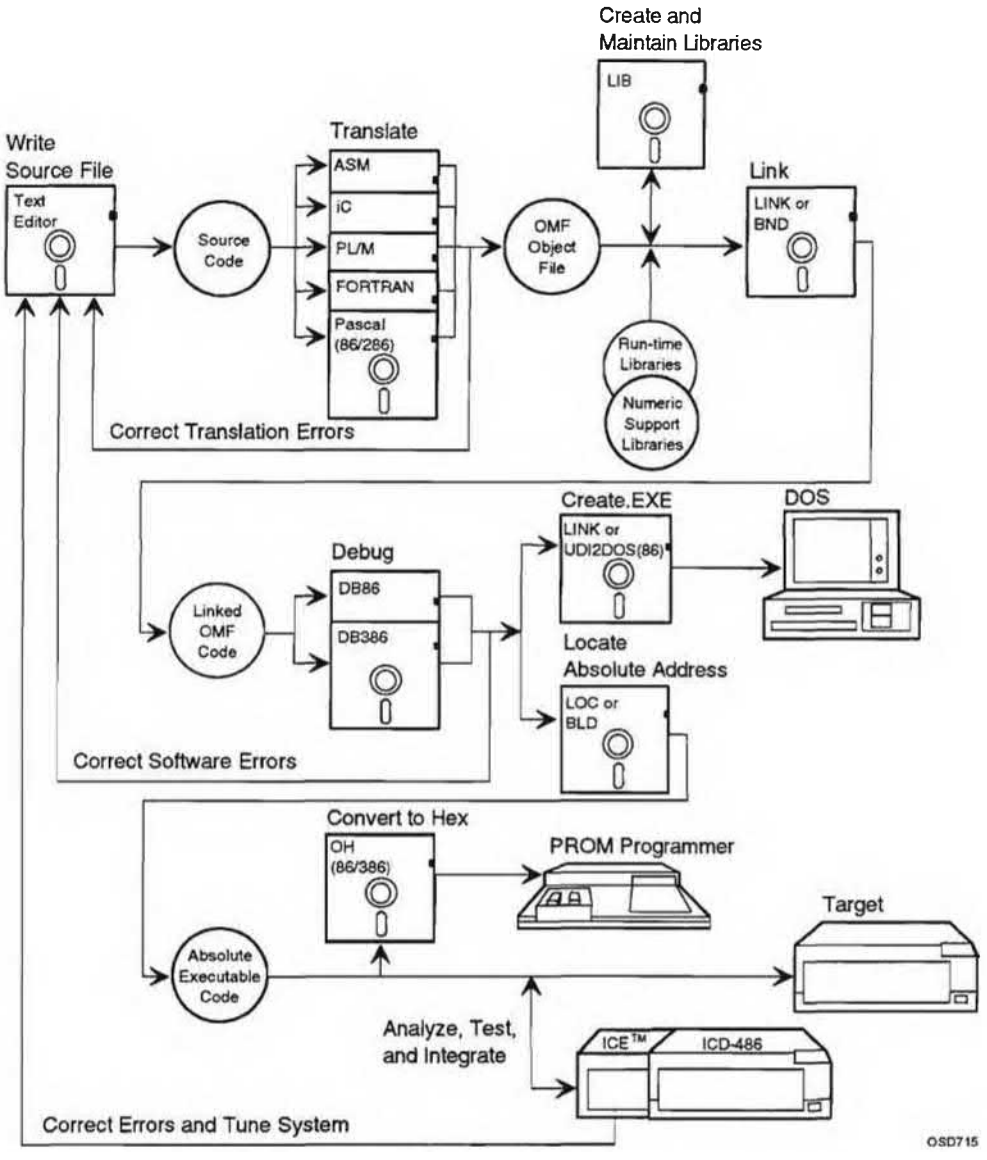

Figure 1-1 Developing an iC-86/286/386 Application

1.1.1 Using the Run-time Libraries

iC-86/286/386 includes a run-time library that supports the entire ANSI C library definition and provides a useful variety of supplementary functions and macros. These supplementary library facilities are defined by the IEEE Std 1003.1-1988 Portable Operating System Interface for Computer Environments (POSIX), the AT&T System V Interface Definition (SVID), or widely used non-standard libraries.

C: A Reference Manual describes the contents and use of the ANSI library. The *iC-86/286/386 Library Supplement* is a supplement to the C reference manual and to your *iC-86/286/386 User's Guide for DOS Systems*. The library supplement describes the iC-86/286/386 run-time libraries, and provides the detailed description of supplementary functions and macros.

See the Getting Started section of the *iC-86/286/386 Library Supplement* to decide how to proceed with using the libraries. See Chapter 2 of this *iC-86/286/386 Compiler User's Guide for DOS Systems* for the names of the library files provided, and for linking and binding information.

1.1.2 Debugging

At logical stages in the application development, use a symbolic debugger (such as DB86 or DB386) or an in-circuit emulator to debug and test the application. iC-86/286/386 supports debugging by enabling you to specify the amount of symbolic information in the object code and to customize the output listing. See Chapter 3 for detailed information on each control. Use the following controls when compiling modules for debugging:

- The `preprint` control creates a listing file of the code after preprocessing but before translation.
- The `type` control includes function and data type definition (`typedef`) information in the object file for intermodule type checking and for debuggers.
- The `debug` control includes symbolic information in the object file which is used by Intel symbolic debuggers and emulators.
- The `line` control includes source-line number information in the object file, which debuggers use to associate source code with translated code.

- The `code` control generates a pseudo-assembly language listing of the compiled code.
- The `optimize(0)` control ensures the most obvious match between the source text and the generated object code.
- The listing selection and format controls customize the contents and appearance of the output listings.
- The debugging information generated by the iC-86/286/386 compilers is compatible with current versions of Intel high-level debuggers and in-circuit emulators capable of loading Intel's object module format (OMF).

1.1.3 Optimizing

Optimized code is more compact and efficient than unoptimized code. The iC-86/286/386 compilers have several controls to adjust the level of optimization performed on your code. See Chapter 3 for detailed information on each control. The following controls adjust optimization:

- The `align|noalign` control specifies whether to generate aligned data structures that use more space than non-aligned structures, but permit quicker memory access.
- The `optimize` control specifies the level of optimization the compiler performs when generating object code. The iC-86/286/386 compilers provide four levels of optimization: 0, 1, 2, and 3; the higher the number, the more extensive the optimization. Object code generated with a higher level of optimization usually occupies less space in memory and executes faster than the code generated with a lower level of optimization. However, the compiler takes longer to generate code at a high level of optimization than at a low level. See Chapter 2 for examples of the code generated at each optimization level.
- The `small`, `compact`, `medium`, `large`, and `flat` controls set the memory segmentation model. For iRMX I and II applications, you can use only `compact` and `large`; for other iC-86/286 applications, you can also use `small` and `medium`. For iRMX III applications, you can use only `compact`; for other iC-386 applications, you can also use `small` and `flat`. See Chapter 3 for an explanation of each control and Chapter 4 for an explanation of each memory model.

1.1.4 Using the Utilities

The Intel utilities also support modular application development. A list of all the publications for the utilities is included at the end of this chapter. The following utilities aid in the software development process:

- LIB86, LIB286, or LIB386 organizes frequently used object modules into libraries. See the *86,88 Utilities User's Guide*, the *286 Utilities User's Guide*, or the *Intel386™ Family Utilities User's Guide* for information on LIB86, LIB286, or LIB386, respectively.
- LINK86, BND286, or BND386 links or binds together object modules from Intel translators. The linker or binders produce a relocatable, loadable module or a module for incremental binding. See the *86,88 Utilities User's Guide*, the *286 Utilities User's Guide*, or the *Intel386™ Family Utilities User's Guide* for information on LINK86, BND286, or BND386, respectively.
- CREF86 lists 86 intermodule cross-references. See the *86,88 Utilities User's Guide* for information on CREF86.
- MAP286 or MAP386 creates feature descriptions of 286 or 386™ object modules. See the *286 Utilities User's Guide* or the *Intel386™ Family Utilities User's Guide* for information on MAP286 or MAP386, respectively.
- LOC86 changes a relocatable 86 object module into an absolute object module. See the *86,88 Utilities User's Guide* for information on LOC86.
- BLD286 or BLD386 builds an executable bootloadable 286 or 386 system. See the *286 Utilities User's Guide* or the *Intel386™ Family System Builder User's Guide* for information on BLD286 or BLD386, respectively.
- LINK86 or OVL286 divides large 86 or 286 programs into overlays. See the *86,88 Utilities User's Guide* for information on LINK86 or the *286 Utilities User's Guide* for information on OVL286.
- OH86 or OH386 converts object code into hexadecimal form for programming for ROM. See the *86,88 Utilities User's Guide* or the product release notes for the Intel386 family utilities for information on OH86 or OH386, respectively.

1.1.5 Programming for Embedded ROM Systems

Use the `rom` compiler control to locate constants with code in the object module. See the `ram | rom` control entry in Chapter 3 for more information on the `rom` control. Link or bind your object modules with startup code tailored for an embedded ROM environment. See Chapter 2 for information on customizing the startup code. Use the LOC86, BLD286, or BLD386 utility to assign absolute addresses to your linked application.

Absolutely located Intel OMF object code is ready to use with the Intel iPPS PROM programming software. The OH86 and OH386 utilities convert absolute OMF86 or OMF386 code into hexadecimal form for use with non-Intel PROM programming utilities.

1.1.6 Running iC-86 Code Under DOS

Either of two DOS-hosted utilities create DOS-executable (.EXE) files, as follows:

- Use the LINK86 `exe` control (available in LINK86 V3.0 and above) when linking to create a DOS-executable file.
- Use the UDI2DOS operating system interface utility to convert an OMF86-loadable file to a DOS-executable file.

See the 86/88 utilities manuals and associated release notes for information on the LINK86 `exe` control and on using UDI2DOS.

1.2 Compiler Capabilities

The iC-86, iC-286, and iC-386 compilers translate C source files and produce code for the 86, 88, 186, or 188 processors; for the 286 processor; or for the Intel386 or Intel486™ processors, respectively.

The executable programs can be targeted for the following environments:

- an 86, 88, 186, or 188 processor-based system running the DOS operating system
- a 286, Intel386, or Intel486 processor-based system executing in real mode and running the DOS operating system
- an 86/88/186/188, 286, or Intel386/Intel486 processor-based system running the iRMX® I, II, or III operating system, respectively
- a custom-designed 86/88/186/188, 286, or Intel386/Intel486 processor-based system

The iC-86/286/386 instruction sets are fully upward compatible, but they are not downward compatible. Table 1-1 shows how to use iC-86 to produce efficient code for PCs running the DOS operating system.

Table 1-1 Using iC-86 For DOS Applications

PC Processor	Numeric Coprocessor	iC-86 Compiler Controls
86 or 88	none or 8087	mod86 and nomod287
186 or 188	none or 8087	mod186 and nomod287
80C186	none or 80C187	mod186 and modc187
286 or i386™ (real mode)	none	mod186 and nomod287
286 or i386 (real mode)	i287™	mod186 and mod287
i386 (real mode)	i387™	mod186 and modc187
i486™ (real mode)	on-chip FPU	mod186 and modc187

The iC-86 compiler generates floating-point instructions for the 8087, 80C187, or the Intel287™ numeric coprocessor; see the `mod287 | modc187 | nomod287` controls in Chapter 3. The iC-286 compiler generates

floating-point instructions for the Intel287 numeric coprocessor. The iC-386 compiler generates floating-point instructions for the Intel387™ numeric coprocessor and the Intel486 processor floating-point unit.

The iC-86/286/386 compilers and libraries conform to the 1989 American National Standard for Information Systems - Programming Language C (ANS X3.159-1989), and provides some useful extensions enabled by the `extend` compiler control. See Chapter 3 for information on the `extend` control. See Chapter 10 for a detailed discussion of the iC-86/286/386 implementation of the C programming language.

1.3 Compatibility With Other Development Tools

Table 1-2 shows the compatible Intel assemblers, compilers, debuggers, and utilities.

Table 1-2 Assemblers, Compilers, Debuggers, and Utilities

Tool	Tool Name for Each Processor		
	86/88/186/188 Family	286 Family	Intel386™, Intel486™ and 376™ Family
assembler	ASM86	ASM286	ASM386
C compiler	iC-86	iC-286	iC-386
FORTRAN compiler	Fortran-86	Fortran-286	Fortran-386
Pascal compiler	Pascal-86	Pascal-286	
PL/M compiler	PL/M-86	PL/M-286	PL/M-386
software debugger	DB86		DB386
linker or binder	LINK86	BND286	BND386
absolute locator	LOC86	BLD286	BLD386
librarian	LIB86	LIB286	LIB386
cross-reference utility	CREF86	MAP286	MAP386
overlay generator	LINK86	OVL286	
object-to-hex converter	OH86		OH386

The iC-86/286/386 compilers are largely compatible with previous Intel C compilers. The `extend` control enables the compilers to recognize the `alien`, `far`, and `near` keywords. See Chapter 3 for more information on the `extend` control. See Chapter 4 for information on the `far` and `near` keywords. See Chapter 10 for information on the `alien` keyword.

Modules compiled by the iC-86/286/386 compilers can refer to object modules created with Intel assemblers and other Intel compilers. Use only Intel compilers or translators to ensure compatibility with the memory segmentation model of the application. Chapter 4 explains memory segmentation models. Chapter 7 describes facilities that aid interfacing with assembler modules. Chapter 8 discusses the function-calling conventions of iC-86/286/386.

1.4 About This Manual

The *iC-86/286/386 Compiler User's Guide for DOS Systems* describes how to use the iC-86/286/386 compilers in the DOS environment. It is one of two iC-86/286/386 manuals: the other is the *iC-86/286/386 Library Supplement.* These manuals apply to Versions 4.5 and later of the iC-86/286/386 compilers and libraries and describe Intel extensions to the 1989 ANSI C standard.

The iC-86/286/386 manuals do not teach either programming techniques or the C language. Intel provides a book, *C: A Reference Manual*, by Harbison and Steele, that gives a complete description of the C programming language, recent extensions, the 1989 ANSI C standard, and standard run-time library functions.

1.4.1 Related Publications

Tables 1-3 through 1-5 identify additional publications that describe the other development tools you are most likely to use when programming with iC-86/286/386. (iC-86/286/386 manuals are described on the preceding page.) The tables also identify the programmer's reference manuals for the processors for which the iC-86/286/386 compilers generate object code. To order Intel publications, contact your local Intel field sales office or write to the Intel Literature Department, Intel Corporation, 3065 Bowers Avenue, Santa Clara, CA 95052.

Table 1-3 86/88 Tool and Processor Publications

Title	Number	Contents
An Introduction to ASM86	121689	introduces 86/88 assembly language
ASM86 Assembly Language Reference Manual	480774	assembly language for 86/88 processors
ASM86 Macro Assembler Operating Instructions	122390	assembler operation
86,88 Utilities User's Guide	122395	utilities for 86/88 processors
Operating System Interface Libraries Manual	480775	Universal Development Interface functions
8087 Support Library Reference Manual	480776	numeric coprocessor library reference
80C187 Support Library Reference Manual	483834	numeric coprocessor library reference
DB86 Software Debugger User's Guide	481850	software debugger operation
8086/8088 Programmer's and Hardware Reference	240487	architecture, assembly language, and hardware reference

Table 1-4 286 Tool and Processor Publications

Title	Number	Contents
ASM286 Assembly Language Reference Manual	122435	assembly language for the 286 processor
ASM286 Macro Assembler Operating Instructions	122440	assembler operation
286 Utilities User's Guide	122450	utilities for 286 processor
286 System Builder User's Guide	122445	utility for building complete systems
80287 Support Library Reference Manual	122460	Intel287™ numeric coprocessor libraries
80286 Programmer's Reference Manual	210498	286 architecture and assembly language
80286 Hardware Reference Manual	210760	hardware design of the 286 microprocessor

Table 1-5 Intel386™ and Intel486™ Tool and Processor Publications

Title	Number	Contents
ASM386 Assembly Language Reference Manual	480251	assembly language for the Intel386 and Intel486 processors
ASM386 Macro Assembler Operating Instructions for DOS Systems	451290	assembler operation in DOS environment
Intel386™ Family System Builder User's Guide	481342	utility for building complete systems
Intel386™ Family Utilities User's Guide	481343	utilities for binding, mapping, and maintaining libraries
80386 System Software Writer's Guide	231499	advanced programming guidelines
80387 Support Library Reference Manual	455497	numeric coprocessor libraries
386™ DX Microprocessor Programmer's Reference Manual	230985	Intel386 DX architecture and assembly language
387™ DX Microprocessor Programmer's Reference Manual	231917	Intel387™ DX coprocessor architecture and numerics assembly instructions

Table 1-5 Intel386™ and Intel486™ Tool and Processor Publications (continued)

Title	Number	Contents
386™ SL Microprocessor Superset Programmer's Reference Manual	240815	describes how to program a highly integrated SL SuperSet system
386™ SX Microprocessor Programmer's Reference Manual	240331	Intel386 SX architecture and assembly language
i486™ Programmer's Reference Manual	240486	Intel486 architecture and assembly language

See the *Customer Literature Guide*, order number 210620, to identify other appropriate user's guides and manuals.

1.5 Trademarks

Intel and iRMX are registered trademarks, and Intel386, Intel387, Intel287, Intel486, i386, i486, i287, ICE, and 376 are trademarks of Intel Corporation.

IBM and PC AT are registered trademarks and PC XT is a trademark of International Business Machines Corporation.

UNIX is a registered trademark of UNIX System Laboratories, Inc.

VAX and VMS are trademarks of Digital Equipment Corporation.

Contents

2

Compiling and Linking or Binding on DOS

Compiling and Linking or Binding on DOS

2

This chapter provides the information you need to compile and link an iC-86/286/386 program in the DOS environment. This chapter contains many examples, and you can study the examples that are most applicable to your development. If you are an experienced DOS user and have used other Intel development tools, the most important information you need is in Section 2.1.1, Invocation Syntax, and in Section 2.2.1, Linking or Binding iC-86/286/386 Object Files. Less experienced developers can obtain information on all of the following topics:

- Invoking the compiler: syntax, compiler messages, and the files that the compiler uses
- Using DOS batch and command files
- Linking or binding object files: general syntax, how to choose the libraries you need, examples, and how to customize the startup code
- Compiling an example program several ways: preprinting, exploring different instruction sets, examining included and conditional code, and creating type and debug information
- Compiling an example at different optimization levels

2.1 Compiler Invocation on DOS

This section describes the syntax for invoking the iC-86/286/386 compilers on DOS, the messages that the compilers display on the screen, and the files that the compiler uses.

2.1.1 Invocation Syntax

On DOS, the iC-86/286/386 compiler invocation has the following format:

```
[cdev:][cpath]icn86 [sdev:][spath]sfile [controls]
```

Where:

cdev:	is the disk drive (or virtual disk) that contains the compiler. If you do not specify a drive, DOS uses the current drive.
cpath	is the path to the directory that contains the compiler. If you do not specify a directory, DOS uses the current directory or searches directories specified in the DOS `path` command.
`icn86`	is the compiler itself. Use `ic86`, `ic286`, or `ic386`. Case is not significant.
sdev:	is the disk drive that contains the primary source file. If you do not specify a disk drive, DOS uses the current drive.
spath	is the path to the directory that contains the primary source file. If you do not specify a directory, DOS uses the current directory.
sfile	is the name of the primary source file; compilation starts with this file. This source file can cause other files to be included by using the `#include` preprocessor directive. See *C: A Reference Manual*, listed in Chapter 1, for information on the `#include` preprocessor directive.
controls	are the compiler controls. Separate consecutive controls with at least one space. Case is not significant in controls; however, case is significant in some control arguments. See Chapter 3 for the syntax of individual controls.

DOS limits the invocation line to 128 characters. If your screen width is less than 128 characters, an invocation command longer than the screen width automatically wraps to the next screen line. If you want to force an invocation line to continue on another screen line, type the ampersand continuation character (&) at the end of the first line, press the Enter key, and continue typing at the >> prompt on the next screen line, as shown in this example:

```
C:> \intel\ic386\ic386 &
>>  c:\applix\source\trial.c &
>>  object(c:\applix\obj\trial.out) &
>>  type &
>>  debug
```

DOS directory and filenames can be no longer than eight characters each preceding the optional period plus three-character extension. DOS truncates longer names from the right.

2.1.2 Sign-on and Sign-off Messages

The compiler writes information to the screen at the beginning and the end of compilation. On invocation, the compiler displays the following message:

```
system-id iC-n86 COMPILER Vx.y
Copyright years Intel Corporation
```

Where:

system-id	identifies your host operating system.
iC-*n*86	identifies the compiler as either iC-86, iC-286, or iC-386.
V*x.y*	identifies the version of the compiler.

On normal completion, the compiler displays the following message if the diagnostic level is 0:

```
iC-n86 COMPILATION COMPLETE. x REMARK[S], y WARNING[S], z ERROR[S]
```

Where:

`iC-`*n*`86`	identifies the compiler as either iC-86, iC-286, or iC-386.
x, *y* and *z*	indicate how many remarks, warnings, and non-fatal error messages, respectively, the compiler generated. If the diagnostic level is 1 (default), the message does not identify the number of remarks. If the `notranslate` control is in effect, the message does not appear. See Chapter 3 for more information on the `diagnostic` and `notranslate` controls.

On abnormal termination, the compiler displays one of the following messages:

```
iC-n86 FATAL ERROR --
message
COMPILATION TERMINATED
```

The print file lists the error that ended the compilation. If the `noprint` control is in effect, the compiler does not generate a print file and the console displays any diagnostics. See Chapter 3 for more information on the effects of the `diagnostic` control.

2.1.3 Files That the Compiler Uses

Output from the compiler usually consists of one object file and zero, one, or two listing files according to the compiler controls in effect. Figure 2-1 shows the input and output for files that iC-86/286/386 uses. The compiler also uses temporary work files during the compilation process. In your DOS `config.sys` file, the `files` specification controls the maximum number of files that DOS allows open at the same time. See the entries for the `preprint` and `include` controls in Chapter 3 for information on how many files iC-86/286/386 has open at the same time. The installation utility for iC-86/286/386 identifies necessary changes to your system configuration file. See the tabbed Installation section at the end of this manual.

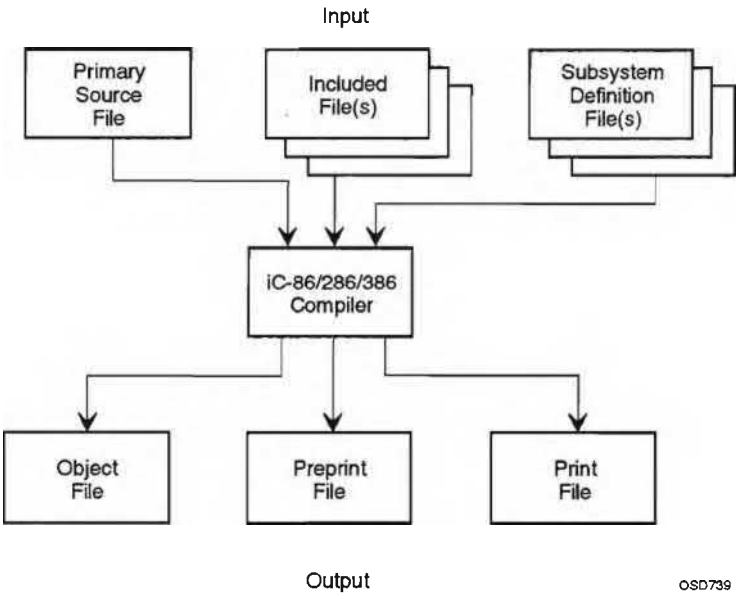

Figure 2-1 Compiler Input and Output Files

2.1.3.1 Work Files

The iC-86/286/386 compilers create and delete temporary work files during compilation. The compiler puts the work files either in the root directory of the `C:` drive or in the directory specified by the `:work:` DOS environment variable. To specify a RAM disk or specific directory for the compiler work files, set `:work:` to point to the specific path location. Using a RAM disk can decrease compilation time. For example, the following command directs the temporary files to the root directory on the `d:` drive:

```
C:> set :work:=d:
```

Be certain not to enter a space between the equals sign (=) and the DOS path designation, `d:` in this example. See your DOS documentation for information on RAM disks and environment variables.

If your host system loses power or some other abnormal event prevents the compiler from deleting its work files, you can delete the work files that remain. Such files have a filename consisting of a series of digits and no extension.

2.1.3.2 Object File

The compiler produces an object file by default. Use the `noobject` control or the `notranslate` control to suppress creation of an object file. See Chapter 3 for more information on the `noobject` and `notranslate` controls.

The default name for the object file is the same as the primary source filename with the `.obj` extension substituted. The compiler places the object file in the directory containing the source file by default. If a file with the same name already exists, the compiler writes over it. To override the filename or directory defaults, use the `object` control.

The object file contains the compiled object module, which is the relocatable code and data generated by the compiler as a result of a successful compilation. You can use many different compiler controls and preprocessor directives to specify the information content and configuration of the object module. See Chapter 3 for information on the object file controls. See *C: A Reference Manual*, listed in Chapter 1, for information on preprocessor directives.

2.1.3.3 Listing Files

The compiler can produce two listing files: a preprint file and a print file. The preprint file contains the source text after preprocessing. The print file can contain the source text and pseudo-assembly language code listings, messages, symbol table information, and summary information about the compilation. See Chapter 6 for more detailed information about the preprint and print files. See Chapter 3 for information about the `preprint` and `print` controls. Figure 2-2 summarizes the controls that create or suppress output files.

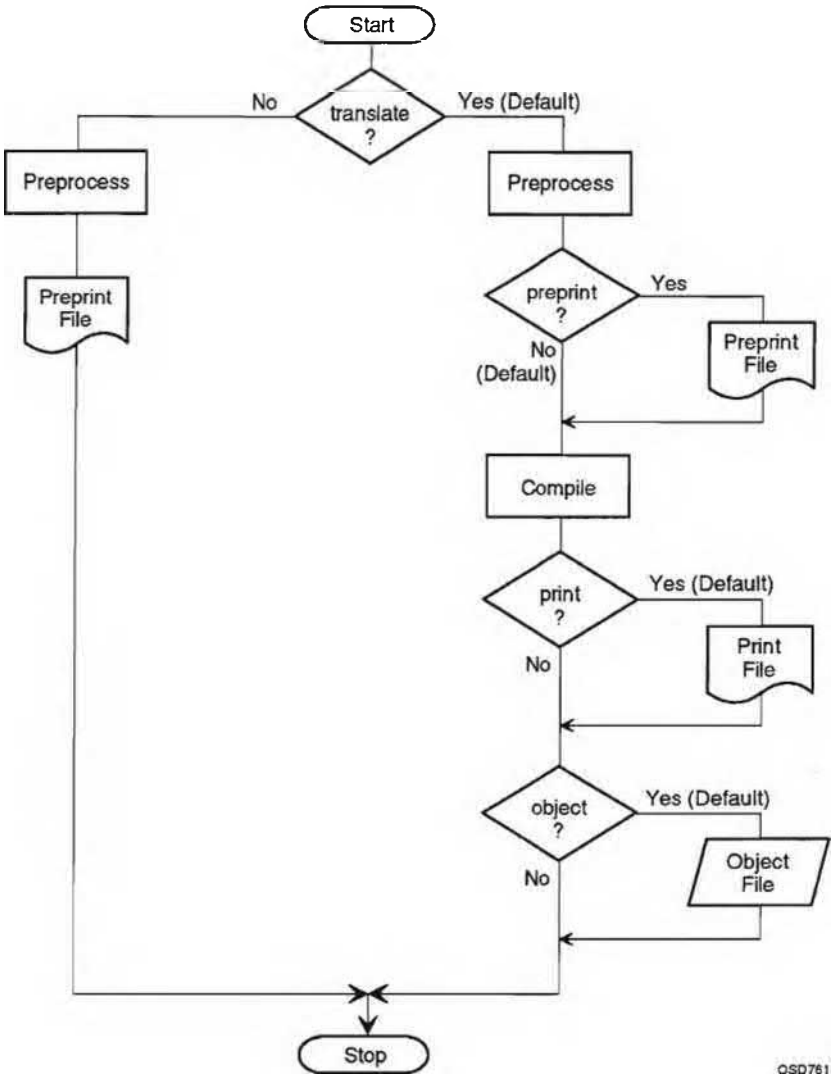

Figure 2-2 Controls That Create or Suppress Files

The compiler generates the preprint file only when the `preprint` or `notranslate` control is specified. The default name for the preprint file is the same as the primary source filename with the `.i` extension substituted. The compiler places the preprint file in the directory containing the source file by default. If a file with the same name already exists, the compiler writes over it. To override the filename or directory defaults, use the `preprint` control.

The preprint file contains an expanded source text listing. The preprint file is especially useful for observing the results of macro expansion, conditional compilation, and file inclusion. Compiling the preprint file produces the same results as compiling the source file, assuming the compiler can expand any macros without errors.

The compiler generates the print file by default. Use the `noprint` control to suppress the print file. The default name for the print file is the same as the primary source filename with the `.lst` extension substituted. The compiler places the print file in the directory containing the source file by default. If a file with the same name already exists, the compiler writes over it. To override the defaults, use the `print` control.

2.2 DOS Batch and Command Files

DOS offers two ways to invoke a series of commands automatically: batch files and command files. This section demonstrates how to use these files to simplify invoking the iC-86/286/386 compiler.

2.2.1 Using DOS Batch Files

A DOS batch file contains one or more commands that DOS executes consecutively. Batch file commands are valid at the DOS command-line prompt and include special commands that are valid only within a batch file. All batch files must have the `.bat` extension.

You can pass arguments to a DOS batch files. In the following example, the `386c.bat` batch file contains a command invoking the iC-386 compiler. Any primary source file with the `.c` extension can be the argument for `386c.bat`. The batch file contains one line:

```
C:\intel\ic386\ic386 %1.c
```

DOS replaces the `%1` parameter with the `prog1` argument in this example. To invoke the batch file, type the pathname of the batch file without its `.bat` extension followed by the name of the primary source file without its `.c` extension. For example:

```
C:> 386c prog1
```

When `386c.bat` executes, DOS replaces the `%1` parameter by `prog1`, resulting in the command:

```
C:\intel\ic386\ic386 prog1.c
```

DOS batch files have several other useful features, such as `if`, `goto`, `for`, and `call` commands. See your DOS documentation for explanations of these and other batch file commands.

Consider the following characteristics when developing a batch file for the iC-86/286/386 compiler:

- If a batch file directly invokes another batch file, control passes to the called batch file but does not return to the calling batch file. Place at most one direct batch file invocation as the last line in a batch file.
- An enhancement available in DOS V3.30 and successive versions enables one batch file to call another batch file and enables control to return to the original batch file. Use the `call` *`filename`* command.
- Batch files can contain command labels and control flow commands such as `if` and `goto`. For example, the following command allows the result of program execution from the previously executed batch file to determine at which label the current batch file continues execution:

  `if errorlevel` *`n`* `goto` *`label`*

The value of *n* is the error code that the last program returned. See the entry for the `diagnostic` control in Chapter 3 for more information on `errorlevel` values. If the error code is the same or greater than the value of *n*, control transfers to the line immediately after *`label`*. The label is any alphanumeric string significant up to eight characters, on its own line, and prepended by a colon.

- Although a batch file can contain multiple DOS commands, each command must fit on a single line (128 characters). You cannot use continuation lines in batch files. To process a longer line, specify a command to redirect input from a file containing the remainder of the line. The redirected file can contain continuation lines.

The following example shows how to redirect additional input from another file, how to use parameters, and how to call another batch file (in DOS 3.30). Figure 2-3 shows the relationships between the `386cl.bat` batch file, the `386cl.ltx` file of filenames, and the `make_map.bat` batch file. This example demonstrates the use of redirection and calling a batch file, and is not a functional example of how to compile and bind an iC-386 program.

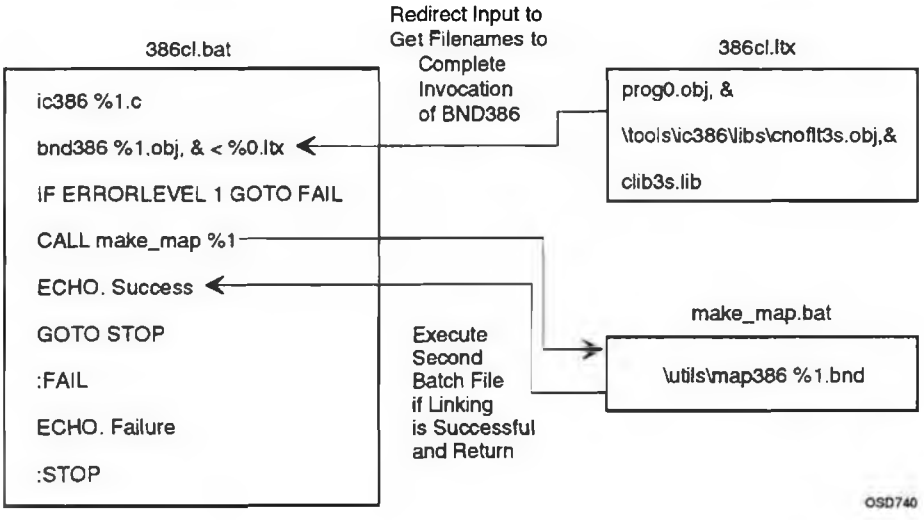

Figure 2-3 Redirecting Input to a DOS Batch File

The DOS batch file `%0` parameter always represents the name of the batch file itself (without the `.bat` extension). In the above example, since `386cl.bat` and `386cl.ltx` have identical names except for the extension, `386cl.bat` can refer to `386cl.ltx` as `%0.ltx`.

To execute the `386cl.bat` batch file and pass `prog1` as an argument, type the following at the DOS command prompt:

```
C:> 386cl prog1
```

When `386cl.bat` executes, it invokes the iC-386 compiler to compile `prog1.c`, then invokes BND386 to bind the resulting object module, `prog1.obj`, to another object module and a library specified in `386cl.ltx`. If the binding is successful, the `make_map.bat` file produces a map file named `prog1.map`.

2.2.2 Using DOS Command Files

You can invoke the DOS command processor, `command.com`, with input redirected from a file called a command file. A DOS command file contains a sequence of DOS commands and `exit` as the final command. Be certain that a carriage return follows the `exit` command, not an end-of-file character. See your DOS documentation for explanations of the `command` and `exit` commands.

For example, the `exemakec.cmd` command file contains the following commands (not a functional example of how to compile and bind an iC-286 program):

```
ic286 prog0.c
ic286 prog1.c
bnd286 progxs.obj, prog0.obj, prog1.obj, &
progxs.lib
exit
```

To sequentially execute the commands in the command file, redirect `exemakec.cmd` to `command.com` by typing the following at the DOS prompt:

```
C:> command < exemakec.cmd
```

Consider the following characteristics when developing a command file for the iC-86/286/386 compiler:

- This method of redirecting commands works for a command file containing a fixed sequence of commands only. You cannot pass arguments to a command file.
- The flow of control is always sequential, from top to bottom of the command file. Command files do not allow conditional commands such as `if` or `goto`.
- You can nest command files. If a command file reinvokes `command.com` with a secondary command file, control returns to the primary command file when the secondary command file exits. To invoke a second command file, insert a line in the first command file such as:

  ```
  command < comfile2.cmd
  ```

  The secondary command file must contain `exit` as its final command followed by a carriage return. If it does not, control does not return to the primary command file until you enter `exit` at the DOS prompt. Control returns to the point in the primary file immediately following the point from which the secondary file was invoked.
- Unlike batch files, command files can contain continuation lines.

If you invoke a command file with output redirected to a file, the command-line interpreter records all commands from the first line of the command file through the command `exit` and all console input and output to the file. For example, the following command invokes the `exemakec.cmd` command file and creates a log file named `exemakec.log` containing a record of all commands:

```
C:> command < exemakec.cmd > exemakec.log
```

2.3 Linking or Binding iC-86/286/386 Object Files

The iC-86/286/386 compiler supports modular, structured development of applications. You can compile and debug application modules separately, then bind them together to create an application. Use the LINK86 linker utility to combine separately translated object modules from the iC-86 compiler. Use the BND286 or BND386 binder utility to combine separately translated object modules from the iC-286 or iC-386 compiler, respectively.

The linker and binders can perform type checking and resolve intermodule references. The binders can automatically select modules from specified libraries to resolve references. See your *86,88 Utilities User's Guide*, the *286 Utilities User's Guide*, or the *Intel386™ Family Utilities User's Guide*, listed in Chapter 1, for complete information on LINK86, BND286, or BND386, respectively.

The general syntax for DOS-hosted LINK86, BND286, and BND386 (without device and path designations) is as follows:

```
link86 input_file_list to output_file [controls]

bnd286 input_file_list [controls]

bnd386 input_file_list [controls]
```

Where:

input_file_list	is one or more names of linkable files separated by commas. A linkable file is generated from a high-level language translator or assembler, or is an incrementally-linked module.
output_file	is the destination file for LINK86 that contains linked object code. (Use the `object` control for BND286 or BND386 to specify a non-default destination file.)
controls	are the linker or binder controls separated by spaces. See the *86,88 Utilities User's Guide for DOS Systems*, the *286 Utilities User's Guide for DOS Systems*, or the *Intel386™ Family Utilities User's Guide*, listed in Chapter 1, for complete information on DOS-hosted LINK86, BND286, or BND386 controls, respectively.

2.3.1 Choosing the Files to Link or Bind

iC-86/286/386 applications can consist of many separately translated modules. The applications can call functions from libraries. To create an executable file, you must use a linker or binder to link all translated code and libraries together. iC-86/286/386 includes several libraries, and you can purchase other libraries, such as the numeric coprocessor libraries included with the appropriate assembler, and you can create your own libraries.

Table 2-1 shows the iC-86 libraries. The `?` character represents `c`, `l`, `m`, or `s`, indicating compact-, large-, medium-, or small-model libraries. Use the `cdosnf?.lib`, `crmxnf1?.lib`, and `clibnf?.lib` libraries for more compact code when your program does not use floating-point numbers.

Table 2-1 iC-86 Libraries

Library Name	Model	Description
clibs.lib clibc.lib clibm.lib clibl.lib	small compact medium large	C run-time library containing all functions that are independent of the target operating system environment
clibnfs.lib clibnfc.lib clibnfm.lib clibnfl.lib	small compact medium large	C run-time library functions that are independent of the target operating system environment except functions that use floating-point numbers
cdoss.lib cdosc.lib cdosm.lib cdosl.lib	small compact medium large	C run-time library containing all functions that interface to the DOS operating system, plus all functions in clib?.lib
cdosnfs.lib cdosnfc.lib cdosnfm.lib cdosnfl.lib	small compact medium large	C run-time library functions that interface to the DOS operating system except functions that use floating-point numbers, plus all functions in clibnf?.lib
crmx1c.lib crmx1l.lib	compact large	C run-time library functions that interface to the iRMX I operating system, plus all functions in clib?.lib
crmxnf1c.lib crmxnf1l.lib	compact large	C run-time library functions that interface to the iRMX I operating system except functions that use floating-point numbers, plus all functions in clibnf?.lib
clib87.lib	all models	C run-time library containing floating-point functions not resolved by the preceding floating-point libraries

Table 2-2 shows the iC-286 libraries. The `?` character represents `c`, `l`, `m`, or `s`, indicating compact-, large-, medium-, or small-model libraries. Use the `crmxnf2?.lib` and `clibnf2?.lib` libraries for more compact code when your program does not use floating-point numbers. Note that the iRMX® I and II C interface libraries are available only in the compact and large memory segmentation models.

Table 2-2 iC-286 Libraries

Library Name	Model	Description
clib2s.lib clib2c.lib clib2m.lib clib2l.lib	small compact medium large	C run-time library containing all functions that are independent of the target operating system environment
clibnf2s.lib clibnf2c.lib clibnf2m.lib clibnf2l.lib	small compact medium large	C run-time library functions that are independent of the target operating system environment except functions that use floating-point numbers
crmx2c.lib crmx2l.lib	compact large	C run-time library containing all functions that interface to the iRMX II operating system, plus all functions in clib2?.lib
crmxnf2c.lib crmxnf2l.lib	compact large	C run-time library functions that interface to the iRMX II operating system except functions that use floating-point numbers, plus all functions in clibnf2?.lib

Table 2-3 shows the iC-386 libraries. The `?` character represents `c`, `f`, or `s`, indicating compact-, flat-, or small-model libraries. Use the `crmxnf3c.lib` and `clibnf3?.lib` libraries for more compact code when your program does not use floating-point numbers. Note that the iRMX III C interface library supports only the compact memory segmentation model.

Table 2-3 iC-386 Libraries

Library Name	Model	Description
clib3s.lib clib3c.lib clib3f.lib	small compact flat	C run-time library containing all functions that are independent of the target operating system environment
clibnf3s.lib clibnf3c.lib clibnf3f.lib	small compact flat	C run-time library functions that are independent of the target operating system environment except functions that use floating-point numbers
crmx3c.lib	compact	C run-time library containing all functions that interface to the iRMX III operating system, plus all functions in clib3c.lib
crmxnf3c.lib	compact	C run-time library functions that interface to the iRMX III operating system except functions that use floating-point numbers, plus all functions in clibnf3c.lib

The library's segmentation model must be compatible with the application's segmentation model and whether you compiled with the `ram` or `rom` control. See Chapter 3 for a description of the `compact`, `flat`, `large`, `medium`, `small`, `ram`, and `rom` compiler controls. See Chapter 4 for a discussion of all the segmentation models for iC-86/286/386.

Figures 2-4 and 2-5 show how to select the segmentation model of the libraries for linking with your program.

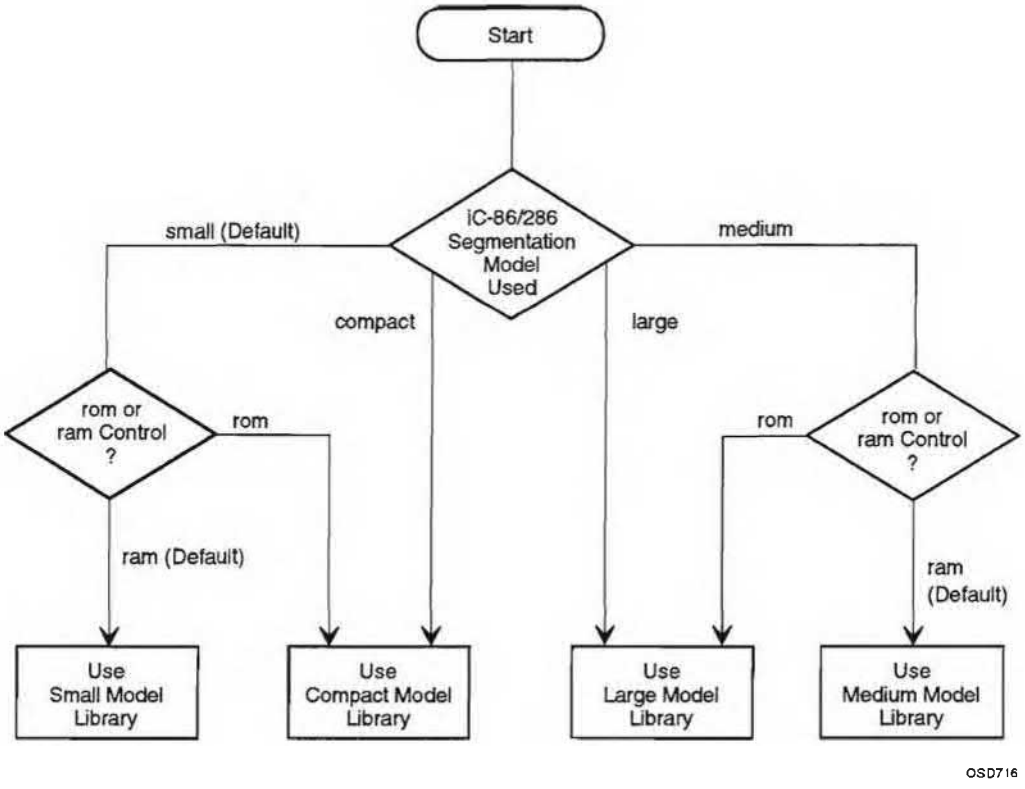

Figure 2-4 Choosing the Correct Segmentation Model of a Library for iC-86 or iC-286

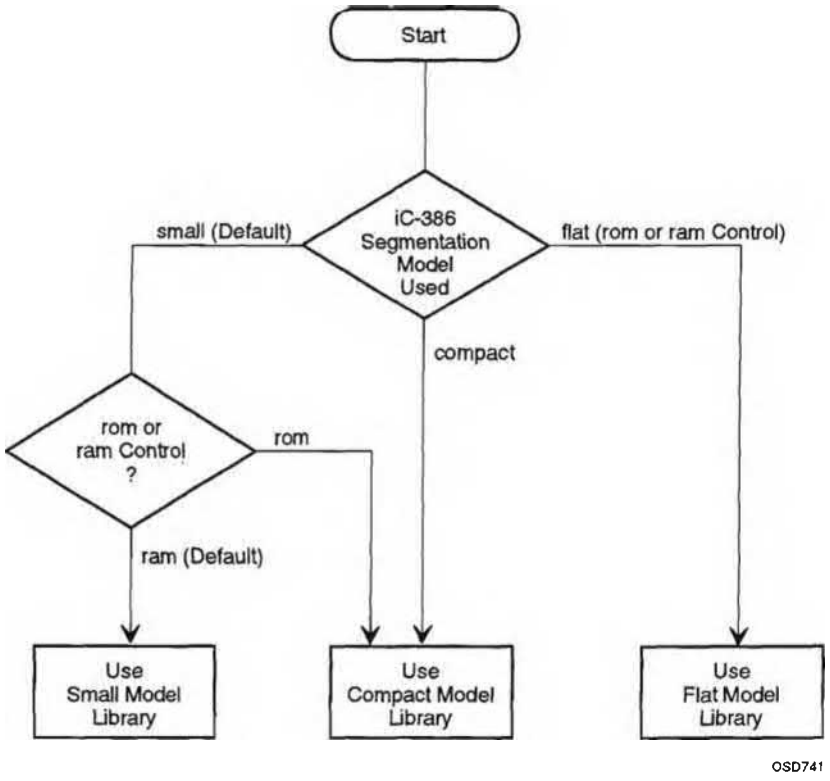

Figure 2-5 Choosing the Correct Segmentation Model of a Library for iC-386

Selecting the correct libraries depends upon whether the program:

- uses floating-point numbers
- uses an 8087 or 80C187 numeric coprocessor or emulator (iC-86)
- uses an Intel287 numeric coprocessor (iC-286)
- uses an Intel387 numeric coprocessor or emulator, or an Intel486 processor floating-point unit (iC-386)
- runs under DOS, an iRMX system, a different system, or no operating system

Figure 2-6 shows how to select the correct libraries for linking with iC-86 modules. The `ce187.lib`, `cl187f.lib`, `8087.lib`, and `80187f.lib` numeric support libraries and the 8087 and 80C187 emulators (`de8087`, `e8087`, and `e80187`) represented in Figure 2-5 are part of your ASM86 package.

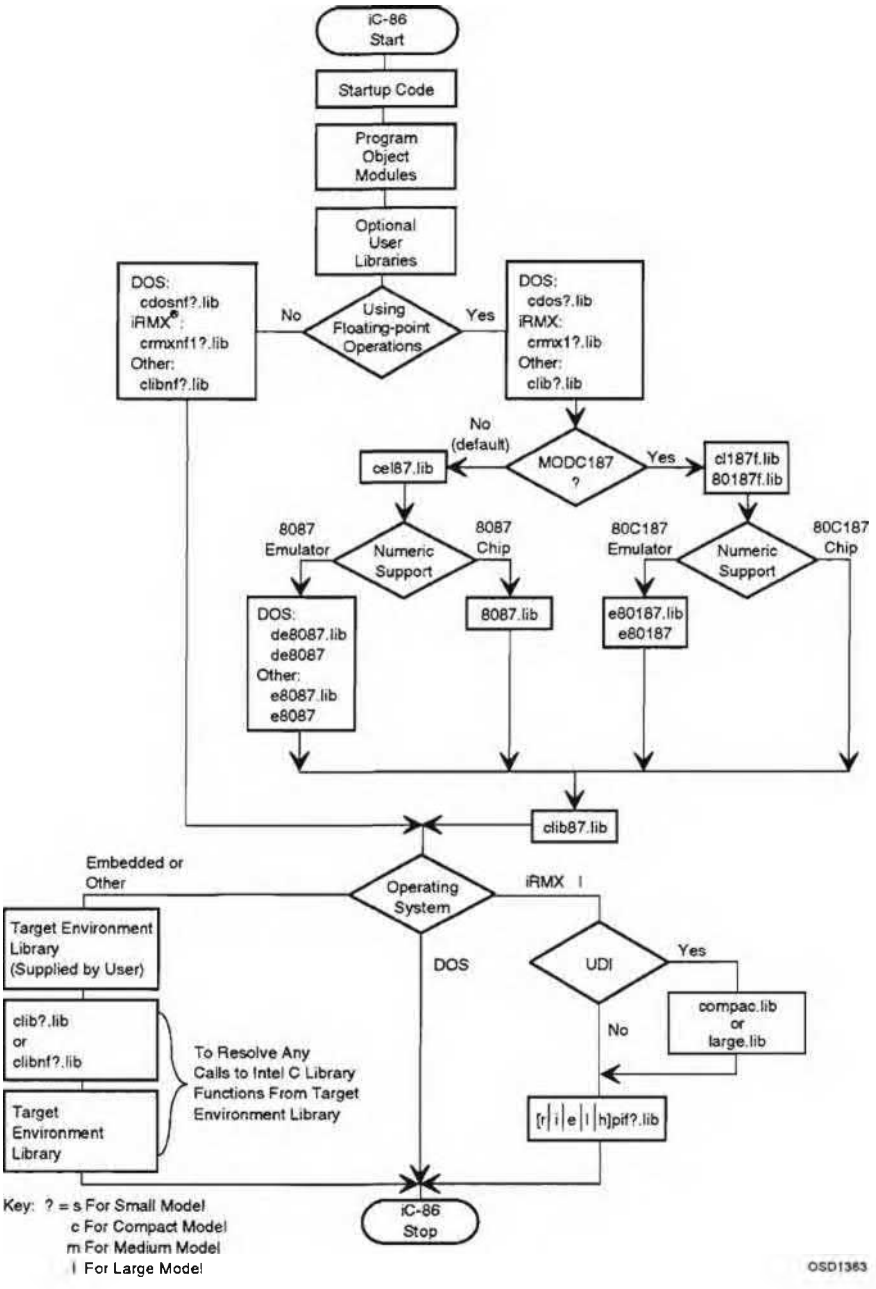

Figure 2-6 Choosing Libraries to Link with iC-86 Modules

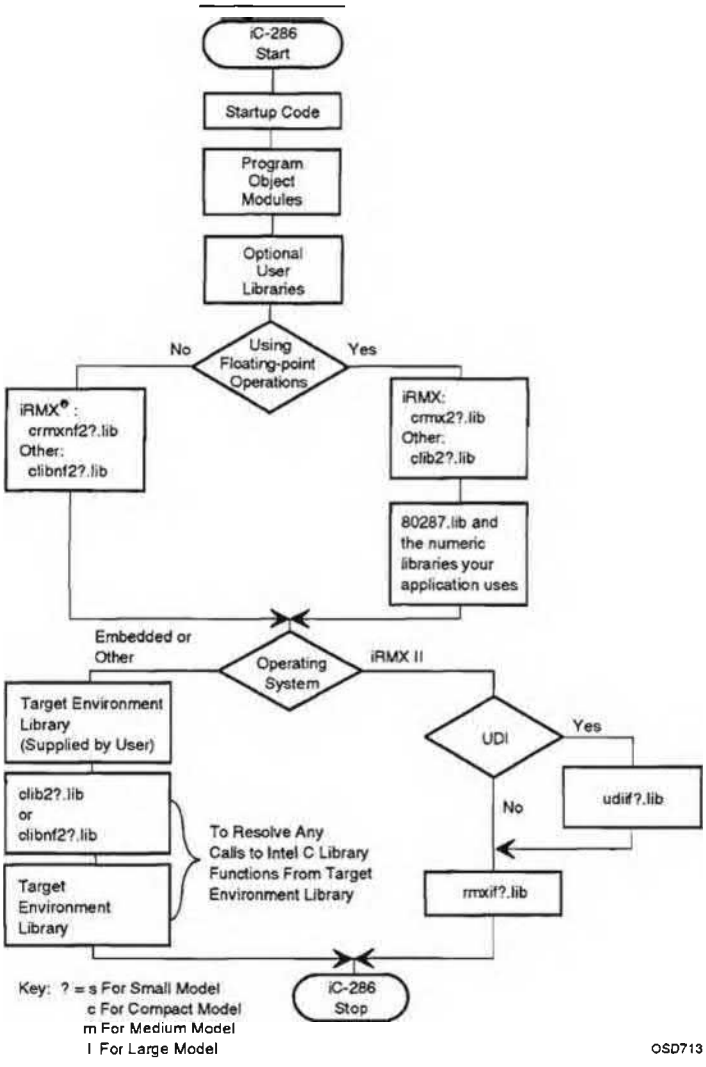

Figure 2-7 Choosing Libraries to Link with iC-286 Modules

Figure 2-7 shows how to select the correct libraries for linking with iC-286 modules. The `cel287.lib` and `80287.lib` numeric support libraries are part of your ASM286 package.

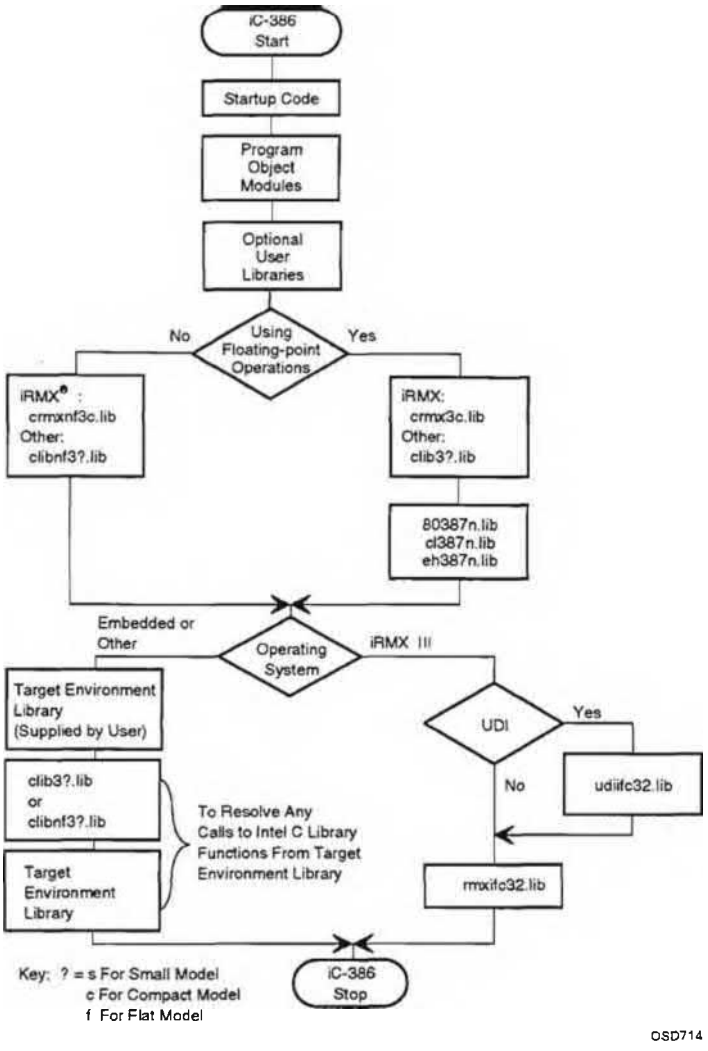

Figure 2-8 Choosing Libraries to Link with iC-386 Modules

Figure 2-8 shows how to select libraries for linking with iC-386 modules. The `80387n.lib`, `cl387n.lib`, and `eh387n.lib` files are part of the Intel387 Support Library.

2.3.1.1 LINK86 Example

This example program uses four C library functions to create and initialize a read-only file under the DOS operating system. The example assumes that all necessary files are in the current directory. Figure 2-9 shows the source code. The following line compiles the `ioexamp.c` source file:

```
C:> ic86 ioexamp.c
```

The following LINK86 invocation links the object module with the startup code and libraries and creates an executable file named `ioexamp.exe`:

```
C:> link86 cstdoss.obj, &
>>  ioexamp.obj,        &
>>  cdosnfs.lib         &
>>  to ioexamp.exe exe
```

Because the program performs no floating-point operations, LINK86 does not need the floating-point functions in `cdoss.lib` to resolve references. Using `cdosnfs.lib` produces a smaller object file.

```
/*
 * File Name:  ioexamp.c
 * This example program creates a file, writes to it, and
 * then closes it.  The mode of the file is then changed to
 * read-only.
 */
#include <fcntl.h>
#include <io.h>
#include <stdio.h>
#include <sys/stat.h>

int main (int argc, char * argv[])
{
     int fh;
     int result;
     char buffer [] = "Hello world!";
     char *pathname = "EXAMPLE.OUT";


     /*
      * Create the file.  If the file cannot be created, then report
      * the error to the user:
      */
     fh = open(pathname, O_CREAT|O_RDWR|O_BINARY, S_IREAD|S_IWRITE);
     if (fh != -1) {
          /*
           * Write the data to the file.  After closing the file,
           * change its mode to READ-ONLY:
           */
          result = write(fh, buffer, sizeof(buffer));
          close(fh);
          result = chmod(pathname, S_IREAD);
          printf("** read-only file was created\n");
     } else {
          /*
           * Report the error:
           */
          printf("** ERROR **:  unable to create file\n");
     }
}
```

Figure 2-9 Source Code for LINK86 Example

2.3.1.2 BND286 Example

This example shows a set of commands that builds one of the C library test suites. The test consists of a main routine (`lib1.c`), four primary test files (`test46.c`, `test47.c`, `test47b.c`, and `test48.c`) and a utility file (`util.c`). The example assumes that all necessary files are in the current directory.

The test itself consists of repeated calls to C library functions. The test runs under the iRMX II operating system. The iRMX interface library makes the appropriate system calls to implement the requested C library functions.

All iC-286 compiler invocations for this example use the compact segmentation model and optimization level zero. The compilation uses the `debug` control so that a source level debugger can help debug the executable file. The compiler invocations are as follows:

```
C:> ic286 lib1.c    compact optimize(0) debug
C:> ic286 test46.c  compact optimize(0) debug
C:> ic286 test47.c  compact optimize(0) debug
C:> ic286 test47b.c compact optimize(0) debug
C:> ic286 test48.c  compact optimize(0) debug
C:> ic286 util.c    compact optimize(0) debug
```

The BND286 invocation binds the object files with the appropriate libraries, as follows:

```
C:> bnd286        &
>>  cstrmx2c.obj, &
>>  lib1.obj,     &
>>  test46.obj,   &
>>  test47.obj,   &
>>  test47b.obj,  &
>>  test48.obj,   &
>>  util.obj,     &
>>  crmx2c.lib,   &
>>  cel287.lib,   &
>>  80287.lib,    &
>>  rmxifc.lib    &
>>  rconfigure    &
>>  object(lib1)
```

First the binder invocation lists the object modules for the C startup code and the six application modules. Next in the list is `crmx2c.lib`, which contains the C run-time library. Then the invocation lists the libraries for the numeric coprocessor (`cel287.lib` and `80287.lib`). The last library to link is the iRMX operating system interface library (`rmxifc.lib`). The `rconfigure` control tells BND286 to configure the object module for the iRMX II operating system. The `object` control names the executable file `lib1`.

The numeric coprocessor libraries are part of the ASM286 package. The iRMX C startup code (`cstrmx2c.obj`) and run-time library (`crmx2c.lib`) are part of iC-286 for VMS hosts. The iRMX II system interface library (`rmxifc.lib`) is part of the iRMX II operating system, and is not supplied with iC-286. If you use iC-286 to create applications that run under another operating system, bind in the startup code, libraries, and operating system interface library for that operating system instead.

2.3.1.3 BND386 Example

This example is cross-compiled to run under the iRMX III operating system and prints the string "Hello, world" on the screen. The example assumes that all necessary files are in the current directory. Figure 2-10 shows the source code. The following line compiles the `hello.c` source file:

```
C:> ic386 hello.c compact
```

```
/*
 * File Name:  hello.c
 * This example writes "Hello, world" on the screen.
 */
#include <stdio.h>


int main (int argc, char * argv[])
{
                printf("Hello, world\n");
}
```

Figure 2-10 Source Code for BND386 Example

The following BND386 invocation links the object module with the startup code and libraries and creates a loadable file named `hello`:

```
C:> bnd386          &
>>  cstrmx3c.obj, &
>>  hello.obj,     &
>>  crmxnf3c.lib, &
>>  rmxifc32.lib  &
>>  renameseg (CODE to CODE32) &
>>  rconfigure    &
>>  object (hello)
```

First, the binder invocation list must specify the object module for the C startup code and the application routine, in that order. Next, the binder links in the target-independent and interface functions library that does not use floating-point numbers (`crmxnf3c.lib`). Last, the binder links in the iRMX III operating system interface library (`rmxifc32.lib`).

The `renameseg` control ensures all library module code segments are named `CODE32`, for combining with iC-386 code segments. The `rconfigure` control causes BND386 to produce a single-task loadable module that can be loaded by the iRMX III system loader. The `object` control names the executable file `hello` instead of the default `hello.bnd`.

Because the program performs no floating-point operations, there are no function references to the floating-point routines in the `crmx3c.lib` library. Binding `crmxnf3c.lib` produces a smaller object file than using `crmx3c.lib`.

The iRMX III C interface libraries, `crmx3c.lib` and `crmxnf3c.lib`, are included with iC-386 for use with applications written for the iRMX III operating system. The iRMX III system interface library (`rmxifc32.lib`) is part of the iRMX III operating system, and is not supplied with iC-386. If you use iC-386 to create applications that run under another operating system, link in the startup code, libraries, and operating system interface library for that operating system instead.

The next example uses some floating-point arithmetic. The example assumes that all necessary files are in the current directory. Figure 2-11 shows the source code. The following line compiles the `float.c` source file:

```
C:> ic386 float.c compact
```

```
/*
 * File Name:  float.c
 * This example calculates the volume of a cylinder, such as a glass.
 * Volume = pi * (diameter / 2)^2 * height
 */
#include <stdio.h>
#include <math.h>
#define PI 3.14159

int main (int argc, char * argv[])
{
     float diam=3;
     int height=7;
     float vol;

     vol = PI * square(diam / 2) * height;
     printf("Your glass can hold %#7.2f cubic inches\n",vol);
}
```

Figure 2-11 Source Code for BND386 Floating-point Example

The BND386 invocation links the object modules and run-time libraries with the appropriate floating-point libraries, as follows:

```
C:> bnd386        &
>>  cstrmx3c.obj, &
>>  float.obj,    &
>>  crmx3c.lib,   &
>>  80387n.lib,   &
>>  cl387n.lib,   &
>>  eh387n.lib,   &
>>  rmxifc32.lib  &
>>  renameseg (CODE to CODE32) &
>>  rconfigure    &
>>  object (float)
```

The application uses the near version of the common elementary functions library. Because the application runs in the compact segmentation memory model, function calls are near calls. See Chapter 3 for a description of the `compact` control. See Chapter 4 for information on the segmentation memory models. See the *80387 Support Library Reference Manual*, listed in Chapter 1, for information on the numeric coprocessor libraries.

2.3.2 Customizing the Startup Code

The iC-86/286/386 package includes an assembly language file that performs several startup tasks: initializing the C library, initializing any hardware systems, invoking the `main()` function, and responding to the return from `main()` if the application does not call `exit()`. The source filename is `cstart.asm`. The startup code can be configured to perform tasks according to the needs of your target system.

The `cstart.asm` code uses many of the macros that the `util.ah` header file defines. Ensure that the DOS `:include:` environment variable is set to the path for `util.ah`. You can customize the startup code by using the `%define` macro definition facility for ASM when you assemble the startup code. See your DOS documentation for information on setting environment variables. See Chapter 7 for an explanation of the `util.ah` assembler header file.

The syntax for assembling the startup code (without device and path designations) is as follows:

```
asmn86 cstart.asm [asm_controls] %define(controls)([header_controls])
```

Where:

`asmn86`	is the assembler. Use `asm86`, `asm286`, or `asm386`.
`asm_controls`	is a sequence of assembler controls. See your *ASM86 Macro Assembler Operating Instructions for DOS*, *ASM286 Macro Assembler Operating Instructions for DOS*, or *ASM386 Macro Assembler Operating Instructions for DOS*, listed in Chapter 1, for information on assembler controls.
`header_controls`	is a sequence of special controls, separated by blanks. Select up to one header control from each of the sets shown in Table 2-4.

Table 2-4 ASM Header Controls for Customizing the Startup Code

Control Sets	Abbreviation	Default	Description
small	sm	X	small segmentation model
compact	cp		compact segmentation model
medium	md		medium segmentation model
large	la		large segmentation model
flat	fl		flat segmentation model
ram	(none)	X	RAM submodel (constants with data)
rom	(none)		ROM submodel (constants with code)
fixedparams	fp	X	FPL calling convention
varparams	vp		VPL calling convention
mod86	(none)	X	86 processor instructions
mod186	(none)		186 processor instructions
asm86	(none)	X	ASM86 assembler
asm286	(none)		ASM286 assembler
asm386	(none)		ASM386 assembler
'module=*name*'	(none)	cq_cstart	module name
'stacksize=*size*'	(none)	0	stack size
dos	(none)	X	DOS operating system
embedded	em		embedded system
rmx1	(none)		iRMX® I operating system
rmx2	(none)		iRMX II operating system
rmx3	(none)		iRMX III operating system

See Chapter 7 for definitions of all of the `header_controls` except the last set (`dos`, `embedded`, `rmx1`, `rmx2`, and `rmx3`). The controls in the last set are defined in the `cstart.asm` source code, not in `util.ah`. Choose one from the last set according to the target environment for your application.

The following examples demonstrate how to specify `header_controls` to produce typical startup code.

1. For a small-model program running under the DOS operating system and on an 86 processor, you can let the *header_controls* default. This sample assembler invocation produces an object file named `cstdoss.obj` and a listing file named `cstart.lst`:

```
C:> asm86 cstart.asm object(cstdoss.obj) &
>>  %define(controls)()
```

2. For a large-model program running on a ROM-based embedded 86 processor (without operating system support), specify three *header_controls*. This sample assembler invocation produces an object file named `cstembl.obj` and a listing file named `cstart.lst`:

```
C:> asm86 cstart.asm object(cstembl.obj) &
>>  %define(controls)(large rom embedded)
```

3. For a compact-model program running in an embedded 286 processor ROM environment (without operating system support), specify four *header_controls*. This sample assembler invocation produces an object file named `cstemb2c.obj` and a listing file named `cstart.lst`:

```
C:> asm286 cstart.asm object(cstemb2c.obj) &
>>  %define(controls)(compact rom asm286 embedded)
```

   The assembler produces an executable code fragment identical to the previous example's, but the 286 assembler generates different object code and segmentation directives depending on the segmentation memory model and target architecture.

4. For a small-model program running in an embedded RAM Intel386™ processor environment, specify `asm386` and `embedded` and let the remaining *header_controls* default. This sample assembler invocation produces an object file named `cst386em.obj` and a listing file named `cstart.lst`:

```
C:> asm386 cstart.asm object(cst386em.obj) &
>>  %define(controls)(asm386 embedded)
```

5. For a compact-model program running on a 386 processor under iRMX III, specify the `asm386`, `compact`, and `rmx3` *header_controls*. This sample assembler invocation produces an object file named `cstrmx3c.obj` and a listing file named `cstart.lst`:

```
C:> asm386 cstart.asm object(cstrmx3c.obj) &
>>  %define(controls)(asm386 compact rmx3)
```

2.4 Compiling an Example Different Ways

This section contains a sample program compiled using the iC-86, iC-286, and iC-386 compilers. The examples explore using preprocessor directives and many controls useful for iC-86, iC-286, or iC-386. Parts of the listing files explain the results of each compilation. See Chapter 3 for more detailed information on each control. See *C: A Reference Manual*, listed in Chapter 1, for information on preprocessor directives.

2.4.1 Example Files

Figure 2-12 shows the location of the files in the tree structure of the disk. The files and directories in this example are:

- `C:\cexample\sievec.c` is the primary source file.
- `C:\cexample\includes\prags.h` is a file that specifies two compiler controls in `#pragma` preprocessor directives: `small`, `extend`, and `optimize(0)`.
- `C:\intel\ic`*n*`86\` is the subdirectory, `ic86`, `ic286`, or `ic386`, containing the compiler.
- `C:\intel\ic`*n*`86\inc\` is the subdirectory containing standard include files, such as `stdio.h`.
- `C:\cexample\` is the current directory when the compiler is invoked.

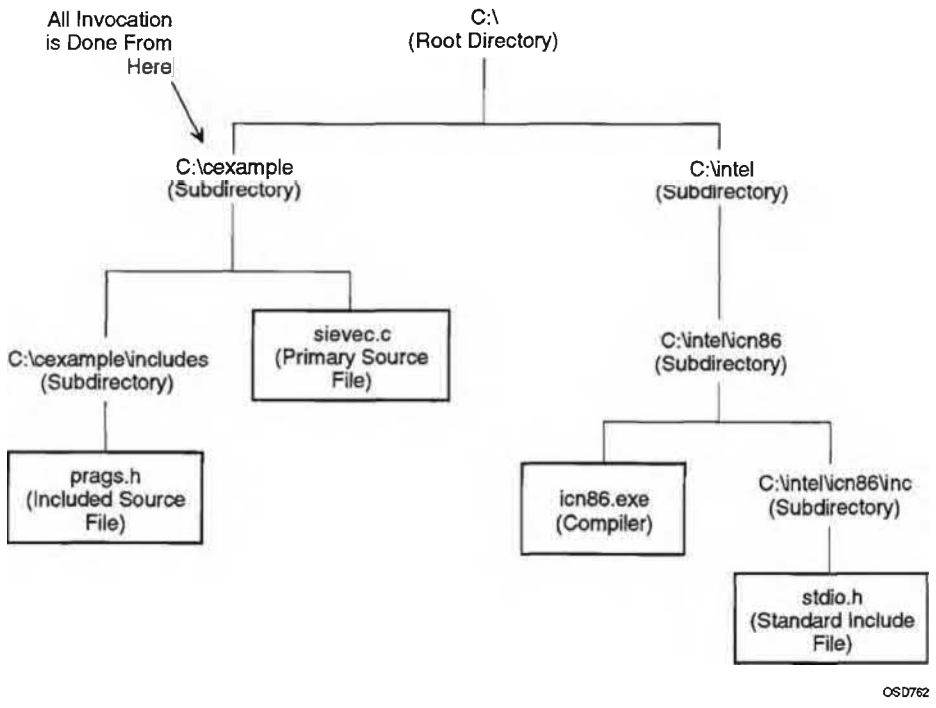

Figure 2-12 Directory Structure for Sieve Example Files

Figure 2-13 shows the complete source text from `sievec.c`. This program prints the prime numbers up to 8,190.

```
/*
 * File Name: sievec.c
 * This program computes prime numbers using the sieve method
 */

#if defined(EXAMPLE)
   #pragma title("Sieve Example")
#endif

#if defined(SCREEN)
   #pragma pagelength(24)
   #pragma pagewidth(80)
   #pragma tabwidth(3)
```

Figure 2-13 Source Code for Sieve Example

```
#elif defined(NPAPER)
   #pragma pagelength(40)
   #pragma pagewidth(75)
   #pragma tabwidth(2)
#endif

#include <stdio.h>

#define TRUE 1
#define FALSE 0
#define MAX 8190
#define EXECUTIONS 2
static char isprime[MAX+1];

int main (int argc, char * argv[])
{
   int i, aprime, j, howmany, n;
   for (n = 1; n <= EXECUTIONS; n++)
      {
         howmany = 0;
         for (i = 0; i <= MAX; i++)
            isprime[i] = TRUE;
         for (i = 2; i <= MAX; i++)
            {
               if (isprime[i])
               {
                  howmany++;
                  aprime = i;
                  for (j = i + aprime; j <= MAX; j += aprime)
                     isprime[j] = FALSE;
               }
            }
         for (i = 0; i <= MAX; i++)
            if (isprime[i]) printf(" %d",i);
      }
}
```

Figure 2-13 Source Code for Sieve Example (continued)

2.4.2 Preprinting the Example Using iC-86

This example discusses the controls and preprocessor directives that create a useful preprint file. Conditional compilation uses macros that the invocation defines. The source text includes a file that contains three `#pragma` preprocessor directives. Table 2-5 shows the controls in effect for the compilation. The `noobject` control overrides other object file controls. The `noprint` control overrides other print file controls. Even though the `noobject` control is in effect, translation occurs and the compiler reports the number of warnings and errors. The default setting of the `diagnostic` control suppresses the reporting of remark messages. See Chapter 3 for detailed information on each control.

This example uses the `preprint.cmd` command file to invoke the compiler. The full pathname to this command file is `C:\cexample\preprint.cmd`. The contents of `preprint.cmd` are as follows:

```
\intel\ic86\ic86 sievec.c &
define(NPAPER) &
include(prags.h) &
searchinclude(\intel\ic86\inc\,includes\) &
preprint &
noprint &
noobject &
define(EXAMPLE)
exit
```

The invocation of the DOS command processor accesses the command file and causes DOS to record the command session in the `preprint.log` file, as follows:

```
C:\CEXAMPLE> command < preprint.cmd > preprint.log
```

Table 2-5 Controls for Preprinting the Sieve Example

Controls	Where Specified
define(NPAPER)	invocation
define(EXAMPLE)	invocation
include(prags.h)	invocation
noobject	invocation
optimize(0)[1]	prags.h
pagelength(40)[2]	sievec.c
pagewidth(75)[2]	sievec.c
preprint	invocation
noprint	invocation
searchinclude(\intel\ic86\inc\,includes\)	invocation
small[1,4]	prags.h
tabwidth(2)[2]	sievec.c
title("Sieve Example")[2]	sievec.c
align[1]	default
nocode[2]	default
nocond[2]	default
nodebug[1]	default
diagnostic(1)	default
noextend[1]	default
fixedparams[1]	default
noline[1]	default
list[2]	default
nolistexpand[2]	default
nolistinclude[2]	default
mod86[1]	default
nomod287[1]	default
modulename(SIEVEC)[1]	default
ram[1]	default
signedchar[1]	default
nosymbols[2]	default
translate[3]	default
type[1]	default
noxref[1]	default

[1]The noobject and noprint controls override this control.
[2]The noprint control overrides this control.
[3]The preprint control overrides this control.
[4]This is the default segmentation model.

Figure 2-14 shows the contents of `preprint.log` after command processing.

```
operating-system-message

C:\CEXAMPLE> \intel\ic86\ic86 sievec.c &
>> define(NPAPER) &
>> include(prags.h) &
>> searchinclude(\intel\ic86\inc\,includes\) &
>> preprint &
>> noprint &
>> noobject &
>> define(EXAMPLE)

system-id iC-86 COMPILER Vx.y
Copyright years Intel Corporation
iC-86 COMPILATION COMPLETE.      Ø WARNINGS,     Ø ERRORS


C:\CEXAMPLE> exit
```

Figure 2-14 Command Log File for the Sieve Preprint Example

The `preprint` control causes the compiler to generate a preprint file. The `noprint` control causes suppression of the print file. The `noobject` control causes suppression of the object file. The only output file resulting from this compilation is the preprint file.

The preprint file contains the source text after preprocessing. Preprocessing includes expanding macros, conditionally selecting source text, and including other files.

The preprint file represents all files included by the `include` control and the `#include` preprocessor directive by the `#line` preprocessor directive followed by the included text. The `#line` preprocessor directive also appears in the preprint file to mark the first line of the primary source file.

Figure 2-15 shows the first few lines of the preprint file, `\cexample\sievec.i`, resulting from this compilation.

```
#line 1 "includes\prags.h"
#pragma small
#pragma optimize(0)
#line 1 "sievec.c"
/*
 * File Name: sievec.c
 * This program computes prime numbers using the sieve method
 */
   #pragma title("Sieve Example")
            #pragma pagelength(40)
   #pragma pagewidth(75)
   #pragma tabwidth(2)
#line 1 "\intel\ic86\inc\stdio.h"
/* stdio.h - standard I/O header file
.
.
.
```

Figure 2-15 Part of the Sieve Example Preprint File

2.4.2.1 Macros and Conditional Compilation

The `define` control in the compiler invocation defines the macros `NPAPER` and `EXAMPLE`. Case is significant in the arguments to the `define` control, so the macros are in uppercase to match the use of the macros in the source text.

The code of the primary source file specifies the conditional compilation of the following source text:

```
#if defined(EXAMPLE)
   #pragma title("Sieve Example")
#endif

#if defined(SCREEN)
   #pragma pagelength(24)
   #pragma pagewidth(80)
   #pragma tabwidth(3)
#elif defined(NPAPER)
   #pragma pagelength(40)
   #pragma pagewidth(75)
   #pragma tabwidth(2)
#endif
```

The `#if defined` preprocessor directive tests whether a macro name has been defined, without regard to the value of the macro. On invocation the `define` control defines `EXAMPLE`, so the preprocessor propagates to the preprint file the following line:

```
#pragma title("Sieve Example")
```

In another instance, the `define` control defines `NPAPER` but not `SCREEN`, so the preprocessor ignores the first set of directives under `#if defined(SCREEN)` and propagates the following lines:

```
#pragma pagelength(40)
#pragma pagewidth(75)
#pragma tabwidth(2)
```

The same source lines demonstrate that `#pragma` preprocessor directives can specify many compiler controls. However, the `title`, `pagelength`, `pagewidth`, and `tabwidth` controls have no effect in this example because they affect only the print file, which the `noprint` control in the compiler invocation suppresses.

2.4.2.2 Include Files

The `include` control in the compiler invocation causes the preprocessor to insert text from the `prags.h` file at the beginning of the primary source file, `sievec.c`. Since `prags.h` is in `\cexample\includes` and not in the current directory `\cexample`, the `searchinclude` control specifies the `includes\` relative path.

The `searchinclude` search path also specifies the `\intel\ic86\inc\` path, for the directory containing the `stdio.h` header file. In the source text, the `#include` preprocessor directive causes the preprocessor to insert text from the `stdio.h` header file at that point in the primary source file.

The `define` and `include` controls are valid only in the compiler invocation; they cannot be used in a `#pragma` preprocessor directive. Instead, use the `#define` and `#include` preprocessor directives. See *C: A Reference Manual*, listed in Chapter 1, for more information about preprocessor directives.

2.4.3 Creating 186 Code and a Custom Print File Using iC-86

This example creates an object file and a custom print file with a pseudo-assembly code listing and a cross-reference table of symbols. The compilation suppresses type-checking and debug information in the object module.

The `mod186` control causes the iC-86 compiler to generate object code using the 186/188 instruction set. (The iC-286 and iC-386 compilers do not use the `mod186` or `mod86` controls.) The 186 and 188 processors can execute this instruction set, but the 86 and 88 processors cannot. A 286, 386, or i486™ processor executing in real mode can also execute the 186/188 instruction set. Table 2-6 shows the controls in effect for the compilation.

This example uses the following compiler invocation:

```
C:\CEXAMPLE> \intel\ic86\ic86 sievec.c &
>> define(NPAPER) &
>> include(prags.h) &
>> searchinclude(\intel\ic86\inc\,includes\) &
>> object(prime.obj) &
>> mod186 &
>> notype &
>> nodebug &
>> print(prime.lst) &
>> nolist &
>> code &
>> xref &
>> diagnostic(Ø)
```

The compiler displays the following lines on the screen:

```
system-id iC-86 COMPILER Vx.y
Copyright years Intel Corporation
iC-86 COMPILATION COMPLETE.      Ø REMARKS,     Ø WARNINGS,     Ø ERRORS
```

Table 2-6 Controls for Creating the 186 Sieve Example

Controls	Where Specified
code	invocation
nodebug[1]	invocation
define(NPAPER)	invocation
diagnostic(0)	invocation
include(prags.h)	invocation
nolist	invocation
mod186	invocation
object(prime.obj)	invocation
optimize(0)	prags.h
pagelength(40)	sievec.c
pagewidth(75)	sievec.c
print(prime.lst)	invocation
searchinclude(\intel\ic86\inc\,includes\)	invocation
small[2]	prags.h
tabwidth(2)	sievec.c
notype	invocation
xref	invocation
align(4)	default
nocond	default
noextend	default
fixedparams	default
noline	default
nolistexpand	default
nolistinclude	default
nomod287	default
modulename(SIEVEC)	default
nopreprint	default
ram	default
signedchar	default
nosymbols[3]	default
title("SIEVEC")	default
translate	default

[1]This is the default setting for this control.
[2]This is the default segmentation model.
[3]The xref control overrides this control.

The `print` and `object` controls specify explicit file names for the print file and object file. The compiler puts the print and object files in the current directory, `C:\cexample`, by default.

The `xref` control causes the compiler to print a symbol table with cross-reference information in the print file. The `xref` control overrides the default `nosymbols` control. The cross-reference information associates each symbol with each line that defines it, declares it, or references it. The cross-reference line numbers are on the far right of the symbol table listing, under each entry in the ATTRIBUTES column.

The print file is `\cexample\prime.lst`. The `pagelength`, `pagewidth`, and `tabwidth` controls affect the format of the print file. The invocation does not define the EXAMPLE macro, so the title of the listing defaults to the module name. The `code` and `nolist` controls affect the contents of the print file. The `nolist` control overrides several controls that affect the source code listing. Figure 2-16 shows the first two pages of the print file, the first page of the symbol table listing from the print file, and the last page of the print file.

```
iC-86  COMPILER   SIEVEC                          mm/dd/yy hh:mm:ss  PAGE   1

system-id iC-86 COMPILER Vx.y, COMPILATION OF MODULE SIEVEC
OBJECT MODULE PLACED IN prime.obj
COMPILER INVOKED BY: C:\intel\ic86\IC86.EXE sievec.c define(NPAPER) include
                -(prags.h) searchinclude(\intel\ic86\inc\,includes\) object
                -(prime.obj) mod186 notype nodebug print(prime.lst) nolist
                -code xref diagnostic(Ø)

 line level  incl
```

Figure 2-16 Parts of the 186 Sieve Example Print File

ASSEMBLY LISTING OF OBJECT CODE

```
                                        ; STATEMENT # 29
                main      PROC NEAR
0000  C80A0000          ENTER   0AH,0H
                @1:
                                        ; STATEMENT # 32
0004  C746F60100        MOV     [BP].n,1H
0009  E9BA00            JMP     @4
                @2:
000C  C746F80000        MOV     [BP].howmany,0H
                                        ; STATEMENT # 35
0011  C746FE0000        MOV     [BP].i,0H
0016  E90F00            JMP     @8
                @6:
0019  8B5EFE            MOV     BX,[BP].i
001C  C687000001        MOV     isprime[BX],1H
                @7:
0021  8B46FE            MOV     AX,[BP].i
0024  40                INC     AX
0025  8946FE            MOV     [BP].i,AX
                @8:
0028  817EFEFE1F        CMP     [BP].i,1FFEH
002D  7F03              JG      $+5H
002F  E9E7FF            JMP     @6
                @9:
                                        ; STATEMENT # 37
0032  C746FE0200        MOV     [BP].i,2H
0037  E94800            JMP     @12
                @10:
003A  8B5EFE            MOV     BX,[BP].i
003D  82BF000000        CMP     isprime[BX],0H
0042  7503              JNZ     $+5H
0044  E93400            JMP     @14
                                        ; STATEMENT # 41
0047  8B46F8            MOV     AX,[BP].howmany
004A  40                INC     AX
004B  8946F8            MOV     [BP].howmany,AX
```

Figure 2-16 Parts of the 186 Sieve Example Print File (continued)

```
iC-86  COMPILER   SIEVEC                          mm/dd/yy hh:mm:ss  PAGE   5
                                    SYMBOL TABLE

  NAME              SIZE    CLASS    ADDRESS    ATTRIBUTES

  _exit             72      Tag                 struct
                                                *66
   open_stream_sem
                     2       Member   0          pointer to void
                                                *69
   open_stream_list
                     2       Member   2          pointer to struct _iobuf
                                                *70
   exit_handler_sem
                     2       Member   4          pointer to void
                                                *71
   exit_handler_count
                     2       Member   6          int
                                                *72
   exit_handler_list
                     64      Member   8          array[32] of pointer to func
                                                -tion returning void
                                                *73
  _heap              8       Tag                 struct
                                                *39
   _malloc_sem       2       Member   0          pointer to void
                                                *42
   _primary_free_list
                     2       Member   2          pointer to struct free_list_
                                                -item
                                                *43
   _secondary_free_list
                     2       Member   4          pointer to struct free_list_
                                                -item
                                                *44
   _secondary_list_count
                     2       Member   6          int
                                                *45
  _iobuf             18      Tag                 struct
.
.
.
MODULE INFORMATION:

     CODE AREA SIZE                = 00D4H    212D
     CONSTANT AREA SIZE            = 0004H      4D
     DATA AREA SIZE                = 1FFFH   8191D
     MAXIMUM STACK SIZE            = 001AH     26D

iC-86 COMPILATION COMPLETE.      0 REMARKS,     0 WARNINGS,     0 ERRORS
```

Figure 2-16 Parts of the 186 Sieve Example Print File (continued)

2.4.4 Creating 86 Code and Linking for DOS Using iC-86

This example creates an object file that is ready to link and run on an 86 processor. The target machine contains an i287 numeric coprocessor. The print file contains the source text and a pseudo-assembly code listing. Table 2-7 shows the controls in effect for the invocation.

Table 2-7 Controls for Creating the DOS Sieve Example

Controls	Where Specified
code	invocation
define(SCREEN)	invocation
include(prags.h)	invocation
mod287	invocation
object(sievedos.obj)	invocation
optimize(3)	prags.h
pagelength(24)	sievec.c
pagewidth(80)	sievec.c
print(sievedos.lst)	invocation
searchinclude(\intel\ic86\inc\,includes\)	invocation
small[1]	prags.h
tabwidth(3)	sievec.c
title("Link For DOS")	invocation
align(4)	default
nocond	default
nodebug	default
diagnostic(1)	default
noextend	default
fixedparams	default
noline	default
list	default
nolistexpand	default
nolistinclude	default
mod86	default
modulename(SIEVEC)	default
nopreprint	default
ram	default
signedchar	default
nosymbols	default
translate	default
type	default
noxref	default

[1]This is the default segmentation model.

This example uses the following compiler invocation:

```
C:\CEXAMPLE> \intel\ic86\ic86 sievec.c &
>> define(SCREEN) &
>> include(prags.h) &
>> searchinclude(\intel\ic86\inc\,includes\) &
>> code &
>> mod287 &
>> print(sievedos.lst) &
>> object(sievedos.obj) &
>> title("Link For DOS")
```

The compiler displays the following lines on the screen:

```
system-id iC-86 COMPILER Vx.y
Copyright years Intel Corporation
iC-86 COMPILATION COMPLETE.     Ø WARNINGS,     Ø ERRORS
```

The print file is `\cexample\sievedos.lst`. Figure 2-17 shows the print file.

```
iC-86  COMPILER   Link for DOS                    mm/dd/yy hh:mm:ss  PAGE   1

system-id iC-86 COMPILER Vx.y, COMPILATION OF MODULE SIEVEC
OBJECT MODULE PLACED IN sievedos.obj
COMPILER INVOKED BY: C:\intel\ic86\IC86.EXE sievec.c define(SCREEN) include(pr ag
             -s.h) searchinclude(\intel\ic86\inc\,includes\) code mod287 prin
             -t(sievedos.lst) object(sievedos.obj) title(Link for DOS)

line level  incl

   1             /*
   2              * File Name: sievec.c
   3              * This program computes prime numbers using the sieve method
   4              */
   5
   6             #if defined(EXAMPLE)
                 #endif
   9
  10             #if defined(SCREEN)
  11                #pragma pagelength(24)
  12                #pragma pagewidth(80)
  13                #pragma tabwidth(3)
  14             #elif defined(NPAPER)
```

Figure 2-17 Part of the DOS Sieve Example Print File

```
                #endif
  19
  20            #include <stdio.h>
  21
  22            #define TRUE 1
  23            #define FALSE 0
  24            #define MAX 8190
  25            #define EXECUTIONS 2
  26            static char isprime[MAX+1];
  27
  28            int main (int argc, char * argv[])
  29            {
  30     1         int i, aprime, j, howmany, n;
  31     1
  32     1         for (n = 1; n <= EXECUTIONS; n++)
  33     1            {
  34     2               howmany = 0;
  35     2               for (i = 0; i <= MAX; i++)
  36     2                  isprime[i] = TRUE;
  37     2               for (i = 2; i <= MAX; i++)
  38     2                  {
```



```
  39     3                     if (isprime[i])
  40     3                     {
  41     4                        howmany++;
  42     4                        aprime = i;
  43     4                        for (j = i + aprime; j <= MAX; j += aprime)
  44     4                           isprime[j] = FALSE;
  45     4                     }
  46     3                  }
  47     2               for (i = 0; i <= MAX; i++)
  48     2                  if (isprime[i]) printf(" %d",i);
  49     2            }
  50     1      }
.
.
.
MODULE INFORMATION:

     CODE AREA SIZE              = 00DAH    218D
     CONSTANT AREA SIZE          = 0004H      4D
     DATA AREA SIZE              = 1FFFH   8191D
     MAXIMUM STACK SIZE          = 001AH     26D

iC-86 COMPILATION COMPLETE.      0 WARNINGS,     0 ERRORS
```

Figure 2-17 Part of the DOS Sieve Example Print File (continued)

LINK86 links the object module to the libraries and creates an executable file for DOS. The following LINK86 command assumes that the DOS search path knows where to find LINK86 and assumes that the libraries and startup code are in the `\intel\ic86` and `\intel\ic86\libs` directories, respectively.

```
C:\CEXAMPLE> link86              &
>> \intel\ic86\lib\cstdoss.obj, &
>> sievedos.obj,                &
>> \intel\ic86\lib\cdoss.lib,   &
>> \intel\ic86\lib\ce187.lib,   &
>> \intel\ic86\lib\8087.lib     &
>> to sieve.exe                 &
>> exe
```

To execute the program, type the program name at the DOS prompt, as follows:

```
C:\CEXAMPLE> sieve
```

2.4.5 Examining Included and Conditional Code Using iC-286

This example creates a print file that contains all of the source text, including uncompiled conditional code, the text from include files, and expanded macros. Table 2-8 shows the controls in effect for the invocation.

Table 2-8 Controls for Creating a Complete Print File for the Sieve Example

Controls	Where Specified
cond	invocation
diagnostic(0)	invocation
define(NPAPER)	invocation
include(prags.h)	invocation
listexpand	invocation
listinclude	invocation
optimize(0)	prags.h
pagelength(40)	sievec.c
pagewidth(75)	sievec.c
searchinclude(\intel\ic386\inc\,includes\)	invocation
small[1]	prags.h
tabwidth(2)	sievec.c
title("Long Listing")	invocation
align(4)	default
nocode	default
nodebug	default
noextend	default
fixedparams	default
noline	default
list	default
modulename(SIEVEC)	default
object(sievec.obj)	default
nopreprint	default
print(sievec.lst)	default
ram	default
signedchar	default
nosymbols	default
translate	default
type	default
noxref	default

[1]This is the default segmentation model.

This example uses the following compiler invocation:

```
C:\CEXAMPLE> \intel\ic286\ic286 sievec.c &
>> define(NPAPER) &
>> include(prags.h) &
>> searchinclude(\intel\ic286\inc\,includes\) &
>> cond &
>> listexpand &
>> listinclude &
>> diagnostic(Ø) &
>> title("Long Listing")
```

The compiler displays the following lines on the screen:

```
system-id iC-286 COMPILER Vx.y
Copyright years Intel Corporation
iC-286 COMPILATION COMPLETE.      Ø REMARKS,     Ø WARNINGS,     Ø ERRORS
```

The print file is `\cexample\sievec.lst` by default. The source text listing includes uncompiled conditional code and the contents of included files. Figure 2-18 shows the first two pages and the last two pages of the print file.

```
iC-286  COMPILER   Long Listing                  mm/dd/yy hh:mm:ss  PAGE   1


system-id iC-286 COMPILER Vx.y, COMPILATION OF MODULE SIEVEC
OBJECT MODULE PLACED IN sievec.obj
COMPILER INVOKED BY: C:\intel\ic286\IC286.EXE sievec.c define(NPAPER) inclu
                -de(prags.h) searchinclude(\intel\ic286\inc\,includes\) con
                -d listexpand listinclude diagnostic(Ø) title(Long Listing)


 line level  incl

    1        1    #pragma small
    2        1    #pragma optimize(Ø)
    1             /*
    2              * File Name: sievec.c
    3              * This program computes prime numbers using the sieve me
                -thod
    4              */
    5
    6             #if defined(EXAMPLE)
                     #pragma title("Sieve Example")
                  #endif
    9
   1Ø             #if defined(SCREEN)
                     #pragma pagelength(24)
                     #pragma pagewidth(8Ø)
                     #pragma tabwidth(3)
                  #elif defined(NPAPER)
   15                #pragma pagelength(4Ø)
   16                #pragma pagewidth(75)
   17                #pragma tabwidth(2)
   18             #endif
   19
   2Ø             #include <stdio.h>
    1        1    /* stdio.h - standard I/O header file
.
.
.
```

Figure 2-18 Parts of the Sieve Example Complete Print File

```
                -ESERVED.
   5        1     */
   6        1
   7        1    #ifndef _stdioh
   8        1    #define _stdioh
   9        1    /*lint -library */
  10        1
  11        1    /*  ../cdos/stdio:  */
  12        1    #pragma fixedparams("rename", "tempnam", "tmpnam")
  13        1
  14        1    /*  ../cflt/stdio:  */
  15        1    #pragma fixedparams("_dtobcd", "_dtos", "_pow_10")
  16        1
  17        1    /*  ../clib/stdio:  */
  18        1    #pragma fixedparams("_putch", "_getch")
  19        1    #pragma fixedparams("_doprnt", "_doscan", "_filbuf", "_fl
                -sbuf", "clearerr")
  20        1    #pragma fixedparams("fclose", "feof", "ferror", "fflush",
                - "fgetc")
  21        1    #pragma fixedparams("fgets", "fopen", "fputc", "fputs", "
                -fread")
  22        1    #pragma fixedparams("freopen", "fseek", "ftell", "fwrite"
                -, "getc")
  23        1    #pragma fixedparams("getchar", "gets", "perror", "putc",
                -"putchar")
  24        1    #pragma fixedparams("puts", "remove", "rewind", "setbuf",
                - "setvbuf")
  25        1    #pragma fixedparams("tmpfile", "ungetc", "vfprintf", "vpr
                -intf", "vsprintf")
  26        1    #pragma fixedparams("fgetpos", "fsetpos")
  27        1
  28        1    #pragma varparams("fprintf", "fscanf", "printf", "scanf",
                - "sprintf", "sscanf")
  29        1
  30        1    /*  ../clib/stdio.ext:  */
  31        1    #pragma fixedparams("fcloseall", "fdopen", "fgetchar", "f
                -ileno", "flushall")
```

Figure 2-18 Parts of the Sieve Example Complete Print File (continued)

```
 23              #define FALSE 0
 24              #define MAX 8190
 25              #define EXECUTIONS 2
 26              static char isprime[MAX+1];
  +              static char isprime[8190+1];
 27
 28              int main (int argc, char * argv[])
 29              {
 30     1           int i, aprime, j, howmany, n;
 31     1
 32     1           for (n = 1; n <= EXECUTIONS; n++)
  +                  for (n = 1; n <= 2; n++)
 33     1               {
 34     2                   howmany = 0;
 35     2                   for (i = 0; i <= MAX; i++)
  +                          for (i = 0; i <= 8190; i++)
 36     2                       isprime[i] = TRUE;
  +                          isprime[i] = 1;
 37     2                   for (i = 2; i <= MAX; i++)
  +                          for (i = 2; i <= 8190; i++)
 38     2                       {
 39     3                           if (isprime[i])
 40     3                           {
 41     4                               howmany++;
 42     4                               aprime = i;
 43     4                               for (j = i + aprime; j <= MAX; j += apr
                ·ime)
  +                                    for (j = i + aprime; j <= 8190; j += ap
                ·rime)
 44     4                                   isprime[j] = FALSE;
  +                                        isprime[j] = 0;
 45     4                           }
 46     3                       }
 47     2                   for (i = 0; i <= MAX; i++)
  +                          for (i = 0; i <= 8190; i++)
 48     2                       if (isprime[i]) printf(" %d",i);
 49     2               }
```



```
 50     1       }

MODULE INFORMATION:

     CODE AREA SIZE                = 00D4H    212D
     CONSTANT AREA SIZE            = 0004H      4D
     DATA AREA SIZE                = 2000H   8192D
     MAXIMUM STACK SIZE            = 001AH     26D

iC-286 COMPILATION COMPLETE.       0 REMARKS,     0 WARNINGS,     0 ERRORS
```

Figure 2-18 Parts of the Sieve Example Complete Print File (continued)

2.4.6 Creating Debug Information Using iC-386

This compilation of the example produces an object file with debug information and a compilation summary. Use a symbolic debugger, such as DB386, to trace program execution and debug the program. Table 2-9 shows the controls in effect for the compilation.

The example in this section uses the following compiler invocation:

```
C:\CEXAMPLE> \intel\ic386\ic386 sievec.c &
>> define(NPAPER) &
>> include(prags.h) &
>> searchinclude(\intel\ic386\inc\,includes\) &
>> nolist &
>> print(debug.lst) &
>> debug
```

The compiler displays the following lines on the screen:

```
system-id iC-386 COMPILER Vx.y
Copyright years Intel Corporation
iC-386 COMPILATION COMPLETE.      Ø WARNINGS,     Ø ERRORS
```

Table 2-9 Controls for Creating Debug Information for the Sieve Example

Controls	Where Specified
debug	invocation
define(NPAPER)	invocation
include(prags.h)	invocation
nolist	invocation
optimize(0)	prags.h
pagelength(40)	sievec.c
pagewidth(75)	sievec.c
print(debug.lst)	invocation
searchinclude(\intel\ic386\inc\,includes\)	invocation
small[1]	prags.h
tabwidth(2)	sievec.c
align(4)	default
nocode	default
codesegment(CODE32)	default
nocond	default

[1]This is the default segmentation model.

2.5 Compiling at Different Optimization Levels

The `optimize` control specifies the compiler's optimization level. The compiler has four optimization levels: 0, 1, 2, and 3, where 0 provides the least optimization and 3 provides the most optimization. Each level performs all the optimizations of the lower levels. Figure 2-20 shows the nesting of the optimization levels. See Chapter 3 for detailed information on the optimize control and an explanation of each type of optimization.

Figure 2-21 shows the source text that demonstrates optimization at each level. Figures 2-22 through 2-25 show the significant results of compiling with iC-386 at different optimization levels. Compiling with iC-86 or iC-286 produces optimizations at the same places.

2.5 Compiling at Different Optimization Levels

The `optimize` control specifies the compiler's optimization level. The compiler has four optimization levels: 0, 1, 2, and 3, where 0 provides the least optimization and 3 provides the most optimization. Each level performs all the optimizations of the lower levels. Figure 2-20 shows the nesting of the optimization levels. See Chapter 3 for detailed information on the optimize control and an explanation of each type of optimization.

Figure 2-21 shows the source text that demonstrates optimization at each level. Figures 2-22 through 2-25 show the significant results of compiling with iC-386 at different optimization levels. Compiling with iC-86 or iC-286 produces optimizations at the same places.

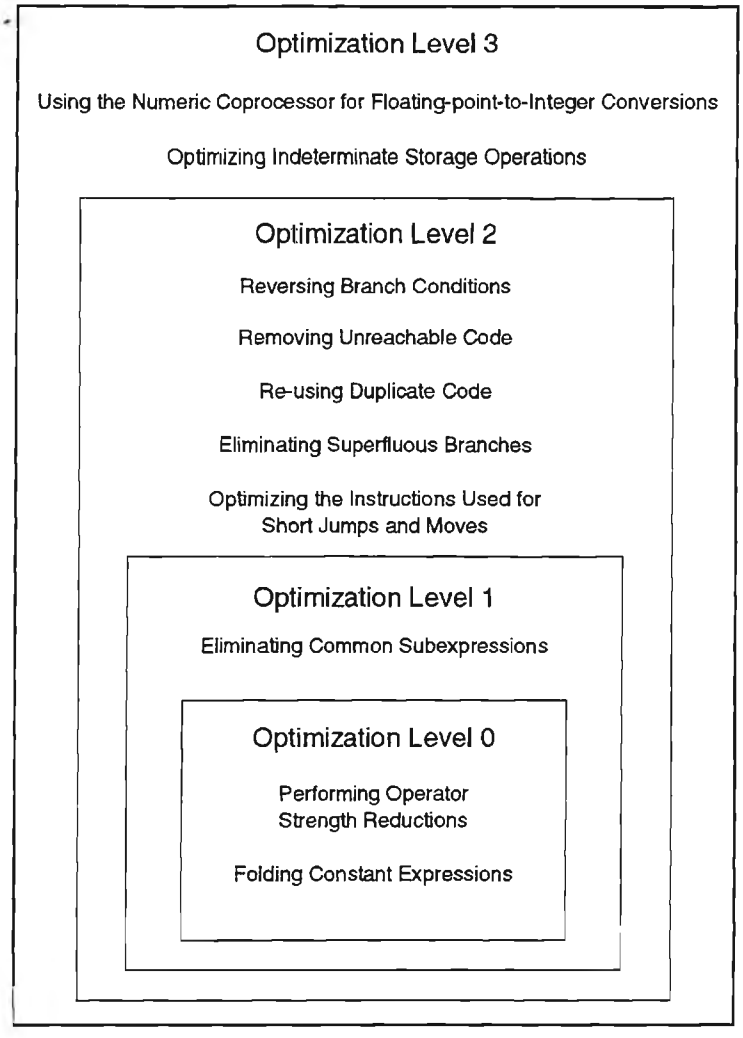

Figure 2-20 Summary of Optimization Levels

```
line

 1 /*************************************************************************/
 2 /* This example shows some of the optimizations that the iC-386 compiler */
 3 /* performs with different values specified for the optimize control.    */
 4 /*************************************************************************/
 5 int i,j,k;
 6 int *a = &j;          /* *a is aliasing j                                 */
 7
 8 int main (int argc, char * argv[])
 9 {
10     i = 1 + j + 1;   /* Folding constants (all levels)                    */
11     k = 3;
12     j = k + 3;
13     i = k + 3;       /* Eliminating common subexpressions (levels 1,2,3)  */
14
15     if (i * 2)       /* Reducing operator strength (all levels)           */
16         i = isquare (i);
17     else             /* Re-using duplicate code (levels 2 and 3)          */
18         i = isquare (j);
19     if (k)
20         goto l1;     /* Branch chaining (levels 2 and 3)                  */
21     else
22         k = 100;
23
24 l1: goto l2;         /* Eliminating superfluous branches (levels 2 and 3) */
25 l2: j = 100;         /* Optimization of pointer indirection --            */
26     *a = 200;        /* Note this step might lead to undesired result,    */
27     i = j;           /* as shown here. (level 3)                          */
28     return;
29
30     k = 200;         /* Eliminating dead code (levels 2 and 3)            */
31 }
```

Figure 2-21 Source Code For Demonstrating Optimization Levels

2.5.1 Results at Optimization Level 0

Figure 2-22 shows the iC-386 pseudo-assembly language code for optimization level 0. At this level, constant-folding occurs in statement #10 and operator strength reduction occurs in statement #15.

```
                                                    ; STATEMENT # 9
                    main        PROC NEAR
00000000  55                    PUSH    EBP
00000001  8BEC                  MOV     EBP,ESP
                    @1:
                                                    ; STATEMENT # 10
00000003  8B0504000000          MOV     EAX,j
00000009  81C002000000          ADD     EAX,2H
0000000F  890500000000          MOV     i,EAX
                                                    ; STATEMENT # 11
00000015  C705080000000300000000
                                MOV     k,3H
                                                    ; STATEMENT # 12
0000001F  8B0508000000          MOV     EAX,k
00000025  81C003000000          ADD     EAX,3H
0000002B  890504000000          MOV     j,EAX
                                                    ; STATEMENT # 13
00000031  8B0508000000          MOV     EAX,k
00000037  81C003000000          ADD     EAX,3H
0000003D  890500000000          MOV     i,EAX
                                                    ; STATEMENT # 15
00000043  8B0500000000          MOV     EAX,i
00000049  D1E0                  SAL     EAX,1
0000004B  0F8416000000          JZ      @2
                                                    ; STATEMENT # 16
00000051  FF3500000000          PUSH    i           ; 1
00000057  E800000000            CALL    isquare
0000005C  890500000000          MOV     i,EAX
00000062  E911000000            JMP     @3
                                                    ; STATEMENT # 17
                    @2:
                                                    ; STATEMENT # 18
00000067  FF3504000000          PUSH    j           ; 1
0000006D  E800000000            CALL    isquare
00000072  890500000000          MOV     i,EAX
                    @3:
                                                    ; STATEMENT # 19
00000078  833D0800000000        CMP     k,0H
0000007F  0F840A000000          JZ      @4
                                                    ; STATEMENT # 20
00000085  E90F000000            JMP     l1
0000008A  E90A000000            JMP     @5
                                                    ; STATEMENT # 21
                    @4:
                                                    ; STATEMENT # 22
0000008F  C705080000006400000000
                                MOV     k,64H
```

Figure 2-22 Pseudo-assembly Code at Optimization Level 0

```
                        @5:
                                                        ; STATEMENT # 24
                        l1:
        00000099  E900000000        JMP     l2
                                                        ; STATEMENT # 25
                        l2:
        0000009E  C705040000064000000
                                    MOV     j,64H
                                                        ; STATEMENT # 26

iC-386  COMPILER   Optimization Level 0          mm/dd/yy hh:mm:ss  PAGE   3
                   ASSEMBLY LISTING OF OBJECT CODE


        000000A8  8B050C000000      MOV     EAX,a
        000000AE  C700C8000000      MOV     [EAX],0C8H
                                                        ; STATEMENT # 27
        000000B4  8B0504000000      MOV     EAX,j
        000000BA  890500000000      MOV     i,EAX
                                                        ; STATEMENT # 28
        000000C0  5D                POP     EBP
        000000C1  C20800            RET     8H
                                                        ; STATEMENT # 30
        000000C4  C705080000000C8000000
                                    MOV     k,0C8H
                                                        ; STATEMENT # 31
                        main        ENDP
                                                        ; STATEMENT # 31


MODULE INFORMATION:

     CODE AREA SIZE                = 000000CEH         206D
     CONSTANT AREA SIZE            = 00000000H           0D
     DATA AREA SIZE                = 00000010H          16D
     MAXIMUM STACK SIZE            = 00000014H          20D

iC-386 COMPILATION COMPLETE.       0 WARNINGS,     0 ERRORS
```

Figure 2-22 Pseudo-assembly Code at Optimization Level 0 (continued)

2.5.2 Results at Optimization Level 1

Figure 2-23 shows the changes in statements #12 through #16 when the invocation uses optimization level 1. The code area size decreases from 208 bytes at optimization level 0 to 182 bytes at optimization level 1.

```
                                                ; STATEMENT # 12
0000001F  B803000000      MOV     EAX,3H
00000024  D1E0            SHL     EAX,1
00000026  890504000000    MOV     j,EAX
                                                ; STATEMENT # 13
0000002C  890500000000    MOV     i,EAX
                                                ; STATEMENT # 15
00000032  D1E0            SAL     EAX,1
00000034  0F8416000000    JZ      @2
                                                ; STATEMENT # 16
0000003A  FF3500000000    PUSH    i       ; 1
00000040  E800000000      CALL    isquare
00000045  890500000000    MOV     i,EAX
0000004B  E911000000      JMP     @3
                                                ; STATEMENT # 17
                  @2:
                                                ; STATEMENT # 18
00000050  FF3504000000    PUSH    j       ; 1
00000056  E800000000      CALL    isquare
0000005B  890500000000    MOV     i,EAX
                  @3:
                                                ; STATEMENT # 19
00000061  833D0800000000  CMP     k,0H
00000068  0F840A000000    JZ      @4
                                                ; STATEMENT # 20
0000006E  E90F000000      JMP     l1
00000073  E90A000000      JMP     @5
                                                ; STATEMENT # 21
                  @4:
                                                ; STATEMENT # 22
00000078  C705080000064000000
                          MOV     k,64H
                  @5:
                                                ; STATEMENT # 24
                  l1:
00000082  E900000000      JMP     l2
                                                ; STATEMENT # 25
                  l2:
00000087  C705040000064000000
                          MOV     j,64H
```

Figure 2-23 Part of the Pseudo-assembly Code at Optimization Level 1

2.5.3 Results at Optimization Level 2

Figure 2-24 shows the changes in statements #16 through #24 and #30 when the invocation uses optimization level 2. Labels also change on several instructions. The code area size decreases from 182 bytes at optimization level 1 to 123 bytes at optimization level 2.

```
iC-386  COMPILER   Optimization Level 2            mm/dd/yy hh:mm:ss  PAGE   2
                   ASSEMBLY LISTING OF OBJECT CODE
.
.
.
                                                   ; STATEMENT # 16
 0000002F  FF3500000000     PUSH    i              : 1
 00000035  EB06             JMP     @1
                                                   ; STATEMENT # 17
                   @2:
                                                   ; STATEMENT # 18
 00000037  FF3504000000     PUSH    j              : 1
                   @1:
 0000003D  E800000000       CALL    isquare
 00000042  A300000000       MOV     i,EAX
                                                   ; STATEMENT # 19
 00000047  833D0800000000   CMP     k,0H
 0000004E  750A             JNZ     l1
                                                   ; STATEMENT # 20
                                                   ; STATEMENT # 21
                                                   ; STATEMENT # 22
 00000050  C705080000006400000
                            MOV     k,64H
```

Figure 2-24 Part of the Pseudo-assembly Code at Optimization Level 2

```
                                                    ; STATEMENT # 24
                        l1:
                                                    ; STATEMENT # 25
                        l2:
          0000005A  C705040000006400000
                                    MOV     j,64H
                                                    ; STATEMENT # 26
          00000064  A10C000000      MOV     EAX,a
          00000069  C700C8000000    MOV     [EAX],0C8H
                                                    ; STATEMENT # 27
          0000006F  A104000000      MOV     EAX,j
          00000074  A300000000      MOV     i,EAX
                                                    ; STATEMENT # 28
          00000079  5D              POP     EBP
          0000007A  C20800          RET     8H
                                                    ; STATEMENT # 30
                                                    ; STATEMENT # 31
                            main    ENDP

iC-386  COMPILER   Optimization Level 2          mm/dd/yy hh:mm:ss  PAGE   3
                   ASSEMBLY LISTING OF OBJECT CODE


                                                    ; STATEMENT # 31


MODULE INFORMATION:

     CODE AREA SIZE                = 0000007DH         125D
     CONSTANT AREA SIZE            = 00000000H           0D
     DATA AREA SIZE                = 00000010H          16D
     MAXIMUM STACK SIZE            = 00000014H          20D

iC-386 COMPILATION COMPLETE.       0 WARNINGS,     0 ERRORS
```

Figure 2-24 Part of the Pseudo-assembly Code at Optimization Level 2 (continued)

2.5.4 Results at Optimization Level 3

Figure 2-25 shows the change in statement #27 when the invocation uses optimization level 3. In this case, because a pointer is aliasing a variable, the change introduces an error. The code area size stays the same from optimization level 2, but one assembly instruction substitutes for two in statement #27.

```
                                                    ; STATEMENT # 12
0000001A  B803000000        MOV     EAX,3H
0000001F  D1E0              SHL     EAX,1
00000021  A304000000        MOV     j,EAX
                                                    ; STATEMENT # 13
00000026  A300000000        MOV     i,EAX
                                                    ; STATEMENT # 15
0000002B  D1E0              SAL     EAX,1
0000002D  7408              JZ      @2
                                                    ; STATEMENT # 16
0000002F  FF3500000000      PUSH    i               ; 1
00000035  EB06              JMP     @1
                                                    ; STATEMENT # 17
                  @2:
                                                    ; STATEMENT # 18
00000037  FF3504000000      PUSH    j               ; 1
                  @1:
0000003D  E800000000        CALL    isquare
00000042  A300000000        MOV     i,EAX
                                                    ; STATEMENT # 19
00000047  833D0800000000    CMP     k,0H
0000004E  750A              JNZ     l1
                                                    ; STATEMENT # 20
                                                    ; STATEMENT # 21
                                                    ; STATEMENT # 22
00000050  C705080000006400000000
                            MOV     k,64H
                                                    ; STATEMENT # 24
                  l1:
                                                    ; STATEMENT # 25
                  l2:
0000005A  C705040000006400000000
                            MOV     j,64H
                                                    ; STATEMENT # 26
00000064  A10C000000        MOV     EAX,a
00000069  C700C8000000      MOV     [EAX],0C8H
                                                    ; STATEMENT # 27
0000006F  C705000000006400000000
                            MOV     i,64H
                                                    ; STATEMENT # 28
00000079  5D                POP     EBP
0000007A  C20800            RET     8H
                                                    ; STATEMENT # 30
                                                    ; STATEMENT # 31
                  main      ENDP
```

Figure 2-25 Part of the Pseudo-assembly Code at Optimization Level 3

```
iC-386  COMPILER   Optimization Level 3            mm/dd/yy hh:mm:ss  PAGE   3
                   ASSEMBLY LISTING OF OBJECT CODE


                                                        ; STATEMENT # 31


MODULE INFORMATION:

     CODE AREA SIZE                = 0000007DH         125D
     CONSTANT AREA SIZE            = 00000000H           0D
     DATA AREA SIZE                = 00000010H          16D
     MAXIMUM STACK SIZE            = 00000014H          20D

iC-386 COMPILATION COMPLETE.       0 WARNINGS,     0 ERRORS
```

Figure 2-25 Part of the Pseudo-assembly Code at Optimization Level 3 (continued)

Contents

3

Compiler Controls

Compiler Controls 3

The compiler controls specify compiler options such as the location of source text files, the amount of debugging information in the object module, and the format and location of the output listings. You need not use any controls when you invoke the compiler. Most of the controls have default settings. Table 3-1 provides default settings and a brief description of each control.

This chapter contains the following topics:

- how controls affect the compilation
- where to use controls
- alphabetical reference of controls

3.1 How Controls Affect the Compilation

Each control affects the compilation in one of three ways:

Source-processing controls	specify the names and locations of input files or define macros at compile time.
Object-file content controls	determine the internal configuration of the object file.
Listing controls	specify the names, locations, and contents of the output listing files.

3.2 Where to Use Controls

Use a compiler control once, freely, or only on invocation, depending on which kind of control it is, as follows:

Primary controls	apply to the entire module. Specify a primary control in the compiler invocation or in a `#pragma` preprocessor directive. A primary control in a `#pragma` preprocessor directive must precede the first executable statement or data definition statement in the source text. A primary control in the invocation line overrides any contradictory control specified in a `#pragma`.
General controls	can change freely within a module. Specify a general control as often as necessary in the compiler invocation and in `#pragma` preprocessor directives anywhere in the source text.
Invocation-only controls	must never appear in a `#pragma` preprocessor directive. Specify an invocation-only control as often as necessary in the invocation line.

Case is not significant in control names, though it can be significant in arguments to controls. DOS preserves the case of arguments to controls. Other systems can require quotation marks (") around arguments to controls to preserve case.

Table 3-1 lists the controls with descriptions, defaults, precedence, effects, and usage classes. Some controls optionally use one or more arguments, indicated by `[a]`. Some controls require one or more arguments, indicated by *a*. Certain controls override other controls, even if stated explicitly. Table 3-1 summarizes such precedence.

Table 3-1 Compiler Controls Summary

Control	Description, Default, and Precedence	Effect	Usage
align [*a*] noalign [*a*]	Aligns or suppresses aligning all structures to specified byte boundaries. Default: align all on 2-byte boundaries (iC-86/286) or 4-byte boundaries (iC-386).	Object	General
code nocode	Generates or suppresses pseudo-assembly object code in print file. Default: nocode.	Listing content	General
codesegment *a*[1]	Names iC-386 code segment. Default: CODE32.	Object	Primary
compact flat[1] large medium small	Specifies segment allocation and segment register addressing in object module. Default: small.	Object	Primary
cond nocond	Includes or suppresses uncompiled conditional code in print file. Default: nocond.	Listing content	General
datasegment *a*[1]	Names iC-386 data segment. Default: DATA.	Object	Primary
debug nodebug	Includes or suppresses debug information in object module. Default: nodebug. nodebug overrides line.	Object	Primary
define *a*	Defines a macro.	Source	Invocation
diagnostic *a*	Specifies level of diagnostic messages. Default: diagnostic level 1.	Listing content	Primary
eject	Inserts form feed in print file. format	Listing	General
extend noextend	Recognizes Intel extensions or not. Default: noextend.	Source	General
fixedparams [*a*] varparams [*a*]	Specifies FPL or VPL function-calling convention. Default: fixedparams for all functions.	Object	General

[1] iC-386 only.

Table 3-1 Compiler Controls Summary (continued)

Control	Description, Default, and Precedence	Effect	Usage
include *a*	Specifies file to process before primary source file.	Source	Invocation
interrupt *a*	Specifies function to be an interrupt handler.	Object	General
line noline	Includes or suppresses source line number debug information in object file. Default: line if debug or noline if nodebug.	Object	Primary
list nolist	Includes or suppresses source code in print file. Default: list. nolist overrides cond, listexpand, listinclude.	Listing content	General
listexpand nolistexpand	Includes or suppresses macro expansion in print file. Default: nolistexpand.	Listing content	General
listinclude nolistinclude	Includes or suppresses include files' text in print file. Default: nolistinclude. nolistinclude overrides listexpand and cond for include files.	Listing content	General
long64[1] nolong64	Sets size for objects declared with long type. Default: nolong64.	Object	Primary
mod86[2] mod186	Uses 86/88 processor instructions or 186/188 instruction set. Default: mod86.	Object	Primary
mod287[2] modc187 nomod287	Generates floating-point instructions for i287™, 80C187, 8087 or numeric coprocessor. Default: nomod287.	Object	Primary
mod486[1] nomod486	Uses i486™ processor instructions or i386™ instruction set. Default: nomod486.	Object	Primary
modulename *a*	Names object module. Default: *sourcename*.	Object	Primary

[1] iC-386 only.
[2] iC-86 only.

Table 3-1 Compiler Controls Summary (continued)

Control	Description, Default, and Precedence	Effect	Usage
object [*a*] noobject	Generates and names or suppresses object file. Default: object named *sourcename*.obj. noobject overrides all object controls except as affects the print file.	Object	Primary
optimize *a*	Specifies level of optimization. Default: optimization level 1.	Object	Primary
pagelength *a*	Specifies number of lines per page in print file. Default: 60.	Listing format	Primary
pagewidth *a*	Specifies number of characters per line in print file. Default: 120.	Listing format	Primary
preprint [*a*] nopreprint	Generates and names or suppresses preprint file. Default: nopreprint if translate or preprint *sourcename*.i if notranslate.	Listing content	Invocation
print [*a*] noprint	Generates and names or suppresses print file. Default: print file named *sourcename*.lst. noprint overrides all listing controls except preprint.	Listing content	Primary
ram rom	Puts constants in data segment or code segment. Default: ram (constants in data segment).	Object	Primary
searchinclude *a* nosearchinclude	Specifies path to prepend to include files or limits path to source directory plus DOS :include: path. Default: nosearchinclude.	Source	General
signedchar nosignedchar	Sign-extends or zero-extends char objects when promoted. Default: signedchar.	Object	Primary
subsys *a*	Reads a subsystem specification file.	Object	Primary
symbols nosymbols	Generates or suppresses identifier list in print file. Default: nosymbols.	Listing content	Primary
tabwidth *a*	Specifies number of characters between tabstops in print file. Default: 4.	Listing format	Primary
title "*a*"	Places title on each page of print file. Default: "*modulename*".	Listing format	Primary

Table 3-1 Compiler Controls Summary (continued)

Control	Description, Default, and Precedence	Effect	Usage
translate notranslate	Compiles or suppresses compilation after preprocessing. Default: translate. notranslate overrides all object and listing controls. notranslate implies preprint.	Source	Invocation
type notype	Generates or suppresses type information in object module. Default: type.	Object	Primary
xref noxref	Adds or suppresses identifier cross-reference information in print file. Default: noxref. xref overrides nosymbols.	Listing content	Primary

3.3 Alphabetical Reference of Controls

The entries in this section describe in detail the syntax and function of each compiler control.

Square brackets ([]) enclose optional arguments for controls. If you do not specify optional arguments for a particular control, do not use an empty pair of parentheses.

Some controls use an optional list of arguments. Separate multiple argument definitions with commas. Brackets surrounding a comma and an ellipsis ([,...]) indicate an optional list with entries separated by commas.

Enclose a control argument in quotation marks (") if the argument contains spaces or any of the following characters:

```
#  !  %  '  ~  -  &  $  @  {  }  `  (
```

Enter all other punctuation as shown, for example, pound signs (#) and equals signs (=).

Syntax

```
align [(structure_tag[=size] [,...])]
noalign [(structure_tag [,...])]

#pragma align [(structure_tag[=size] [,...])]
#pragma noalign [(structure_tag [,...])]
```

Where:

structure_tag	is a structure tag defined in the source text (not a structure identifier).
size	is the number of bytes. The *size* can be 1 for unaligned (byte alignment), 2 for alignment to byte addresses evenly divisible by 2, or 4 for alignment to byte addresses evenly divisible by 4.

Abbreviation

```
[no]al
```

Default

```
align
```

The default value for *size* is 2 bytes for iC-86 and iC-286, or 4 bytes for iC-386. The compiler attempts to place structure components so that they do not cross 2-byte (iC-86/286) or 4-byte (iC-386) boundaries.

Discussion

Use the `align` control to minimize the number of alignment boundaries a structure component can cross. The compiler allocates memory for an aligned-structure component on the next alignment boundary if the component would otherwise span that boundary. If a structure component is larger than the space between alignment boundaries, the component starts on an alignment boundary and still crosses one or more boundaries. Use the `noalign` control or the `align` control with a *`size`* of `1` to allocate structure components on adjacent bytes, leaving no unused bytes.

The processor can require less time to access aligned structures. However, aligned structures can occupy more space than unaligned structures in memory. The compiler attaches no symbol or value to holes. The third example shows a map of how the compiler allocates memory for an aligned structure. The fourth example shows a map of how the compiler allocates memory for an unaligned structure.

Bit fields smaller than one byte cannot cross byte boundaries regardless of alignment. Although an unaligned structure cannot contain any unused bytes, it can contain undefined bits.

To specify 2-byte alignment (iC-86/286 default) or 4-byte alignment (iC-386 default) for all structures, use the `align` control without arguments. To specify byte alignment for all structures, use the `noalign` control without arguments. To specify alignment for all structures of a given type, identify them by *`structure_tag`*. Do not specify structure or type definition identifiers. To ensure alignment, specify the alignment for the structure tag before defining the actual structure.

The `notranslate` control overrides the `align` and `noalign` controls. The `noobject` control overrides the `align` and `noalign` controls except for their effect on the print file.

See *C: A Reference Manual*, listed in Chapter 1, for more information on structures.

Examples

The following examples show different uses of the `align` and `noalign` controls.

1. In this example, only structures of the type in *`argument_list`* are unaligned; all other structures in the subsequent source text are aligned on 2-byte boundaries for iC-86 and iC-286 or 4-byte boundaries for iC-386. Use the following in the compiler invocation:

   ```
   noalign (argument_list)
   ```

   Or use the following in the source text:

   ```
   #pragma noalign (argument_list)
   ```

2. This example aligns all structures of the types in the argument list on the specified boundaries; all other structures in the subsequent source text are allocated regardless of word boundaries. Use the following in the compiler invocation:

   ```
   noalign align (argument_list)
   ```

   Or, use the following in the source text:

   ```
   #pragma noalign
   #pragma align (argument_list)
   ```

3. This example aligns components of a structure on even-byte boundaries. The structure is declared as follows:

```
struct std_struct
{
   unsigned char m1a;
   unsigned char m1b;
   unsigned long m4a;
   unsigned m2a;
   unsigned mba:5;
   unsigned mbb:7;
   unsigned mbc:6;
   double m8a;
};
```

To align all structures of a particular type, use a type definition as follows:

```
typedef struct std_struct
{
   unsigned char m1a;
   unsigned char m1b;
   unsigned long m4a;
   unsigned m2a;
   unsigned mba:5;
   unsigned mbb:7;
   unsigned mbc:6;
   double m8a;
} std_struct_id;
```

In either case, specify the *structure_tag*, not a type identifier, in the `align` control:

```
align (std_struct=2)
```

Figure 3-1 shows how the iC-86 and iC-286 compilers allocate a `std_struct` structure.

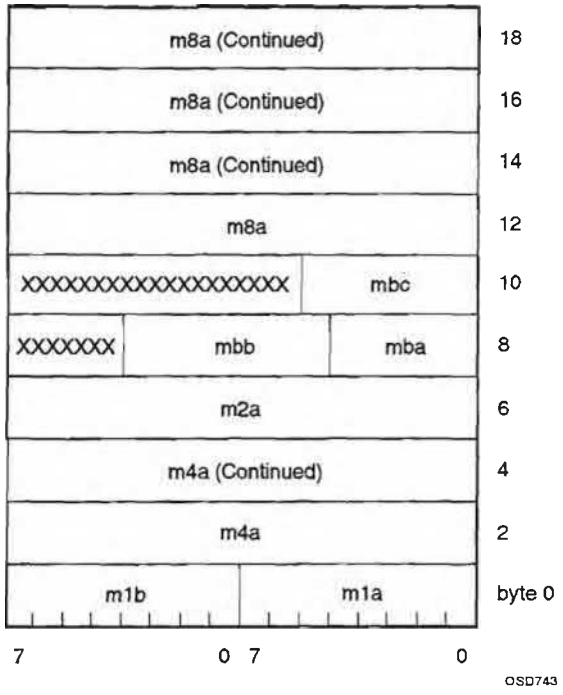

Figure 3-1 Effect of iC-86 and iC-286 align Control on Example Structure Type

Figure 3-2 shows how the iC-386 compiler allocates a `std_struct` structure, assuming the `nolong64` control is in effect.

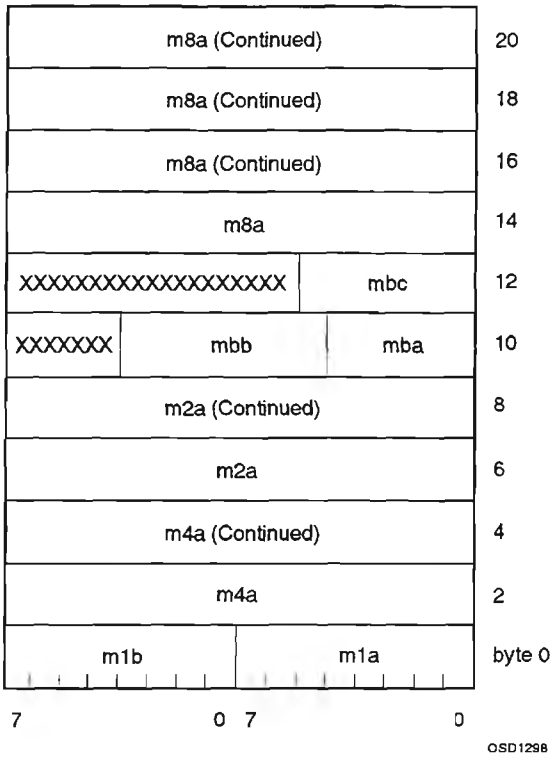

Figure 3-2 Effect of iC-386 align Control on Example Structure Type

4. This example aligns the components of the structure in the previous example on 1-byte (unaligned) boundaries. Use the following control in the compiler invocation:

   ```
   noalign (std_struct)
   ```

   (The `align (std_struct=1)` control achieves the same alignment.)

   Figure 3-3 shows how the iC-86 and iC-286 compilers allocate a `std_struct` structure.

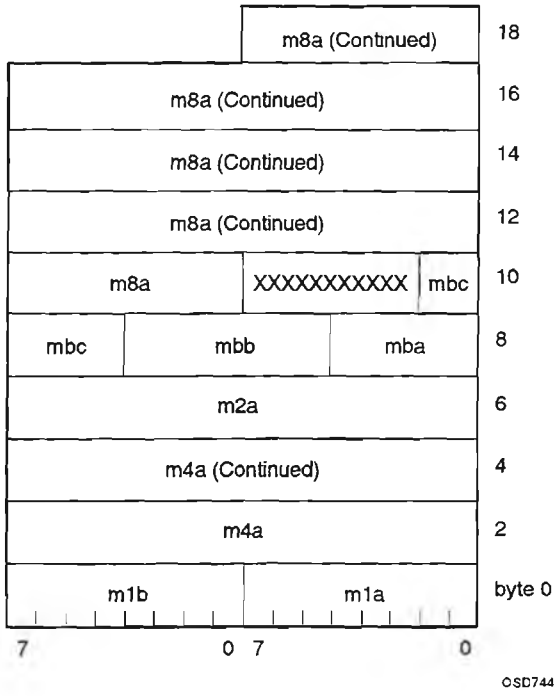

Figure 3-3 Effect of iC-86 and iC-286 noalign Control on Example Structure Type

Figure 3-4 shows how the iC-386 compiler allocates a `std_struct` structure, assuming the `nolong64` control is in effect.

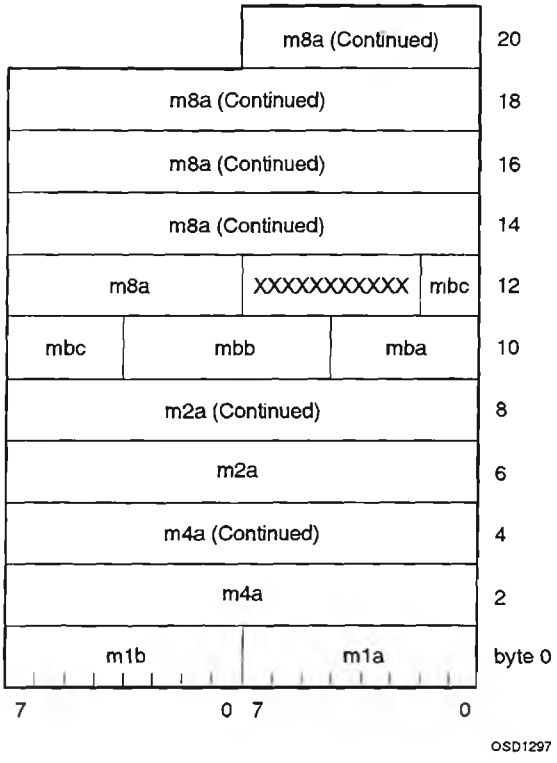

Figure 3-4 Effect of iC-386 noalign Control on Example Structure Type

Cross-references

```
long64 | nolong64
object | noobject
translate | notranslate
```

Syntax

```
[no]code
#pragma [no]code
```

Abbreviation

```
[no]co
```

Default

```
nocode
```

Discussion

Use the `code` control to produce a pseudo-assembly language listing equivalent to the object code that the compiler generates. The compiler places this listing in the print file following the source text listing. Use the `nocode` control (default) to suppress the pseudo-assembly language listing.

The `noobject` control does not override the `code` control. The `noprint` control causes the compiler to suppress all of the print file, even if `code` is specified. The `notranslate` control overrides the `code` control.

Use the `code` control for the following purposes:

- To view the effects of different levels of optimization set by the `optimize` control.
- To view the differences in code the compiler generates under the `mod86` and `mod186` controls or `mod287`, `nomod287`, and `modc187` (iC-86) or the `mod486` and `nomod486` controls (iC-386).

- To view the differences in pointer types the compiler generates under the `extend` or `noextend` controls.
- To detect errors when debugging at the assembly-code level.

See Chapter 5 for more information on the print file.

Cross-references

`extend | noextend`
`mod86 | mod186`
`mod287 | nomod287 | modc187`
`mod486 | nomod486`
`object | noobject`
`optimize`
`print | noprint`
`translate | notranslate`

Syntax

iC-386: `codesegment (code_segment_name)`

iC-386: `#pragma codesegment (code_segment_name)`

Where:

code_segment_name	is the name of the iC-386 code segment in the object module.

Abbreviation

`cs`

Default

The iC-386 compiler uses `CODE32` or the subsystem identifier as specified in the subsystem definition file.

Discussion

Use the iC-386 `codesegment` control to name the code segment in the object module. The code segment name is used by the BND386 binder and BLD386 builder. This name also appears in output from the MAP386 mapper. See the *Intel386™ Family Utilities User's Guide* and the *Intel386™ Family System Builder User's Guide*, listed in Chapter 1, for information on BND386, MAP386, and BLD386.

This control is provided for compatibility with C-386, Intel's previous compiler for Intel386™ processor code.

NOTE

Do not use the `codesegment` control in an invocation that specifies the `subsys` control. The compiler issues an error or a warning, depending on whether the `subsys` control is found in the invocation line or in a `#pragma` preprocessor directive, respectively.

Cross-references

`datasegment`
`modulename`
`subsys`

Syntax

```
compact
#pragma compact
```

Abbreviation

```
cp
```

Default

Of the four iC-86 and iC-286 memory model specifications (`small`, `compact`, `medium`, and `large`) and the three iC-386 memory model specifications (`small`, `compact`, and `flat`), the default is `small`.

Discussion

Use the `compact` control to specify the compact segmentation model. The compiler produces an object module containing a code segment, a data segment, and a separate stack segment. The linker or binder combines the code segments for all modules into a single code segment in memory and the data segments for all modules into a single data segment in memory, and reserves a separate segment in memory for the stack. The compact segmentation model is efficient in both program size and memory access, and offers the maximum possible space for the stack.

For 86 and 286 processors, code, data, or stack segments each can occupy up to 64 kilobytes of memory. For Intel386 processors, each segment can occupy up to 4 gigabytes of memory.

The processor addresses the compact model program's code segment relative to the CS register, the data segment relative to the DS register, and the stack segment relative to the SS register. Depending on whether the `rom` or `ram` control is in effect, the compiler places constants in the code segment or data segment, respectively. All functions have near pointers and calls. All data pointers are far pointers. See the `extend | noextend` control and Chapter 4 for more information about the `far` and `near` keywords.

The `notranslate` control overrides the `compact` control. The `noobject` control overrides the `compact` control except for its effect on the print file.

See Chapter 2 for more information on the availability of run-time libraries for the various memory models.

See Chapter 4 for more specific information on segmentation and the compact memory model.

Cross-references

`extend | noextend`	`object | noobject`
`flat`	`ram | rom`
`large`	`small`
`medium`	`translate | notranslate`

Syntax

```
[no]cond
#pragma [no]cond
```

Abbreviation

```
[no]cd
```

Default

```
nocond
```

Discussion

Use the `cond` control to include in the program listing code not compiled because of conditional preprocessor directives. Use the `nocond` control (default) to suppress listing of code eliminated by conditional compilation.

Regardless of these controls, the conditional preprocessor directives (`#if`, `#ifdef`, `#ifndef`, `#elif`, `#else`, and `#endif`) delimiting the code appear in the source text listing in the print file.

The `nolist`, `notranslate`, and `noprint` controls override the `cond` control. If any of these is in effect, the compiler does not list any source text. The `nolistinclude` control overrides the `cond` control for include files. Neither `cond` nor `nocond` has any effect on the preprint file.

See Chapter 5 for more information on the preprint and print files. See Chapter 2 for an example of the effect of the `cond` and `nocond` controls and other listing specifications on the print file. See *C: A Reference Manual*, listed in Chapter 1, for more information on conditional compilation.

Cross-references

```
list | nolist
listinclude | nolistinclude
print | noprint
translate | notranslate
```

Syntax

iC-386: `datasegment (data_segment_name)`

iC-386: `#pragma datasegment (data_segment_name)`

Where:

data_segment_name	is the name of the iC-386 data segment in the object module.

Abbreviation

`ds`

Default

The iC-386 compiler uses `DATA` or the subsystem identifier as specified in the subsystem definition file.

Discussion

Use the iC-386 `datasegment` control to name the data segment in the object module. The data segment name is used by the BND386 binder and BLD386 builder. This name also appears in output from the MAP386 mapper. See the *Intel386™ Family Utilities User's Guide* and the *Intel386™ Family System Builder User's Guide*, listed in Chapter 1, for information on BND386, MAP386, and BLD386.

This control is provided for compatibility with C-386, Intel's previous compiler for Intel386 processor code.

NOTE

Do not use the `datasegment` control in an invocation that specifies the `subsys` control. The compiler issues an error or a warning, depending on whether the `subsys` control is found in the invocation line or in a `#pragma` preprocessor directive, respectively.

Cross-references

`codesegment`
`modulename`
`subsys`

Syntax

```
[no]debug
#pragma [no]debug
```

Abbreviation

```
[no]db
```

Default

```
nodebug
```

Discussion

Use the `debug` control to place symbolic debug information used by symbolic debuggers in the object module. Use the `nodebug` control (default) to suppress symbolic debug information. Suppressing symbolic debug information reduces the size of the object module. Debug information is composed of the name, relative address, and type of every object and function definition, and the relative address of each source line both in the source file and in the object file.

The `noobject` and `notranslate` controls override the `debug` and `nodebug` controls.

Choose one of the following combinations of the `debug` or `nodebug` and `type` or `notype` controls to aid debugging:

`type debug`	to include all debug and type information (`debug` implies `line`). This combination allows both type checking and symbolic debugging.
`type debug noline`	to include debug and type information, but no source line numbers. This combination enables linker type checking and symbolic debugging, but not source-level debugging.
`type nodebug`	to include type definition information for external and public symbols only. This combination allows type checking by the linker or binder. Use this combination to reduce the size of the object module when you are not using a symbolic debugger.
`notype nodebug`	to suppress all debug and type information. This combination reduces the size of the object module by omitting information not necessary for execution.

The `optimize` control can further reduce the size of the object module. However, higher levels of optimization reduce the ability of most symbolic debuggers to accurately correlate debug information to the source code. The `line` control causes the compiler to place source file and object file line-number debug information in the object file. The `symbols` control puts a listing of all identifiers and their types into the print file. The `xref` control puts a cross-reference listing of all identifiers into the print file.

Cross-references

`line | noline`
`object | noobject`
`optimize`
`symbols | nosymbols`
`translate | notranslate`
`type | notype`
`xref | noxref`

Syntax

```
define (name[=body] [,...])
```

Where:

name	is the name of a macro.
body	is the text (i.e., value) of the macro. If the *body* contains blanks or punctuation, surround the entire *body* with quotation marks (").

Abbreviation

```
df
```

Default

If the definition contains no *body*, the default value of the macro is 1.

Discussion

Use the `define` control to create an object-like macro at invocation time. The body of an object-like macro contains no formal parameters. A macro so defined in the compiler invocation is in effect for the entire module, until the `#undef` preprocessor directive removes it. An attempt to redefine a macro in a `#define` preprocessor directive causes an error.

Available memory limits the number of active macro definitions, including macros defined in the compiler invocation and macros defined with `#define` in your source text. Macros are useful when used with conditional compilation preprocessor directives to select source text at compile time. Do not use the `define` control for function-like macros; use the `#define` preprocessor directive in the source text instead.

See *C: A Reference Manual*, listed in Chapter 1, for more information on macros and preprocessor directives. See Chapter 2 for an example using the `define` control.

Example

In this example, using the `define` control in the invocation determines the result of conditional compilation. The invocation contains the following control:

```
define (SYS)
```

The source text contains the following lines:

```
#if SYS
#define PATHLENGTH 128
#else
#define PATHLENGTH 45
#endif
```

The value of the symbol `SYS` defaults to 1. `PATHLENGTH` gets the value 128.

Syntax

```
diagnostic (level)
#pragma diagnostic (level)
```

Where:

level is the value 0, 1, or 2. The values correspond to all diagnostic messages, no remarks, and only errors, respectively.

Abbreviation

```
dn
```

Default

diagnostic level 1

Discussion

Use the `diagnostic` control to specify the level of diagnostic messages that the compiler produces. A remark points out a questionable construct, such as using an undeclared function name. A warning points out an erroneous construct, such as a pointer type mismatch. An error points out a construct that is not part of the C language, such as a syntax error.

Use the different levels of the `diagnostic` control as follows:

`diagnostic (Ø)`	for the compiler to issue all remarks, warnings, and errors.
`diagnostic (1)`	(the default) for the compiler to issue warnings and errors but no remarks.
`diagnostic (2)`	for the compiler to issue only error messages.

The compiler's exit status is equal to the highest level of diagnostic reported. For example, if the diagnostic level is 2, the compiler's exit status is zero if the program contains no errors but could contain remarks or warnings. At level 2, the compiler's exit status is non-zero only if the program contains errors. Table 3-2 shows the DOS errorlevel values for the exit status of the compiler at different diagnostic levels.

Table 3-2 DOS Errorlevel Values

Diagnostic Level	Fatal Errors	Errors	Warnings	Remarks	Errorlevel
2	no	no	not used	not used	0
	no	yes	not used	not used	1
	yes	yes or no	not used	not used	2
1 (default)	no	no	no not used	0	
	no	no	yes not used	1	
	no	yes	yes or no	not used	2
	yes	yes or no	yes or no	not used	3
0	no	no	no no	0	
	no	no	no yes	1	
	no	no	yes yes or no	2	
	no	yes	yes or no	yes or no	3
	yes	yes or no	yes or no	yes or no	4

The `notranslate` control causes preprocessing diagnostics to appear at the console. The `noprint` control causes the compiler to display all diagnostic messages at the console.

Cross-references

`print | noprint`
`translate | notranslate`

eject

General control
Causes form feed

Syntax

```
eject
#pragma eject
```

Abbreviation

```
ej
```

Discussion

Use the `eject` control to cause a form feed in the print file at the point where the control is specified. If you specify the `eject` control on the invocation line, the form feed occurs before the text of any source file is listed.

The `noprint` and `notranslate` controls suppress the print file, causing the `eject` control to have no effect.

The `pagelength`, `pagewidth`, `tabwidth`, and `title` controls also affect the format of the print file. See Chapter 5 for a description of the print file.

Cross-references

```
pagelength
pagewidth
tabwidth
title
```

Syntax

```
[no]extend
#pragma [no]extend
```

Abbreviation

```
[no]ex
```

Default

```
noextend
```

Discussion

Use the `extend` control to enable the compiler to recognize the non-ANSI `alien`, `far`, and `near` keywords in the source text, and to allow the dollar sign (`$`) to be a non-significant character in identifiers in the source text. Use the `noextend` control (default) to suppress recognition of Intel's extensions. These extensions allow compatibility with earlier versions of Intel C.

See the `fixedparams` and `varparams` controls for information on calling convention compatibility with earlier versions of Intel C.

See Chapter 4 for more information on the `far` and `near` keywords. See Chapter 10 for more information on the `alien`, `far`, and `near` keywords.

Cross-references

```
fixedparams
ram | rom
varparams
```

Syntax

```
fixedparams [(function [,...])]
#pragma fixedparams [(function [,...])]
```

Where:

function is the name of a function defined in the source text. Function-name arguments are case-significant.

Abbreviation

```
fp
```

Default

Of the two calling convention specifications (`fixedparams` and `varparams`), the default is `fixedparams`. If you specify the `fixedparams` control but do not supply a *function* argument, the `fixedparams` control applies to all functions in the subsequent source text.

Discussion

Use the `fixedparams` control (default) to require the specified functions to use the fixed parameter list (FPL) calling convention. Most of Intel's non-C compilers generate object code for function calls using the FPL calling convention. Some earlier versions of Intel C use the variable parameter list (VPL) calling convention.

A function's calling convention dictates the sequence of instructions that the compiler generates to manipulate the stack and registers during a call to a function. The FPL calling convention is as follows:

1. The calling function pushes the arguments onto the stack with the leftmost argument pushed first before control transfers to the called function.
2. The called function removes the arguments from the stack before returning to the calling function.

See Chapter 8 for more detailed information on the FPL and VPL calling conventions.

The FPL calling convention uses fewer instructions and therefore occupies less space in memory and executes more quickly than the VPL calling convention. See the `varparams` control for more information on the VPL calling convention.

A calling convention specified without an argument in the compiler invocation affects functions throughout the entire module. If a function uses a calling convention other than the one in effect for the compilation, specify the calling convention before declaring the function.

If FPL is in effect globally, you can use an ellipsis in a prototype or declaration to declare a VPL function, or use the `varparams` control. If VPL is in effect globally, you must use the `fixedparams` control in a `#pragma` preprocessor directive to declare an FPL function.

If `notranslate` is specified, the compiler does not generate object code and the calling convention control has no effect. If `noobject` is specified, the effect of the calling convention control on the object code can be seen in the print file, although the compiler does not produce a final object file.

NOTE

An error occurs if a function in the source text explicitly declares a variable parameter list and also is named in the *function* list for the `fixedparams` control. In the following example, the ellipsis in the `fvprs` function prototype indicates a VPL convention for this function. Specifying the `fixedparams (fvprs)` control in this case causes a compilation error:

```
#include <stdarg.h>
fvprs (int a, ...);
```

See the `extend | noextend` control for other information on code compatibility with previous versions of Intel C. See the `varparams` control for information on the variable parameter list calling convention.

Examples

The following examples show different uses of the `fixedparams` and `varparams` controls.

1. This combination of controls specifies the variable parameter list convention (VPL) for all functions in the source file except those in the argument list. Use the controls on the invocation line as follows:

   ```
   varparams fixedparams (argument_list)
   ```

   Or use the controls in `#pragma` preprocessor directives as follows:

   ```
   #pragma varparams
   #pragma fixedparams (argument_list)
   ```

2. This control specifies the fixed parameter list convention (FPL) for all functions in the source file except those in the argument list. Use the `varparams` control on the invocation line to override the default for the functions in the argument list as follows:

   ```
   varparams (argument_list)
   ```

   Or use the `varparams` control in a `#pragma` preprocessor directive as follows:

   ```
   #pragma varparams (argument_list)
   ```

Cross-references

`extend | noextend`
`object | noobject`
`translate | notranslate`
`varparams`

Syntax

iC-386: `flat`

iC-386: `#pragma flat`

Abbreviation

`fl`

Default

Of the three iC-386 memory model specifications (`small`, `compact`, and `flat`), the default is `small`.

Discussion

Use the iC-386 `flat` control to specify the flat segmentation model. The compiler produces an object module containing a code segment including constants, a data segment, and a stack segment. Using the `flat` control with BLD386, the builder maps all of the code, data, and stack segments into a single segment in memory. The flat segmentation model is efficient in both program size and memory access, but does not take advantage of the segmentation protection provided by the hardware.

The processor addresses the flat model program's code segment relative to the CS register, the data segment relative to the DS register, and the stack segment relative to the SS register. CS contains the selector for an execute-read segment. DS and SS contain the selector for a read-write segment that is aliased to the execute-read segment.

The rom or ram controls have little effect, because the compiler always places constants in the code segment. The rom or ram control does affect the value of the _ROM_ predefined macro. See Chapter 5 for information on predefined macros.

All functions have near pointers and calls. All data pointers are near pointers. See the extend | noextend control and Chapter 4 for more information about the far and near keywords.

The notranslate control overrides the flat control. The noobject control overrides the flat control except for its effect on the print file.

See Chapter 2 for more information on the availability of run-time libraries for the various memory models.

See Chapter 4 for more specific information on segmentation and the flat memory model.

Cross-references

compact	ram | rom
extend | noextend	small
object | noobject	translate | notranslate

Syntax

```
include (filename [,...])
```

Where:

filename is the file specification (including a device name and directory name or pathname, if necessary) to be included and compiled before the primary source file. You do not have to enclose a *filename* in quotation marks, even if it contains a pathname.

Abbreviation

```
ic
```

Discussion

Use the `include` control to insert and compile text from files other than the primary source file. These files are called include files. The compiler processes include files in the order specified in the *filename* list before processing the primary source file.

Use the `listinclude` control to list the contents of the include files in the source code listing in the print file. Use the `searchinclude` control to specify a search path for include files. Use the `preprint` control and the `notranslate` control together to view the resulting order and names of include files without compilation.

Files included by the `include` control on the invocation line are within the scope of all macros defined by the `define` control on the invocation line, regardless of the order of the controls. Files included by the `include` control on the invocation line precede the scope of macros defined by the `#define` preprocessor directive in the primary source file. If more than one `include` control occurs in the invocation, the compiler includes files in the order specified in the invocation line.

The maximum number of filenames in an instance of the `include` control is 19. The maximum number of files open simultaneously during compilation is system-dependent. The maximum nesting level of include files is 10, unless the `preprint` control is in effect, in which case the maximum nesting level is 7.

The iC-86/286/386 compiler on DOS has two added facilities for searching for files. The compiler maps slashes (/) in filenames to backslashes (\). When a pathname begins with an environment variable, the compiler uses the value of the environment variable as the directory path prefix and applies the mappings to all filenames including prefixes specified with the `searchinclude` control.

See Chapter 2 for an example of using the `include` control on DOS. See Chapter 5 for a description of the print file.

Cross-references

```
listinclude
preprint | nopreprint
searchinclude | nosearchinclude
```

Syntax

iC-86:	`interrupt (`*`function`*`[=`*`n`*`] [,...])`
iC-286 and iC-386:	`interrupt (`*`function`*` [,...])`
iC-86:	`#pragma interrupt (`*`function`*`[=`*`n`*`] [,...])`
iC-286 and iC-386:	`#pragma interrupt (`*`function`*` [,...])`

Where:

`function`	is the name of a function defined in the source text.
`n`	is an integer from 0 to 255 for iC-86 only.

Abbreviation

`in`

Discussion

Use the `interrupt` control to specify a function in the source text to handle some condition signaled by an interrupt. An interrupt-handler function must be of type `void` and can neither take arguments nor return a value. The interrupt designation must precede the function definition. The `interrupt` control causes the compiler to generate prolog and epilog code to save and restore registers and return from the interrupt.

For an 86-family target processor, you can use the `interrupt` control to associate the interrupt handler with an interrupt number, *n*. The iC-86 compiler produces an interrupt vector entry for each interrupt handler. The interrupt vectors are an array of pointer values beginning at physical address 0. The *n*th entry is at location 4 * *n*, and contains the location of the interrupt handler associated with the interrupt number *n*. If you specify more than one interrupt handler for the same vector number *n*, the compiler associates only the first function with the interrupt number.

For a 286 or Intel386-family target processor, use the BLD286 or BLD386 system builder to create a gate for the interrupt handler and place the gate in the interrupt descriptor table (IDT). See the *System Builder User's Guide*, listed in Chapter 1.

The `notranslate` control overrides the `interrupt` control. The `noobject` control overrides the `interrupt` control except for its effect on the print file.

See Chapter 6 for examples using the `interrupt` control. See *C: A Reference Manual*, listed in Chapter 1, for information on the `void` function type.

Cross-references

`object | noobject`
`translate | notranslate`

Syntax

```
large
#pragma large
```

Abbreviation

```
la
```

Default

Of the four iC-86 and iC-286 memory model specifications (`small`, `compact`, `medium`, and `large`), the default is `small`. For iC-386, the `large` control has the same effect as the `compact` control. The following discussion applies to iC-86 and iC-286 only.

Discussion

Use the `large` control to specify the large segmentation model. The compiler produces an object module containing a code segment, a data segment, and a separate stack segment. The linker or binder creates a separate code segment for each module's code and a separate data segment for each module's data, and creates a single stack segment. The large segmentation model offers a total amount of code and data space limited only by the target system.

For 86 and 286 processors, each code, data, or stack segment can occupy up to 64 kilobytes of memory. For Intel386 processors, each segment can occupy up to 4 gigabytes of memory.

The processor addresses the large model program's currently active code segment relative to the CS register, the currently active data segment relative to the DS register, and the stack segment relative to the SS register. Depending on whether the `rom` or `ram` control is in effect, the compiler places each module's constants in its code segment or data segment, respectively. All functions have far pointers and calls. All data pointers are far pointers. See the `extend | noextend` control and Chapter 4 for more information about the `far` and `near` keywords.

The `notranslate` control overrides the `large` control. The `noobject` control overrides the `large` control except for its effect on the print file.

See Chapter 2 for more information on the availability of run-time libraries for the various memory models.

See Chapter 4 for more specific information on segmentation and the large memory model.

Cross-references

`compact`	`object	noobject`
`extend	noextend`	`ram	rom`
`flat`	`small`
`medium`	`translate	notranslate`

Syntax

```
[no]line
#pragma [no]line
```

Abbreviation

```
[no]ln
```

Default

`line`	when the `debug` control is in effect.
`noline`	when the nodebug control is in effect.

Discussion

Use the `line` control (default) to cause the compiler to create source line number information in the object file. Use the `noline` control to suppress this information, reducing the object file size by as much as 80%. Source line number information is useful when using a symbolic debugger for source-level debugging.

The `noline` control was used in conjunction with the `debug` control to generate the iC-86/286/386 libraries. This combination of controls retains all debug information other than line information, which is only useful if the source code for each function is available to the debugger.

The `nodebug` control, the `noobject` control, and the `notranslate` control override the `line` control.

Cross-references

debug | nodebug
object | noobject
translate | notranslate

Syntax

```
[no]list
#pragma [no]list
```

Abbreviation

```
[no]li
```

Default

```
list
```

Discussion

Use the `list` control (default) to generate a listing of the source text. The compiler places the source listing in the print file. Use the `nolist` control to suppress the source listing.

The `noprint` and `notranslate` controls suppress the entire print file, even if `list` is specified. The `nolist` control overrides the `cond` control and the `listexpand` and `listinclude` controls.

Several other controls affect the contents of the listing, as follows:

- The `code` control causes pseudo-assembly code to appear after the source listing.
- The `cond` control causes uncompiled conditional code to appear in the listing.
- The `listexpand` control causes macros to be expanded in the listing.
- The `listinclude` control causes text from include files to appear in the listing.

The `eject`, `pagewidth`, `pagelength`, `tabwidth`, and `title` controls affect the format of the print file.

See Chapter 5 for a description of the print file. See Chapter 2 for examples of the effect that listing controls have on the print file.

Cross-references

`code | nocode`
`cond | nocond`
`eject`
`listexpand | nolistexpand`
`listinclude | nolistinclude`
`pagelength`
`pagewidth`
`print | noprint`
`tabwidth`
`title`
`translate | notranslate`

Syntax

```
[no]listexpand
#pragma [no]listexpand
```

Abbreviation

```
[no]le
```

Default

```
nolistexpand
```

Discussion

Use the `listexpand` control to show the results of macro expansion in the source text listing in the print file. Use the `nolistexpand` control (default) to suppress the results of macro expansion. Neither control has any effect on the preprint file.

The compiler marks the macro expansion lines in the listing with a plus (+) in the line-number column. Macro expansions appear only in the listing for compiled code. If the preprocessor suppresses compilation of conditional code, the listing does not include the expansion of any macro invocations in the suppressed code.

Use the `cond` control to list uncompiled conditional code.

The `nolist`, `notranslate`, and `noprint` controls override the `listexpand` control. If any of these is in effect, the compiler does not list any source text. The `nolistinclude` control overrides the `listexpand` control for include files.

See Chapter 5 for a description of the print file.

Cross-references

`cond | nocond`
`list | nolist`
`listinclude | nolistinclude`
`print | noprint`
`translate | notranslate`

Syntax

```
[no]listinclude
#pragma [no]listinclude
```

Abbreviation

```
[no]lc
```

Default

```
nolistinclude
```

Discussion

Use the `listinclude` control to list the text of include files in the source text listing in the print file. Use the `nolistinclude` control (default) to suppress the listing of include files. Neither control has any effect on the preprint file.

The compiler lists files included with the `include` control before the first line of source listing. The compiler adds the text of files included with the `#include` preprocessor directive after the line with the `#include` directive. The compiler lists include files in the order they are specified.

The `nolist`, `notranslate`, and `noprint` controls override the `listinclude` control.

When the `nolistinclude` control is in effect, diagnostic messages for include files appear in the print file as follows:

- For files included with the `include` control, diagnostic messages precede the first line of source text.
- For files included with the `#include` preprocessor directive, diagnostic messages appear on the lines immediately after the `#include` directive.

The compiler lists diagnostic messages in the order in which the associated conditions occur. Use the `diagnostic` control to specify the level of messages the compiler issues.

See Chapter 5 for a description of the print file. See Chapter 2 for an example of the effect of listing controls on the print file.

Cross-references

`diagnostic`
`include`
`list|nolist`
`print|noprint`
`translate|notranslate`

Syntax

iC-386: `[no]long64`

iC-386: `#pragma [no]long64`

Abbreviation

`[no]l64`

Default

`nolong64`

Discussion

Use the iC-386 `long64` control to specify that objects declared with the `long` type qualifier are 64 bits in length. Use the `nolong64` control (default) to specify that objects declared with `long` are 32 bits in length.

The `notranslate` control overrides the `long64` and `nolong64` controls. The `noobject` control overrides the `long64` and `nolong64` controls except for their effect on the print file.

See Chapter 10 for information on iC-386 data types. See *C: A Reference Manual*, listed in Chapter 1, for information on specifying type qualifiers.

Cross-references

`object | noobject`
`translate | notranslate`

medium

Primary control
Specifies the medium
segmentation memory model

Syntax

```
medium
#pragma medium
```

Abbreviation

```
md
```

Default

Of the four iC-86 and iC-286 memory model specifications (`small`, `compact`, `medium`, and `large`), the default is `small`. For iC-386, the `medium` control has the same effect as the `small` control. The following discussion applies to iC-86 and iC-286 only.

Discussion

Use the `medium` control to specify the medium segmentation model. The compiler produces an object module containing a code segment, a data segment, and a separate stack segment. The linker or binder creates a separate code segment for each module's code, combines the data segments for all modules into a single data segment, and reserves space in the data segment to accommodate all stack activity. The medium segmentation model offers efficient data access with code space limited only by the target system.

For 86 and 286 processors, each code segment and the combined data and stack segment can occupy up to 64 kilobytes of memory.

The processor addresses the medium model program's currently active code segment relative to the CS register, the data in the combined data and stack segment relative to the DS register, and the stack in the combined data and stack segment relative to the SS register (which has the same value as the DS register). Depending on whether the `rom` or `ram` control is in effect, the compiler places each module's constants in the corresponding code segment or in the combined data and stack segment. All functions have far pointers and calls. If the `ram` control is in effect, all data pointers are near pointers. If the `rom` control is in effect, all data pointers are far pointers. See the `extend | noextend` control and Chapter 4 for more information about the `far` and `near` keywords.

The `notranslate` control overrides the `medium` control. The `noobject` control overrides the `medium` control except for its effect on the print file.

See Chapter 2 for more information on the availability of run-time libraries for the various memory models.

See Chapter 4 for more specific information on segmentation and the medium memory model.

Cross-references

`compact`	`object	noobject`
`extend	noextend`	`ram	rom`
`flat`	`small`
`large`	`translate	notranslate`

mod86 | mod186

Primary control, iC-86 only
Generates 86/88 processor code or
186/188 processor code

Syntax

iC-86: `mod86`
`mod186`

iC-86: `#pragma mod86`
`#pragma mod186`

Abbreviation

(none)

Default

`mod86`

Discussion

Use the iC-86 `mod86` control to cause the compiler to generate code for the 86/88 processor. Use the `mod186` control to cause the compiler to generate code for the 186/188 processor. The 186/188 instruction set includes short forms of some instructions.

The `notranslate` control overrides the `mod86` and `mod186` controls. The `noobject` control overrides the `mod86` and `mod186` controls except for their effect on the print file.

NOTE

Object modules compiled with the `mod186` control do not execute properly on the 86 or 88 processors. These modules produce execution errors with unpredictable results, such as hanging your system.

See the *ASM86 Assembly Language Reference Manual*, listed in Chapter 1, for descriptions of the 86/88 and 186/188 instruction sets.

Example

Using the 186/188 instruction set instead of the 86/88 instruction set may result in a program requiring less code space. The iC-86 compiler produces the following three lines when the `mod86` control is in effect:

```
0002   55            PUSH   BP
0003   8BEC          MOV    BP,SP
0005   81EC0C00      SUB    SP,0CH
```

The iC-86 compiler produces the following equivalent line when the `mod186` control is in effect:

```
0002   C80C0000      ENTER  0CH,0H
```

Cross-references

`object | noobject`
`translate | notranslate`

mod287 | modc187 | nomod287

Primary control, iC-86 only
Generates code for i287™, 80C187 or
8087 numeric coprocessor

Syntax

```
[no]mod287
modc187

#pragma [no]mod287
#pragma modc187
```

Abbreviation

(none)

Default

`nomod287`

Discussion

Use the iC-86 `mod287` control to cause the compiler to generate code for the Intel287 numeric coprocessor, without FWAIT instructions. This code is more efficient for some systems, for example, an AT-class system with a 286 processor (executing in real mode) and an Intel287 numeric coprocessor. Use the `nomod287` control (default) to cause the compiler to generate code for the 8087 numeric coprocessor, including FWAIT instructions. Do not use `mod287` for code to be linked to the E8087 or DE8087 emulator.

Use the `modc187` control to cause the compiler to generate code for the 80C187 numeric coprocessor. Also use the `mod186` control to generate efficient code for the 80C186 processor used in conjunction with the 80C187.

Code generated with the `mod287` control in effect executes correctly on a system containing an Intel287, Intel 387, or 80C187 numeric coprocessor (or true emulator) or Intel486 processor with on-chip FPU. Such code will not execute correctly on an 8087.

Similarly, code generated with the `modc187` control will execute correctly on an 80C187, Intel387, or Intel486 processor (or true emulator), but not on an 8087 or Intel287.

The `notranslate` control overrides the `mod287`, `modc187`, and `nomod287` controls. The noobject control overrides the `mod287`, `modc187`, and `nomod287` controls except that the effect of the instruction set control on the object code can be seen in the print file, although the compiler does not produce a final `object` file.

Cross-references

```
mod186
object | noobject
translate | notranslate
```

mod486 | nomod486

Primary control, iC-386 only
Generates i486™ processor code or
i386™ processor code

Syntax

iC-386: `[no]mod486`

iC-386: `#pragma [no]mod486`

Abbreviation

(none)

Default

`nomod486`

Discussion

Use the iC-386 `mod486` control to cause the compiler to generate code for the i486™ processor. This code is particularly suited for fast execution on i486 processor-based systems. The code includes code alignment for the CALL instruction, and different instruction sequences to take advantage of the on-chip cache. Use the `nomod486` control (default) to cause the compiler to generate code for the i386™ processor, which also executes on the i486 processor.

If `notranslate` is specified, the compiler does not generate object code and the instruction set control has no effect. If `noobject` is specified, the effect of the instruction set control on the object code can be seen in the print file, although the compiler does not produce a final object file.

NOTES

An object module compiled with the `mod486` control can execute on an i386 processor, but may execute more slowly than if compiled with the `nomod486` control.

Do not execute a `mod486`-compiled object module that contains i486 processor built-in functions on an i386 processor. The behavior of such code on an i386 processor is unpredictable.

Cross-references

`object | noobject`
`translate | notranslate`

modulename

Primary control
Names object module

Syntax

```
modulename (name)
#pragma modulename (name)
```

Where:

name is the name of the object module (not the object file).

Abbreviation

```
mn
```

Default

The compiler uses the source filename without its extension. For example, the compiler names the object module `main` for the source file `main.c`.

Discussion

Use the `modulename` control to name the object module.

The object module name is used by the linker, binder, locator, and builder. LINK86 does not have the facility to rename object modules, but BND286 and BND386 do have such a facility. The object module name also appears in the print file.

The `notranslate` control overrides the `modulename` control. The `noobject` control overrides the `modulename` control except for its effect on the print file.

NOTE

A `#pragma` preprocessor directive specifying the `modulename` control must precede any `#pragma` directives that specify the `subsys` control.

Cross-references

`object | noobject`
`subsys`
`translate | notranslate`

object | noobject

Primary control
Generates and names
or suppresses object file

Syntax

```
object [(filename)]
noobject
```

```
#pragma object [(filename)]
#pragma noobject
```

Where:

filename is the file specification (including a device name and directory name or pathname, if necessary) in which the compiler places the object code.

Abbreviation

```
[no]oj
```

Default

```
object
```

By default the compiler places the object file in the directory containing the source file. The compiler composes the default object filename from the source filename, as follows:

sourcename.obj

Where:

sourcename is the filename of the primary source file without its file extension.

For example, by default the compiler creates an object file named `main.obj` for the source file `main.c`.

Discussion

Use the `object` control to specify a non-default name or directory for the object file. Use the `noobject` control to suppress creation of an object file.

The `notranslate` control suppresses all translation of source code to object code and suppresses creation of the object file and the print file. The `noobject` control does not suppress translation, and the compiler can produce a print file. The `noobject` control overrides other object file controls except for their effect on the print file.

To place a pseudo-assembly language version of the object code in the print file, use the `code` control.

Cross-references

`code | nocode`
`translate | notranslate`

optimize

Primary control
Specifies the level of optimization

Syntax

```
optimize (level)
#pragma optimize (level)
```

Where:

level	is 0, 1, 2, or 3. The values correspond to the levels of optimization, with 0 being the lowest level (least optimization) and 3 being the highest level (most optimization).

Abbreviation

```
ot
```

Default

optimization level 1

Discussion

Use the `optimize` control to improve the space usage and execution efficiency of a program. Use level 0 when debugging to ensure the closest match between a line of source text and the generated object code for that line. Each optimization level performs all the optimizations of all lower levels. Figure 3-5 summarizes the optimizations performed at each level.

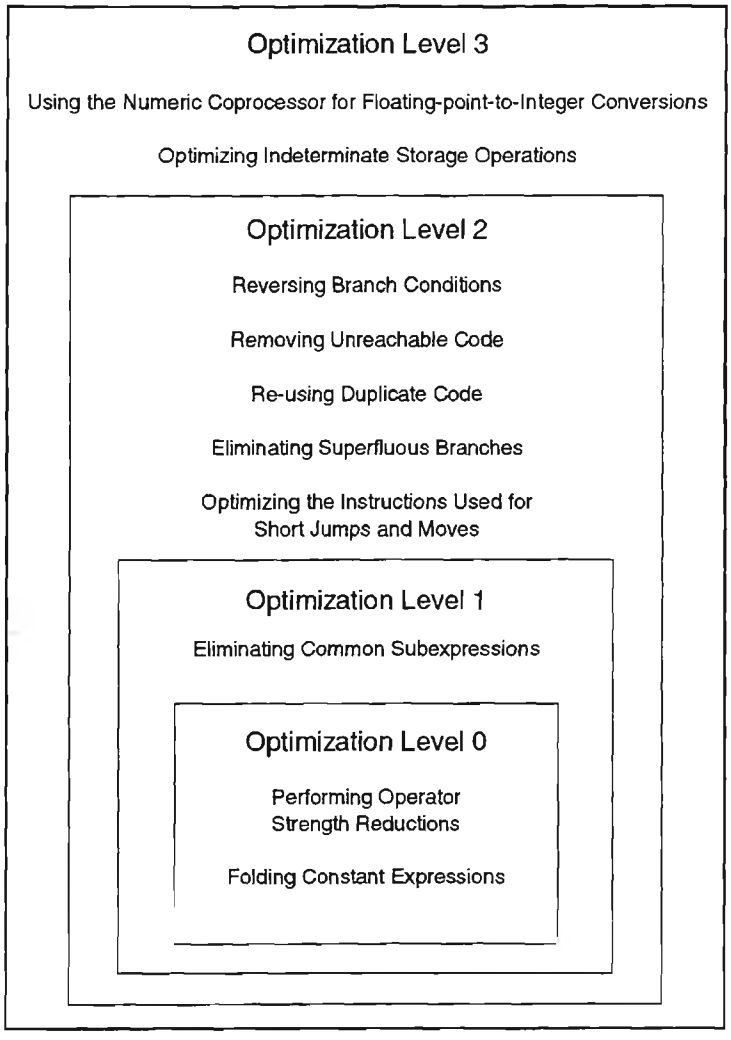

Figure 3-5 Summary of Optimization Levels

The `optimize` control is a primary control. Use it in the compiler invocation or in a `#pragma` preprocessor directive. A primary control in a `#pragma` preprocessor directive must precede the first line of data definition or executable source text. A primary control in the invocation overrides any contradictory control in a `#pragma` preprocessor directive.

See the `compact`, `debug | nodebug`, `line | noline`, `type | notype`, and `small` controls for other ways to optimize code size. See Chapter 2 for an example program compiled at all four levels of optimization.

Folding of Constant Expressions at All Levels

The compiler recognizes operations involving constant operands and removes or combines them to save memory space or execution time. Addition with 0, multiplication by 1, and operations on two or more constants fall into this category. For example, the expression `a+2+3` becomes `a+5`.

Reducing Operator Strength at All Levels

The compiler substitutes quick operations for longer ones, such as shifting left by 1 instead of multiplying by 2. The substituted instruction requires less space and executes faster. The addition of identical subexpressions can also generate left shift instructions.

Eliminating Common Subexpressions at Levels 1, 2, and 3

If an expression reappears in the same block of source text, the compiler generates object code to reuse rather than recompute the value of the expression. It generates code to save intermediate results during expression evaluation in registers and on the stack for later use. The compiler also recognizes commutative forms of subexpressions. For example, in the following block of code the compiler generates code to compute the value of `c*d/3` for the first expression and to save and retrieve it for the second expression:

```
a = b + c*d/3;
c = e + d*c/3;
```

Optimizing the Machine Code of Short Jumps and Moves at Levels 2 and 3

The compiler saves space in the object code by using shorter forms for identical machine instructions.

Eliminating Superfluous Branches at Levels 2 and 3

The compiler combines consecutive or multiple branches into a single branch.

Reusing Duplicate Code at Levels 2 and 3

Duplicate code can be identical code at the ends of two converging paths, or it can be machine instructions immediately preceding a loop identical to those ending the loop. In the first case, the compiler inserts code on only one path and inserts a jump to that path in the other path. In the second case, the compiler generates a branch to reuse the code generated at the beginning of the loop.

Removing Unreachable Code at Levels 2 and 3

The compiler eliminates code that can never be executed. The optimization that removes the unreachable code takes a second pass through the generated object code and finds areas which can never be reached due to the control structures created in the first pass.

Reversing Branch Conditions at Levels 2 and 3

The compiler optimizes the evaluation of Boolean expressions, so only the shorter of two mutually exclusive conditions is evaluated. For example, the following `if` statement on the left has the execution order of its branches reversed:

```
if (!a)                              if (a)
{                                    {
   /* (block 1) */                      /* (block 2) */
}                    /* becomes */   }
else                                 else
{                                    {
   /* (block 2) */                      /* (block 1) */
}                                    }
```

Optimizing Indeterminate Storage Operations at Level 3

The indeterminate storage operations involve pointer indirection. When code assigns a pointer to refer to a variable, it creates an alias for that variable. A variable referenced by a pointer has two aliases: the pointer and the name of the variable itself. Use optimization level 3 when the compiler need not insert code to guard against aliasing.

The compiler performs this level 3 optimization as follows:

- When the code assigns an expression to a variable, the compiler generates code to evaluate the expression and assign the result to the variable. The result also remains in the register used in evaluating the expression.
- When the code subsequently uses the same alias to access the variable, the compiler does not generate code to access the variable; instead it inserts a reference to the register.
- The compiler refers to the same register each time the code uses the alias. Run-time performance is improved because accessing the register executes faster than accessing the variable in memory.

This optimization can introduce errors when the code uses multiply-aliased variables. The compiler does not insert code to check for intermediate references to a variable using a different alias. If the code modifies a variable using a different alias, the value in the variable is not necessarily the same as the value in the register referenced by the compiler. For example, in the following code under optimization level 3, y erroneously acquires the value 1 instead of 2. If the optimization level is less than 3, the compiler codes the assignment correctly:

```
int x,y;
int *a = &x;                    /*  *a is aliasing x    */
x = 1;                          /*  put a value in x    */
*a = 2;                         /*  x now has value 2   */
y = x;                          /*  TROUBLE at level 3! */
```

See *C: A Reference Manual*, listed in Chapter 1, for more information on indirection and aliasing.

Using the Numerics Coprocessor for Floating-point-to-integer Conversions at Level 3

Unsafe conversions of floating-point types to integral types may occur at optimization level 3. The 1989 ANSI C standard specifies that these conversions must use truncation. At optimization level 3, the numerics coprocessor controls the method used in rounding. After RESET, the rounding mode of the numeric coprocessor is round-to-nearest. Therefore, at optimization level 3, the conversion of floating-point types to integral types usually uses rounding, contrary to the standard. At lower optimization levels, these conversions use truncation, which is according to the standard.

Cross-references

`code|nocode`	`line|noline`
`compact`	`small`
`debug|nodebug`	`type|notype`

pagelength

Primary control
Specifies lines per page
in the print file

Syntax

```
pagelength (lines)
#pragma pagelength (lines)
```

Where:

lines is the length of a page in lines. This value can range from 10 to 32767.

Abbreviation

```
pl
```

Default

60 lines per page

Discussion

Use the `pagelength` control to specify the maximum number of lines printed on a page of the print file before a form feed is printed. The number of lines on a page includes the page headings.

The `noprint` and `notranslate` controls suppress the print file, causing the `pagelength` control to have no effect.

The `eject`, `pagewidth`, `tabwidth`, and `title` controls also affect the format of the print file.

See Chapter 5 for a description of the print file. See Chapter 2 for an example of the effect of the `pagelength` control on the print file.

Cross-references

`eject`	`tabwidth`
`pagewidth`	`title`
`print|noprint`	`translate|notranslate`

pagewidth

Primary control

Specifies line length in the print file

Syntax

```
pagewidth (chars)
#pragma pagewidth (chars)
```

Where:

chars is the line length in number of characters. This value ranges from 72 through 132.

Abbreviation

```
pw
```

Default

120 characters

Discussion

Use the `pagewidth` control to specify the maximum length, in characters, of lines in the print file.

The `noprint` and `notranslate` controls suppress the print file, causing the `pagewidth` control to have no effect.

The `eject`, `pagelength`, `tabwidth`, and `title` controls also affect the format of the print file.

See Chapter 5 for a description of the print file. See Chapter 2 for an example of the effect of the `pagewidth` control on the print file.

Cross-references

`eject`	`tabwidth`
`pagelength`	`title`
`print	noprint`	`translate	notranslate`

preprint | nopreprint

Invocation control
Generates or suppresses a preprocessed source text listing file

Syntax

```
preprint [(filename)]
nopreprint
```

Where:

filename	is the file specification (including a device name and directory name or pathname, if necessary) in which the compiler places the preprint information.

Abbreviation

```
[no]pp
```

Default

`nopreprint`	when the translate control is in effect.
`preprint`	when the notranslate control is in effect.

By default, the compiler places the preprint file in the directory containing the source file. The compiler composes the default preprint filename from the source filename as follows:

sourcename`.i`

Where:

sourcename	is the filename of the primary source file without its file extension.

For example, by default the compiler creates a preprint file named `proto.i` for the source file `proto.c`.

Discussion

Use the `preprint` control to create a file containing the text of the source after preprocessing. Use the `nopreprint` control (default) to suppress creation of a preprint file. Preprocessing includes file inclusion, macro expansion, and elimination of conditional code. The preprint file is the intermediate source text after preprocessing and before compilation.

The preprint file is especially useful for observing the results of macro expansion, conditional compilation, and the order of include files. If the preprint file contains no errors, compiling the preprint file produces the same results as compiling the primary source file and any files included in the compiler invocation.

The `preprint` control creates a file different from the print file. The `eject`, `pagelength`, `pagewidth`, `tabwidth`, and `title` controls have no effect on the preprint file.

When the `preprint` control is in effect, the maximum nesting level of include files is 7.

See Chapter 5 for a description of the preprint and print files. See Chapter 2 for an example of the effect of the `preprint` or `nopreprint` controls and other listing controls.

Cross-references

`include`
`print | noprint`

print | noprint

Primary control

Generates or suppresses the print file

Syntax

```
print [(filename)]
noprint

#pragma print (filename)
#pragma noprint
```

Where:

filename is the file specification (including a device name and directory name or pathname, if necessary) in which the compiler places the print information.

Abbreviation

```
pr
```

Default

```
print
```

By default the compiler places the print file in the directory containing the source file. The compiler composes the default print filename from the source filename, as follows:

sourcename`.lst`

Where:

sourcename is the filename of the primary source file without its file extension.

For example, the compiler creates a print file named `main.lst` for the source file `main.c`.

Discussion

Use the `print` control to produce a text file of information about the source and object code. Use the `noprint` control to suppress the print file. The `noprint` control causes the compiler to display diagnostic messages only at the console.

The `noprint` control overrides all other listing controls except the `preprint` control. The `notranslate` control overrides the `print` control. The `noprint` control causes diagnostic messages to appear at the console.

The `print` control creates a print file different from the preprint file.

The `code | nocode`, `cond | nocond`, `diagnostic`, `list | nolist`, `listexpand | nolistexpand`, `listinclude | nolistinclude`, `symbols | nosymbols`, and `xref | xref` controls affect the contents of the print file. The `eject`, `pagewidth`, `pagelength`, `tabwidth`, and `title` controls affect the format of the print file.

See Chapter 5 for a description of the print file. See Chapter 2 for an example of the effect of listing controls on the print file.

Cross-references

code | nocode
cond | nocond
diagnostic
eject
list | nolist
listexpand | nolistexpand
listinclude | nolistinclude
pagelength
pagewidth
preprint | nopreprint
symbols | nosymbols
tabwidth
title
translate | notranslate
xref | noxref

Syntax

```
ram
rom

#pragma ram
#pragma rom
```

Abbreviation

(none)

Default

`ram`

Discussion

Use the `ram` control (default) to place constants in the data segment in memory. When the `ram` control is in effect, the compiler initializes to zero all static variables not explicitly initialized in the source text.

Use the `rom` control to place constants in the code segment in memory. When the `rom` control is in effect, the compiler does not initialize any static variables not explicitly initialized in the source text. Also, the compiler produces warning messages for all static variables the code explicitly initializes.

Constants can be defined in the code or defined by the compiler. Constants include the values of string literals, floating-point literals, and static variables declared with the `const` attribute specifier.

The `ram` or `rom` controls have little effect when used with the `flat` control (iC-386 only), because the compiler always places constants in the code segment. The `rom` or `ram` control does affect the value of the `_ROM_` predefined macro. See Chapter 5 for information on predefined macros.

The `compact`, `flat`, `large`, `medium`, and `small` controls determine the segmentation model for the object code. The segmentation model determines how many code and data segments are present in the object code.

The `notranslate` control overrides the `ram` and `rom` controls. The `noobject` control overrides the `ram` and `rom` controls except for their effect on the print file.

See Chapter 4 for more specific information on segmentation. See *C: A Reference Manual*, listed in Chapter 1, for information on the `const` attribute specifier and the `static` storage class.

Cross-references

`compact`
`flat`
`large`
`medium`
`object | noobject`
`small`
`translate | notranslate`

Syntax

```
searchinclude (pathprefix [,...])
nosearchinclude

#pragma searchinclude (pathprefix [,...])
#pragma nosearchinclude
```

Where:

pathprefix is a string of characters that the compiler prepends to the filename argument of an instance of the `include` or `subsys` control, or to the file argument of an `#include` preprocessor directive. If the path prefix contains special characters such as the slash (/), enclose the *pathprefix* in quotation marks (").

Abbreviation

```
[no]si
```

Default

```
nosearchinclude
```

The three default path prefixes are derived from the directory containing the primary source file, the `:include:` environment variable, and the null prefix (current directory). The compiler always uses the path prefix in the `:include:` environment variable after the list specified by the `searchinclude` control.

Discussion

Use the `searchinclude` control to specify a list of possible path prefixes for include files. Use the `nosearchinclude` control (default) to limit the compiler to the three default search path prefixes. Each *`pathprefix`* argument is a string that, when concatenated to a filename, specifies the relative or absolute path of a file (including a device name and directory name, if necessary). The compiler tries each prefix in the order in which they are specified, until a legal filename is found. If a legal filename is not found, the compiler issues an error.

The DOS `:include:` environment variable can specify a path prefix to the name of a directory containing include files.

Include files are files specified with the `include` control or the `subsys` control in the compiler invocation or with the `#include` preprocessor directive in the source text. In the `#include` preprocessor directive, source files are surrounded by quotation marks (`"`), and header files are surrounded by angle brackets (`<>`).

When the compiler searches for a file specified in the `include` control, or when it searches for a source file (surrounded by quotation marks) specified in an `#include` preprocessor directive, the compiler tests the prefixes in the following order:

1. the source directory prefix
2. the directories specified by the `searchinclude` list
3. the directory or directories specified by the `:include:` environment variable, if defined
4. the null prefix, that is, the current directory

When the compiler searches for a header file (surrounded by angle brackets) specified in an `#include` preprocessor directive, the compiler tests the prefixes in the following order:

1. the directories specified by the `searchinclude` list
2. the directory or directories specified by the `:include:` environment variable, if defined
3. the source directory prefix
4. the null prefix, that is, the current directory

The maximum number of path prefixes for the `searchinclude` control is 19. The maximum number of files open simultaneously during compilation is system dependent.

The iC-86/286/386 compiler on DOS has two added facilities for searching for files. The compiler maps slashes (/) in filenames to backslashes (\). When a pathname begins with an environment variable, the compiler uses the value of the environment variable as the directory path prefix and applies the mappings to all filenames including prefixes specified with the `searchinclude` control.

The `searchinclude` and `nosearchinclude` controls are general controls. Use them freely in the compiler invocation or in `#pragma` preprocessor directives. Specifying the `searchinclude` control more than once adds to the search path prefix list. Specifying the `nosearchinclude` control after the `searchinclude` control does not eliminate the `searchinclude` list.

Cross-references

```
include
subsys
```

signedchar | nosignedchar

Primary control
Sign-extends or zero-extends
char objects when promoted

Syntax

```
[no]signedchar
#pragma [no]signedchar
```

Abbreviation

```
[no]sc
```

Default

```
signedchar
```

Discussion

Use the `signedchar` control (default) to specify that objects declared to be the `char` data type are treated as if they were declared as the `signed char` data type. The compiler sign-extends these objects when they are converted to a data type that occupies more memory.

Use the `nosignedchar` control to specify that objects declared as the `char` data type are treated as if they were declared as the `unsigned char` data type. The compiler zero-extends these objects when they are converted to a data type that occupies more memory.

If `notranslate` is specified, the compiler does not generate object code and the `signedchar` and `nosignedchar` controls have no effect. If `noobject` is specified, the effect of the `signedchar` and `nosignedchar` controls on the object code can be seen in the print file, although the compiler does not produce a final object file.

The `signedchar` control does not affect the interpretation of objects specifically declared as either `signed char` or `unsigned char` data types.

See *C: A Reference Manual*, listed in Chapter 1, for more information on the `char`, `unsigned char`, and `signed char` data types and data type conversion.

Cross-references

`object | noobject`
`translate | notranslate`

small

Primary control
Specifies the small segmentation memory model

Syntax

```
small
#pragma small
```

Abbreviation

```
sm
```

Default

Of the four iC-86 and iC-286 memory model specifications (`small`, `compact`, `medium`, and `large`) and the three iC-386 memory model specifications (`small`, `compact`, and `flat`), the default is `small`.

Discussion

Use the `small` control (default) to specify the small segmentation model. The compiler produces an object module containing a code segment, a data segment, and a separate stack segment. The linker or binder combines the code segments for all modules into a single code segment, combines the data segments for all modules into a single data segment, and reserves space in the data segment to accommodate all stack activity. The small segmentation model is efficient in both program size and memory access.

For 86 and 286 processors, code, data, or stack segments each can occupy up to 64 kilobytes of memory. For Intel386 processors, each segment can occupy up to 4 gigabytes of memory.

The processor addresses the small model's code segment relative to the CS register, the data in the combined data and stack segment relative to the DS register, and the stack in the combined data and stack segment relative to the SS register (which has the same value as the DS register). Depending on whether the `rom` or `ram` control is in effect, the compiler places the constants from each module in the corresponding code segment or the combined data and stack segment, respectively. All functions have `near` pointers and calls. If the `ram` control is in effect, all data pointers are `near` pointers. If the `rom` control is in effect, all data pointers are `far` pointers. See the `extend|noextend` control and Chapter 4 for more information about the `far` and `near` keywords.

The `notranslate` control overrides the `small` control. The `noobject` control overrides the `small` control except for its effect on the print file.

See Chapter 2 for more information on the availability of run-time libraries for the various memory models.

See Chapter 4 for more specific information on segmentation and the small memory model.

Cross-references

`compact`	`medium`
`extend|noextend`	`object|noobject`
`flat`	`ram|rom`
`large`	`translate|notranslate`

subsys

Primary control
Reads a subsystem specification

Syntax

```
subsys (filename [,...])
#pragma subsys (filename [,...])
```

Where:

filename is the file specification (including a device name and directory name or pathname, if necessary) in which the compiler finds the subsystem definition.

Abbreviation

(none)

Default

(none)

Discussion

Use the `subsys` control to cause the compiler to read one or more files for subsystem definitions. The compiler searches for the named files the same way that it searches for source files surrounded by quotation marks in the `#include` preprocessor directive. See the entry for the `searchinclude` control in this chapter for the search method. See Chapter 9 for how to define subsystems.

The compiler preserves case distinction in identifiers in `exports` lists. The compiler always ignores dollar signs (`$`) in identifiers, even if the `extend` control is not in effect. The compiler ignores valid PL/M controls unrelated to segmentation, such as `$IF` and `$INCLUDE`. The compiler ignores lines whose first character is not a dollar sign (`$`).

A subsystem can export only function and variable names with file scope. The compiler implicitly modifies declarations of exported symbols, if necessary, by inserting the `far` keyword in the appropriate place. The modifications occur even if the `extend` control is not in effect.

If `notranslate` is specified, the compiler does not generate object code and the `subsys` control has no effect. If `noobject` is specified, the effect of the `subsys` control on the object code can be seen in the print file, although the compiler does not produce a final object file.

NOTES

A `#pragma` preprocessor directive specifying the `modulename` control must precede any `#pragma` directives that specify the `subsys` control.

Do not use the `codesegment` or `datasegment` controls in an invocation that specifies the `subsys` control. The compiler issues an error or a warning message, depending on whether the `subsys` control is found in the invocation line or in a `#pragma` preprocessor directive.

See *C: A Reference Manual*, listed in Chapter 1, for information on the `#include` preprocessor directive and the scope of identifiers. See Chapter 9 for a detailed discussion of subsystems. See the `extend | noextend` control for more information on the `far` keyword. See Chapter 4 for more information on segmentation memory models and how to use the `far` keyword.

Cross-references

`code segment`
`data segment`
`extend | noextend`
`modulename`
`object | noobject`
`searchinclude | nosearchinclude`
`translate | notranslate`

symbols | nosymbols

Primary control
Generates or suppresses identifier list in print file

Syntax

```
[no]symbols
#pragma [no]symbols
```

Abbreviation

```
[no]sb
```

Default

```
nosymbols
```

Discussion

Use the `symbols` control to include in the print file a table of all identifiers and their attributes from the source text. Use the `nosymbols` control (default) to suppress the table.

The `noprint` and `notranslate` controls override `symbols`. The `xref` control causes the compiler to generate a cross-referenced symbol table even if the `nosymbols` control is specified.

See Chapter 5 for a description of the print file. See Chapter 2 for an example of the effect of the `symbols` or `nosymbols` controls on the print file.

Cross-references

print | noprint
translate | notranslate
xref | noxref

tabwidth

Primary control
Specifies characters per tab stop in the print file

Syntax

```
tabwidth (width)
#pragma tabwidth (width)
```

Where:

width is a value from 1 to 80. This value is the number of characters from tab stop to tab stop in the print file.

Abbreviation

```
tw
```

Default

4 characters per tab stop

Discussion

Use the `tabwidth` control to specify the number of characters between tab stops in the print file.

The `noprint` and `notranslate` controls suppress the print file, causing the `tabwidth` control to have no effect.

The `eject`, `pagewidth`, `pagelength`, and `title` controls also affect the format of the print file.

See Chapter 2 for an example of the effect of the `tabwidth` control on the print file.

Cross-references

`eject`
`pagelength`
`pagewidth`
`print | noprint`
`title`
`translate | notranslate`

title

Primary control
Specifies the print file title

Syntax

```
title ("string")
#pragma title ("string")
```

Where:

string is the title.

Abbreviation

```
tt
```

Default

The compiler uses the object module name.

Discussion

Use the `title` control to specify the print file title. The compiler places the title at the top of each page of the print file.

To specify no title, use at least one blank in the title string. Do not use the null string.

A title can be up to 60 characters long. A narrow page width can restrict a title to fewer than 60 characters. In such cases, the compiler truncates the title from the right.

The `noprint` and `notranslate` controls suppress the print file, causing the `title` control to have no effect.

The `eject`, `pagewidth`, `pagelength`, and `tabwidth` controls also affect the format of the print file.

See Chapter 5 for a description of the print file.

Cross-references

`eject`
`modulename`
`object | noobject`
`pagelength`
`pagewidth`
`print | noprint`
`tabwidth`
`translate | notranslate`

translate | notranslate

Invocation control
Compiles or suppresses compilation after preprocessing

Syntax

```
[no]translate
```

Abbreviation

```
[no]tl
```

Default

```
translate
```

Discussion

Use the `translate` control (default) to cause the compilation to continue after preprocessing. Translation includes parsing the input, checking for errors, generating code, and producing an object module. Use the `notranslate` control to cause compilation to cease after preprocessing.

The `notranslate` control implies the `preprint` control. The `notranslate` control overrides all other object and listing controls except for their effect on the print file. The `notranslate` control causes preprocessing diagnostic messages to appear at the console.

Cross-references

`object | noobject`
`preprint | nopreprint`

Syntax

```
[no]type
#pragma [no]type
```

Default

```
type
```

Abbreviation

```
ty
```

Discussion

Use the `type` control (default) to include type information for public and external symbols in the object module. Use the `notype` control to suppress generation of type information. Suppressing type information reduces the size of the object module.

Type information can be useful to other tools in the application development process. The binder uses type information to perform type checking across modules. A debugger or an emulator uses type information to display symbol attributes.

The `noobject` and `notranslate` controls cause `type` and `notype` to have no effect.

See the discussion of the `debug` control for information on combining controls that affect the size of the object module, such as the `line` control.

The `optimize` control can further reduce the size of the object module. However, higher levels of optimization reduce the ability of most symbolic debuggers to accurately correlate debug information to the source code. The `line` control causes the compiler to place source line number debug information in the object file. The `symbols` control puts a listing of all identifiers and their types into the print file. The `xref` control puts a cross-reference listing of all identifiers into the print file.

See *C: A Reference Manual*, listed in Chapter 1, for more information on type definitions.

Cross-references

`debug | nodebug`
`line | noline`
`object | noobject`
`optimize`
`symbols | nosymbols`
`translate | notranslate`
`xref | noxref`

Syntax

```
varparams [(function [,...])]
#pragma varparams [(function [,...])]
```

Where:

function is the name of a function defined in the source text. Case is significant in function-name arguments.

Abbreviation

```
vp
```

Default

Of the two calling convention specifications (`fixedparams` and `varparams`), the default is `fixedparams`. If you specify `varparams` but do not supply a *function* argument, the `varparams` control applies to all functions in the subsequent source text.

Discussion

Use the `varparams` control to require the specified functions to use the variable parameter list (VPL) calling convention. Most of Intel's non-C compilers generate object code for function calls using the fixed parameter list (FPL) calling convention. Some earlier versions of Intel C use the variable parameter list calling convention.

A function's calling convention dictates the sequence of instructions that the compiler generates to manipulate the stack and registers during a call to a function. The VPL calling convention is as follows:

1. The calling function pushes the arguments onto the stack with the rightmost argument pushed first before control transfers to the called function.
2. The calling function removes the arguments from the stack after control returns from the called function.

See Chapter 8 for more detailed information on the FPL and VPL calling conventions.

The VPL calling convention provides more flexibility than the FPL calling convention. Use the VPL calling convention for functions that take a variable number of parameters. See the `fixedparams` control for more information on the FPL calling convention.

A calling convention specified without an argument in the compiler invocation affects functions throughout the entire module. If a function uses a calling convention other than the one in effect for the compilation, specify the calling convention before declaring the function.

If FPL is in effect globally, you can use an ellipsis in a prototype or declaration to declare a VPL function, or use the `varparams` control. If VPL is in effect globally, you must use the `fixedparams` control in a `#pragma` preprocessor directive to declare an FPL function.

If `notranslate` is specified, the compiler does not generate object code and the calling convention control has no effect. If `noobject` is specified, the effect of the calling convention control on the object code can be seen in the print file, although the compiler does not produce a final object file.

NOTE

An error occurs if a function in the source text explicitly declares a variable parameter list and also is named in the *function* list for the `fixedparams` control. In the following example, the ellipsis in the `fvprs` function prototype indicates a VPL convention for this function. Specifying the `fixedparams (fvprs)` control in this case causes an error:

```
#include <stdarg.h>
fvprs (int a, ...);
```

The `varparams` and `fixedparams` controls are general controls. Use them freely in the compiler invocation or in `#pragma` preprocessor directives. If you specify both controls without arguments in the invocation, the compiler acts on the most recently encountered control. These controls only affect the subsequent source text and remain in effect until the compiler encounters a contrary control or the end of the source text.

See the entry for the `extend | noextend` control in this chapter for other information on code compatibility with previous versions of Intel C. See the entry for the `fixedparams` control for information on the fixed parameter list calling convention.

Examples

The following examples show different uses of the `fixedparams` and `varparams` controls.

1. This combination of controls specifies variable parameter list convention (VPL) for all functions in the source file except those in the argument list. Use the controls on the invocation line as follows:

   ```
   varparams fixedparams (argument_list)
   ```

   Or use the controls in `#pragma` preprocessor directives as follows:

   ```
   #pragma varparams
   #pragma fixedparams (argument_list)
   ```

2. This control specifies fixed parameter list convention (FPL) for all functions in the source file except those in the argument list. Use the `varparams` control on the invocation line to override the default for the function in the argument list as follows:

   ```
   varparams (argument_list)
   ```

   Or use the `varparams` control in a `#pragma` preprocessor directive as follows:

   ```
   #pragma varparams (argument_list)
   ```

Cross-references

`extend | noextend`
`fixedparams`
`object | noobject`
`translate | notranslate`

Syntax

```
[no]xref
#pragma [no]xref
```

Abbreviation

```
[no]xr
```

Default

```
noxref
```

Discussion

Use the `xref` control to add cross-reference information to the symbol table listing in the print file. Use the `noxref` control (default) to suppress the cross-reference information.

The `noprint` and `notranslate` controls override the `xref` control. The `xref` and `symbols` controls are similar, except that the `xref` control adds a cross-reference listing of identifiers from the source program. The `xref` control causes the compiler to generate a cross-referenced symbol table even if the `nosymbols` control is specified.

The print file lists the cross-reference line numbers on the far right under the "Attributes" column in the symbol table listing. The "Attributes" column describes the data or function type. A number with an asterisk (*) indicates the line where the object or function is declared. A number without an asterisk indicates a line where the object or function is accessed. The cross-reference line numbers refer to the line numbers in the source text listing in the print file. See Chapter 2 for an example of a cross-reference listing.

See Chapter 5 for more information on the symbol table and the print file. See Chapter 2 for an example of the effect of the `xref` and `noxref` controls on the symbol table in the listing.

Cross-references

```
print | noprint
symbols | nosymbols
translate | notranslate
```

Contents

4

Segmentation Memory Models

Segmentation Memory Models

4

This chapter discusses how segmentation memory models manage code, data, and stacks for the 86, 286, and Intel386™ segmented architectures. This chapter contains the following topics:

- how the linker and binder combine the compiler-created segments
- characteristics of the small, compact, medium, large, and flat memory models
- how to use and interpret the `far` and `near` keywords

The iC-86 and iC-286 compilers use four segmentation memory models: small, compact, medium, and large. The iC-386 compiler uses three segmentation memory models: small, compact, and flat. For iC-386, the medium model is equivalent to the small model, and the large model is equivalent to the compact model. Section 4.2 explains each model in detail.

The small and flat segmentation models are the most efficient models. Data access is less efficient in the compact and large models, but these models separate the data segment from the stack and enable access to other data segments without specifying the `far` keyword. Code access is less efficient in the medium and large models.

A segment for an 86 or 286 processor must not exceed 64 kilobytes. An Intel386 processor segment can be as large as 4 gigabytes. Figure 4-1 shows how to choose a segmentation model if your 86 or 286 application outgrows its current model.

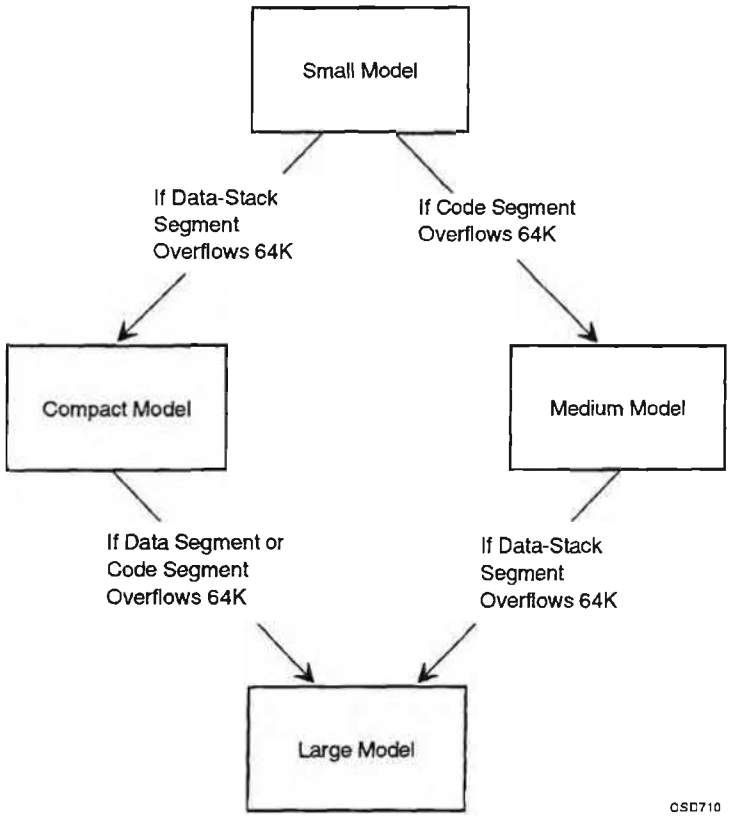

Figure 4-1 Choosing a Segmentation Memory Model for iC-86 or iC-286

4.1 How the Linker and Binder Combine Segments

Segmentation divides a program into units that contain the program's code, data, and stack. Segmentation makes references to memory locations more efficient. The compiler places information defining segment attributes and content into each object module. The linker or binder combines the compiler's segments according to their definitions, thereby implementing the segmentation memory model.

A segment represents a contiguous set of memory locations, but does not necessarily have a fixed address or fixed size until placed in memory for execution. The LOC86 locator, BLD286 or BLD386 system builder, or operating system loader assigns a fixed address to a segment and establishes its size. The maximum size of an 86 or 286 segment is 64 kilobytes, and of an Intel386 processor segment is 4 gigabytes.

4.1.1 Combining iC-86 Segments With LINK86

The LINK86 linker combines segments from its input modules if they have the following characteristics:

- the same segment name
- the same overlay name
- the same combine-type
- a combined length no greater than 64 kilobytes

The group name identifies segments that must be kept within the same 64 kilobytes segment in memory.

The class name identifies segments that share common attributes and should be kept together in the same area of memory, but not necessarily in the same segment.

The iC-86 compiler places in each object module the following segment definition characteristics for each compiler-created segment:

- the segment name
- a null overlay name for all segments
- the combine-type
- the size of the segment
- the group name
- the class name

The iC-86 compiler assigns the word segment alignment attribute to all segments, and does not assign the inpage attribute. Any alignment attribute except byte can result in unused memory between combined segments. The alignment attributes are as follows:

- Byte, which means a segment starts at any address.
- Word, which means a segment starts at an address that is a multiple of 2, starting from address 0H (for example, 0H, 2H, 4H, ...). iC-86 assigns this attribute to all segments.
- Paragraph, which means a segment starts at an address that is a multiple of 16, starting from address 0H (for example, 0H, 10H, 20H, ...).
- Page, which means a segment starts at an address that is a multiple of 256, starting from address 0H (for example, 0H, 100H, 200H, ...).
- Inpage, which means a segment starts at an address according to one of the previous alignment attributes, and not cross a page boundary. An inpage segment must not be larger than 256 kilobytes. iC-86 does not assign this attribute.

4.1.2 Combining iC-286 Segments With BND286

The BND286 binder combines segments from the input object modules if they have the following characteristics:

- the same segment name
- the same kind of contents, i.e., code or data
- the same privilege level
- compatible access rights
- compatible combine-types
- a combined length no greater than 64 kilobytes

The iC-286 compiler places in each object module the following segment definition characteristics for each compiler-created segment:

- the segment name
- whether the segment is code or data
- privilege level 3
- segment access rights: non-conforming, not present, and not expand-down for all segments; and whether code is readable or data is writable
- the combine-type
- the size of the segment

See Chapter 6 for more information on the characteristics of a 286 processor segment.

4.1.3 Combining iC-386 Segments With BND386

The BND386 binder combines segments from the input object modules if they have the following characteristics:

- the same segment name
- the same kind of contents, i.e., code or data
- the same privilege level
- compatible granularity and default operand and address size
- compatible access rights
- compatible combine-types
- a combined length no greater than 4 gigabytes

The iC-386 compiler places in each object module the following segment definition characteristics for each compiler-created segment:

- the segment name
- whether the segment is code or data
- privilege level 3
- byte granularity and 32-bit operand and address size
- segment access rights: non-conforming, not present, and not expand-down for all segments; and whether code is readable or data is writable
- the combine-type
- the size of the segment

See Chapter 6 for more information on the characteristics of an Intel386 processor segment.

4.1.4 How Subsystems Extend Segmentation

A subsystem is a collection of modules that use the same segmentation model. A program can be made up of one or more subsystems. Subsystems allow collections of program modules that are compiled with different segmentation controls to be combined into the same program. For detailed information on the use and syntax of subsystems, see Chapter 9.

4.2 Segmentation Memory Models

The segmentation memory model determines the number of segments and the contents of those segments in the compiler-created object module. The linker or binder uses the segments from each compiled object module to create the linked object module. The `small`, `compact`, `medium`, `large`, and `flat` compiler controls determine the segmentation model that the compiler uses to create an object module.

NOTE

The iRMX® I and II operating systems support only the compact and large segmentation memory models, and the iRMX III operating system supports only the compact segmentation memory model.

There are four components of object code:

- code (executable instructions)
- data (global and static variables)
- stack (function activation records, automatic variables, and any compiler-generated temporary storage not explicitly declared in the source module)
- constants (statically allocated constant objects, character strings and floating-point literals, and other compiler-generated constant values)

The compiler creates a code segment for executable instructions, a data segment for global and static variables, and a stack segment for stack activity. The `ram` and `rom` controls determine whether the compiler puts the constants with the code segment or the data segment. If you specify the `rom` control during compilation, the compiler places the constants in the code segment. If you specify the `ram` control during compilation or accept the default, the compiler places the constants in the data segment.

The segmentation memory model of your application determines the segmentation model of the run-time libraries to which you bind your applications. Figures 4-2 and 4-3 show how to select the segmentation memory model of the libraries for linking or binding with your program.

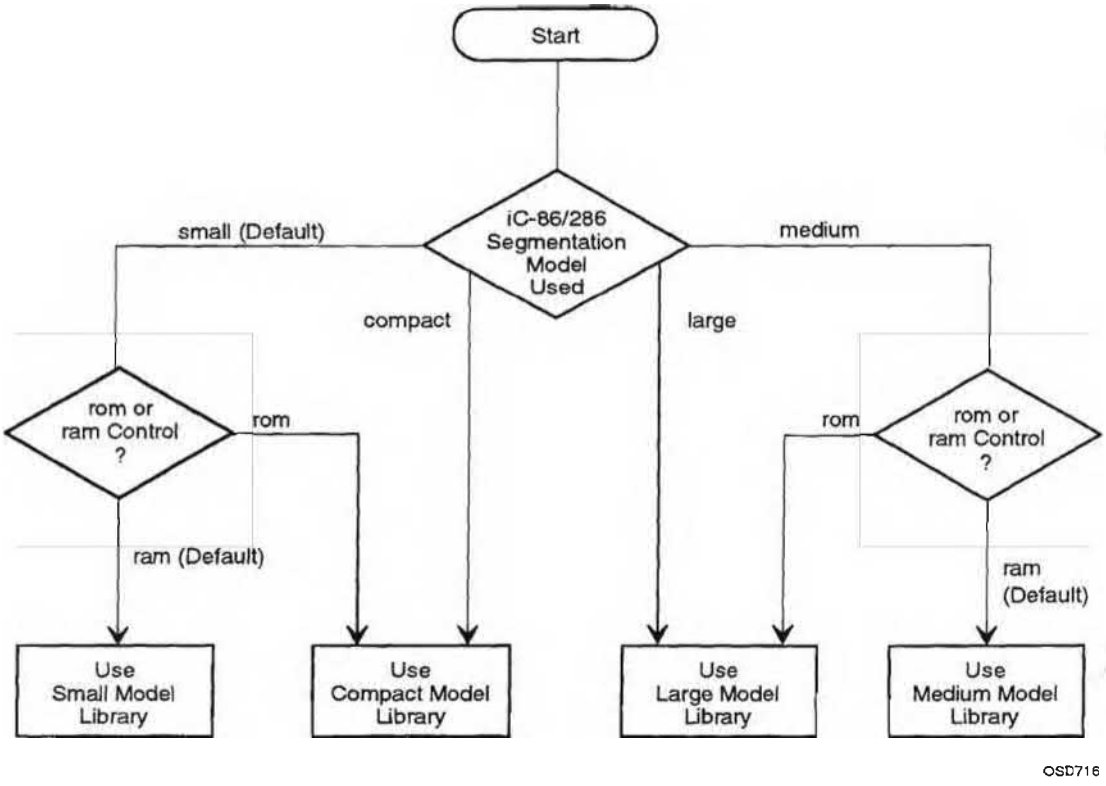

Figure 4-2 Choosing the Segmentation Model of a Library for iC-86 or iC-286

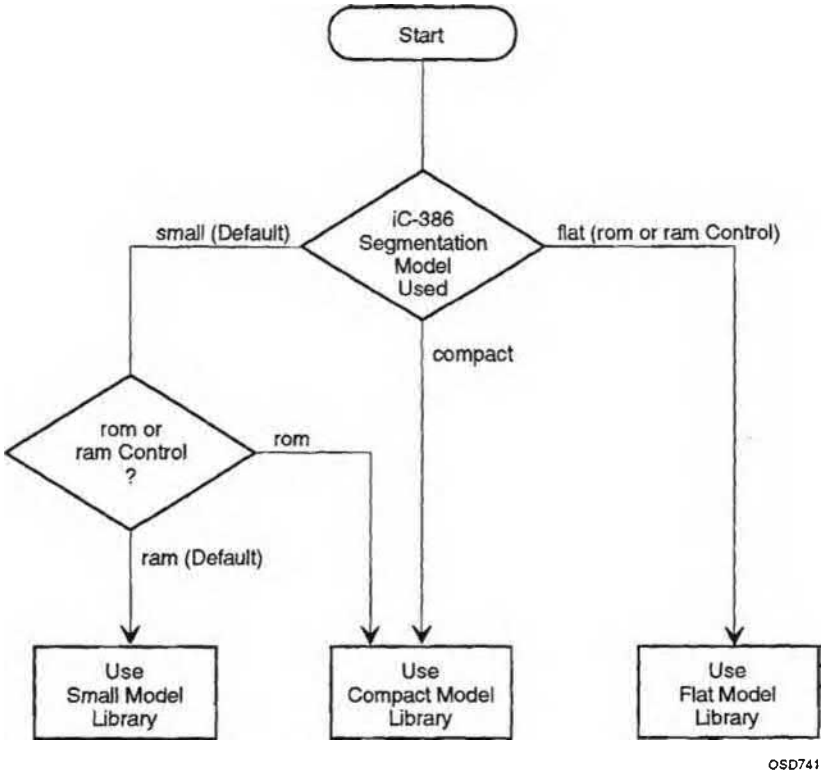

Figure 4-3 Choosing the Segmentation Model of a Library for iC-386

4.2.1 Small Models

Recall that the LINK86 linker combines iC-86 segments with the same name and compatible combine-types, and the linker ensures that segments with the same group name reside in the same 64K-byte segment. The BND286 and BND386 binders combine compiler-generated segments that have the same name, compatible combine-types, and the same access attributes.

A program using the small segmentation memory model contains two segments: `CODE` (iC-86 or iC-286) or `CODE32` (iC-386) and `DATA`. The CS register contains the selector for the code segment. The DS and SS registers contain the selector for the `DATA` segment. For iC-386, the ES register also contains the selector for `DATA`.

Tables 4-1 through 4-3 show the compiler segment definitions for a module compiled with the `small` control. When you specify the `rom` control, the compiler places the constants in the module's code segment. When you specify the `ram` control, the iC-86 compiler creates a segment for the constants, which the linker combines with other `DGROUP` segments to make the data segment. Under the `ram` control, both the iC-286 and iC-386 compilers place the constants in the module's data segment.

Table 4-1 iC-86 Segment Definitions for Small Model Modules

Description	Name	Combine-type	Group
code segment	CODE	concatenate	CGROUP
constant segment (only with ram control)	CONST	concatenate	DGROUP
data segment	DATA	concatenate	DGROUP
stack segment	STACK	overlay additively	DGROUP

Table 4-2 iC-286 Segment Definitions for Small Model Modules

Description	Name	Combine-type	Access
code segment	CODE	normal	execute-read
data segment	DATA	normal	read-write
stack segment	DATA	stack	read-write

Table 4-3 iC-386 Segment Definitions for Small Model Modules

Description	Name	Combine-type	Access
code segment	CODE32	normal	execute-read
data segment	DATA	normal	read-write
stack segment	DATA	stack	read-write

The resulting linked small model module contains one code segment up to 64 kilobytes (iC-86 and iC-286) or 4 gigabytes (iC-386) long, and one combined data-stack segment up to 64 kilobytes (iC-86 and iC-286) or 4 gigabytes (iC-386) long.

The small segmentation memory model is efficient in both program size and memory access. Using the small segmentation memory model restricts your program to 128 kilobytes (iC-86 and iC-286) or 8 gigabytes (iC-386) of memory.

Since all the executable instructions fall within one segment, function pointers are near by default (the offset-only address format). If you specify the `ram` control, all variables, temporary variables, and constants fall within one segment, and data pointers are near by default. If you specify the `rom` control, which places constants in the code segment, data pointers are far (the segment-selector-and-offset address format). See Section 4.3 for an explanation of near and far address formats.

Figures 4-4 and 4-5 show the process of linking or binding a small RAM and a small ROM program from two modules. The relative sizes of the final segments are not to scale. The order of modules in the linker or binder input list affects the order of segments in the output file.

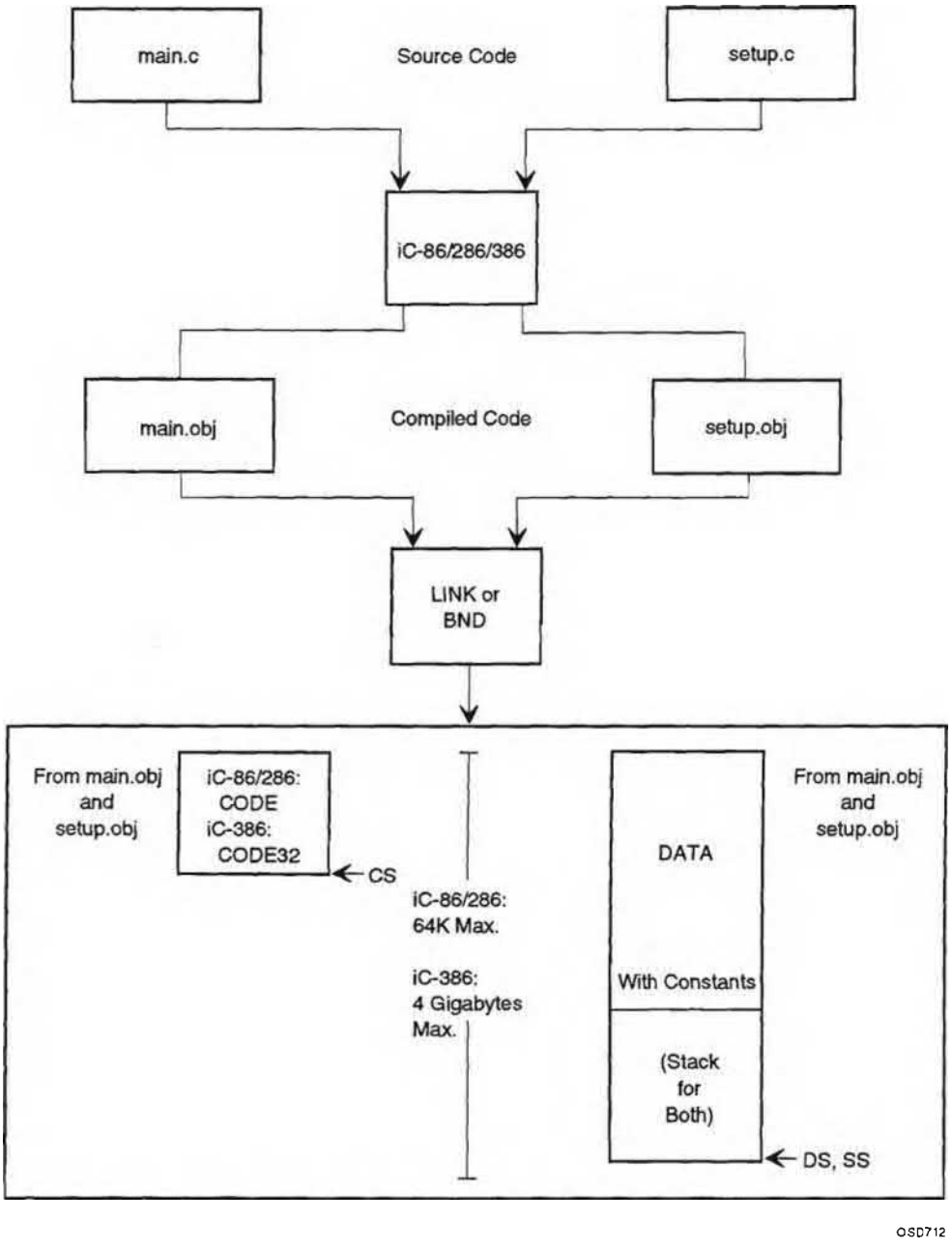

Figure 4-4 Creating a Small RAM Program

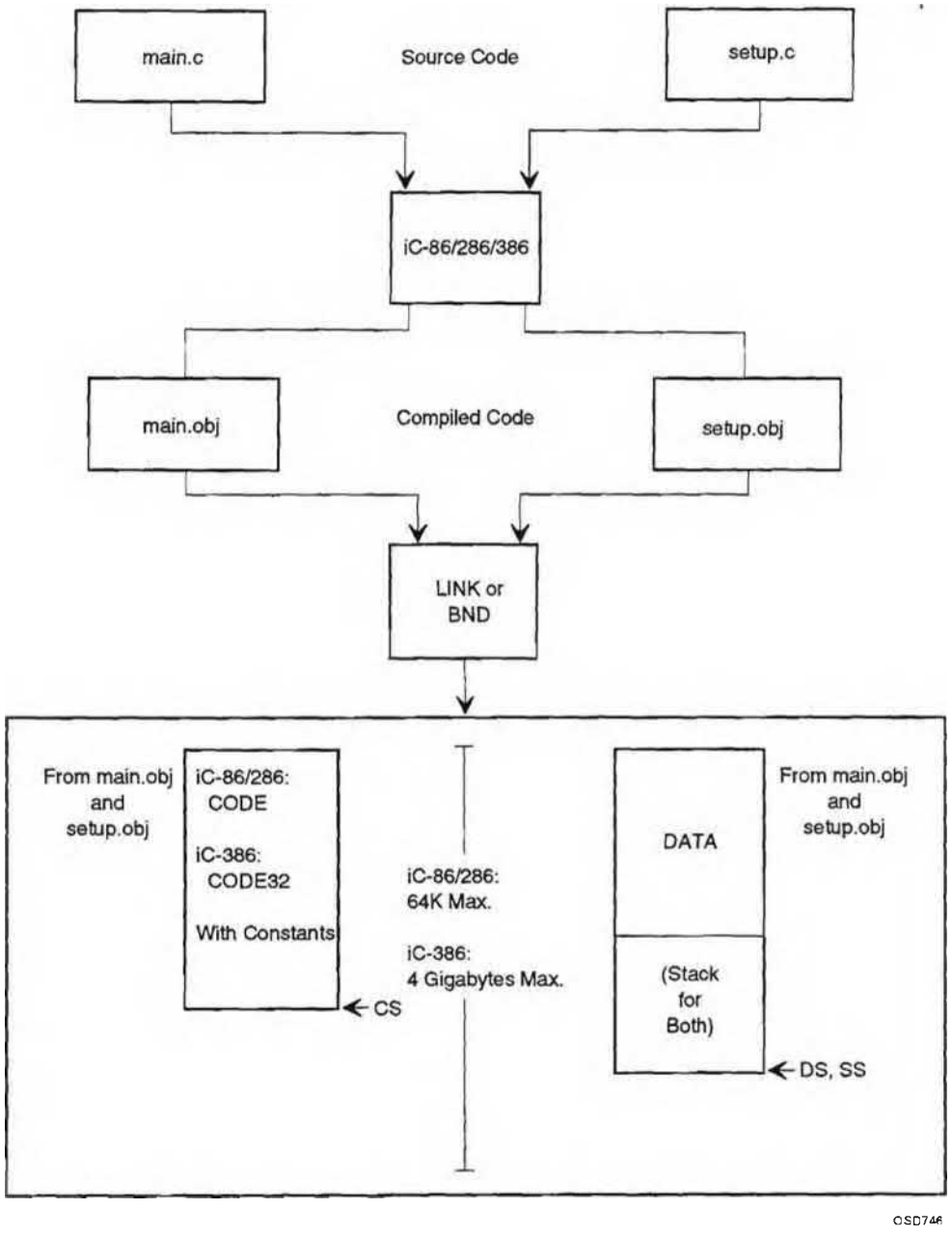

Figure 4-5 Creating a Small ROM Program

4.2.2 Compact Models

Recall that the LINK86 linker combines iC-86 segments with the same name and compatible combine-types, and the linker ensures that segments with the same group name reside in the same 64K-byte segment. The BND286 and BND386 binders combine compiler-generated segments that have the same name, compatible combine-types, and the same access attributes.

A program using the compact segmentation memory model contains three segments: `CODE` (iC-86 or iC-286) or `CODE32` (iC-386), `DATA`, and `STACK`. The CS, DS, and SS registers contain the selectors for the `CODE` or `CODE32`, `DATA`, and `STACK` segments, respectively. For iC-386, the ES register contains the same value as the DS register.

Tables 4-4 through 4-6 show the compiler segment definitions for a module compiled with the `compact` control. When you specify the `rom` control, the compiler places the constants in the module's code segment. When you specify the `ram` control, the iC-86 compiler creates a segment for the constants, which the linker combines with other `DGROUP` segments to make the data segment. Under the `ram` control, both the iC-286 and iC-386 compilers place the constants in the module's data segment.

Table 4-4 iC-86 Segment Definitions for Compact-model Modules

Description	Name	Combine-type	Group
code segment	CODE	concatenate	CGROUP
constant segment (only with ram control)	CONST	concatenate	DGROUP
data segment	DATA	concatenate	DGROUP
stack segment	STACK	overlay additively	

Table 4-5 iC-286 Segment Definitions for Compact-model Modules

Description	Name	Combine-type	Access
code segment	CODE	normal	execute-read
data segment	DATA	normal	read-write
stack segment	STACK	stack	read-write

Table 4-6 iC-386 Segment Definitions for Compact-model Modules

Description	Name	Combine-type	Access
code segment	CODE32	normal	execute-read
data segment	DATA	normal	read-write
stack segment	STACK	stack	read-write

The resulting linked compact model module contains one code segment up to 64 kilobytes (iC-86 and iC-286) or 4 gigabytes (iC-386) long, one data segment up to 64 kilobytes (iC-86 and iC-286) or 4 gigabytes (iC-386) long, and one stack segment up to 64 kilobytes (iC-86 and iC-286) or 4 gigabytes (iC-386) long.

The compact segmentation memory model is efficient in program size, and offers the maximum possible space for stack activity. Using the compact segmentation memory model restricts your program to 192 kilobytes (iC-86 and iC-286) or 12 gigabytes (iC-386) of memory, but has a full 64 kilobytes (iC-86 and iC-286) or 4 gigabytes (iC-386) for stack activity, and allows access to multiple data segments.

Since all the executable instructions fall within one segment, function pointers are near by default (the offset-only address format). Since data (constants, program variables, or temporary variables) can be in different segments (code, data, or stack), data pointers are far by default (the segment-selector-and-offset address format). See Section 4.3 for an explanation of near and far address formats.

Because all data pointers are far pointers by default, a compact model program can dynamically allocate one or more additional data segments up to 64 kilobytes (iC-86/286) or 4 gigabytes (iC-386) long.

Figures 4-6 and 4-7 show the process of linking or binding a compact RAM and a compact ROM program from two modules. The relative sizes of the final segments are not to scale. The order of modules in the binder input list affects the order of segments in the output file.

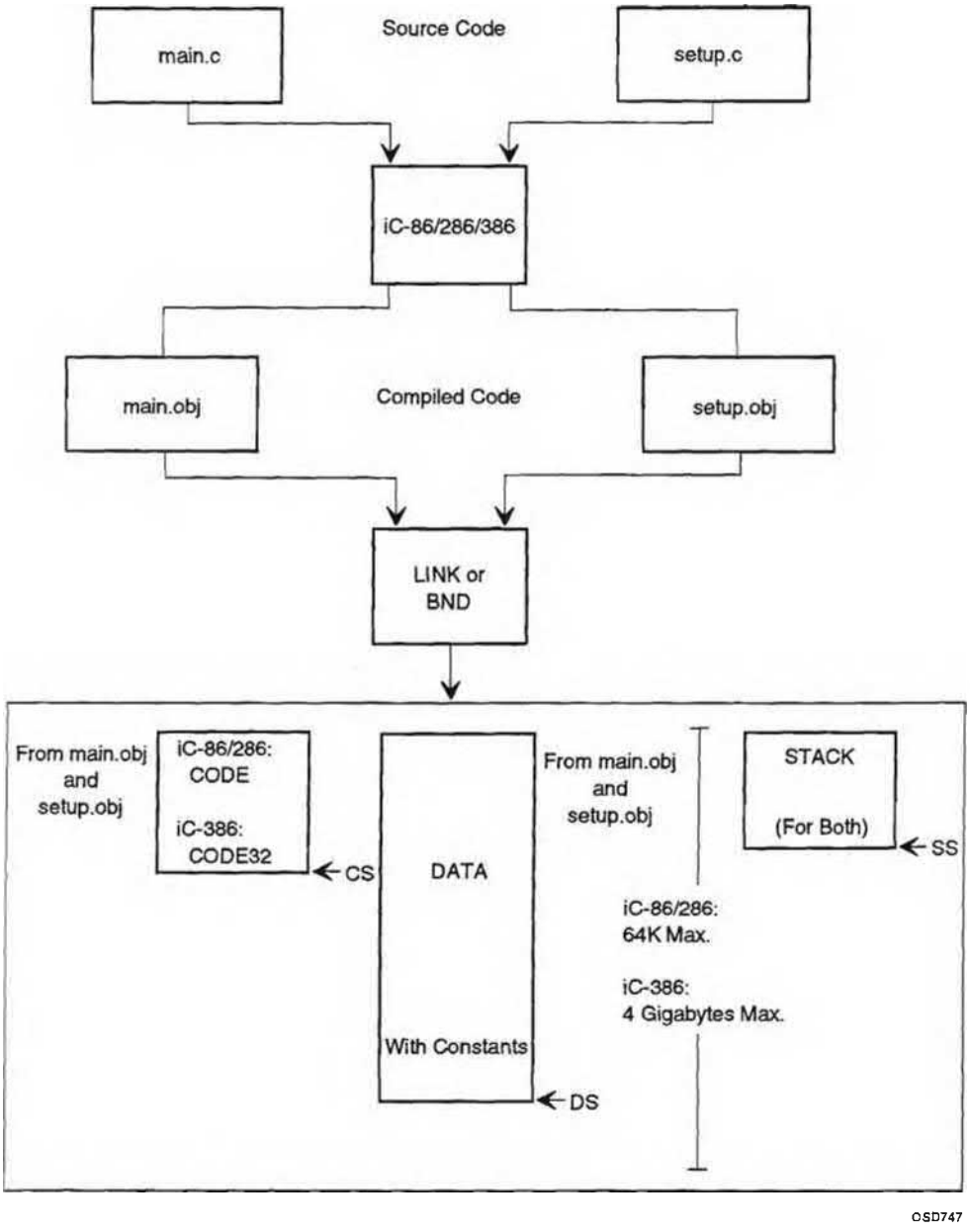

Figure 4-6 Creating a Compact RAM Program

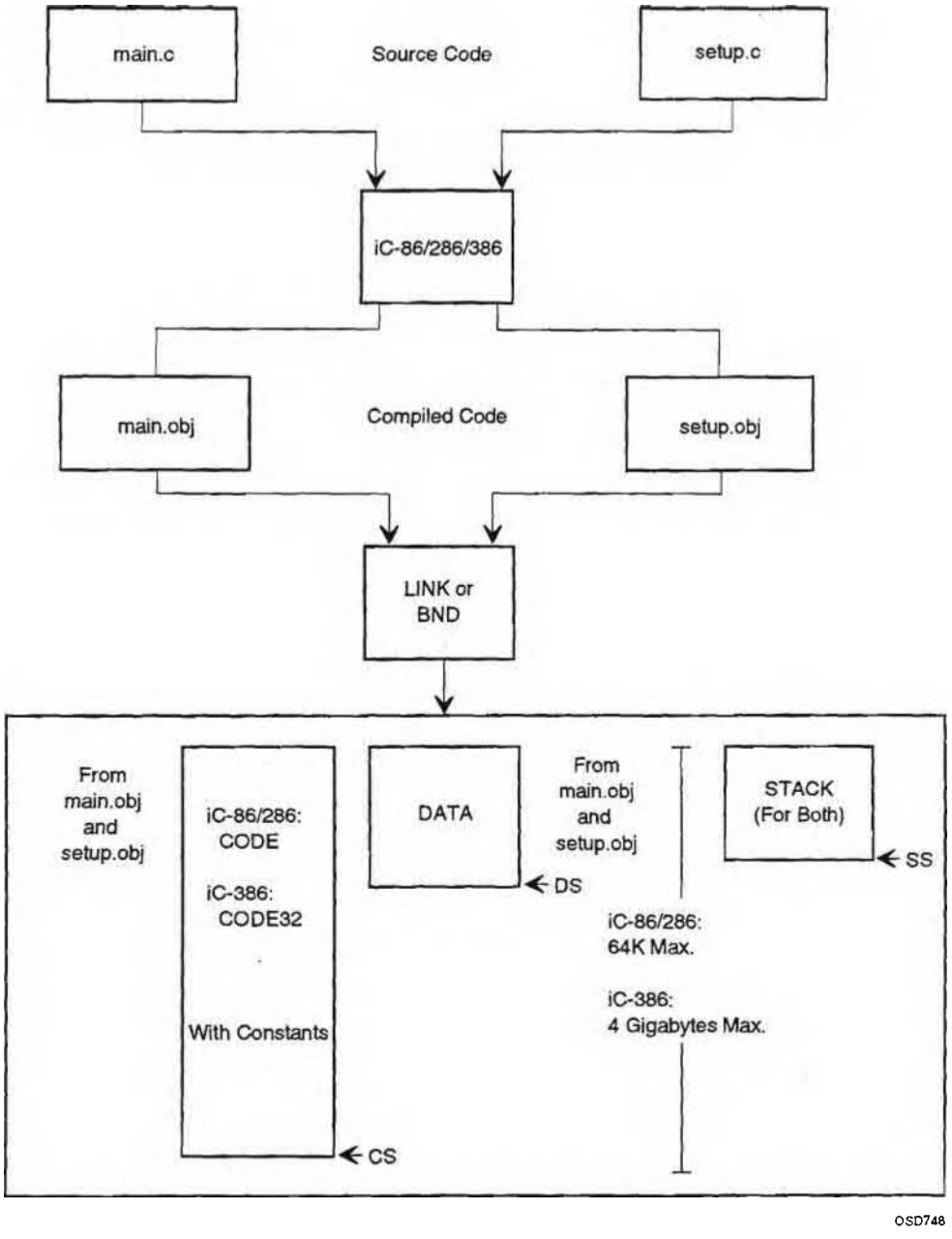

Figure 4-7 Creating a Compact ROM Program

4.2.3 Medium Models (iC-86 and iC-286)

Recall that the LINK86 linker combines iC-86 segments with the same name and compatible combine-types, and the linker ensures that segments with the same group name reside in the same 64 kilobyte segment. The BND286 binder combines compiler-generated segments that have the same name, compatible combine-types, and the same access attributes.

NOTE

For iC-386, the medium model is equivalent to the small model.

A program using the medium segmentation memory model contains as many code segments as input modules, and one combined data-stack segment, `DATA`. The value in the CS register changes during execution to point to the currently active code segment. The DS, ES, and SS registers contain the selector for the `DATA` segment.

Tables 4-7 and 4-8 show the compiler segment definitions for a module compiled with the `medium` control. When you specify the `rom` control, the compiler places the constants in the module's code segment. When you specify the `ram` control, the iC-86 compiler creates a segment for the constants, which the linker combines with other `DGROUP` segments to make the data segment. Under the `ram` control, the iC-286 compiler places the constants in the module's data segment.

Table 4-7 iC-86 Segment Definitions for Medium-model Modules

Description	Name	Combine-type	Group
code segment	*module*_CODE	concatenate	
constant segment (only with ram control)	CONST	concatenate	DGROUP
data segment	DATA	concatenate	DGROUP
stack segment	STACK	overlay additively	DGROUP

Table 4-8 iC-286 Segment Definitions for Medium-model Modules

Description	Name	Combine-type	Access
code segment	*module*_CODE	normal	execute-read
data segment	DATA	normal	read-write
stack segment	DATA	stack	read-write

The resulting linked medium module contains one 64K-byte code segment for each separately compiled module and one 64K-byte combined data-stack segment.

The medium segmentation memory model offers maximum program code space, efficient data and stack size, and (with the `ram` control) efficient data access. Using the medium segmentation memory model enables your program to have 64 kilobytes of memory available per module for executable instructions. Program code space is limited only by the number of modules you define and the total memory available in your target system.

Since all the executable instructions do not fall within one segment, function pointers are by default far (the segment-selector-and-offset address format). If you specify the `ram` control, all variables, temporary variables, and constants fall within one segment, and data pointers are by default near (the offset-only address format). If you specify the `rom` control, which groups constants with code, data pointers are far. See Section 4.3 for an explanation of near and far address formats.

Figures 4-8 and 4-9 show the process of linking or binding a medium RAM and medium ROM program from two modules. The relative sizes of the final segments are not to scale. The order of modules in the linker or binder input list affects the order of segments in the output file.

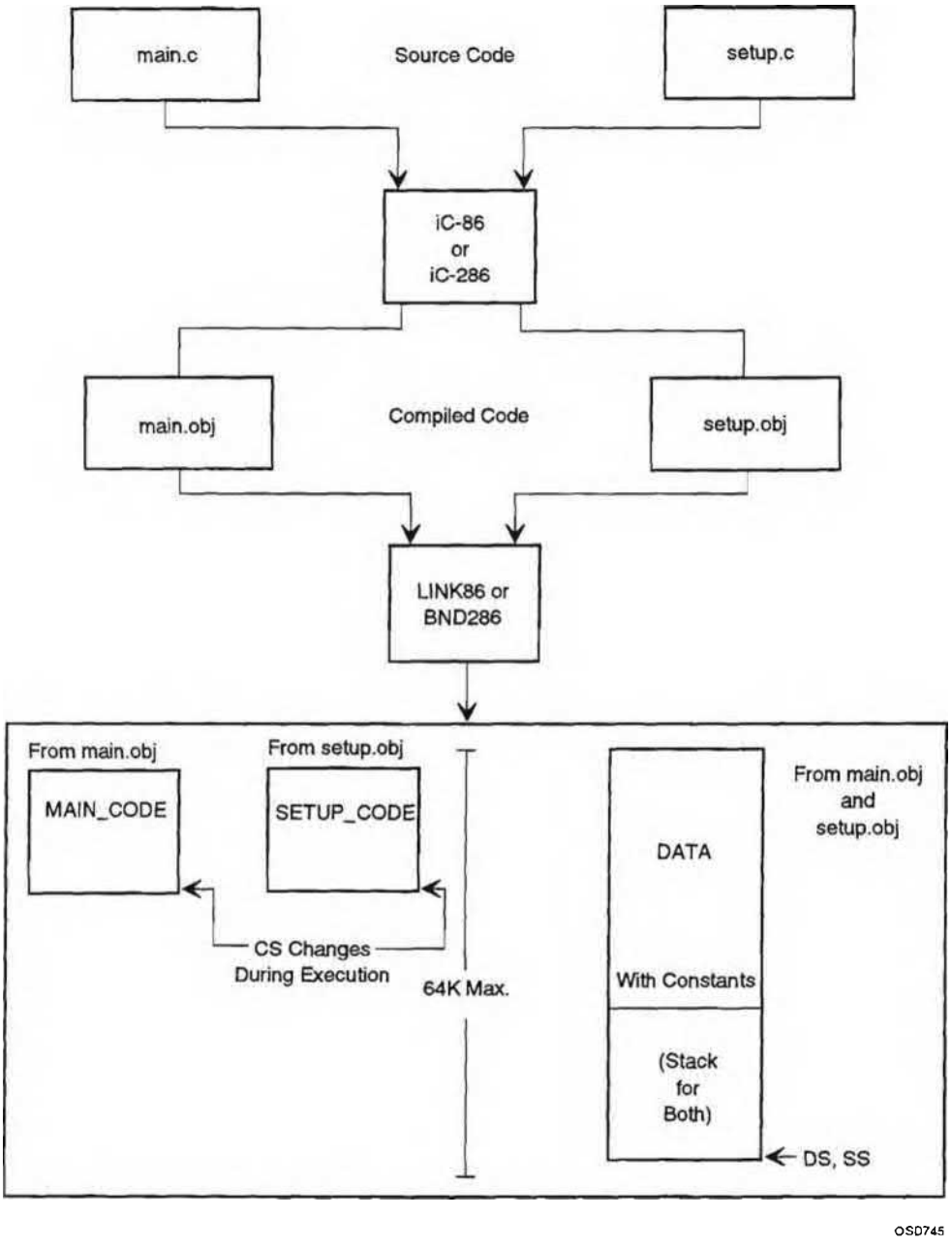

Figure 4-8 Creating an iC-86 or iC-286 Medium RAM Program

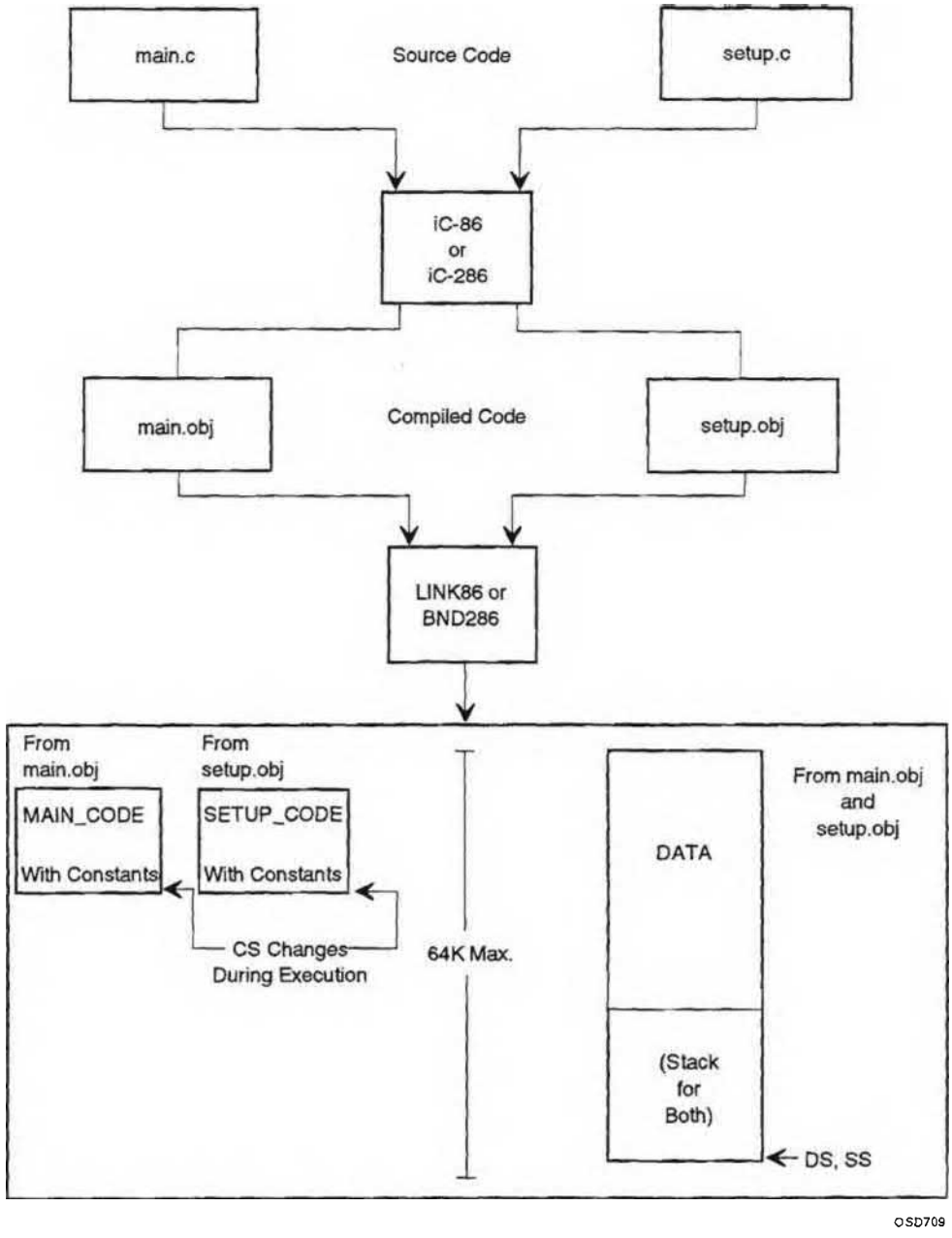

Figure 4-9 Creating an iC-86 or iC-286 Medium ROM Program

4.2.4 Large Models (iC-86 and iC-286)

Recall that the LINK86 linker combines iC-86 segments with the same name and compatible combine-types, and the linker ensures that segments with the same group name reside in the same 64 kilobyte segment. The BND286 binder combines compiler-generated segments that have the same name, compatible combine-types, and the same access attributes.

NOTE

For iC-386, the large model is equivalent to the compact model.

A program using the large segmentation memory model contains as many code segments as input modules, as many data segments as input modules, and one stack segment, `STACK`. The values in the CS and DS registers change during execution to point to the currently active code and data segments, respectively. The SS register contains the selector for the `STACK` segment. The ES register contains the same value as the DS register.

Tables 4-9 and 4-10 show the compiler segment definitions for a module compiled with the `large` control. When you specify the `rom` control, the both compilers place the constants in the module's code segment. When you specify the `ram` control, both compilers place the constants in the module's data segment.

Table 4-9 iC-86 Segment Definitions for Large-model Modules

Description	Name	Combine-type	Group
code segment	*module*_CODE	concatenate	
data segment	*module*_DATA	concatenate	
stack segment	STACK	overlay additively	

Table 4-10 iC-286 Segment Definitions for Large-model Modules

Description	Name	Combine-type	Access
code segment	*module*_CODE	normal	execute-read
data segment	*module*_DATA	normal	read-write
stack segment	STACK	stack	read-write

The resulting linked large module contains one 64 kilobyte code segment for each separately compiled module, one 64 kilobyte data segment for each separately compiled module, and one 64 kilobyte stack segment.

The large segmentation memory model offers maximum program code space, maximum program data space, and maximum program stack space. Using the large segmentation memory model enables your program to have 64 kilobytes of memory available per module for executable instructions, 64 kilobytes of memory per module for data, and 64 kilobytes of memory for all stack activity. Program size is limited only by the number of modules you define and the total memory available in your target system.

Since all the executable instructions do not fall within one segment, function pointers are by default far (the segment-selector-and-offset address format). Since data (constants, program variables, or temporary variables) can be in different segments (code, data, or stack), data pointers are by default far (the segment-selector-and-offset address format). See Section 4.3 for an explanation of near and far address formats.

Figures 4-10 and 4-11 show the process of linking or binding a large RAM and large ROM program from two modules. The relative sizes of the final segments are not to scale. The order of modules in the linker or binder input list affects the order of segments in the output file.

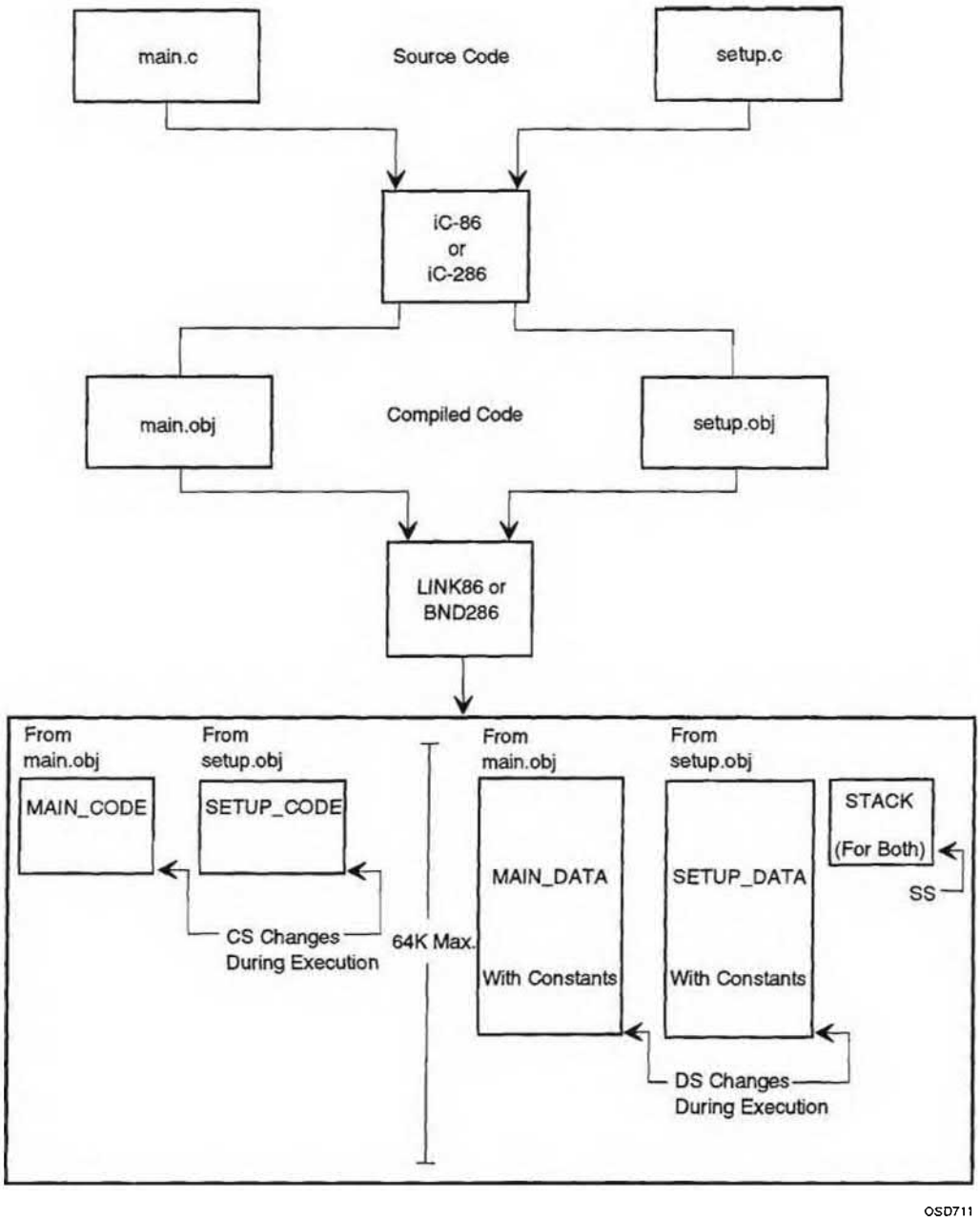

Figure 4-10 Creating an iC-86 or iC-286 Large RAM Program

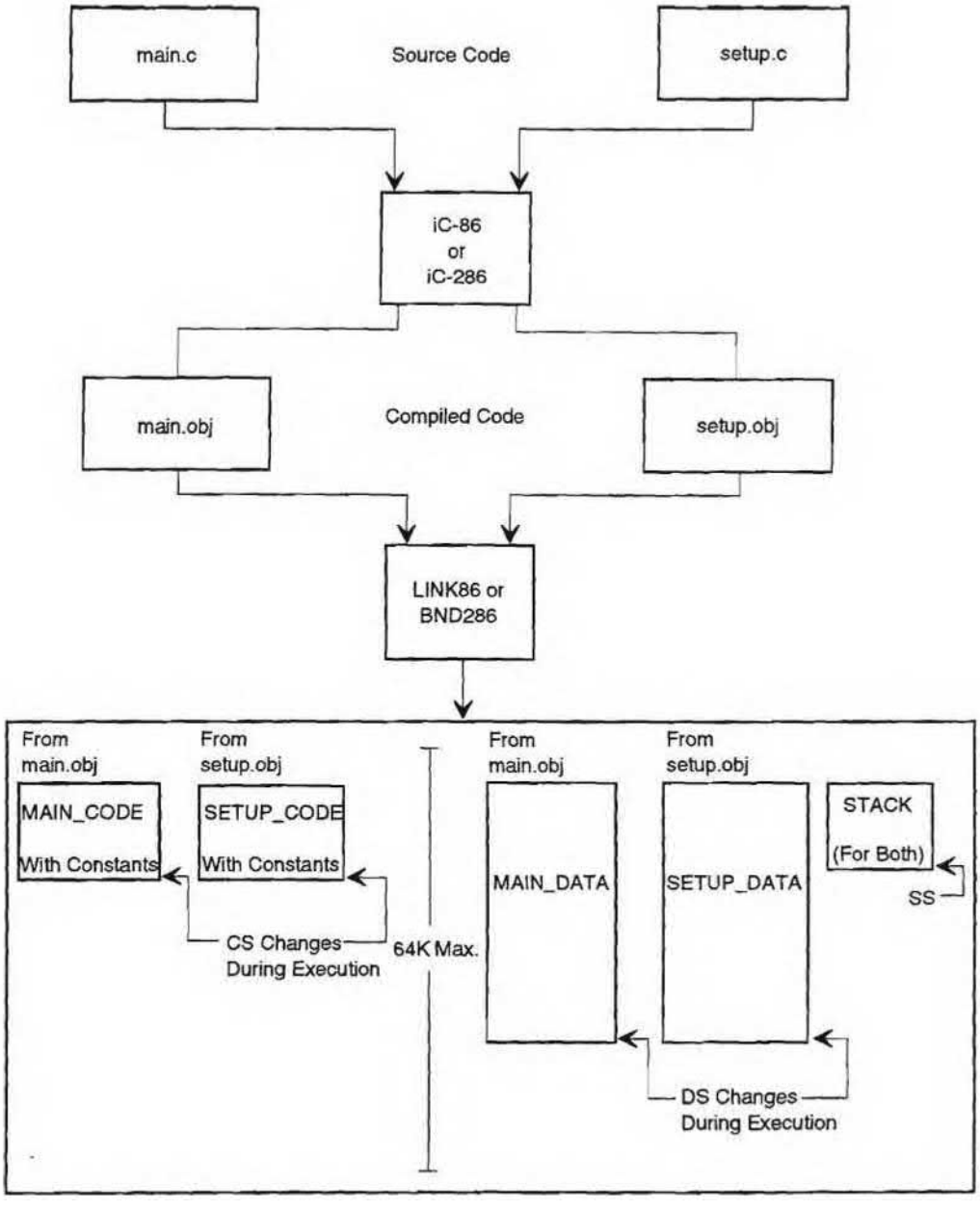

Figure 4-11 Creating an iC-86 or iC-286 Large ROM Program

4.2.5 Flat Model (iC-386 Only)

The iC-386 compiler creates segment definitions for the flat model of segmentation the same as for the compact model, but all function and data pointers are near by default (the offset-only address format). The system builder, BLD386, combines the bound segments into one linear segment by setting the CS, DS, and SS registers to the same selector, and adjusting all offsets relative to the common base address. See the *Intel386™ Family System Builder User's Guide*, listed in Chapter 1, for more information on how the builder creates a flat-model system from bound segments. See Section 4.3 for an explanation of near and far address formats.

Table 4-11 shows the iC-386 compiler segment definitions for a module compiled with the `flat` control. Table 4-12 shows the changes to the segment names after the module is processed by BLD386.

Table 4-11 iC-386 Segment Definitions for Flat-model Modules

Description	Name	Combine-type	Access
code segment	CODE32	normal	execute-read
data segment	DATA	normal	read-write
stack segment	STACK	stack	read-write

Table 4-12 BLD386 Segment Names for Flat-model Programs

iC-386 Name	BLD386 Name
CODE32	_phantom_code_
DATA	_phantom_data_
STACK	_phantom_data_

Whether you specify the `rom` control or the `ram` control, the iC-386 compiler places the constants with the code segment. The `rom` and `ram` controls only affect initialization of static variables. See the entry for the `rom` and `ram` controls in Chapter 3 for more information on the initialization of static variables.

The resulting bound and built flat model system contains one segment up to 4 gigabytes long containing all code, data, and space for stack activity.

The flat segmentation memory model is efficient in size and memory access, but disables many of the protection features that the other memory models provide. Data, stack, and code are not protected from run-time segment overruns. Using the flat segmentation memory model restricts your program to 4 gigabytes of memory.

Since all the executable instructions fall within one segment, function pointers are near by default (the offset-only address format). Since data (constants, program variables, or temporary variables) fall within one segment, data pointers are also near by default (the offset-only address format). See Section 4.3 for an explanation of near and far address formats.

Figure 4-12 shows the process of building a flat model program from the bound segments of two compiled modules, and other bound modules, such as startup code. The relative sizes of the segments are not to scale. The relative positions of code, data, and stack within the built program depend on the definitions in the build file and system data structures, such as descriptor tables. See the *Intel386™ Family System Builder User's Guide*, listed in Chapter 1, for more information on how the builder positions code, data, stack, and system data structures within the built program.

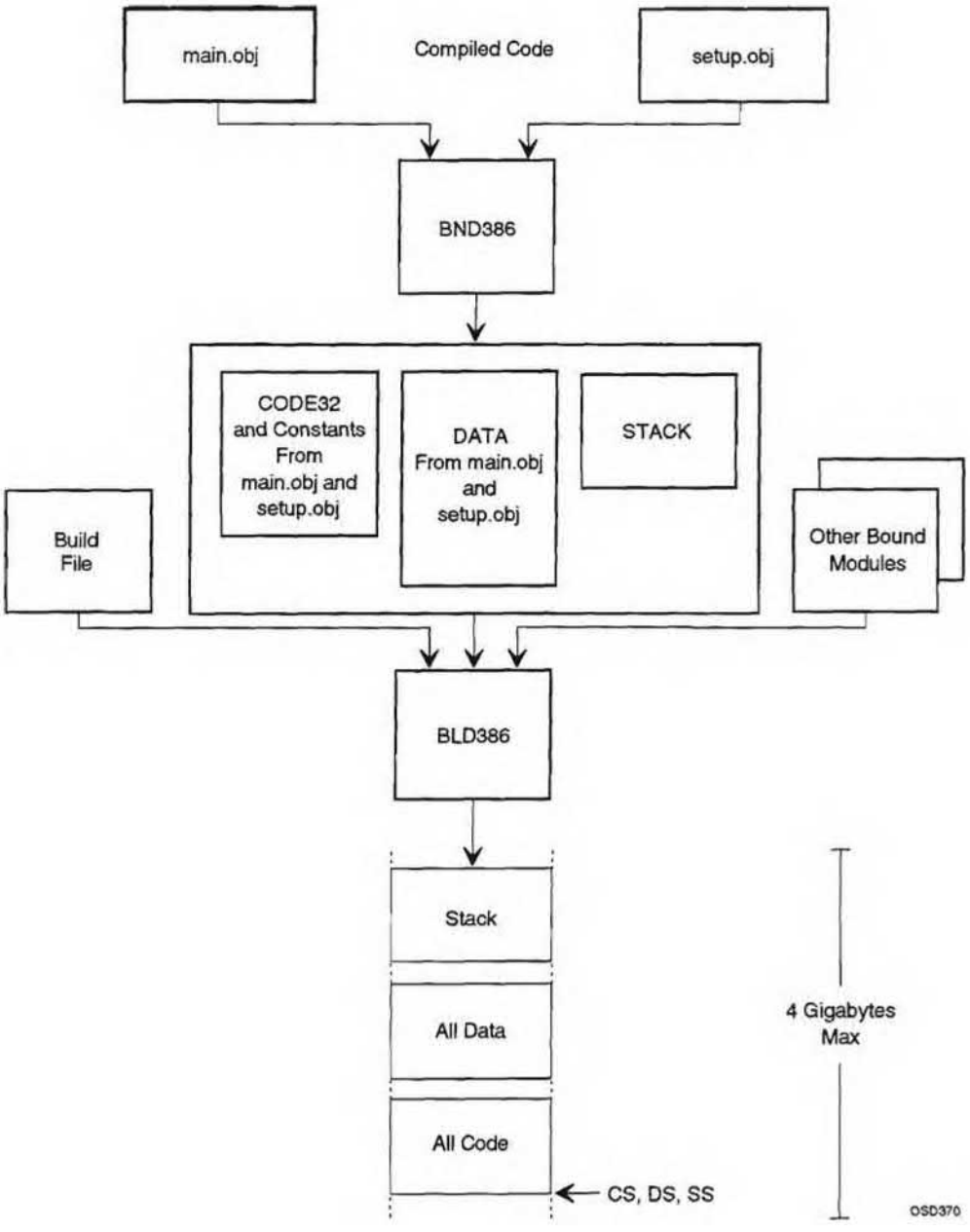

Figure 4-12 Binding and Building an iC-386 Flat-model Program

4.3 Using near and far

The `near` and `far` keywords are type qualifiers that allow programs to override the default address size generated for a data or code reference, which is determined by the segmentation memory model. You must compile programs that use the `near` and `far` keywords with the `extend` control. See Chapter 3 for information about the `extend` control. Table 4-13 shows the default address sizes for all segmentation memory models.

Table 4-13 Segmentation Models and Default Address Sizes

Segmentation Model	Code Reference	Data Reference
small RAM	offset	offset
small ROM	offset	selector and offset
compact RAM	offset	selector and offset
compact ROM	offset	selector and offset
medium RAM	selector and offset	offset
medium ROM	selector and offset	selector and offset
large RAM	selector and offset	selector and offset
large ROM	selector and offset	selector and offset
flat RAM or ROM	offset	offset

The `near` type qualifier causes the compiler to generate an offset-only address. An offset-only address occupies less space and results in quicker execution than a selector-and-offset address. An offset-only address can reference memory only within its segment. The `far` type qualifier causes the compiler to generate a segment-selector-and-offset address. A selector-and-offset address can reference all addressable memory.

Use the `far` type qualifier for the following reasons:

to write code that executes in different memory models	You can compile a module using different segmentation models for different applications. To ensure that the code executes properly under all models, use the `far` type qualifier when non-local data and code references are required.
to call a library that requires a selector-and-offset call	Some libraries require access through a selector-and-offset call.
to refer to code or data in a subsystem of another segmentation model	In multiple segmentation model applications, non-local references can require the `far` type qualifier. See Chapter 9 for information on using multiple subsystems to mix segmentation models within an application.
to call a function at a different privilege level or handle an interrupt	Functions at different privilege levels are always in different segments. A call to an interrupt handler is a far call except in the flat segmentation memory model.

Use the `near` type qualifier for the following reasons:

to discard the selector portion of an address	Casting a pointer to `near` discards the selector. Reference through an offset-only pointer is more efficient.
to override the default data address size	For efficient data references, override the default far references to data that occur when the DS register already has the correct selector.
to override the default code address size	For efficient code references, override the default far references to code that occur when the CS register already has the correct selector.

4.3.1 Addressing Under the Segmentation Models

In small and medium model programs, the CS register contains the code segment selector and the DS and SS registers contain the data segment selector. In medium model programs, the selector in the CS register changes during execution as the current code segment changes.

In compact and large model programs, the CS register contains the code segment selector, the DS register contains the data segment selector, and the SS register contains the stack segment selector. In large model programs, the selectors in the CS and DS registers change during execution as the current segments change. The data and code segments are paired by module, and the CS and DS registers always change together.

In flat model programs, the CS, DS, and SS registers contain selectors that all point to the same base address.

In all models, a reference to a selector-and-offset object requires a load to a segment register. In iC-86 and iC-286, the ES register is typically used to de-reference selector-and-offset addresses. In iC-386, the FS and GS registers are typically used to de-reference selector-and-offset addresses, and the ES register is expected to contain the same value as the DS register.

A variable or a function is "near" if the segmentation model assigns offset-only addresses by default, or if the variable or function is declared with the `near` type qualifier. A variable or a function is "far" if the segmentation model assigns selector-and-offset addresses by default, or if the variable or function is declared with the `far` type qualifier.

In a call to a near function, the processor uses the segment selector in the CS register with the offset-only address of the function to form the address of the function. In a reference to a near variable, the processor uses the segment selector in the DS register with the offset-only address of the variable to form the address of the variable.

In a call to a far function, the processor loads the segment selector portion of the address into the CS register, and then uses the CS register with the offset portion of the function's address to form the address of the function. In a reference to a far variable, the processor loads the segment selector portion of the address into the ES register (86 or 286) or FS or GS register (Intel386 CPU) if neither contains the necessary selector. Then the processor uses either the ES, FS, or GS register with the offset portion of the variable's address to form the address of the variable.

4.3.2 Using far and near in Declarations

The `near` and `far` type qualifiers can occur anywhere in a list of declaration specifiers. Declaration specifiers include storage-class specifiers and type specifiers. Declaration specifiers can also occur after an asterisk (*) in a pointer declarator. See *C: A Reference Manual*, identified in Chapter 1, for information on the syntax of declarations, declaration specifiers, storage-class specifiers, type specifiers, and pointer declarators.
See Chapter 10 for the way iC-86/286/386 extends the syntax of declarators.

You can declare any variable or function with either the `near` or `far` type qualifier to indicate whether it is declared in the same segment from which it is referenced or in a different one. You can specify whether a pointer variable contains a near or a far address.

For example, the following declarations override the default addresses in a module where all addresses are near by default:

```
        int far m;              /* m is a local integer that */
                                /*  is referenced from some  */
                                /*  other segment            */

extern int far n;               /* n is an integer in some   */
                                /*  other segment            */
                                /*  being referenced here    */

        int far * mn_ptr;       /* mn_ptr is a local pointer */
                                /*  to an integer like m or  */
                                /*  n in a different segment */

extern int far * far nm_ptr;/* nm_ptr is a pointer in    */
                                /*  some other segment to an */
                                /*  integer like n or m in a */
                                /*  different segment        */

extern int * far k_ptr;         /* k_ptr is a pointer in     */
                                /*  some other segment to a  */
                                /*  local integer in this    */
                                /*  segment                  */
```

4.3.3 Examples Using far

All of the examples that follow assume the compilation uses the `small` control. In these examples, each single letter in an identifier stands for a type or a type qualifier. The identifiers are spelled so that if you read each letter in the identifier from left to right, the types the letters stand for create a description of the example declaration. Interpret the phrase "far *something*" to be the same as "*something* in a different segment". The identifiers and types in the examples are as follows:

`i`	int
`F`	far
`f`	function returning
`p`	pointer to

1. This example declares two integers. The integer `i` is in the current data segment, referenced through the DS register. The integer `Fi` is in a different data segment, and a reference causes a load to a segment register. The address of `i`, `&i`, is a near address (offset-only). The address of `Fi`, or `&Fi`, is a far address (selector-and-offset). If the `extern` storage class specifier did not exist in the declaration of `Fi`, references to `Fi` would use near addresses, but the address of `Fi` would still be a far address.

```
extern int      i;    /* Where "i" is read as "int"      */

extern int far Fi;    /* Where "Fi" is read as "far int" */
```

2. This example declares two functions. Calls to `fi` are near calls, and calls to `Ffi` are far calls. The address of `fi`, or `&fi`, is a near address. The address of `Ffi`, or `&Ffi`, is a far address. If the `extern` storage class specifier did not exist in the declaration of `Ffi`, calls to `Ffi` would still be far calls.

```
extern int      fi();    /* Where "fi" is read as        */
                         /* "function returning int"     */

extern int far Ffi();    /* Where "Ffi" is read as       */
                         /* "far function returning int" */
```

3. This example declares four pointer variables. The addresses of `pi` and `pFi` are near addresses, and the addresses of `Fpi` and `FpFi` are far addresses. The values of `pi` and `Fpi` are near addresses (near pointers), and those of `pFi` and `FpFi` are far addresses (far pointers). Reference to `Fpi`, `FpFi`, `*pFi`, or `*FpFi` causes a load to a segment register.

```
extern int     *       pi;
extern int     * far   Fpi;
extern int far *       pFi;
extern int far * far FpFi;
```

4. This example declares four functions that return pointers. Calls to `fpi` and `fpFi` are near calls. Calls to `Ffpi` and `FfpFi` are far calls. Both `fpi` and `Ffpi` return near pointers, and `fpFi` and `FfpFi` return far pointers.

```
extern int     *       fpi();
extern int     * far   Ffpi();
extern int far *       fpFi();
extern int far * far FfpFi();
```

Reading the last identifier from left to right, the type of `FfpFi` is read "far function returning pointer to far int." Reading the declarator inside-out (right-to-left), which is the standard way of reading complex C declarators, gives "function returning far pointer to far int," as follows:

Element	Interpretation
`FfpFi()`	"function returning"
`* far`	"far pointer to"
`int far`	"far int"

Such an inside-out interpretation is illogical because a function's return value must be in a register, not in memory (as a far pointer would be). Adding parentheses and writing the same declaration as follows preserves inside-out interpretation and matches the left-to-right reading of the letters in `FfpFi`:

```
extern int far * (far FfpFi)();
```

Element	Interpretation
`int far`	"far int"
`*`	"pointer to"
`(far FfpFi)()`	"far function returning"

The last declaration uses a non-standard type qualifier syntax explained in Chapter 10.

5. This example declares four variables whose values point to a function. Such functions can be called indirectly. Reference to `pfi` or `pFfi` uses the DS register. Reference to `Fpfi` or `FpFfi` causes a load into a segment register. Calls through `pfi` or `Fpfi` are near calls. Calls through `pFfi` or `FpFfi` are far calls.

```
extern int     (*       pfi)();
extern int     (* far  Fpfi)();
extern int far (*      pFfi)();
extern int far (* far FpFfi)();
```

6. This example declares eight pointers to functions that return pointers. Three different kinds of memory references can occur: referencing the pointer to a function, calling the function, and referencing the value indirectly specified by the return value of the function. Reference to `Fpfpi`, `FpFfpi`, `FpfpFi`, and `FpFfpFi` all cause a load into a segment register; these functions are declared with the `far` type qualifier in the third column. Calls to `pFfpi`, `FpFfpi`, `pFfpFi`, and `FpFfpFi` are far calls; these functions are declared with the `far` type qualifier in the second column. The values returned by `pfpFi`, `FpfpFi`, `pFfpFi`, and `FpFfpFi` are far pointers; these functions are declared with the `far` type qualifier in the first column.

```
extern int     *     (*        pfpi)();
extern int     *     (* far   Fpfpi)();
extern int     * far (*       pFfpi)();
extern int     * far (* far  FpFfpi)();
extern int far *     (*       pfpFi)();
extern int far *     (* far  FpfpFi)();
extern int far * far (*      pFfpFi)();
extern int far * far (* far FpFfpFi)();
```

Contents

5

Listing Files

5 Listing Files

The iC-86/286/386 compilers provide listing information in two optional listing files: the preprint file and the print file. These two files embody two phases in compiling. The preprint file contains the source text after textual preprocessing, such as including files and expanding macros. The print file contains information about the results of compiling, that is, using the source text to create object code. The term "compiling" often refers to both the preprocessing and compiling phases as one.

By default, the compiler does not generate a preprint file; use the `preprint` control to produce a preprint listing file. By default, the DOS- and iRMX® system-hosted compilers generate a print file; use the `noprint` control to suppress the print file. The VMS-hosted compilers also generate a print file by default, except when used interactively with DCL-style syntax, as described in Chapter 2. See Chapter 3 for more information about the `preprint` and `noprint` controls. See Chapter 2 for examples of invocations that produce print and preprint files.

5.1 Preprint File

This section describes the preprint file generated by the preprocessing phase of the compiler. The preprint file contains the preprocessor output, which is used as input for the compiling phase. Compiling the preprint file produces the same results as compiling the source file, assuming the compiler can expand any macros without errors.

The compiler preprocesses the source text, to produce the preprint text as follows:

- Expands macros by substituting the body, or textual value, of each macro for each occurrence of its name.

- Inserts source text from files specified with the `include` compiler control or the `#include` preprocessor directive; inserts the `#line` preprocessor directive to bracket sections of included source text in the preprint file.
- Eliminates parts of the source text based on the `#if`, `#ifdef`, `#ifndef`, `#else`, `#elif`, and `#endif` conditional compilation directives.
- Propagates the preprocessor directives `#line`, `#error`, and `#pragma` from the source text to the preprocessed source text.

5.1.1 Macros

Use the `define` control or the `#define` preprocessor directive to define a textual value for a macro name. The preprocessor substitutes the textual value everywhere the macro name appears in the subsequent source text. See Chapter 3 for more information on using the `define` control to define macros, and see Chapter 2 for examples using the `#define` preprocessor directive. See *C: A Reference Manual*, listed in Chapter 1, for detailed information on the `#define` preprocessor directive.

The iC-86/286/386 compilers provide several predefined macros for your convenience. Table 5-1 shows these macros and their values. See Chapter 3 for information on the `long64 | nolong64`, `mod86 | mod186`, `mod287 | nomod287`, `mod486 | nomod486`, `optimize`, `rom`, and `ram` controls. See Chapter 4 for information on segmentation memory models and addressing formats.

Table 5-1 iC-86/286/386 Predefined Macros

Name	Value
__DATE__	date of compilation (if available)
__FILE__	current source filename
__LINE__	current source line number
__STDC__	conformance to ANSI C standard: 1 indicates conformance
__TIME__	time of compilation (if available)

Table 5-1 iC-86/286/386 Predefined Macros (continued)

Name	Value
ARCHITECTURE	86 for iC-86 compiler and mod86 control (default) 186 for iC-86 compiler and mod186 control 286 for iC-286 compiler 386 for iC-386 compiler and nomod486 control (default) 486 for iC-386 compiler and mod486 control
_FAR_CODE_	default address size for function pointers and default range for function calls: 1 (far) for medium and large segmentation models 0 (near) for small, compact, and flat segmentation models
_FAR_DATA_	default address size for data pointers: 1 (far) for all ROM, compact RAM, and large RAM segmentation models 0 (near) for small RAM, medium RAM, and flat segmentation models
LONG64	default type size for long data types in iC-386: 1 for 8-byte long data types if using long64 control 0 for 4-byte long data types if using nolong64 control
NPX	generate FWAIT instructions for numeric coprocessor: 87 for iC-86 and nomod287 control (generate FWAITs) 287 for iC-86 and mod287 control (no FWAITs)
OPTIMIZE	current optimization level as set by optimize control: 0, 1, 2, or 3
ROM	placement of constants with code or data: 1 if using rom control 0 if using ram control

5.1.2 Include Files

Use the `include` control in the compiler invocation or the `#include` preprocessor directive in the source text to specify an include file. The preprocessor inserts the contents of a file included with the `include` control before the first line of the source file. The preprocessor inserts the contents of a file included with the `#include` preprocessor directive into the source text in place of the line containing the `#include` directive. See Chapter 3 for more information on the `include` control.

Paired occurrences of the `#line` preprocessor directive bracket the included text. The compiler inserts the `#line` directive in the preprint listing file at the beginning of the included text and another `#line` directive at the end of the included text.

5.1.3 Conditional Compilation

Conditional preprocessor directives delimit sections of source text to be compiled only if certain conditions are met. The preprocessor evaluates the conditions and determines which sections of source text are kept. The source text that is not kept does not appear in the preprint file unless the `cond` control is in effect. See Chapter 3 for more information on the `cond | nocond` control.

The conditional directives are `#if`, `#else`, `#elif`, `#endif`, `#ifdef`, and `#ifndef`. The `#if` directive can take a special `defined` operator. See *C: A Reference Manual*, listed in Chapter 1, for information on these directives and the `defined` operator.

5.1.4 Propagated Directives

The preprocessor propagates the directives `#line`, `#error`, and `#pragma` from the source text to the preprint file to ensure that the preprint text is equivalent to the source text after preprocessing. See Chapter 11 and *C: A Reference Manual*, listed in Chapter 1, for information on these directives. See Chapter 3 for a complete list of controls that a `#pragma` directive can use.

5.2 Print File

This section describes the print file generated by the compiling phase of the compiler. The print file contains information about the source text read into the compiler and the object code generated by the compiler. See Chapter 2 for several examples of a print file. The following controls (and the equivalent DCL-style qualifiers) affect the format and contents of the print file:

```
code | nocode      listexpand | nolistexpand      pagelength
cond | nocond      listinclude | nolistinclude    pagewidth
diagnostic         modulename                     tabwidth
eject              symbols | nosymbols            title
list | nolist      xref | noxref
```

Table 5-2 shows the compiler controls that affect the entire print file format.

Table 5-2 Controls That Affect the Print File Format

Control	Effect
eject	specifies a form feed (new page)
pagelength	determines number of lines per page
pagewidth	determines number of characters per line
tabwidth	determines number of characters per tab stop

5.2.1 Print File Contents

The print file contains the following sections:

page header	identifies the compiler and the object module name and gives the date and time of compilation.
compilation heading	identifies the host operating system, the compiler, the object module name, and describes the parameters with which the compiler was invoked.
source text listing	is the listing of the C program.

remark, warning, and error messages	are generated by the compiler and are listed with the source text.
pseudo-assembly listing	is a listing of the assembly language object code produced by the compiler. The code does not contain all the assembler directives necessary for a complete assembly language program.
symbol table and cross-reference	provide symbolic information and cross-reference information.
compilation summary	tabulates the size of the output module, the number of diagnostic messages, and the completion status (successful termination or fatal error) of the compilation.

5.2.2 Page Header

Each page of the output listing file begins with a page header. The page header describes the compiler, identifies the module compiled, and shows the date and page number.

The following page header shows the iC-386 compiler compiling the module `MAIN` on the 25th of January, 1991. This example shows the header from the first page of the print file.

```
iC-386  COMPILER   MAIN                          01/25/91 10:28:20  PAGE   1
```

Page numbers range from 1 to 999, then start over at 0.

5.2.3 Compilation Heading

The compilation heading is on the first page of the print file. The compilation heading gives the name of the object module, the pathname of the object module file, and the compiler controls specified in the compiler invocation. It also identifies the compiler version and host system.

For example, the compiler is invoked on a DOS host system as follows:

```
C:\CEXAMPLE> ic386 main.c define(NPAPER) &
>> include(prags.h) &
>> searchinclude(\intel\ic386\inc\,includes\)
```

The compiler processes the `main.c` source file and puts the object module into the file `main.obj`. The compilation heading shows the host operating system, the compiler version, the module name, and the controls used on invocation, as follows:

```
system-id        iC-386 COMPILER Vx.y, COMPILATION OF MODULE MAIN
OBJECT MODULE PLACED IN main.obj
COMPILER INVOKED BY: \INTEL\IC386\IC386.EXE main.c define(NPAPER) include(p
               -rags.h) searchinclude(\intel\ic386\inc\,includes\)
```

If the invocation includes the `modulename` control and uses the `noobject` control to suppress the object file, the invocation looks like the following:

```
C:\CEXAMPLE> ic386 main.c define(NPAPER) &
>> include(prags.h) &
>> searchinclude(\intel\ic386\inc\,includes\) &
>> modulename(NewName) &
>> noobject
```

The resulting compilation heading shows the different module name in the first line, and shows the lack of object file in the second line, as follows:

```
system-id        iC-386 COMPILER Vx.y, COMPILATION OF MODULE NEWNAME
NO OBJECT MODULE PRODUCED
COMPILER INVOKED BY: \INTEL\IC386\IC386.EXE main.c define(NPAPER) include(p
               -rags.h) searchinclude(\intel\ic386\inc\,includes\) modulen
               -ame(NewName) noobject
```

5.2.4 Source Text Listing

The source text listing contains a formatted image of the source text. It also gives the statement number, block nesting level, and include nesting level of each source text statement. If a source line is too long to fit on one line, it continues on as many following lines as are needed. Continued lines contain a hyphen (-) in column 17, followed by the source text.

Statement numbers range from 1 to 99999. Error, warning, and remark messages, when present, refer to the statement numbers in the source text listing. Statement numbers do not always correspond to the sequence of lines in the source text: source text lines that end in a backslash (\) are continuations of the previous line. The listing statement numbers do not increment for continuation lines.

The block nesting level describes how many source text block control constructs surround the statement. It ranges from 0 (for a statement outside of any function definition) to 99. When its value is 0, this field is blank.

The include nesting level describes how many `#include` preprocessor directives or instances of the `include` control the preprocessor encountered to get to this statement in the source text. For the input source file, the nesting depth is 0, and this field is blank. Each nested `#include` preprocessor directive or `include` control increments the include nesting level. The include nesting level column has a value only if the `listinclude` control is in effect. The maximum nesting of include files depends on the number of files open simultaneously during compilation and can vary with the operating system. See Chapter 11 for limitations on the number of nested include files and see the Installation section for more information on the files that your operating system uses.

In addition to the format controls shown in Table 5-2, Table 5-3 shows the compiler controls that affect the source text listing portion of the print file. See Chapter 3 for complete descriptions of these controls.

Table 5-3 Controls That Affect the Source Text Listing

Control	Effect
cond | nocond	Generates or suppresses uncompiled conditional code.
diagnostic	Determines class of messages that appear.
list | nolist	Generates or suppresses source text listing.
listexpand | nolistexpand	Generates or suppresses macro expansion listing.
listinclude | nolistinclude	Generates or suppresses text of include files.

5.2.5 Remarks, Warnings, and Errors

Compiler messages indicate errors (including fatal errors), warnings, and remarks. The source text listing contains these messages. The compiler prints each message on a separate line immediately following the offending statement. If the offending statement is not printed, the compiler prints the messages in the listing as the compiler generates them.

Use the `diagnostic` control to suppress generation of lower-level messages. See Chapter 3 for information on the `diagnostic` control.

5.2.6 Pseudo-assembly Listing

The pseudo-assembly listing is an assembly language equivalent to the object code produced in compilation. It contains a location counter, a source statement number, and the equivalent assembly code. The location counter is a hexadecimal value that represents an offset address relative to the start of the object code.

The assembler cannot assemble the pseudo-assembly language listing; it is not a complete program. It describes the object code produced by the compiler and is useful for noticing program variations, such as those that result from changing optimization levels.

Use the `code` or `nocode` control to generate or suppress the pseudo-assembly listing. See Chapter 3 for information on the `code | nocode` control.

5.2.7 Symbol Table and Cross-reference

The symbol table lists all objects and their attributes from the compiled code. The table includes the name, type, size, and address of each object. The table can optionally include source text cross-reference information. The compiler generates the table in alphabetical order by identifier. A source module can declare a unique identifier more than once, but each object, even if named by a duplicate identifier, appears as a separate entry in the symbol table.

Use the `symbols` or `nosymbols` control to generate or suppress the symbol table. Use the `symbols` and `xref` controls together to generate additional cross-reference information. See Chapter 3 for information on these controls.

5.2.8 Compilation Summary

The final line of the compilation summary in the print file is identical to the sign-off message displayed on the screen when the compilation is complete. Before this final line, the compiler lists information about the compiled object module.

If the compilation completes normally (without errors), the compilation summary is similar to the following example:

```
MODULE INFORMATION:

     CODE AREA SIZE              = 0000028BH         651D
     CONSTANT AREA SIZE          = 000002A7H         679D
     DATA AREA SIZE              = 00000000H          0D
     MAXIMUM STACK SIZE          = 0000001AH          26D

iC-386 COMPILATION COMPLETE.     0 WARNINGS,     0 ERRORS
```

If the compilation ends with a fatal error, the following line is displayed on the console:

```
COMPILATION TERMINATED
```

Contents

6

Processor-specific Facilities

Processor-specific Facilities 6

This chapter describes the functions, macros, and data types available in the `i86.h`, `i8086.h`, `i186.h`, `i286.h`, `i386.h`, and `i486.h` header files. These facilities enable the program to manipulate the unique characteristics of the *n*86 family of processors. This chapter contains the following topics:

- making selectors, far pointers, and near pointers
- using special control functions
- examining and modifying the flags register
- examining and modifying the I/O ports
- enabling and causing interrupts, with guidelines for creating interrupt handlers
- manipulating the protected mode features of the 286, Intel386™ and Intel486™ processors
- manipulating the special control, test, and debug registers in the Intel386 and Intel486 processors
- managing the data cache and paging translation lookaside buffer using special Intel486 processor instructions
- manipulating the 8087, Intel287™, and Intel387™ numeric coprocessors, and the Intel486 floating-point unit

The functions and macros take the place of assembly language routines you usually need to write, saving coding time. The functions and macros also improve run-time performance, because the compiler generates in-line instructions instead of generating calls to your assembly language routines.

Six header files define the functions, macros, and data types. The header files are designed so that your code includes only the file named for the target processor, and your application has access to all appropriate features.

Tables 6-1 through 6-6 list the function names in the header files and the section in this chapter that discusses the function. The function names are available only if your code includes the appropriate header file, and if your code does not redeclare the function names.

The `i86.h` header file defines functions, macros, and data types that apply to the entire line of *n*86 processors, the 8087, Intel287, and Intel387 coprocessor, and the Intel486 processor floating-point unit. Two functions are not defined for Intel386 and Intel486 processors, as noted.

Table 6-1 Built-in Functions in i86.h

Function	Section	Function	Section	Function	Section
buildptr	6.1	halt	6.2,6.5	outword	6.4
causeinterrupt	6.5	inbyte	6.4	restorerealstatus[1]	6.9.5
disable	6.5	initrealmathunit	6.9	saverealstatus[1]	6.9.5
enable	6.5	inword	6.4	setflags	6.3
getflags	6.3	lockset	6.2	setrealmode	6.9.2
getrealerror	6.9.3	outbyte	6.4		

[1]Not for Intel386 and Intel486 processors. See the i386.h header file for substitute definitions.

The `i8086.h` header file uses the `#include` preprocessor directive to include the contents of the `i86.h` header file. The `i8086.h` header file contains a function that applies to *n*86 processors executing in real mode only. This header file is not part of iC-286 or iC-386.

Table 6-2 Built-in Function in i8086.h

Function	Section
setinterrupt	6.5

The `i186.h` header file uses the `#include` preprocessor directive to include the contents of the `i86.h` header file. The `i186.h` header file contains functions that apply to 186 and higher processors.

Table 6-3 Built-in Functions in i186.h

Function	Section	Function	Section	Function	Section
blockinbyte	6.4	blockoutbyte	6.4	blockinword	6.4
blockoutword	6.4				

The `i286.h` header file uses the `#include` preprocessor directive to include the contents of the `i186.h` header file, which similarly includes the contents of the `i86.h` header file. The `i286.h` header file contains functions, macros, and data types that apply to 286 and higher processors in protected mode only.

Table 6-4 Built-in Functions in i286.h

Function	Section	Function	Section	Function	Section
adjustrpl	6.6.4	gettaskregister	6.6.1	segmentwritable	6.6.3
cleartaskswitchedflag	6.6.2	restoreglobaltable	6.6.1	setlocaltable	6.6.1
getaccessrights	6.6.3	restoreinterrupttable	6.6.1	setmachinestatus	6.6.2
getlocaltable	6.6.1	saveglobaltable	6.6.1	settaskregister	6.6.1
getmachinestatus	6.6.2	saveinterrupttable	6.6.1	waitforinterrupt	6.5
getsegmentlimit	6.6.3	segmentreadable	6.6.3		

The `i386.h` header file uses the `#include` preprocessor directive to include the contents of the `i286.h` header file, which enables access to the functions and macros in the `i186.h` and `i86.h` header files, as well. The `i386.h` header file contains functions and macros that apply to the Intel386 and Intel486 processors in protected mode.

Table 6-5 Built-in Functions in i386.h

Function	Section	Function	Section	Function	Section
blockinhword	6.4	gettestregister	6.7	saverealstatus[1]	6.9.5
blockouthword	6.4	inhword	6.4	setcontrolregister	6.7
getcontrolregister	6.7	outhword	6.4	setdebugregister	6.7
getdebugregister	6.7	restorerealstatus[1]	6.9.5	settestregister	6.7

[1]These functions are defined differently from those in the i86.h header file.

The `i486.h` header file uses the `#include` preprocessor directive to include the contents of the `i386.h` header file, which enables access to the functions and macros in the `i286.h`, `i186.h`, and `i86.h` header files, as well. The `i486.h` header file contains functions and macros that apply to Intel486 processors in protected mode.

Table 6-6 Built-in Functions in i486.h

Function	Section	Function	Section	Function	Section
byteswap	6.8	invalidatetlbentry	6.8	wbinvalidatedatacache	6.8
invalidatedatacache	6.8				

The header files are include files, not libraries; use the `#include` preprocessor directive or the `include` control to include one of the headers when compiling. Do not bind to the header files.

6.1 Making Selectors, Far Pointers, and Near Pointers

The `selector` data type and the `buildptr` function, defined in the `i86.h` header file, construct far pointers (segment-selector-and-offset) and extract the selector portion from far pointers.

A value of type `selector` refers to the 16-bit selector portion of a far pointer. This data type is compatible with PL/M `SELECTOR` data type. The `selector` type is similar to the `void *` type for type checking:

- The compiler implicitly converts a value of type `selector` to any pointer type, and vice versa. An explicit cast is unnecessary. When the compiler converts a far pointer to the `selector` type, the compiler discards the offset portion of the far pointer. When the compiler converts a selector to a far pointer type, the compiler supplies an offset of zero.
- Conversion between the `selector` type and any integral type requires an explicit cast. When the compiler converts a selector to an integral type, it zero-extends to fill, or it truncates high-order bits to shorten. When the compiler converts an integral value to the `selector` type, it sign-extends signed values and zero-extends unsigned values to fill, or it truncates high-order bits to shorten.

The `buildptr` function takes two arguments: a selector and an offset. The function returns a far pointer. The prototype for `buildptr` is as follows:

```
void far * buildptr (selector    sel,
                     void near * offset);
```

The offset argument can be zero, and the value that `buildptr` returns is equivalent to casting a selector to a far pointer type, as the following expressions show:

```
(void far *) sel

/* is the same as */

buildptr (sel, 0)
```

Implicit conversion from a far pointer to a near pointer (offset-only) results in a warning message. To retrieve the offset portion from a far pointer, explicitly cast to a near pointer, as the following expression shows:

```
(void near *) farptr
```

6.2 Using Special Control Functions

The `lockset` and `halt` functions in the `i86.h` header file provide special control over processing. See Section 6.5 for information on functions that control the processor interrupt mechanisms.

The `lockset` function takes two arguments: a pointer to a byte and a byte value. The function generates an exchange instruction (XCHG) with a LOCK prefix. The prototype for `lockset` is as follows:

```
unsigned char lockset (unsigned char * lockptr,
                       unsigned char   newbytevalue);
```

The exchange operation puts *newbytevalue* into the byte pointed to by *lockptr* and returns the value previously pointed to by *lockptr*. The LOCK prefix ensures that the processor has exclusive use of any shared memory during the exchange operation.

The `halt` function enables interrupts and halts the processor. It generates a set interrupt instruction (STI) to enable interrupts, followed by a halt instruction (HLT). The prototype for `halt` is as follows:

```
void halt (void);
```

6.3 Examining and Modifying the FLAGS Register

The `getflags` and `setflags` functions in the `i86.h` header file provide access to the FLAGS register for 86 and 286 processors, or the EFLAGS register for Intel386 and Intel486 processors. In Intel386 and Intel486 processors, the EFLAGS register contains the FLAGS register in its low-order 16 bits. Table 6-7 lists several macros in the `i86.h`, `i286.h`, `i386.h`, and `i486.h` header files that isolate individual flags from the FLAGS and EFLAGS registers.

NOTE

In this section, the text refers to a 16-bit word and a 32-bit double word, according to other Intel386 and Intel486 processor documentation. In C programming literature, a word is the amount of storage reserved for an integer, which is 16 bits for iC-86 and iC-286, and 32 bits for iC-386.

The `getflags` function takes no arguments, and returns a 16-bit unsigned integer for iC-86/286 or a 32-bit unsigned integer for iC-386. Use it to retrieve the value of the FLAGS or EFLAGS register, respectively. The prototype for `getflags` is as follows:

```
unsigned int getflags (void);
```

The `setflags` function takes as an argument a 16-bit unsigned integer for iC-86/286 or a 32-bit unsigned integer for iC-386. Use it to set the value of the FLAGS or EFLAGS register, respectively. The prototype for `setflags` is as follows:

```
void setflags (unsigned int wordvalue);
```

The FLAGS register contains the processor flags reflecting the execution and results of various operations. Figure 6-1 shows the format of the 86/286 FLAGS and Intel386 and Intel486 EFLAGS register.

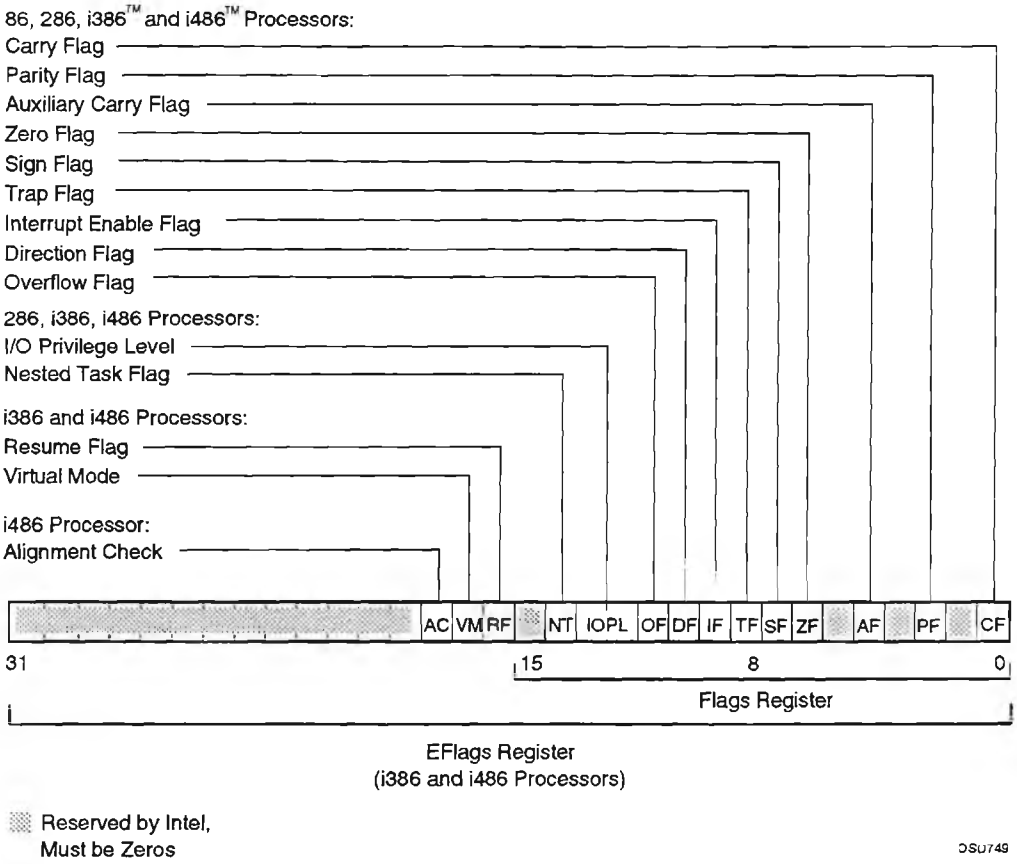

Figure 6-1 FLAGS and EFLAGS Register

Table 6-7 lists the names of the macros in the `i86.h`, `i286.h`, `i386.h`, and `i486.h` header files and describes the meaning of the corresponding fields of the flags register. These macro names must be uppercase in the source text.

Table 6-7 Flag Macros

Name	Value	Meaning
FLAG_CARRY	0x0001	This flag is set when a subtraction causes a borrow into, or an addition causes a carry out of, the high-order bit of the result.
FLAG_AUXCARRY	0x0010	This flag is set when a subtraction causes a borrow into, or an addition causes a carry out of, the low-order 4 bits of the result.
FLAG_PARITY	0x0004	This flag is set when the modulo 2 sum of the low-order 8 bits of the result of an operation is 0 (even parity).
FLAG_ZERO	0x0040	This flag is set when the result of an operation is 0.
FLAG_SIGN	0x0080	This flag is set when the high-order bit of the result of an operation is set, that is, when a signed value is negative.
FLAG_TRAP	0x0100	This flag controls the generation of single-step interrupts. When this flag is set, an internal single-step interrupt occurs after each instruction is executed.
FLAG_INTERRUPT	0x0200	This flag, when set, enables the processor to recognize external interrupts.
FLAG_DIRECTION	0x0400	This flag, when set, makes string operations process characters progressing from higher to lower addresses.
FLAG_OVERFLOW	0x0800	This flag is set when an operation results in a carry into but not a carry out of the high-order bit of the result, or a carry out of but not a carry into the high-order bit of the result (e.g., signed overflow).

Table 6-7 Flag Macros (continued)

Name	Value	Meaning
FLAG_IOPL[1]	0x3000	These two bits define the current task's I/O privilege level, controlling the task's right to execute certain I/O instructions.
FLAG_NESTED[1]	0x4000	This flag is set when the processor executes a task switch. The flag indicates that the back-link field of the task state segment is valid.
FLAG_RESUME[2]	0x10000	This flag, when set, disables debug exceptions so that an instruction can be restarted after a debug exception without immediately causing another debug exception.
FLAG_VM[2]	0x20000	This flag, when set, indicates that the current task is a virtual 86 program.
FLAG_ALIGNCHECK[3]	0x40000	This flag, when set, causes interrupt 17, generating a fault for a memory reference to a mis-aligned address, such as a word at an odd address. This flag is ignored if the privilege level is less than 3.

[1]For 286 and higher processors.
[2]For Intel386 and Intel486 processors.
[3]For Intel486 processors only.

Use the functions and flag macros to set or clear particular flags, as shown in the following examples.

1. This example shows a short program that tests the carry bit:

```
#include <i86.h>
#include <limits.h>
#include <stdio.h>

int main (int argc, char * argv[])
{
  unsigned char i,j;
  unsigned short is_carry;

    /* Test the carry bit */
    i = UCHAR_MAX;
    j = 1;
    j += i;                /* overflow, carry = 1 */
    is_carry = getflags() & FLAG_CARRY;

    if (is_carry == FLAG_CARRY)
      printf("overflow\n");
    return 0;
}
```

2. This example shows a function that ensures that interrupts are disabled before processing, then restores interrupts to their original state before returning:

```
f()
{
  unsigned short int_stat;
    int_stat = getflags() & FLAG_INTERRUPT;
    disable();                    /* See Section 6.5 */

    /* processing */

    setflags (getflags() | int_stat);
}
```

6.4 Examining and Modifying the Input/Output Ports

The functions `inbyte`, `inword`, `outbyte`, and `outword` in the `i86.h` header file, and `inhword` and `outhword` in the `i386.h` header file perform reading from and writing to processor I/O ports. The functions `blockinbyte`, `blockinword`, `blockoutbyte`, and `blockoutword` in the `i186.h` header file, and `blockinhword` and `blockouthword` in the `i386.h` header file perform block reading from and block writing to processor I/O ports.

NOTE

In this section, the text refers to a 16-bit word and a 32-bit double word, according to Intel386 and Intel486 processor documentation. In C programming literature, a word is the amount of storage reserved for an integer, which is 16 bits for iC-86 and iC-286, and 32 bits for iC-386.

The `inbyte`, `inword`, and `inhword` functions take the hardware input port number as an argument. The `inbyte` function returns an 8-bit byte for all processors. The `inword` function returns a 16-bit word for 86 and 286 processors, or a 32-bit double word for Intel386 and Intel486 processors. The `inhword` function returns a 16-bit word for Intel386 and Intel486 processors. The function prototypes are as follows:

```
unsigned char  inbyte  (unsigned short port);

unsigned int   inword  (unsigned short port);

unsigned short inhword (unsigned short port);
```

The `outbyte`, `outword`, and `outhword` functions take two arguments: the hardware output port number and the value to send to the port. The `outbyte` function sends an 8-bit byte to an output port for all processors. The `outword` function sends a 16-bit word for 86 and 286 processors, or a 32-bit double word for Intel386 and Intel486 processors. The `outhword` function sends a 16-bit word for Intel386 and Intel486 processors. The function prototypes are as follows:

```
void outbyte  (unsigned short port,
               unsigned char  bytevalue);

void outword  (unsigned short port,
               unsigned int   word_or_dwordvalue);

void outhword (unsigned short port,
               unsigned short wordvalue);
```

The `blockinbyte`, `blockinword`, and `blockinhword` functions take three arguments: the hardware input port number, a pointer to the initial byte in the destination, and the byte, word, or double word count. The `blockinbyte` function reads 8-bit bytes from an input port for all processors. The `blockinword` function reads 16-bit words for 86 and 286 processors, or 32-bit double words for Intel386 and Intel486 processors. The `blockinhword` function reads 16-bit words for Intel386 and Intel486 processors. The function prototypes are as follows:

```
void blockinbyte  (unsigned short   port,
                   unsigned char *  destinationptr,
                   unsigned int     bytecount);

void blockinword  (unsigned short   port,
                   unsigned int *   destinationptr,
                   unsigned int     word_or_dwordcount);

void blockinhword (unsigned short   port,
                   unsigned short * destinationptr,
                   unsigned int     wordcount);
```

The `blockoutbyte`, `blockouthword`, and `blockoutword` functions take three arguments: the hardware port number, a pointer to the initial byte in the source location, and a byte, word, or double word count. The `blockoutbyte` function copies 8-bit bytes from a location in memory to an output port for all processors. The `blockoutword` function copies 16-bit words for 86 and 286 processors, or 32-bit double words for Intel386 and Intel486 processors. The `blockouthword` function copies 16-bit words for Intel386 and Intel486 processors. The function prototypes are as follows:

```
void blockoutbyte (unsigned short         port,
                   unsigned char const *  sourceptr,
                   unsigned int           bytecount);

void blockoutword (unsigned short         port,
                   unsigned int const *   sourceptr,
                   unsigned int         word_or_dwordcount);

void blockouthword (unsigned short         port,
                    unsigned short const * sourceptr,
                    unsigned int           wordcount);
```

6.5 Enabling and Causing Interrupts

The `enable`, `disable`, `causeinterrupt`, and `halt` functions in the `i86.h` header file provide control over the interrupt process. The `setinterrupt` function in the `i8086.h` header file establishes an iC-86 function as an interrupt handler for a particular interrupt vector. The `waitforinterrupt` function in the `i286.h` header file causes the 286, Intel386, and Intel486 processors to perform a task switch while in a nested interrupt task.

The `enable` function generates a set interrupt instruction (STI). STI sets the interrupt enable flag. The prototype for `enable` is as follows:

```
void enable (void);
```

The `disable` function generates a clear interrupt instruction (CLI). CLI clears the interrupt enable flag. The prototype for `disable` is as follows:

```
void disable (void);
```

The `causeinterrupt` function generates an interrupt instruction (INT). It takes the interrupt number as an argument. The interrupt number must be a constant in the range 0 through 255. The prototype for `causeinterrupt` is as follows:

```
void causeinterrupt (unsigned char interruptnumber);
```

The `halt` function enables interrupts and halts the processor. It generates an STI instruction followed by a halt instruction (HLT). The prototype for `halt` is as follows:

```
void halt (void);
```

The `setinterrupt` function associates an interrupt handler with an interrupt vector number at run time. This operation is only for the 86 and 186 processors (or any processor executing in real mode). The function takes two arguments: the interrupt number and a pointer to the interrupt handler. The interrupt number must be a constant in the range 0 through 255. The prototype for `setinterrupt` is as follows:

```
void setinterrupt (const unsigned char interruptnumber,
                   void far (* handler)(void));
```

The `waitforinterrupt` function generates a return from interrupt instruction (IRET). IRET causes the processor to perform a task switch, saving the status of the outgoing task in its task state segment. The prototype for `waitforinterrupt` is as follows:

```
void waitforinterrupt (void);
```

6.5.1 Hints on Manipulating Interrupts

This discussion applies only to embedded applications or programs not running under an operating system that traps interrupts.

All processors in the *n*86 family have two types of interrupt pins: the non-maskable interrupt (NMI) and the maskable interrupt (INTR). You cannot disable the non-maskable interrupts. You can enable and disable the maskable interrupts.

The following expression determines whether maskable interrupts are enabled:

```
getflags() & FLAG_INTERRUPT
```

The following two statements both enable interrupts; they do not differ in function, but the first is more efficient:

```
enable();

setflags (getflags() | FLAG_INTERRUPT);
```

The following two statements both disable interrupts; they do not differ in function, but the first is more efficient:

```
disable();

setflags (getflags() & ~FLAG_INTERRUPT);
```

Interrupts occur automatically when the associated condition occurs. However, you can force a particular interrupt to occur at any point in your source text by specifying an interrupt number directly. The following statement initiates an integer overflow interrupt:

```
causeinterrupt(4);
```

Each maskable interrupt has an interrupt number designating the condition which causes the interrupt. Interrupt numbers range from 0 to 255. Table 6-8 shows the numbers reserved for specific interrupts. You can use numbers greater than 31 to define your own interrupts. Intel reserves all interrupts from 0 through 31, even if they are not defined for a processor. To specify a handler for an Intel-reserved interrupt, you must use the interrupt numbers as defined in Table 6-8.

Table 6-8 Interrupt Numbers

Number	Meaning	Processor
0	divide error	all
1	debug exceptions	all
2	non-maskable interrupt	all
3	debugger breakpoint	all
4	overflow	all
5	reserved bounds check	86 186 and higher
6	reserved invalid opcode	86 186 and higher
7	reserved coprocessor/device not available	86 186 and higher
8	reserved double fault/system error	86 and 186 286 and higher
9	reserved coprocessor segment overrun reserved	86 and 186 286 and i386™ i486™
10	reserved invalid task state segment	86 and 186 286 and higher
11	reserved segment not present	86 and 186 286 and higher
12	reserved stack fault	86 and 186 286 and higher
13	reserved general protection fault	86 and 186 286 and higher
14	reserved page fault	86, 186 and 286 i386 and i486
15	reserved	all
16	coprocessor/floating-point error	all
17	reserved alignment check	86, 186, 286, and i386 i486
18-31	reserved	all
32-255	user-definable	all

6.5.2 Interrupt Handlers for the 86 and 186 Processors

In the 86 and 186 processors, each interrupt number indexes an interrupt vector. The interrupt vectors are an absolutely located array of entries beginning at location 0 in memory. The *n*th vector is at location 4*n*, and contains the address of the interrupt handler associated with interrupt number *n*. Each vector is a four-byte value containing the segment-selector-and-offset address of the interrupt handler. See Chapter 4 for information on segment-selector-and-offset addressing.

Two iC-86 facilities manipulate interrupt handlers for the 86 and 186 processors (or any processor executing in real mode): the `interrupt` control and the `setinterrupt` built-in function. See Chapter 3 for additional information on the `interrupt` control.

- The `interrupt` control causes the compiler to do the following at compile time:
  - Generate prolog and epilog code for the interrupt handler for saving and restoring registers and returning from the interrupt.
  - Optionally generate an interrupt vector for the interrupt handler, statically associating the handler with a specific interrupt number.
- The `setinterrupt` built-in function dynamically associates an interrupt handler (one that already has the proper prolog and epilog code) with a specific interrupt number.

Always use the `interrupt` control to make a function into an interrupt handler. Use the following criteria to determine whether to use the `interrupt` control or the `setinterrupt` built-in function to associate an interrupt handler with a specific interrupt number:

- If the interrupt vector table is in ROM, use the `interrupt` control for static association.
- If the interrupt vector is in RAM, use the `setinterrupt` built-in control for dynamic association.
- If the application runs under DOS and you link the application with the `exe` control for LINK86, use the `setinterrupt` built-in function.

- If the application runs under DOS and you use UDI2DOS to create the executable version, use the `interrupt` control or the `setinterrupt` function.
- If the application runs under the iRMX® operating system, use the `interrupt` control or the `setinterrupt` function.

The following examples use the control and the built-in function differently, depending on the application.

1. This example, containing two modules, shows user-defined interrupts for the interrupt numbers 100 and 200. The first module forces the interrupts. The second module uses the `interrupt` control to declare the interrupt functions and associate them with the interrupt numbers 100 and 200. This example runs under DOS and uses UDI2DOS to create the executable file. Figure 6-2 shows the source text.

```
/* first module of DOS application */

#include <i8086.h>
#include <stdio.h>

int reach1 = 0;
int reach2 = 0;

main()
{
   causeinterrupt(100);
   if (reach1 == 1)
      printf("handler1 was reached\n");

   causeinterrupt(200);
   if (reach2 == 1)
      printf("handler2 was reached\n");
}
```

Figure 6-2 Example DOS Interrupt Handlers

```
/*-------------------------------------------------------*/
/* second module of DOS application */

#pragma interrupt("handler1"=100, "handler2"=200)

extern int reach1, reach2;

void handler1(void)
{
   reach1 = 1;
}

void handler2(void)
{
   reach2 = 1;
}
```

Figure 6-2 Example DOS Interrupt Handlers (continued)

2. This example shows similar code for an embedded application with the interrupt vector table in RAM. The second module uses the `interrupt` control to declare the interrupt functions and the first module uses the `setinterrupt` built-in function to associate the interrupt handlers with the interrupt numbers 100 and 200. Figure 6-3 shows the source text.

```
/* first module of embedded application */

#include<i8086.h>

extern void far handler1(void);
extern void far handler2(void);
int reach1 = 0;
int reach2 = 0;

main()
{
   setinterrupt(100,handler1);
   setinterrupt(200,handler2);
```

Figure 6-3 Example Embedded Interrupt Handlers

```
    causeinterrupt(100);
    if (reach1 == 1)
       /* handler1 was reached */;
    causeinterrupt(200);
    if (reach2 == 1)
       /* handler2 was reached */;
}

/*-------------------------------------------------------*/

/* second module of embedded application */

#pragma interrupt("handler1", "handler2")

extern int reach1, reach2;

void handler1(void)
{
   reach1 = 1;
}

void handler2(void)
{
   reach2 = 1;
}
```

Figure 6-3 Example Embedded Interrupt Handlers (continued)

To make a function into an unassigned interrupt handler, use the `interrupt` control without an assignment. You can create the interrupt vector at a later time and link the handler to the program. Similarly, you can have a library of interrupt handlers that are not yet associated with an interrupt vector. Any program can link in any of these functions and separately create the interrupt vectors. For example, assume the compiler invocation includes the following control:

```
interrupt(int_0,int_1,int_2,int_3,int_4)
```

Somewhere in the source text the following can occur:

```
#include <i8086.h>

/* in declarations */

extern void far int_0(void);
extern void far int_1(void);
extern void far int_2(void);
extern void far int_3(void);
extern void far int_4(void);

/* in executable code */

setinterrupt(0,int_0);
setinterrupt(1,int_1);
setinterrupt(2,int_2);
setinterrupt(3,int_3);
setinterrupt(4,int_4);
```

6.5.3 Interrupt Handlers for 286 and Higher Processors

The 286 and higher processors executing in protected mode require an interrupt descriptor table (IDT). This table can be anywhere in memory. The interrupt descriptor table register (IDTR) is a system register that holds the address of the IDT. The startup code initializes this register. You can manipulate this register with the `saveinterrupttable` and `restoreinterrupttable` functions described in Section 6.6.1.

The entries in the IDT are task, trap, or interrupt gates. A gate is a special control-transfer descriptor which acts like a sophisticated interrupt vector. It contains the address of the handler and some access information. Its position in the IDT determines which interrupt it handles. Figure 6-4 shows the format of a gate. The special descriptors for a task state segment (TSS) and the local descriptor table (LDT) share the four-bit type field but differ in other fields from the gate descriptor. See the appropriate programmer's reference manuals listed in Chapter 1, for more information on descriptors.

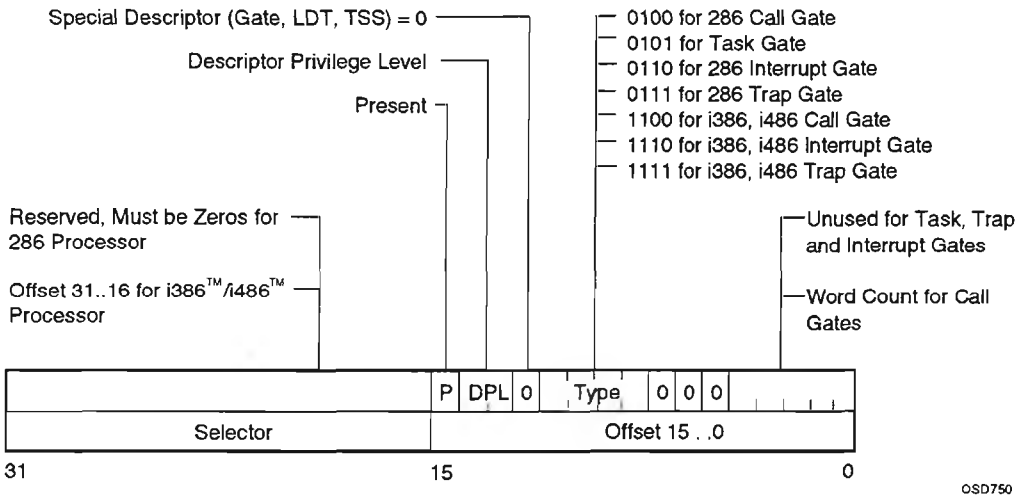

Figure 6-4 Gate Descriptor for 286 and Higher Processors

High-priority hardware interrupts often use an interrupt gate for automatically disabling interrupts upon invocation. Software-invoked interrupts often use trap gates since trap gates do not disable the maskable hardware interrupts. Sometimes low-priority interrupts (for example, a timer) use a trap gate to enable other devices of higher priority to interrupt the handler of the lower priority interrupt. Task gates cause a task switch, which includes saving all of the processor registers and isolating the address space and privilege level of the handler. A task resumes execution on each invocation instead of starting from the initial entry point.

To make an iC-286 or iC-386 function into an interrupt handler, use the `interrupt` control. This control causes the compiler to generate prolog and epilog code for an interrupt handler to save and restore registers. See Chapter 3 for more information on the `interrupt` control.

The easiest way to associate an iC-286 or iC-386 interrupt handler with a processor interrupt is to use the system builder utility, BLD286 or BLD386. Use the build file to create a gate, associate it with the handler, and position it in the IDT. See the *286 System Builder User's Guide* or the *Intel386™ Family System Builder User's Guide*, both listed in Chapter 1, for more information on the builders and build files.

For example, assume functions with the external names `int_0`, `int_1`, `int_2`, `int_3`, and `int_4` are interrupt handlers. In the build file, the following text creates interrupt gates at descriptor privilege level 0 and inserts the gates into the IDT:

```
BUILDFILENAME;
--  (other build specifications here)

GATE int_0_gate (INTERRUPT, DPL = 0, ENTRY = int_0),
     int_1_gate (INTERRUPT, DPL = 0, ENTRY = int_1),
     int_2_gate (INTERRUPT, DPL = 0, ENTRY = int_2),
     int_3_gate (INTERRUPT, DPL = 0, ENTRY = int_3),
     int_4_gate (INTERRUPT, DPL = 0, ENTRY = int_4);

TABLE IDT (ENTRY = (0:int_0_gate,
                    1:int_1_gate,
                    2:int_2_gate,
                    3:int_3_gate,
                    4:int_4_gate));

--  (other build specifications here)
END
```

6.6 Protected Mode Features of 286 and Higher Processors

The functions in the `i286.h` header file enable iC-286 and iC-386 programs to manipulate the system address registers and the machine status word, to retrieve attributes of a segment descriptor, and to adjust the requested privilege level (RPL) of a selector. The functions that access the global descriptor table (GDT) and local descriptor table (LDT) return the `descriptor_table_reg` data type. The `i286.h` header provides macros for isolating information from the machine status word and from segment descriptors.

See the appropriate programmer's reference manual listed in Chapter 1, for more information on the architecture of the 286, Intel386, and Intel486 processors, address translation, and protected mode features.

6.6.1 Manipulating System Address Registers

The system address registers are the task register (TR), the global descriptor table register (GDTR), the interrupt descriptor table register (IDTR), and the local descriptor table register (LDTR).

The `gettaskregister` function returns the contents of the task register (TR). The prototype for `gettaskregister` is as follows:

```
selector gettaskregister (void);
```

The `settaskregister` function loads a selector into the task register (TR). Only protected mode code at privilege level 0 can execute this function. It takes the selector value as its argument. The prototype for `settaskregister` is as follows:

```
void settaskregister (selector sel);
```

The `descriptor_table_reg` structure type describes the register value returned by the `saveglobaltable` and `saveinterrupttable` functions. The structure definition is as follows:

```
#if _LONG64_
   typedef unsigned int base_addr;
#else
   typedef unsigned long base_addr;
#endif

#pragma NOALIGN("descriptor_table_reg")

struct  descriptor_table_reg
{
   unsigned short limit;
   base_addr      base;
};
```

The `saveglobaltable` function copies the contents of the global descriptor table register (GDTR) into a specific 6-byte location of type `descriptor_table_reg`. The function takes a pointer to this destination as an argument. The prototype for `saveglobaltable` is as follows:

```
void saveglobaltable
           (struct descriptor_table_reg * destinationptr);
```

The `restoreglobaltable` function loads a value of type `descriptor_table_reg` into the global descriptor table register (GDTR). Only protected mode code at privilege level 0 can execute this function. The function takes a pointer to the `descriptor_table_reg` 6-byte area as an argument. The prototype for `restoreglobaltable` is as follows:

```
void restoreglobaltable
          (struct descriptor_table_reg const * sourceptr);
```

The `saveinterrupttable` function copies the contents of the interrupt descriptor table register (IDTR) into a specific 6-byte location of type `descriptor_table_reg`. The function takes a pointer to this destination as an argument. The prototype for `saveinterrupttable` is as follows:

```
void saveinterrupttable
          (struct descriptor_table_reg * destinationptr);
```

The `restoreinterrupttable` function loads a value of type `descriptor_table_reg` into the interrupt descriptor table register (IDTR). Only protected mode code at privilege level 0 can execute this function. The function takes a pointer to the `descriptor_table_reg` 6-byte area as an argument. The prototype for `restoreinterrupttable` is as follows:

```
void restoreinterrupttable
          (struct descriptor_table_reg const * sourceptr);
```

The `getlocaltable` function returns the contents of the local descriptor table register (LDTR). The prototype for `getlocaltable` is as follows:

```
selector getlocaltable (void);
```

The `setlocaltable` function loads a value of type `selector` into the local descriptor table register (LDTR). Only protected mode code at privilege level 0 can execute this function. It takes the selector value as an argument. The prototype for `setlocaltable` is as follows:

```
void setlocaltable (selector sel);
```

6.6.2 Manipulating the Machine Status Word

The machine status word (MSW) contains four bits that indicate the status and configuration of the processor. In the Intel386 and Intel486 processors, the machine status word is the lower word in control register 0 (CR0). Figure 6-5 shows the format of the machine status word.

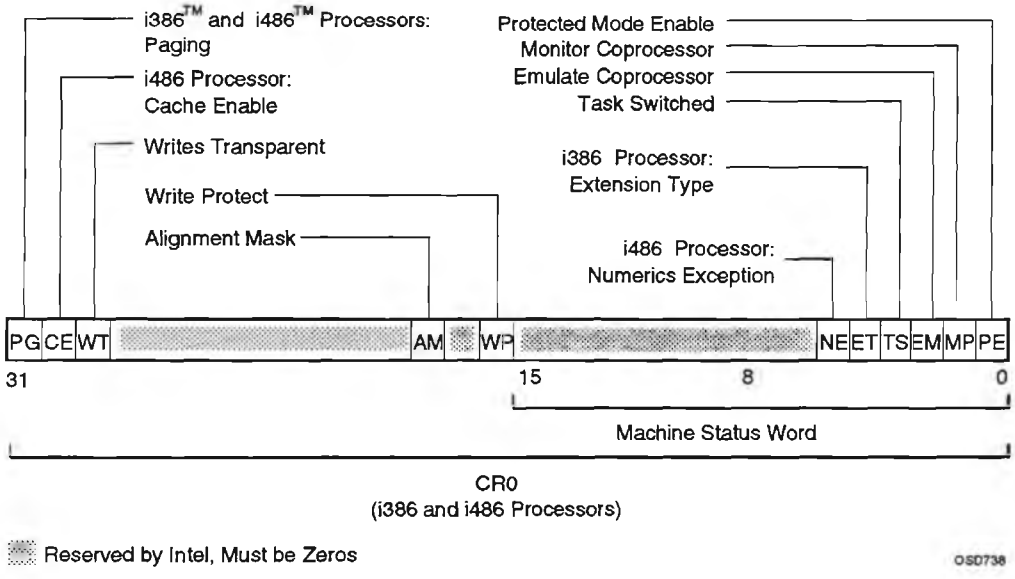

Figure 6-5 Machine Status Word of 286 and Higher Processors

The `getmachinestatus` function returns the contents of the machine status word. The prototype for `getmachinestatus` is as follows:

```
unsigned short getmachinestatus (void);
```

The `setmachinestatus` function loads a value into the machine status word. The compiler generates a short jump to the next instruction to clear the instruction prefetch queue. Only code at privilege level 0 can execute this function. The function takes the value for the machine status word as an argument. The prototype for `setmachinestatus` is as follows:

```
void setmachinestatus (unsigned short wordvalue);
```

The `cleartaskswitchedflag` function clears the task flag in the machine status word. Only code at privilege level 0 can execute this function. The prototype for `cleartaskswitchedflag` is as follows:

```
void cleartaskswitchedflag (void);
```

Four macros isolate particular fields in the machine status word. Table 6-9 lists the names of the machine status word macros in the `i286.h` header file and describes the meaning of the corresponding fields of the machine status word. These macro names must be uppercase in the source text.

Table 6-9 Machine Status Word Macros for 286 and Higher Processors

Name	Value	Meaning
MSW_PROTECTION_ENABLE	0x0001	This bit, when set, places the processor into protected mode and cannot be cleared except by RESET.
MSW_MONITOR_COPROCESSOR	0x0002	This bit, when set, makes WAIT instructions cause interrupt number 7 if the task-switched flag is set.
MSW_EMULATE_COPROCESSOR[1]	0x0004	This bit, when set, makes ESC instructions cause interrupt number 7 to enable coprocessor emulation.
MSW_TASK_SWITCHED	0x0008	This bit, when set, makes the next coprocessor instruction cause interrupt number 7 so software can test whether the coprocessor context belongs to the current task.

[1]Not meaningful for Intel486 processor.

6.6.3 Accessing Descriptor Information

A segment descriptor contains several attributes in its access rights byte. Figures 6-6 and 6-7 show the format of a 286 segment descriptor or Intel386 and Intel486 segment descriptor, respectively.

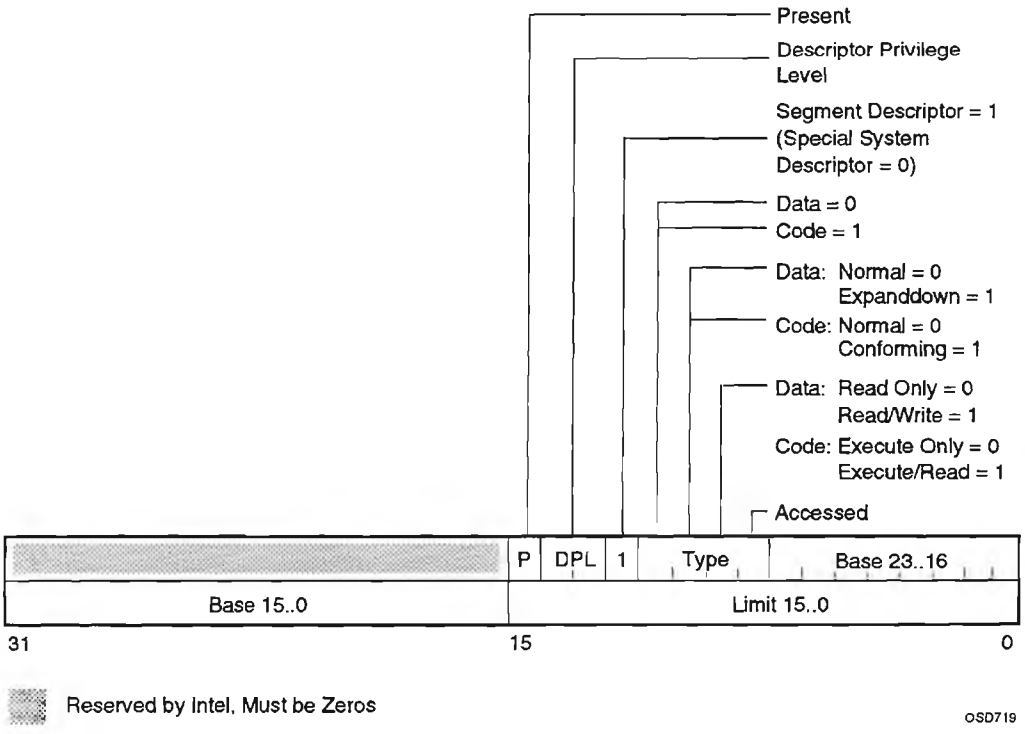

Figure 6-6 Segment Descriptor for 286 Processor

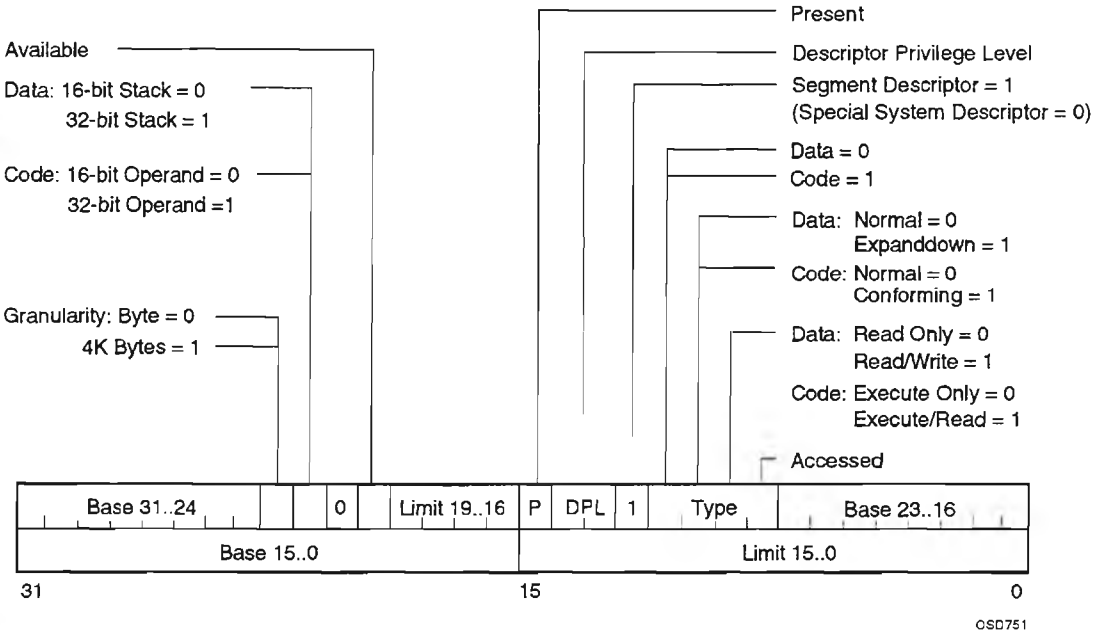

Figure 6-7 Segment Descriptor for Intel386™ and Intel486™ Processors

The `getsegmentlimit` function sets the zero flag and returns the limit of the segment indicated by the selector argument if the following conditions are met (or clears the zero flag and returns an undefined value otherwise):

- The selector argument is non-null.
- The selector denotes a descriptor within the bounds of the GDT or the LDT.
- If the descriptor is for a data segment, its descriptor privilege level must be greater than or equal to the current privilege level.
- If the descriptor is for a nonconforming code segment, its descriptor privilege level must be greater than or equal to the current privilege level.

- If the descriptor is for a nonconforming code segment, its descriptor privilege level must be greater than or equal to the selector's requested privilege level.
- If the descriptor is for a conforming code segment, its descriptor privilege level can be any value.

The `getsegmentlimit` function takes the selector value as an argument. The prototype is as follows:

```
unsigned int getsegmentlimit (selector sel);
```

The `segmentreadable` function returns a 1 if the segment indicated by the selector argument is readable (or returns a 0 otherwise). A segment is readable if the following conditions are met:

- The selector argument is non-null.
- The selector denotes a descriptor within the bounds of the GDT or the LDT.
- If the segment descriptor is for a code segment, the execute/read bit must be 1.
- If the descriptor is for a data segment, its descriptor privilege level must be greater than or equal to the current privilege level.
- If the descriptor is for a nonconforming code segment, its descriptor privilege level must be greater than or equal to the current privilege level.
- If the descriptor is for a nonconforming code segment, its descriptor privilege level must be greater than or equal to the selector's requested privilege level.
- If the descriptor is for a conforming code segment, its descriptor privilege level can be any value.

The `segmentreadable` function takes a selector value as an argument. The prototype is as follows:

```
int segmentreadable (selector sel);
```

The `segmentwritable` function returns 1 if the segment indicated by the selector argument is writable (or returns a 0 otherwise). A segment is writable if the following conditions are met:

- The selector argument is non-null.
- The selector denotes a descriptor within the bounds of the GDT or the LDT.
- The segment descriptor denotes a data segment.
- The descriptor's read/write bit must be 1.
- The descriptor privilege level of the segment must be greater than or equal to the current privilege level.

The `segmentwritable` function takes a selector value as an argument. The prototype is as follows:

```
int segmentwritable (selector sel);
```

The `getaccessrights` function returns the access rights of the segment indicated by the selector argument and sets the zero flag if the following conditions are met (or clears the zero flag and returns an undefined value otherwise):

- The selector argument is non-null.
- The selector denotes a descriptor within the bounds of the GDT or the LDT.
- If the descriptor is for a data segment, its descriptor privilege level must be greater than or equal to the current privilege level.
- If the descriptor is for a nonconforming code segment, its descriptor privilege level must be greater than or equal to the current privilege level.
- If the descriptor is for a nonconforming code segment, its descriptor privilege level must be greater than or equal to the selector's requested privilege level.
- If the descriptor is for a conforming code segment, its descriptor privilege level can be any value.

The `getaccessrights` function takes a selector value as an argument. The return value is four bytes with the access rights in the byte above the low-order byte The prototype for `getaccessrights` is as follows:

```
unsigned int getaccessrights (selector sel);
```

A segment descriptor and a special descriptor have several fields in common: the present bit, the descriptor privilege level, and the segment or special descriptor bit. Figure 6-8 shows the format of a special descriptor, such as a gate, local descriptor table (LDT), or task state segment (TSS).

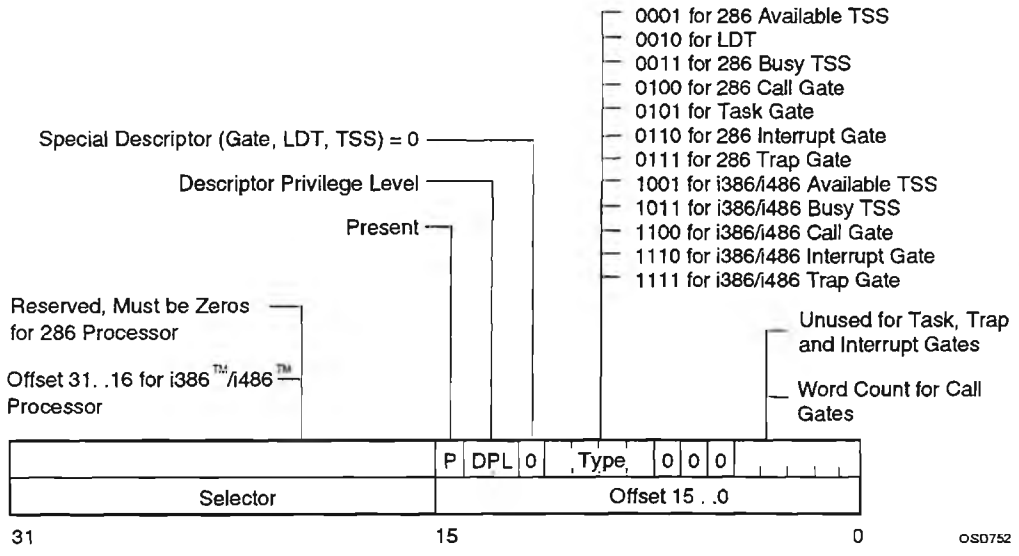

Figure 6-8 Special Descriptor for 286 and Higher Processors

Table 6-10 lists the names of the macros in the `i286.h` header file that isolate information for all descriptors (segment and special) and describes the meaning of the corresponding fields of the access byte. Refer to Figures 6-6 and 6-7 for the format of a segment descriptor. These macro names must be uppercase in the source text.

Table 6-10 General Descriptor Access Rights Macros for 286 and Higher Processors

Name	Value	Meaning
AR_SEGMENT	0x1000	This bit is 1 for a segment descriptor and 0 for a special descriptor, such as a gate.
AR_PRIV_MASK	0x6000	These two bits indicate the descriptor privilege level of the segment.
AR_PRESENT	0x8000	This bit indicates whether or not the segment is present in memory.
AR_PRIVILEGE(x)[1]		Isolates the descriptor privilege level in the low-order bits of a word.
AR_PRIV_SHIFT	13	Used by AR_PRIVILEGE to shift the descriptor privilege level bits.

[1]The macro definition is as follows:
#define AR_PRIVILEGE(X) (((X) & AR_PRIV_MASK) >> AR_PRIV_SHIFT)

Table 6-11 lists the names of the macros in the `i286.h` header file that isolate information for segment descriptors and describes the meaning of the corresponding fields of the segment descriptor access byte. Refer to Figures 6-4 and 6-5 for the format of a segment descriptor. These macro names must be uppercase in the source text.

Table 6-11 Segment Descriptor Access Rights Macros for 286 and Higher Processors

Name	Value	Meaning
AR_ACCESSED	0x0100	If the AR_SEGMENT bit is 1 and the AR_EXECUTABLE bit is 0, this bit is set to 1 when the segment is accessed or the selector for the segment is loaded into a selector register.
AR_WRITABLE	0x0200	If the AR_SEGMENT bit is 1 and the AR_EXECUTABLE bit is 0, this bit is 1 for a writable data segment and 0 for a read-only data segment.
AR_READABLE	0x0200	If the AR_SEGMENT bit is 1 and the AR_EXECUTABLE bit is 1, this bit is 1 for a readable code segment and 0 for an execute-only code segment.
AR_EXPAND_DOWN	0x0400	If the AR_SEGMENT bit is 1 and the AR_EXECUTABLE bit is 0, this bit is 1 for an expand-down data segment and 0 for a non-expand-down data segment.
AR_CONFORMING	0x0400	If the AR_SEGMENT bit is 1 and the AR_EXECUTABLE bit is 1, this bit is 1 for a conforming code segment and 0 for a non-conforming code segment.
AR_EXECUTABLE	0x0800	If the AR_SEGMENT bit is 1, this bit is 1 for a code segment and 0 for a data segment.

Table 6-12 lists the names of the macros in the `i286.h` header file that isolate information for special descriptors and describes the meaning of the corresponding fields of the segment descriptor access byte. These macro names must be uppercase in the source text.

Table 6-12 Special Descriptor Access Rights Macros for 286 and Higher Processors

Name	Value	Meaning
AR_CALL_GATE	0x0000	If the AR_SEGMENT bit is 0 and the AR_GATE bit is 1, the low-order type bits are 00 for a call gate.
AR_TSS	0x0100	If the AR_SEGMENT bit is 0 and the AR_GATE bit is 0, this bit is 1 for an available task state segment.
AR_TASK_GATE	0x0100	If the AR_SEGMENT bit is 0 and the AR_GATE bit is 1, the low-order type bits are 01 for a task gate.
AR_BUSY	0x0200	If the AR_SEGMENT bit is 0 and the AR_GATE bit is 0, this bit is 1 for a busy task state segment.
AR_INTR_GATE	0x0200	If the AR_SEGMENT bit is 0 and the AR_GATE bit is 1, the low-order type bits are 10 for an interrupt gate.
AR_GATE_MASK	0x0300	These two bits indicate the gate type.
AR_TRAP_GATE	0x0300	If the AR_SEGMENT bit is 0 and the AR_GATE bit is 1, the low-order type bits are 11 for a trap gate.
AR_GATE	0x0400	If the AR_SEGMENT bit is 0, this bit is 1 for a gate and 0 for other special descriptors.
AR_386_TYPE	0x0800	If the AR_SEGMENT bit is 0, this bit is 1 for an i386™ processor call, interrupt, or trap gate and 0 for a 286 processor call, interrupt, or trap gate.
AR_GATE_TYPE(*x*)[1]		Isolates the gate type in the high-order byte of a word.

[1]The macro definition is as follows:
#define AR_GATE_TYPE(X) ((X) & AR_GATE_MASK)

6.6.4 Adjusting Requested Privilege Level

A selector for a processor segment has a two-bit field called requested privilege level (RPL). This field normally contains the descriptor privilege level of the referring or calling code segment (referring code segment if the target is a data segment, calling code segment if the target is a code segment). Through adjustment, the RPL field can represent the descriptor privilege level of the original calling segment in a series of nested calls. Figure 6-9 shows the format of a selector.

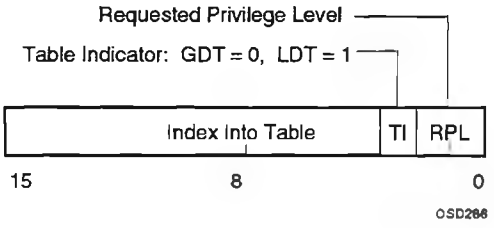

Figure 6-9 Selector for 286 and Higher Processors

Adjusting the RPL field of the selector of a called segment ensures that nested code segment accesses occur at a level no more privileged than the level of the original calling segment.

The `adjustrpl` function is for operating system software, but can execute at any privilege level. The function takes a selector value as an argument (the selector of the called segment). The prototype for `adjustrpl` is as follows:

```
selector adjustrpl (selector sel);
```

The `adjustrpl` function compares its argument with the selector for the code segment that called the routine that invoked `adjustrpl`. The `adjustrpl` function adjusts the selector argument and sets or clears the zero flag in the flags register as follows:

- If the RPL of the argument is more privileged than the RPL of the calling segment, the function sets the zero flag, adjusts the RPL of the selector argument to the lesser privilege level, and returns the adjusted selector.
- If the RPL of the argument is the same or less privileged than the RPL of the calling segment, the function clears the zero flag and returns the selector argument unchanged.

6.7 Manipulating the Control, Test, and Debug Registers of Intel386™ and Intel486™ Processors

The `i386.h` header file contains functions that enable iC-386 programs to examine and set the contents of the control, test, and debug registers. Accessing these registers can be made only from code executing at privilege level 0. Figure 6-10 shows the special registers accessible in the Intel386 and Intel486 processors.

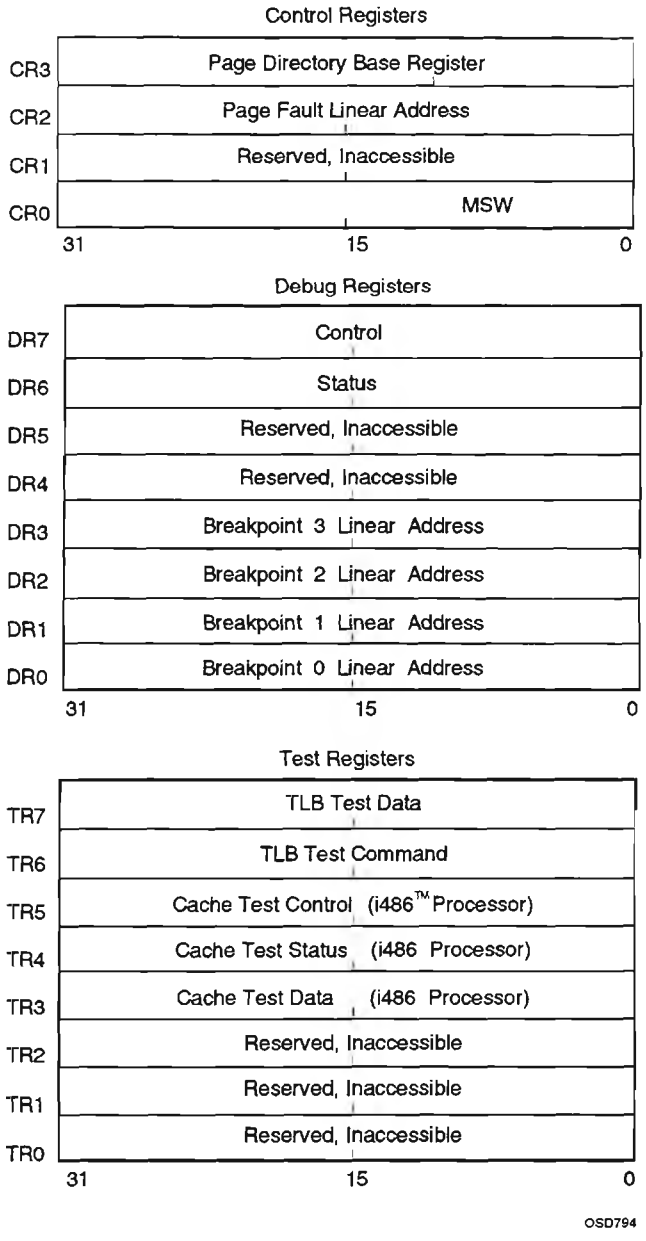

Figure 6-10 Control, Test, and Debug Registers of Intel386™ and Intel486™ Processors

The `getcontrolregister`, `gettestregister`, and `getdebugregister` functions return the 32-bit contents of the specified register. The functions take the register number as an argument. The register number must be a constant. Their prototypes are as follows:

```
unsigned int getcontrolregister (const unsigned char number);

unsigned int gettestregister    (const unsigned char number);

unsigned int getdebugregister   (const unsigned char number);
```

The `setcontrolregister`, `settestregister`, and `setdebugregister` functions load a 32-bit value into the specified register. The functions take the register number and the 32-bit value as arguments. Their prototypes are as follows:

```
void setcontrolregister (const unsigned char number,
                         unsigned int        value);

void settestregister    (const unsigned char number,
                         unsigned int        value);

void setdebugregister   (const unsigned char number,
                         unsigned int        value);
```

Control register 0 (CR0) contains the machine status word in its low-order 16 bits. See Section 6.6.2 for functions and macros that manipulate the machine status word. Figure 6-11 shows the format of control register 0.

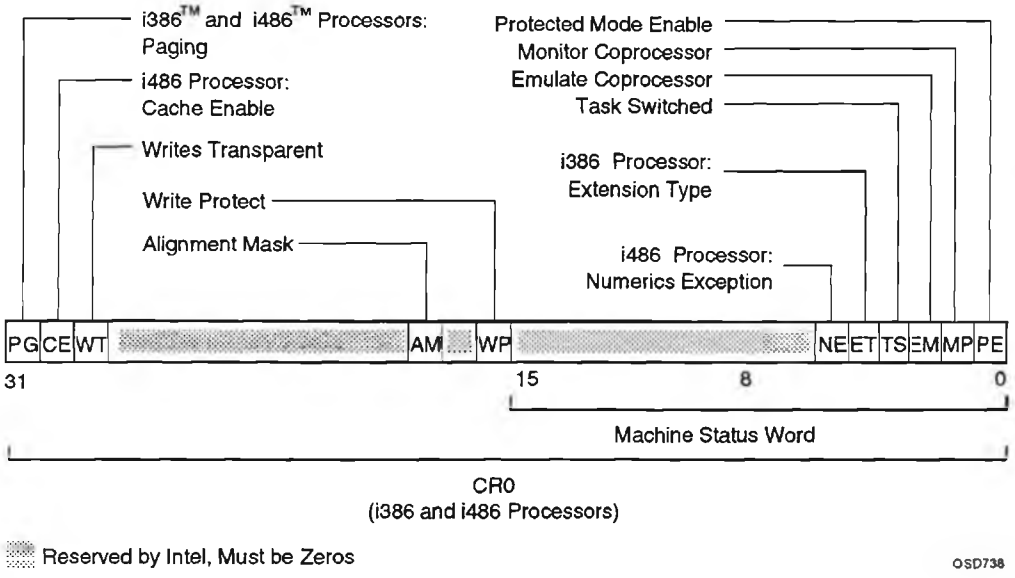

Figure 6-11 Control Register 0 of Intel386™ and Intel486™ Processors

Table 6-13 lists the names of the macros in the `i386.h` header file and describes the meaning of the corresponding fields in the high-order 16 bits of the CR0 control register. These macro names must be uppercase in the source text.

Table 6-13 Control Register 0 Macros for Intel386™ and Intel486™ Processors

Name	Value	Meaning
CR0_EXTENSION_TYPE	0x0010	This bit is 1 if the i387™ coprocessor or the i486™ processor is present, and 0 if the i287™ coprocessor is present.
CR0_PAGING_ENABLED	0x8000	This bit is 1 if paging is enabled, or 0 if paging is disabled.

6.8 Managing the Features of the Intel486™ Processor

The `i486.h` header file contains functions that enable iC-386 programs to manipulate the unique features of the Intel486 processor.

The Intel386 and Intel486 processors execute memory read and write operations from low-order to high-order addresses. This order is called "little endian." The `byteswap` function reverses the order of bytes in a 32-bit double word, converting little endian format to big endian format. This feature is useful for transferring data between the Intel486 processor and foreign processors or peripherals. The function takes a 32-bit double word as its argument, and returns the swapped 32-bit value. The function prototype is as follows:

```
unsigned int byteswap (unsigned int value);
```

The Intel486 processor also contains on-chip caches and provides instructions to manipulate those caches. The `invalidatedatacache` function flushes the internal data cache. Its prototype is as follows:

```
void invalidatedatacache (void);
```

The `wbinvalidatedatacache` function flushes the internal data cache and directs any external cache to write back its contents and flush itself. The function prototype is as follows:

```
void wbinvalidatedatacache (void);
```

The translation lookaside buffer (TLB) is a cache used for page table entries. The `invalidatetlbentry` function marks a single entry in the translation lookaside buffer (TLB) invalid. The function takes an address of a memory location as an argument; the argument must have the address operator (`&`) preceding it. If the TLB contains a valid entry which maps the argument address, that entry is marked invalid. The function prototype is as follows:

```
void invalidatetlbentry (void far * memoryaddress);
```

6.9 Manipulating the Numeric Coprocessor

The `i86.h` header file contains several functions, macros, and data types that enable iC-86/286/386 programs to manipulate a numeric coprocessor, a true software emulator, or the Intel486 processor floating-point unit. See the following manuals, all listed in Chapter 1, for information on numeric coprocessors:

- *8086/8088 Programmer's and Hardware Reference* or *ASM86 Assembly Language Reference Manual* for information on the 8087 numeric coprocessor.
- *80286 Programmer's Reference Manual* or *ASM286 Assembly Language Reference Manual* for information on the i287™ numeric coprocessor.
- *80387 Programmer's Reference Manual* or *ASM386 Assembly Language Reference Manual* for information on the Intel387 numeric coprocessor. The Intel486 processor contains an on-chip floating-point unit (FPU) that operates exactly the same as the Intel387 coprocessor.

This section uses the term "numeric coprocessor" to indicate a coprocessor, emulator, or on-chip unit.

The `initrealmathunit` function initializes the numeric coprocessor, however, normally the iC-86/286/386 startup code initializes the coprocessor. Use the `initrealmathunit` function if the standard startup code is not used. The prototype for `initrealmathunit` is as follows:

```
void initrealmathunit (void);
```

The numeric coprocessor uses 8 numeric data registers, a control word register, a status word register, a tag word register, an instruction pointer and a data pointer. The coprocessor treats the numeric data registers as if they were a stack. Figure 6-12 shows the numeric data register set. Figures 6-13 and 6-14 show the environment registers for the 8087 or i287 coprocessors, and Intel387 coprocessor or Intel486 FPU, respectively.

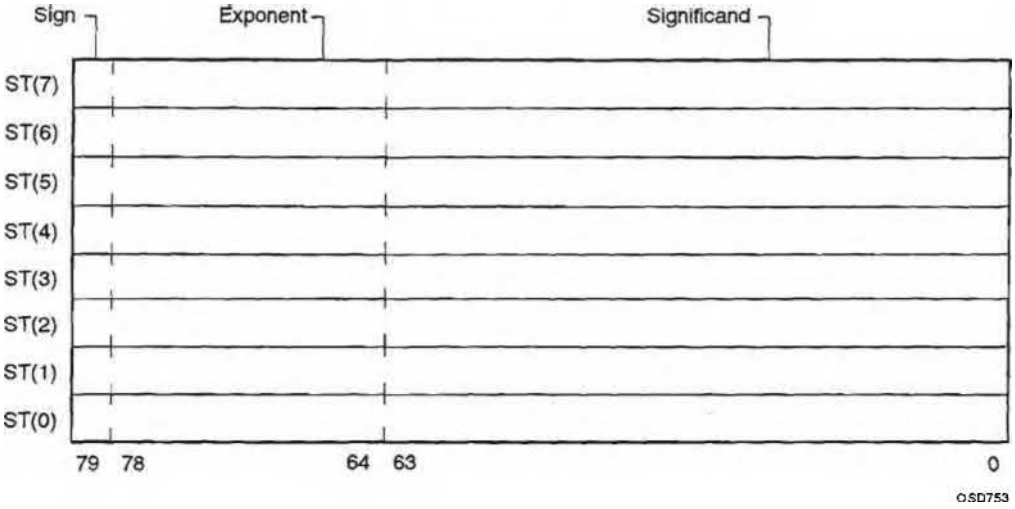

Figure 6-12 Numeric Coprocessor Stack of Numeric Data Registers

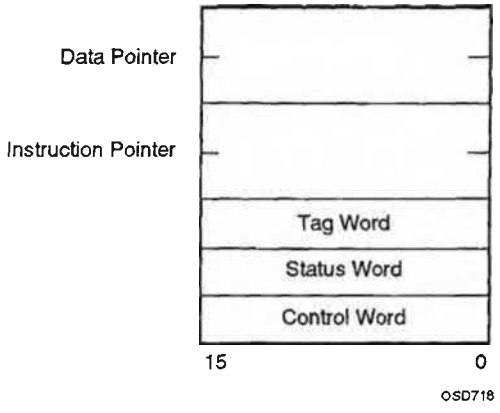

Figure 6-13 8087 or i287™ Numeric Coprocessor Environment Registers

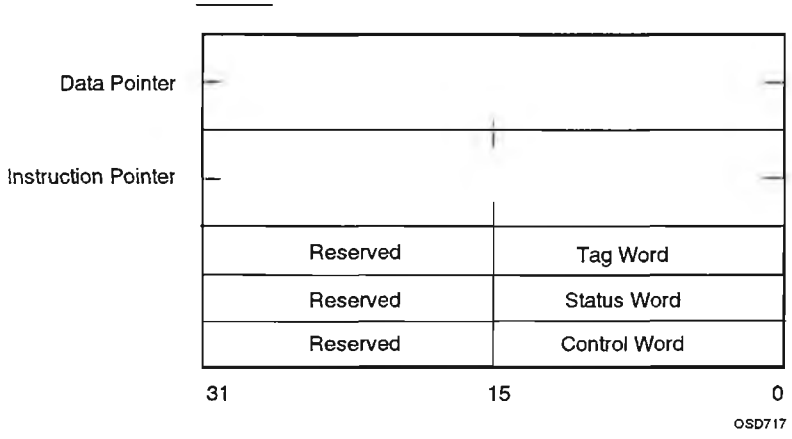

Figure 6-14 Intel387™ Numeric Coprocessor or Intel486™ FPU Environment Registers

The `setrealmode` function sets the fields of the control word. See Section 6.9.2 for more information on the control word and the `setrealmode` function.

The `getrealerror` function retrieves the value of the status word. See Section 6.9.3 for more information on the status word and the `getrealerror` function.

The numeric coprocessor's environment consists of the contents of the control word, status word, tag word, instruction pointer, and data pointer. The numeric coprocessor's state consists of the contents of all the registers. See Section 6.9.5 for data types and functions relative to the numeric data registers, environment, and state.

6.9.1 Tag Word

The tag word contains a 2-bit field for each numeric data register. The tag fields indicate the kind of value in the register and whether or not the register contains a valid value. Figure 6-15 shows the tag word and the possible values for each tag.

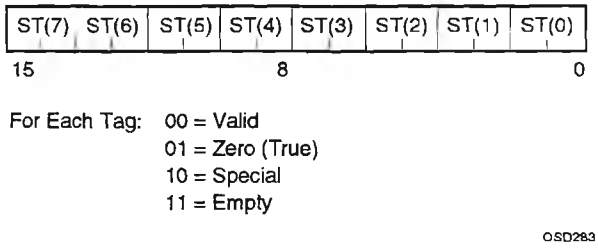

Figure 6-15 Numeric Coprocessor Tag Word

Table 6-14 lists the names of the tag word macros in the `i86.h` header file that isolate a tag from the tag word. These macro names must be uppercase in the source text.

Table 6-14 Numeric Coprocessor Tag Word Macros

Name	Value	Meaning
I87_TAG_MASK	0x0003	Each tag is 2 bits.
I87_TAG(*x,y*)[1]		Isolates the tag for the *y*th numeric register in the low-order bits of a word.
I87_TAG_SHIFT	2	Used by I87_TAG to shift the appropriate tag into position.

[1]The macro definition is as follows:
#define I87_TAG(X,y) (((X).tag >> (I87_TAG_SHIFT * (y))) & I87_TAG_MASK)

6.9.2 Control Word

The control word contains exception mask bits and three sets of control bits. The mask bits correspond to the flags in the status word (refer to Figure 6-17 for the format of the status word). Figure 6-16 shows the format of the control word.

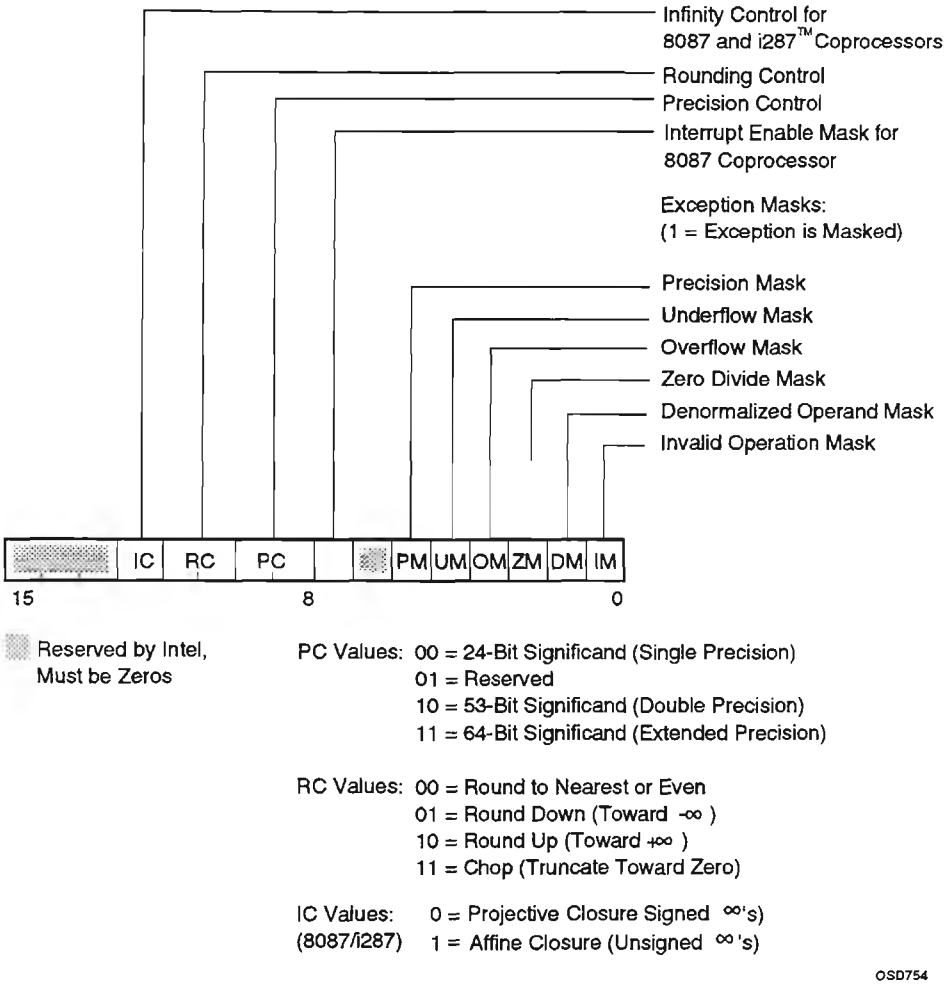

Figure 6-16 Numeric Coprocessor Control Word

The `setrealmode` function loads a value into the control word. The function takes the value as its argument. The prototype for `setrealmode` is as follows:

```
void setrealmode (unsigned short mode);
```

Table 6-15 lists the names of the macros in the `i86.h` header file that isolate information from the control word. These macro names must be uppercase in the source text.

Table 6-15 Numeric Coprocessor Control Word Macros

Name	Value	Meaning
I87_INVALID_OPERATION	0x0001	This bit masks or unmasks the IE bit in the status word.
I87_DENORMALIZED_OPERAND	0x0002	This bit masks or unmasks the DE bit in the status word.
I87_ZERO_DIVIDE	0x0004	This bit masks or unmasks the ZE bit in the status word.
I87_OVERFLOW	0x0008	This bit masks or unmasks the OE bit in the status word.
I87_UNDERFLOW	0x0010	This bit masks or unmasks the UE bit in the status word.
I87_PRECISION	0x0020	This bit masks or unmasks the PE bit in the status word.
I87_CONTROL_PRECISION	0x0300	These two bits control whether a 24-bit, 53-bit, or 64-bit significand is used.
I87_PRECISION_24_BIT	0x0000	The precision bits are 00 for 24-bit significand (single) precision.
I87_PRECISION_53_BIT	0x0200	The precision bits are 10 for 53-bit significand (double) precision.
I87_PRECISION_64_BIT	0x0300	The precision bits are 11 for 64-bit significand (extended) precision.

Table 6-15 Numeric Coprocessor Control Word Macros (continued)

Name	Value	Meaning
I87_CONTROL_ROUNDING	0x0C00	These two bits control the method used in rounding.
I87_ROUND_NEAREST	0x0000	The rounding bits are 00 to round to nearest or even.
I87_ROUND_DOWN	0x0400	The rounding bits are 01 to round down.
I87_ROUND_UP	0x0800	The rounding bits are 10 to round up.
I87_ROUND_CHOP	0x0C00	The rounding bits are 11 to truncate toward zero.
I87_CONTROL_INFINITY[1]	0x1000	This bit controls whether projective closure or affine closure is used to represent infinity.
I87_INFINITY_PROJECTIVE[1]	0x0000	The infinity bit is 0 to use projective closure (unsigned infinity).
I87_INFINITY_AFFINE[1]	0x1000	The infinity bit is 1 to use affine closure (signed infinities).

[1]For 8087 and i287 numeric coprocessors only.

6.9.3 Status Word

The status word contains flags, condition codes, the top of the stack of numeric data registers, and a busy bit. The flag bits correspond to the mask bits in the control word (refer to Figure 6-16 for the format of the control word). Figure 6-17 shows the format of the status word. Tables 6-16 and 6-17 show the values of the condition codes for the 8087 or i287, and Intel387 numeric coprocessors or Intel487 FPU, respectively.

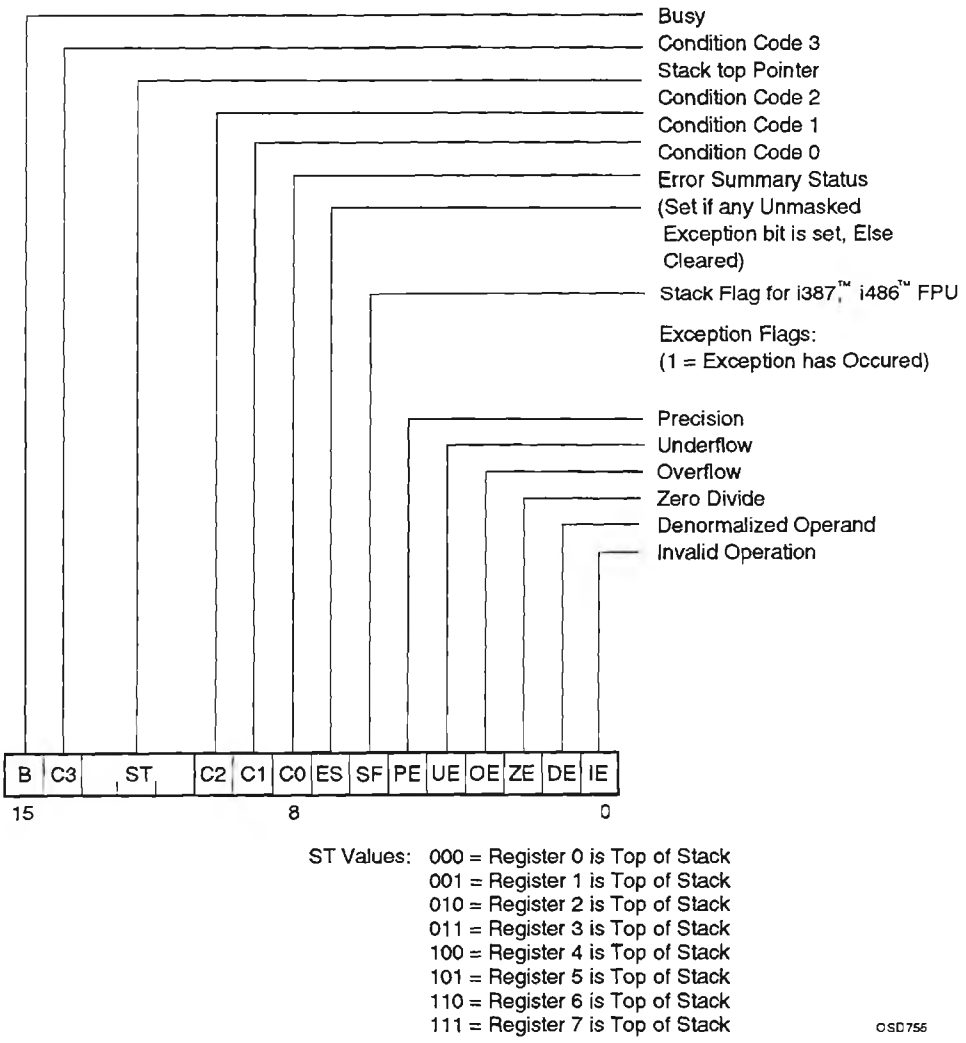

Figure 6-17 Numeric Coprocessor Status Word

Table 6-16 8087 or i287™ Numeric Coprocessor Condition Codes

Instruction Type	C_3	C_2	C_1	C_0	Interpretation
compare, test	0	0	X	0	ST > source or 0 (FTST)
	0	0	X	1	ST < source or 0 (FTST)
	1	0	X	0	ST = source or 0 (FTST)
	1	1	X	1	ST is not comparable
remainder	Q_1	0	Q_0	Q_2	complete reduction with three low bits of quotient in C_0, C_3, and C_1
	U	1	U	U	incomplete reduction
examine	0	0	0	0	valid, positive unnormalized
	0	0	0	1	invalid, positive, exponent=0
	0	0	1	0	valid, negative, unnormalized
	0	0	1	1	invalid, negative, exponent=0
	0	1	0	0	valid, positive, normalized
	0	1	0	1	infinity, positive
	0	1	1	0	valid, negative, normalized
	0	1	1	1	infinity, negative
	1	0	0	0	zero, positive
	1	0	0	1	empty register
	1	0	1	0	zero, negative
	1	0	1	1	empty register
	1	1	0	0	invalid, positive, exponent=0
	1	1	0	1	empty register
	1	1	1	0	invalid, negative, exponent=0
	1	1	1	1	empty register

Key:
ST = top of stack
X = instruction does not affect value
FTST = instruction that compares ST with zero
U = instruction leaves value undefined
Q_n = quotient bit n following complete reduction (C_2=0)

Table 6-17 Intel387™ Numeric Coprocessor or Intel486™ FPU Condition Codes

Instructions	C_3	C_2	C_1	C_0	Interpretation
FCOM, FCOMP, FCOMPP,	0	0	0 or O/U	0	stack top > operand
FTST, FUCOM, FUCOMP,	0	0	0 or O/U	1	stack top < operand
FUCOMPP, FICOM, FICOMP	1	0	0 or O/U	0	stack top = operand
	1	1	0 or O/U	1	unordered
FPREM, FPREM1	Q_1	0	Q_0	Q_2	complete reduction with three low bits of quotient in C_0, C_3, and C_1
	U	1	U	U	incomplete reduction
FXAM	0	0	Sign	0	unsupported
	0	0	Sign	1	NaN
	0	1	Sign	0	normal
	0	1	Sign	1	infinity
	1	0	Sign	0	zero
	1	0	Sign	1	empty
	1	1	Sign	0	denormal
FCHS, FABS, FXCH, FINCTOP, FDECTOP, Constant loads, FXTRACT, FLD, FILD, FBLD, FSTP	U	U	0 or O/U	U	
FIST, FBSTP, FRNDINT, FST, FSTP, FADD, FMUL, FDIV, FDIVR, FSUB, FSUBR, FSCALE, FSQRT, FPATAN, F2XM1, FYL2X, FYL2XP1	U	U	Round or O/U	U	rounding valid when PE bit of status word is set
FPTAN, FSIN, FCOS, FSINCOS	U	0	Round or O/U	U	complete reduction
	U	1	U	U	incomplete reduction
FLDENV, FRSTOR	Loaded	Loaded	Loaded	Loaded	each bit loaded from memory
FLDCW, FSTENV, FSTCW, FSTSW, FCLEX, FINIT, FSAVE	U	U	U	U	undefined

Key:
O/U = When IE and SF bits of status word are set
1 = stack overflow and 0 = stack underflow;
U = instruction leaves value undefined
Q_n = quotient bit n following complete reduction (C_2=0)

The `getrealerror` function returns the contents of the low-order byte of the status word and then clears the exception flags in the status word to zeros. The prototype for `getrealerror` is as follows:

```
unsigned short getrealerror (void);
```

Table 6-18 lists the names of the macros in the `i86.h` header file that isolate information from the status word. These macro names must be uppercase in the source text.

Table 6-18 Numeric Coprocessor Status Word Macros

Name	Value	Meaning
I87_STATUS_ERROR	0x0080	This bit is 1 if any unmasked exception bit is set.
I87_STATUS_STACKTOP_MASK	0x3800	These three bits indicate the numeric register that is at the top of the stack.
I87_STATUS_STACKTOP_SHIFT	11	Used by I87_STATUS_STACKTOP to shift the stack top bits.
I87_STATUS_STACKTOP(*env*)[1]		Isolates the stack top bits in the low-order bits of a word.
I87_STATUS_BUSY	0x8000	This bit is 1 when the coprocessor is executing or 0 when the coprocessor is idle.
I87_STATUS_CONDITION_CODE	0x4700	These four bits are the condition code bits; they reflect the outcome of arithmetic operations.

[1]The macro definition is as follows:
#define I87_STACKTOP(env) (((env).status & I87_STATUS_STACKTOP_MASK) >> \
I87_STATUS_STACKTOP_SHIFT)

Table 6-18 Numeric Coprocessor Status Word Macros (continued)

Name	Value	Meaning
I87_CONDITION_C0	0x0100	This bit is condition code bit 0 (see Tables 6-16 and 6-17).
I87_CONDITION_C1	0x0200	This bit is condition code bit 1 (see Tables 6-16 and 6-17).
I87_CONDITION_C2	0x0400	This bit is condition code bit 2 (see Tables 6-16 and 6-17).
I87_CONDITION_C3	0x4000	This bit is condition code bit 3 (see Tables 6-16 and 6-17).

6.9.4 Data Pointer and Instruction Pointer

The format of the data pointer and instruction pointer differs depending on which numeric coprocessor is used and whether the processor is executing in real mode or protected mode.

6.9.4.1 8087 or i287™ Numeric Coprocessor Data Pointer and Instruction Pointer

Figure 6-18 shows the real mode format of data pointer and instruction pointer for the 8087 or i287 numeric coprocessor, and the protected mode format of the pointers for the i287 numeric coprocessor.

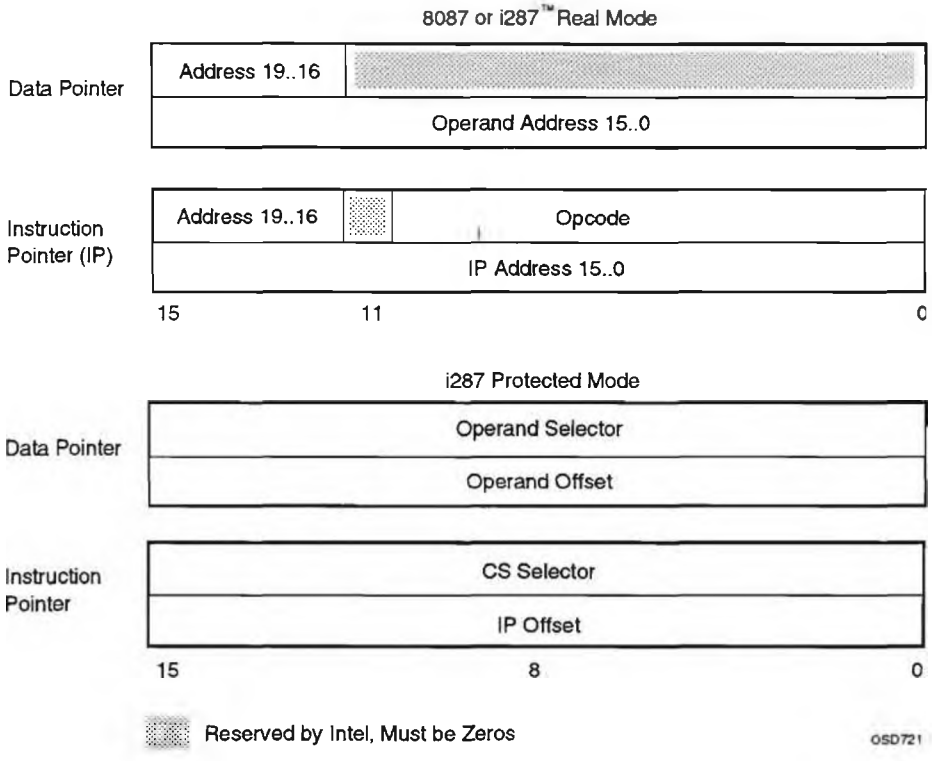

Figure 6-18 8087 or i287™ Numeric Coprocessor Data Pointer and Instruction Pointer

The `i87_real_address` and `i87_address` data types define the structure of the information in the data pointer or instruction pointer for the 8087 or i287 coprocessor. The `I87_REAL_ADDRESS` macro computes a far pointer from the real mode address information in the data pointer or instruction pointer of the 8087 or i287 coprocessor.

The i87_real_address structure type accommodates the value of the real mode data pointer. The opcode field is undefined for the data pointer. The i87_real_address structure definition is as follows:

```
#pragma ALIGN("i87_real_address")
struct  i87_real_address
{
   unsigned offset:    16, : 0;
   unsigned opcode:    11, : 1;
   unsigned selector:   4, : 0;
};
```

The i87_address union type accommodates the value of the real mode or protected mode data pointer or instruction pointer. The i87_address union definition is as follows:

```
union i87_address
{
   struct i87_real_address real;
   void far * protected;
};
```

The I87_REAL_ADDRESS macro computes a far pointer from an i87_address union. This macro name must be in uppercase in the source text. The macro definition is as follows:

```
#define I87_REAL_ADDRESS(addr)                                  \
        buildptr((selector) ((addr).selector & 0xF0000),  \
                 (void near * (addr).offset)
```

6.9.4.2 Intel387™ Numeric Coprocessor and Intel486™ FPU Data Pointer and Instruction Pointer

Figure 6-19 shows the real mode and protected mode formats of the data pointer and instruction pointer for the Intel387 numeric coprocessor or Intel486 FPU.

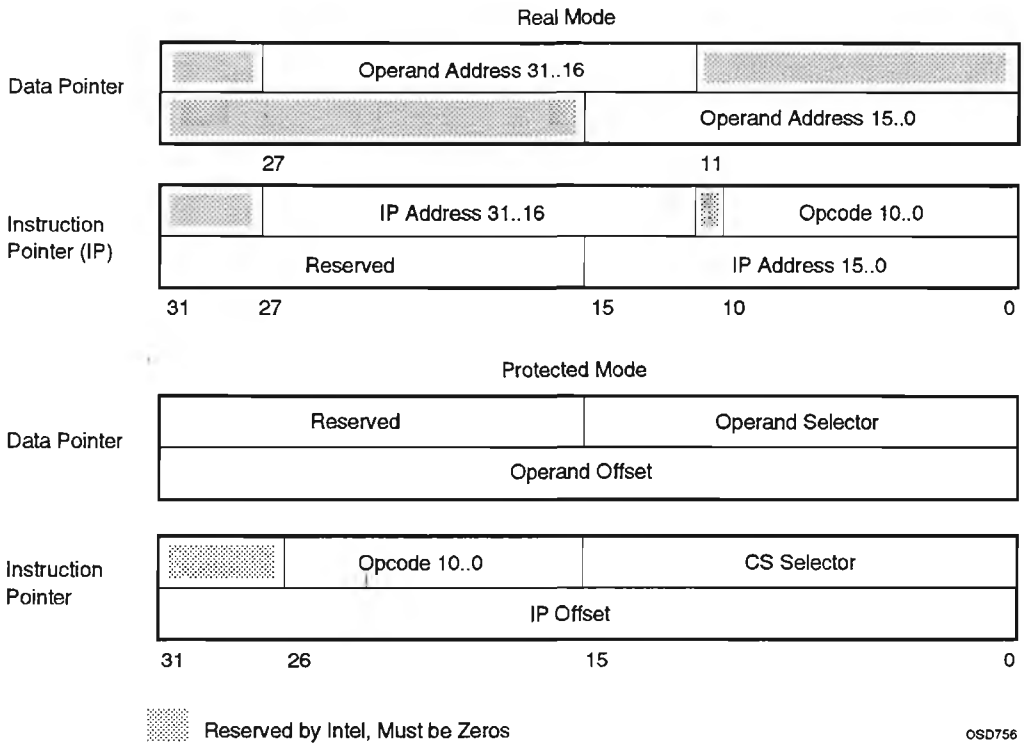

Figure 6-19 Intel387™ Numeric Coprocessor or Intel486™ FPU Data Pointer and Instruction Pointer

The `i387_real_address`, `i387_protected_addr`, and `i387_address` data types define the structure of the information in the data pointer or instruction pointer for the Intel387 numeric coprocessor or Intel486 FPU.

The i387_real_address structure type accommodates the value of the real mode data pointer or instruction pointer. The opcode field is undefined for the data pointer. The structure definition is as follows:

```
#pragma ALIGN("i387_real_address")
struct  i387_real_address
{
   unsigned ip1   : 16, : 16;
   unsigned opcode: 11, : 1;
   unsigned ip2   : 16, : 4;
   unsigned op1   : 16, : 16, : 12;
   unsigned op2   : 16, : 4;
};
```

The i387_protected_addr structure type accommodates the value of the protected mode data pointer or instruction pointer. The opcode field is undefined for the data pointer. The structure definition is as follows:

```
#pragma ALIGN("i387_protected_addr")
struct  i387_protected_addr
{
   unsigned ip_offset: 32;
   unsigned cs_sel   : 16;
   unsigned opcode   : 11, : 5;
   unsigned op_offset: 32;
   unsigned op_sel   : 16, : 16;
};
```

The i387_address union type accommodates the value of the real mode or protected mode data pointer or instruction pointer. The union definition is as follows:

```
union i387_address
{
   struct i387_real_address   real;
   struct i387_protected_addr prot;
};
```

6.9.5 Saving and Restoring the Numeric Coprocessor State

The numeric coprocessor's environment is the contents of the control word, status word, tag word, instruction pointer, and data pointer. The numeric coprocessor's state is the contents of the environment registers plus the numeric data register stack. Refer to Figures 6-12 through 6-14 for the general format of these registers.

The `i87_environment` and `i387_environment` data types define the environment for the 8087 or i287 coprocessors, and the Intel387 coprocessor or Intel486 FPU, respectively. The `i87_tempreal` data type and the `tempreal_t` typedef define the format of one numeric register. The `i87_state` and `i387_state` data types define the structure of all the registers for the 8087 or i287 coprocessors, and the Intel387 coprocessor or Intel486 FPU, respectively. The `saverealstatus` and `restorerealstatus` functions manipulate the entire state of the numeric coprocessor.

The `i87_environment` structure type defines the 8087 or i287 numeric coprocessor environment. The structure definition is as follows:

```
#pragma ALIGN("i87_environment")
struct i87_environment
{
   unsigned           control: 16, : 0;
   unsigned           status : 16, : 0;
   unsigned           tag    : 16, : 0;
   union i87_address  instruction;
   union i87_address  operand;
};
```

The `i387_environment` structure type defines the Intel387 numeric coprocessor or Intel486 FPU environment. The structure definition is as follows:

```
#pragma ALIGN("i387_environment")
struct i387_environment
{
   unsigned           control: 16, : 16;
   unsigned           status : 16, : 16;
   unsigned           tag    : 16, : 16;
   union i387_address ptrs_n_opcode;
};
```

The i87_tempreal structure type and tempreal_t typedef define the fields in one numeric register. You can define the SBITFIELD macro to control whether the one-bit sign field is signed or unsigned. The definitions for i87_tempreal and tempreal_t are as follows:

```
#pragma NOALIGN ("i87_tempreal")
struct  i87_tempreal
{
   char     significand[8];
   unsigned exponent: 15;
#if defined(SBITFIELD)
   signed   sign    :  1;
#else
   unsigned sign    :  1;
#endif
};

typedef struct i87_tempreal tempreal_t;
```

The i87_state structure defines the state of the 8087 or i287 numeric coprocessor. The structure definition is as follows:

```
struct i87_state
{
   struct i87_environment environment;
   tempreal_t             stack[8];
};
```

The i387_state structure defines the state of the Intel387 numeric coprocessor or Intel486 FPU. The structure definition is as follows:

```
struct i387_state
{
   struct i387_environment environment;
   tempreal_t              stack[8];
};
```

The saverealstatus function copies the contents of the numeric coprocessor state into a specific location of type i87_state for the 8087 or i287 coprocessor, or i387_state for the Intel387 coprocessor or Intel486 FPU. The function takes a pointer to this destination as an argument.

The prototype for saverealstatus for 87 or 287 coprocessors is as follows:

```
void saverealstatus (struct i87_state * destinationptr);
```

The prototype for `saverealstatus` for the Intel387 coprocessor or Intel486 FPU is as follows:

```
void saverealstatus (struct i387_state * destinationptr);
```

The `restorerealstatus` function loads values into all the numeric coprocessor registers. The function takes as an argument a pointer to the `i87_state` save area for the 8087 or i287 coprocessor, or the `i387_state` save area for the Intel387 coprocessor or Intel486 FPU.

The prototype for `restorerealstatus` for 8087 or i287 coprocessors is as follows:

```
void restorerealstatus (struct i87_state const * sourceptr);
```

The prototype for `restorerealstatus` for the Intel387 coprocessor or Intel486 FPU is as follows:

```
void restorerealstatus (struct i387_state const * sourceptr);
```

Contents

7

Assembler Header File

Assembler Header File 7

The `util.ah` header file contains macros that help interface assembly routines to iC-86/286/386 programs. To use these facilities, include the header file in your assembly routines. The `util.ah` assembler header file provides the following facilities:

- segmentation and linkage directives and generic data type specifiers for any standard memory model (small, compact, medium, large, or flat)
- standard prolog and epilog for conformance to either the variable parameter list (VPL) or the fixed parameter list (FPL) calling convention
- simple directives for using parameters and automatic variables

To select these features, use header controls that the `util.ah` macros recognize. The source for the `util.ah` header file is common for ASM86, ASM286, and ASM386. See Section 7.6.5 for several examples.

7.1 Macro Selection

The macros defined in `util.ah` fall into five groups, as follows:

Flag macros	indicate segmentation model, calling convention, and instruction set used in the assembly.
Register macros	are generic register names and expand to appropriate registers depending on the calling convention.
Segment macros	are names of segments or groups as determined by segmentation model.
Type macros	are generic data type specifications and expand to appropriate types depending on segmentation model.

Operation macros — are instructions or directives for commonly used assembly language operations.

Ensure that the `:include:` environment variable contains the path for the `util.ah` file. For example, set `:include:` as follows:

```
C:> set :include:=\intel\ic86\lib\
```

Use the following line in your assembly source text to include `util.ah`:

```
$include(:include:util.ah)
```

The expansion of the macros in `util.ah` depends on the value of a macro named `controls`, which contains a list of header controls that specify the behavior of the `util.ah` macros. Table 7-1 lists these header controls.

Table 7-1 Assembler Header Controls for Macro Selection

Header Control	Abbr.	Description
asm86[1]		generate code for ASM86
asm286[2]		generate code for ASM286
asm386[3]		generate code for ASM386
small	sm	generate code for small memory model
compact	cp	generate code for compact memory model
medium[4]	md	generate code for medium memory model
large[4]	la	generate code for large memory model
flat[5]	fl	generate code for flat memory model
fixedparams	fp	generate prolog/epilog for FPL calling convention
varparams	vp	generate prolog/epilog for VPL calling convention
mod86[1]		generate 86 processor code
mod186[1]		generate 186 processor code
'module=*name*'[6]		set module name
ram		generate code for RAM sub-model
rom		generate code for ROM sub-model
'stacksize=*size*'[6]		set size of the stack segment

[1]For ASM86 applications only.
[2]For ASM286 applications only.
[3]For ASM386 applications only.
[4]For ASM86 or ASM286 applications only.
[5]For ASM386 applications only.
[6]Use single quotation marks around these header controls on the assembler invocation line.

If you include `util.ah`, you must define the `controls` macro in the assembler invocation or in the assembly source text before the line including `util.ah`. Otherwise, the assembler reports an undefined macro error. You can define the `controls` macro with an empty value; any header controls that you do not specify take on their default settings. Table 7-2 lists the default settings for the header controls.

Table 7-2 Assembler Header Control Defaults

Header Controls	Default
asm86, asm286, or asm386	asm86
small, compact, medium, large, or flat	small
mod86 or mod186	mod86
fixedparams or varparams	fixedparams
module=*name*	module=anonymous
ram or rom	ram
stacksize=*size*	stacksize=0

You can define the `controls` macro in the assembler invocation, or in the source text, or both places, as follows:

- If you define the `controls` macro in the assembler invocation, provide a definition for the `controls` macro each time you assemble the program. Thus, each time you assemble the program you can specify any header control settings or define the `controls` macro with an empty value, letting the unspecified controls take on their default settings.
- If you define the `controls` macro in the assembly source text as a simple list of header controls, you can change the header control settings only by modifying the source text. When the assembler processes a macro definition, it discards any existing definition of that macro, so defining the `controls` macro in the assembler invocation has no effect.
- You can define the `controls` macro in the assembler invocation, then use that definition of it as part of a redefinition of the `controls` macro in the assembly source text. This forces some header control settings to take effect any time you invoke the assembler for that source text. You can also override other header control settings and let some header controls take on their global default settings.

The DOS syntax for the assembler invocation is as follows:

```
asmn86 file [asm_controls] %define(controls)([header_controls])
```

Where:

`asmn86`	is `asm86`, `asm286`, or `asm386`.
`file`	is the source file to assemble.
`asm_controls`	are controls for the assembly. See the *ASM Macro Assembler Operating Instructions* for your host system, listed in Chapter 1, for information on ASM controls.
`header_controls`	are header controls from Table 7-1, separated by spaces.

Within the source text, the syntax for defining the `controls` macro and including the `util.ah` header file is as follows:

```
%define(controls)
       ([file_default_ctls] %controls [file_override_ctls])
$include(:include:util.ah)
```

If you specify conflicting controls, the last one encountered by the assembler takes effect. The precedence levels of the header controls are as follows:

- The *`file_override_ctls`*, specified last in the `controls` definition in the source text, have the highest precedence. The *`file_override_ctls`* always take effect, overriding any conflicting control in the *`header_controls`* or *`file_default_ctls`*.
- The *`header_controls`*, specified in the assembler invocation (and expanded in the source text from the `%controls` embedded in the `controls` definition), have second precedence. The *`header_controls`* take effect when they do not conflict with the *`file_override_ctls`*. A control in the *`header_controls`* overrides any conflicting control in the *`file_default_ctls`*.

- The `file_default_ctls`, specified first in the `controls` definition in the source text, have third precedence. The `file_default_ctls` take effect whenever they do not conflict with the `header_controls` or `file_override_ctls`.
- The global default controls, listed in Table 7-2, have the lowest precedence. The global default controls take effect only when they do not conflict with the `file_override_ctls`, `header_controls`, or `file_default_ctls`.

Figure 7-1 shows the precedence relationship depending on where controls are placed.

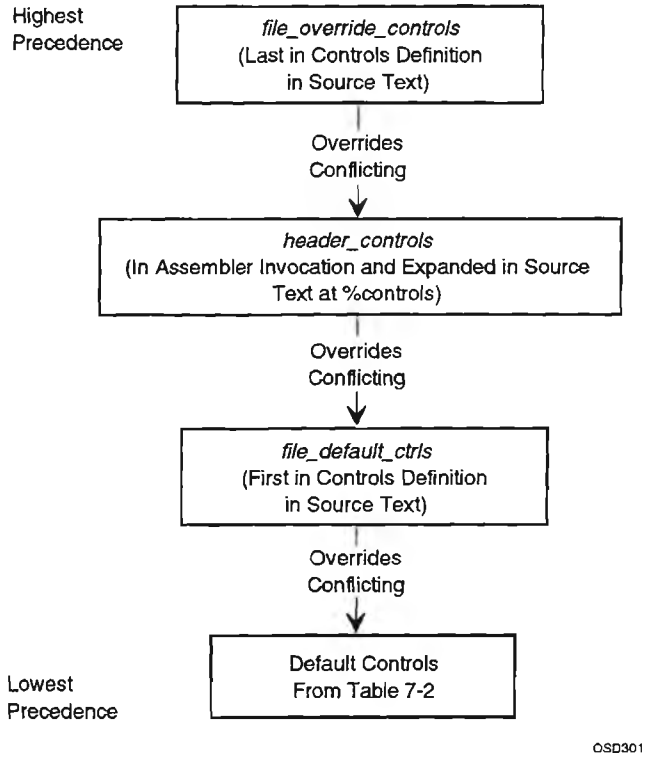

Figure 7-1 Precedence Levels of Assembler Header Controls

The following examples demonstrate invoking the assembler with header controls to select macros.

1. This example invokes the ASM86 assembler with non-default assembler settings and header controls. The assembler processes the source text in the file `utest.asm` using the 186 processor instruction set and compact model, producing an object module with variable parameter list (VPL) calling convention.

```
C:> asm86 utest.asm %define(controls)(mod186 cp vp)
```

2. This example defines `controls` in the assembly source text. The header control settings specify ASM386, the small model, and the ROM submodel.

```
%define(controls)(asm386 sm rom)
$include(:include:util.ah)
```

3. This example defines header control defaults partly different from the global default controls. The assembly source text contains the following:

```
%define(controls)
       (cp vp 'stacksize=50' %controls 'module=utl')
```

This definition of the `controls` macro sets the following defaults:

- The object module is compact model rather than small.
- The calling convention is variable parameter-list (VPL) rather than fixed parameter list (FPL).
- The stack size is 50 rather than 0.
- The module name is `utl` instead of `anonymous` and cannot be overridden; its position after `%controls` indicates that it is a file override control.

The assembler invocation for ASM286 on DOS is as follows:

```
C:> asm286 utest.asm %define(controls)(asm286 sm rom)
```

The `controls` defined in the assembler invocation override only the file default controls that specify the memory model, as follows:

- The object module is small ROM model rather than compact RAM.
- The calling convention is VPL and the stack size is 50, as specified in the file default controls.

7.2 Flag Macros

The value of a flag macro is either 1 (set) or 0. Use flag macros in ASM macro programming language `%if` constructs. See the *ASM Macro Assembler Operating Instructions* manual for your system, listed in Chapter 1, for more information on the macro programming language.

Use the flag macros to test the following conditions:

`%const_in_code`	indicates that constants are in the code segment; set by the `rom` header control.
`%far_code`	indicates that function pointers are far; set by the `medium` or `large` header controls.
`%far_data`	indicates that data pointers are far; set by the `compact`, `large`, or `rom` header controls.
`%far_stack`	indicates that the stack is in a separate segment, that is, the SS register value is not the same as the DS register value; set by the `compact` or `large` header controls.
`%fpl`	indicates that the calling convention is fixed parameter list (FPL); set by the `fixedparams` header control.
`%i186_instrs`	indicates whether to use or simulate instructions available only in 186 and higher instruction sets; set by the `mod186`, `asm286`, or `asm386` header controls.

`%i86_asm`
`%i286_asm`
`%i386_asm`
	indicates code specific to a particular architecture when code is common between products targeted for 86, 286, or Intel386™ processors; set by `asm86`, `asm286`, or `asm386` header controls, respectively.

`%set_ds`	indicates that each module has its own data segment; set by the `large` header control.

Table 7-3 lists which flag macros are set when you specify various header controls.

Table 7-3 Assembler Flag Macros Set by Header Controls

Header Control	Flag Macros Set
asm86	%i86_asm
asm286	%i286_asm %i186_instrs
asm386	%i386_asm %i186_instrs
compact	%far_data %far_stack
medium	%far_code
large	%far_code %far_data %far_stack %set_ds
fixedparams	%fpl
mod186	%i186_instrs
rom	%const_in_code %far_data

7.3 Register Macros

You can use a register macro as an instruction operand in place of the register name. Table 7-4 shows macros useful in specifying operands to instructions.

Table 7-4 Assembler Register Macros

Macro	ASM86 Expansion	ASM286 Expansion	ASM386 Expansion
%ax	ax	ax	eax
%bx	bx	bx	ebx
%cx	cx	cx	ecx
%dx	dx	dx	edx
%bp	bp	bp	ebp
%sp	sp	sp	esp
%si	si	si	esi
%di	di	di	edi

The registers referenced by the following register macros depend on whether you specify the `fixedparams` or `varparams` header control, as follows:

`%retoff` is the register that holds the offset portion of a pointer return value. The `%retoff` macro expands to `bx` (for `fixedparams`) or `ax` (for `varparams`) for ASM86 and ASM286, and `eax` for ASM386.

`%retsel` is the register that holds the selector portion of a pointer return value. The `%retsel` macro expands to `es` (for `fixedparams`) or `dx` (for `varparams`) for ASM86 and ASM286, and `edx` for ASM386.

7.4 Segment Macros

Each segment macro expands to the name of a segment. The memory model determines the segment names. The segment names conform exactly to those used by C and PL/M. You can use these names as instruction operands and in segmentation directives.

The segment macros correspond to the names of segments. These segment names, and what each macro expands to, are as follows:

`%cgroup` the segment to which the CS register points

`%code` the code segment name

`%const` the constant segment name

`%data` the data segment name

`%stack` the stack segment name

`%dgroup` the segment to which the DS register points

`%sgroup` the segment to which the SS register points

Tables 7-5 through 7-7 show the segment macro expansion by model for each assembler.

Table 7-5 ASM86 Segment Macro Expansion by Memory Model

Macro	Model	Sub-model	Expansion
%code	small or compact	RAM or ROM	CODE
	medium or large	RAM or ROM	*module-id*_CODE
%cgroup	small or compact	RAM or ROM	CGROUP
	medium or large	RAM or ROM	%code
%data	small, compact, or medium	RAM or ROM	DATA
	large	RAM or ROM	*module-id*_DATA
%dgroup	small, compact, or medium	RAM or ROM	DGROUP
	large	RAM or ROM	%data
%stack	small, compact, medium, or large	RAM or ROM	STACK
%sgroup	small or medium	RAM or ROM	DGROUP
	compact or large	RAM or ROM	STACK
%const	small, compact, medium, or large	ROM	%code
	small, compact, or medium	RAM	CONST
	large	RAM	%data

Table 7-6 ASM286 Segment Macro Expansion by Memory Model

Macro	Model	Sub-model	Expansion
%code	small or compact	RAM or ROM	CODE
	medium or large	RAM or ROM	*module-id*_CODE
%cgroup	small, compact, medium, or large	RAM or ROM	%code
%data	small, compact, or medium	RAM or ROM	DATA
	large	RAM or ROM	*module-id*_DATA
%dgroup	small, compact, medium, or large	RAM or ROM	%data
%stack	small or medium	RAM or ROM	DATA
	compact or large	RAM or ROM	STACK
%sgroup	small, compact, medium, or large	RAM or ROM	%stack
%const	small, compact, medium, or large	RAM	%data
	small, compact, medium, or large	ROM	%code

Table 7-7 ASM386 Segment Macro Expansion by Memory Model

Macro	Model	Sub-model	Expansion
%code	small, compact, or flat	RAM or ROM	CODE32
%cgroup	small, compact, or flat	RAM or ROM	%code
%data	small, compact, or flat	RAM or ROM	DATA
%dgroup	small, compact, or flat	RAM or ROM	%data
%stack	small or flat	RAM or ROM	DATA
	compact	RAM or ROM	STACK
%sgroup	small, compact, or flat	RAM or ROM	%stack
%const	small, compact, or flat	RAM	%data
	small, compact, or flat	ROM	%code

The following example uses `%data` to bracket static variable data:

```
%data segment
;assembler commands, e.g.,
  var dw Ø
%data ends
```

This example expands to the following, except under the large model:

```
DATA segment
;assembler commands, e.g.,
  var dw Ø
DATA ends
```

Under the large model, the example expands to the following:

```
module-id_DATA segment
;assembler commands, e.g.,
  var dw Ø
module-id_DATA ends
```

7.5 Type Macros

You can use a type macro wherever an ASM data type (such as `byte`, `word`, `dword`, etc.) can be used.

The type macros correspond to the data types of objects as follows:

`%fnc`	the type of a global function
`%fnc_ptr`	the size of a pointer to a function
`%ptr`	the size of a pointer to data
`%reg_size`	the size of a pointer
`%int`	the size of an integer
`%dint`	the size of a double integer

Tables 7-8 through 7-10 show the type macro expansion by model for each assembler.

Table 7-8 ASM86 Type Macro Expansion by Memory Model

Macro	Model	Sub-model	Expansion
%fnc	small or compact	RAM or ROM	near
	medium or large	RAM or ROM	far
%fnc_ptr	small or compact	RAM or ROM	word
	medium or large	RAM or ROM	dword
%ptr	small or medium	RAM	word
	small or medium	ROM	dword
	compact or large	RAM or ROM	dword
%reg_size	small, compact, medium, or large	RAM or ROM	word ptr
%int	small, compact, medium, or large	RAM or ROM	word
%dint	small, compact, medium, or large	RAM or ROM	dw

Table 7-9 ASM286 Type Macro Expansion by Memory Model

Macro	Model	Sub-model	Expansion
%fnc	small or compact	RAM or ROM	near
	medium or large	RAM or ROM	far
%fnc_ptr	small or compact	RAM or ROM	word
	medium or large	RAM or ROM	dword
%ptr	small or medium	RAM	word
	small or medium	ROM	dword
	compact, or large	RAM or ROM	dword
%reg_size	small, compact, medium, or large	RAM or ROM	word ptr
%int	small, compact, medium, or large	RAM or ROM	word
%dint	small, compact, medium, or large	RAM or ROM	dw

Table 7-10 ASM386 Type Macro Expansion by Memory Model

Macro	Model	Sub-model	Expansion
%fnc	small, compact, or flat	RAM or ROM	near
%fnc_ptr	small, compact, or flat	RAM or ROM	dword
%ptr	small or flat	RAM or ROM	dword
	compact	RAM or ROM	pword
%reg_size	small, compact, or flat	RAM or ROM	dword ptr
%int	small, compact, or flat	RAM or ROM	dword
%dint	small, compact, or flat	RAM or ROM	dd

7.6 Operation Macros

The operation macros are grouped in four different classes according to their function as follows:

External declaration macros	expand to declarations of external variables, constants, and functions.
Instruction macros	expand to code simulating instructions or the instructions themselves, depending on the instruction set used.

Conditional macros	expand to instructions that test or load data pointers. The expansion depends on whether data pointers have selectors.
Function definition macros	expand to the basic parts of a function definition.

7.6.1 External Declaration Macros

Use the external declaration macros as follows:

`%extern(`*`type`*`,` *`vname`*`)`	to declare an external variable where *`type`* is a valid assembler data type or a type macro, and *`vname`* is a variable name; can be used only outside all functions and segments.
`%extern_const(`*`type`*`,` *`cname`*`)`	to declare an external constant where *`type`* is a valid assembler data type or a type macro, and *`cname`* is a constant name; can be used only outside all functions and segments.
`%extern_fnc(`*`fname`*`)`	to declare an external function where *`fname`* is a function name; can be used only outside all functions and segments.

Tables 7-11 through 7-13 show the external definition macro expansion by model for each assembler. See Tables 7-5 through 7-7 for expansion of the `%const` segment macro.

Table 7-11 ASM86 Type Macro Expansion by Memory Model

Macro	Model	Sub-model	Expansion
%extern	small, compact, or medium	RAM or ROM	DATA segment extrn *vname:type* DATA ends
	large	RAM or ROM	extrn *vname:type*
%extern_const	small or compact medium	RAM or ROM RAM	%const segment extrn *%cname:%type* %const ends
	medium large	ROM RAM or ROM	extrn *%cname:%type*
%extern_fnc	small or compact	RAM or ROM	CODE segment extrn *fname*:near CODE ends
	medium or large	RAM or ROM	extrn *fname*:far

Table 7-12 ASM286 Type Macro Expansion by Memory Model

Macro	Model	Sub-model	Expansion
%extern	small, compact, or medium	RAM or ROM	DATA segment extern *vname:type* DATA ends
	large	RAM or ROM	extrn *vname:type*
%extern_const	small or compact medium RAM	RAM or ROM RAM	%const segment extrn *%cname:%type* %const ends
	medium large	ROM RAM or ROM	extrn *%cname:%type*
%extern_fnc	small or compact	RAM or ROM	CODE segment extrn *fname*:near CODE ends
	medium or large	RAM or ROM	extrn *fname*:far

Table 7-13 ASM386 External Declaration Macro Expansion by Memory Model

Macro	Model	Sub-model	Expansion
%extern	small, compact, or flat	RAM or ROM	DATA segment extrn *vname:type* DATA ends
%extern_const	small or compact flat	RAM RAM or ROM	CONST segment extrn *aconst:type* CONST ends
	small or compact	ROM	CODE32 segment extrn *aconst:type* CODE32 ends
%extern_fnc	small, compact, or flat	RAM or ROM	CODE32 segment extrn *fname*:near CODE32 ends

7.6.2 Instruction Macros

The instruction macros provide compatibility between 86 and higher processor instruction sets.

`%enter locals, level` (for 86 instructions) expands to code that simulates the `enter` instruction. The `level` argument is only a placeholder. The `locals` argument indicates the value to subtract from the sp register. Note that `%enter` uses spaces rather than parentheses to delimit the beginning and end of its parameter list.

`%enter` (for 186, 286, or Intel386 instructions) expands to the `enter` instruction.

`%leave` expands to code that simulates the `leave` instruction for the 86 instruction set, or the `leave` instruction for 186 and higher instruction sets.

`%pusha` expands to code that simulates the `pusha` instruction for the 86 instruction set, the `pusha` instruction for the 186 and 286 instruction sets, or the `pushad` instruction for the Intel386 instruction set.

`%popa`	expands to code that simulates the `popa` instruction for the 86 instruction set, the `popa` instruction for the 186 and 286 instruction sets, or the `popad` instruction for the Intel386 instruction set.
`%pushf`	expands to `pushf` for the 86, 186, and 286 instruction sets, or `pushfd` for the Intel386 instruction set.
`%movsx`	expands to `mov` for the 86, 186, and 286 instruction sets, or `movsx` for the Intel386 instruction set.
`%movzx`	expands to `mov` for the 86, 186, and 286 instruction sets, or `movzx` for the Intel386 instruction set.

7.6.3 Conditional Macros

The conditional macros select source text for assembly depending on whether data pointers have selectors (the far address format). The conditional macros expand as follows:

`%mov|`*`lsr`*	expands to `mov` if `%far_data` is not set, or to the register load instruction you specify as the *`lsr`* argument if `%far_data` is set. Use this macro as an instruction mnemonic for loading a data pointer. The *`lsr`* argument can be either `lds`, `les`, `lfs`, or `lgs`. Note that `%mov` uses a vertical bar (|) rather than parentheses to delimit its argument.
`%if_sel(`*`text`*`)`	expands only if data pointers have selectors. The *`text`* argument is the source text to be conditionally assembled. This macro is equivalent to the following: `%if (%far_data) then (`*`text`*`) fi`
`%if_nsel(`*`text`*`)`	expands only if data pointers do not have selectors. The *`text`* argument is source text to be conditionally assembled. This macro is equivalent to the following: `%if (not %far_data) then (`*`text`*`) fi`

7.6.4 Function Definition Macros

The following entries describe the function macros in detail in their order of use, as follows:

`%function`	open a function definition
`%param`	define a parameter name
`%param_flt`	define a floating-point parameter name
`%auto`	define a local automatic variable
`%prolog`	generate a function prolog
`%epilog`	generate a function epilog
`%ret`	generate a return instruction
`%endf`	close a function definition

Syntax

```
%function(fname)
```

Where:

fname is the name of the function to be opened.

Discussion

Use `%function` as the first statement in a function definition, to open the function definition.

For ASM86 or ASM286 small or compact model, the `%function` macro expands to the following:

```
CODE segment
  fname proc near
  public fname
```

For ASM86 or ASM286 medium or large model, the `%function` macro expands to the following:

```
module-id_CODE segment
  fname proc near
  public fname
```

For ASM386 all models, the `%function` macro expands to the following:

```
CODE32 segment
  fname proc near
  public fname
```

%param

Define a parameter name

Syntax

```
%param(type, pname)
```

Where:

type	is the data type of the parameter.
pname	is the name of the parameter, which is defined as a macro such that *%pname* expands to a valid reference to the parameter.

Discussion

Use `%param` to define a parameter name. Use `%param` only between `%function` and `%prolog`. When you define a parameter of data type *type*, the size of the parameter block increases by the number of bytes occupied by a parameter of data type *type*.

Regardless of whether the calling convention is fixed parameter list (FPL) or variable parameter list (VPL), parameters must be declared in the order that their corresponding arguments occur in the ASM function call expression.

Syntax

```
%param_flt(type, fpname)
```

Where:

type	is the data type of the parameter
fpname	is the name of the floating-point parameter, which is defined as a macro such that *%fpname* expands to a valid reference to the floating-point parameter.

Discussion

Use `%param_flt` to define a floating-point parameter name. Use `%param_flt` only between `%function` and `%prolog`.

If you specify the `varparams` header control, the effect of `%param_flt` is identical to that of `%param`. If you specify the `fixedparams` header control, `%param_flt` has no effect, since floating-point arguments are passed on the numeric coprocessor stack instead of on the processor stack. In general, you must handle floating-point arguments with a construct such as the following:

```
%if (not %fpl) then (
   fld %fpname              ; load the argument
) fi
.                           ; body of code
.
.
```

%auto

Define a local automatic variable

Syntax

```
%auto(type, mname)
```

Where:

type	can be any valid assembler data type or a type macro.
mname	is the name of the variable, which is defined as a macro such that *%mname* expands to a valid reference to the variable.

Discussion

Use `%auto` to define a local automatic variable. Use `%auto` only between `%function` and `%prolog`. When you define a local automatic variable of data type *type*, the size of the local area allocated by `%prolog` increases by the number of bytes occupied by a variable of data type *type*.

Syntax

```
%prolog(registers)
```

Where:

registers is a list of segment registers and general registers. However, the macro ignores all but the DS, ES, DI, and SI registers for ASM86 or ASM286; or DS, ES, EDI, and ESI registers for ASM386. Separate the register names with spaces.

Discussion

Use `%prolog` to generate a prolog function. Use `%prolog` only after `%function` and before any other instructions. Use `%prolog` whenever you use `%epilog`, `%param`, `%param_flt`, or `%auto`, and be sure to use `%prolog` after `%parm`, `%parm_flt`, and `%auto`. You must also use `%epilog` whenever you use `%prolog`.

Of the registers you list in the *registers* argument list, the prolog function pushes only those that the calling convention requires to be preserved. The prolog function performs the following tasks:

- pushes registers
- pushes BP for ASM86 or ASM286, or EBP for ASM386 (the base pointer register) and initializes it for use as a local frame pointer using the `ENTER` assembler instruction
- sets SP for ASM86 or ASM286, or ESP for ASM386 using the `ENTER` assembler instruction
- allocates space for automatic variables

In addition, for ASM86 and ASM286 large model, if the `%data` segment macro has been expanded, the prolog performs the following:

- pushes DS (the data segment register)
- loads the data segment address into DS

%epilog

Generate a function epilog

Syntax

```
%epilog
```

Discussion

Use `%epilog` to generate a function epilog. Use `%epilog` only immediately before a return instruction. The epilog deallocates space for automatic variables (allocated by the `%auto` function macro) and pops registers pushed by the `%prolog` function macro. The epilog also issues the `LEAVE` assembler instruction, thereby restoring the BP register for ASM86 or ASM286, or the EBP register for ASM386; and the SP register for ASM86 or ASM286, or the ESP register for ASM386.

Syntax

```
%ret
```

Discussion

Use `%ret` to generate a return instruction. The expansion of `%ret` depends on whether you specify the `varparams` or the `fixedparams` header control, as follows:

Under the `varparams` header control, `%ret` expands to the following:

```
ret
```

Under the `fixedparams` header control, `%ret` expands to the following:

```
ret paramsize
```

The *paramsize* is the sum of the sizes of all the parameters declared with `%param`. The *paramsize* must be an even value, since parameters are word-aligned.

%endf

Close a function definition

Syntax

```
%endf(fname)
```

Where:

fname is the name of the function to be closed.

Discussion

Use `%endf` as the last statement in a function definition to close the function definition. The `%endf` macro always expands to the following:

```
fname endp
```

7.6.5 Examples Using Assembler Macros

This section contains several examples that use flag macros, register macros, conditional macros, and function definition macros.

1. This example uses the following ASM source code:

```
mov al, byte ptr %if_sel(es:)[ebx]
%if_nsel(
push  ds
pop   es
)
rep stosb
```

For an ASM86 or ASM286 compact RAM, large RAM, or any ROM model, the expansion is as follows:

```
mov al, byte ptr es:[bx]
rep stosb
```

For an ASM86 or ASM286 small RAM or medium RAM model, the expansion is as follows:

```
mov al, byte ptr [bx]
push  ds
pop   es
rep stosb
```

For an ASM386 compact RAM, compact ROM, or small ROM model, the expansion is as follows:

```
mov al, byte ptr es:[ebx]
rep stosb
```

For an ASM386 small RAM or flat model, the expansion is as follows:

```
mov al, byte ptr [ebx]
push  ds
pop   es
rep stosb
```

2. This example shows assembler source text that assembles correctly for ASM86, ASM286, and ASM386. This example is not a working function, but demonstrates expansion under the different assemblers. The example uses the following DOS assembler invocations:

```
C:> asm86 ex2.asm %define (controls)()

C:> asm286 ex2.asm %define (controls)(asm286)

C:> asm386 ex2.asm %define (controls)(asm386)
```

Figure 7-2 shows the contents of the `ex2.asm` source file, and Figure 7-3 shows the expansion of the source file, and Figures 7-3 through 7-5 show the expanded code under the three assemblers for the default small memory model.

```
%define(controls)(module=ex2 stacksize=100 %controls)
$include(:include:util.ah)

; ext_var is an external variable
%extern(%int, ext_var)

; abc is a function that adds three values:  its input argument
; plus 10 plus the offset of an external variable
; and returns the sum represented as a pointer.
%function(abc)
%param(%int, p_word)
%auto(%int, a_word)
    %prolog()
    mov     %a_word, 10
    mov     %cx, %p_word
    add     %cx, %a_word
    add     %cx, offset ext_var
    mov     %retoff, %cx
    %if_sel(mov %cx, seg ext_var
            mov %retsel, %cx)
    %epilog
    %ret
%endf(abc)
    end
```

Figure 7-2 Assembler Source for Accessing the Address of an External Variable

```
$include(:include:util.ah)
; macros and defines for assembly language code
        name    ex2

        code    segment para public 'code'
        code    ends
         data   segment para public 'data'
         data   ends
        memory  segment para memory 'memory'
        memory  ends
        stack   segment para stack 'stack'
                        db      64H dup (?)
                ex2_tos label word
        stack   ends
                const   segment para public 'const'
                const   ends
                cgroup  group   code
                dgroup  group    data           , const
                assume  cs:cgroup
                assume  ds:dgroup
                assume  es:nothing
                assume  ss:dgroup
; ext_var is an external variable
                 data   segment
                        extrn    ext_var:word
                 data   ends
; abc is a function that adds three values:  its input
; plus 10 plus the offset of an external variable
; and returns the sum represented as a pointer.
        code    segment
        abc     proc    near
                public  abc
                push    bp
                mov     bp, sp
                        sub     sp,     02H
    mov     word ptr [bp - 02H], 10
    mov     cx, word ptr [bp + 04H + 02H - 02H]
    add     cx, word ptr [bp - 02H]
    add     cx, offset ext_var
    mov     bx, cx
                mov     sp, bp
                pop     bp
    ret 02H
        abc     endp
        code    ends
    end
```

Figure 7-3 ASM86 Expansion of Assembler Source for Accessing the Address of an External Variable

```
$include(:include:util.ah)
; macros and defines for assembly language code

        name    ex2
        code    segment er public
        code    ends
         data   segment rw public
         data   ends
        data    stackseg        64H
                assume  ds: data
                assume  es:nothing
                assume  ss:data
; ext_var is an external variable
                 data   segment
                        extrn    ext_var:word
                 data   ends
; abc is a function that adds three values:  its input
; plus 10 plus the offset of an external variable
; and returns the sum represented as a pointer.
        code    segment
        abc     proc    near
                public  abc
        enter   02H, 0
    mov     word ptr [bp - 02H], 10
    mov     cx, word ptr [bp + 04H + 02H - 02H]
    add     cx, word ptr [bp - 02H]
    add     cx, offset ext_var
    mov     bx, cx
        leave
    ret 02H
        abc     endp
        code    ends
    end
```

Figure 7-4 ASM286 Expansion of Assembler Source for Accessing the Address of an External Variable

```
$include(:include:util.ah)
: macros and defines for assembly language code
        name    ex2
        code32  segment er public
        code32  ends
         data   segment rw public
         data   ends
        data    stackseg        64H
                assume  ds: data
                assume  es:nothing
                assume  ss:data
: ext_var is an external variable
                 data   segment
                        extrn    ext_var:dword
                 data   ends
: abc is a function that adds three values:  its input argument
: plus 10 plus the offset of an external variable
: and returns the sum represented as a pointer.
        code32  segment
        abc     proc    near
                public  abc
        enter   04H, 0
    mov     dword ptr [ebp - 04H], 10
    mov     ecx, dword ptr [ebp + 08H + 04H - 04H]
    add     ecx, dword ptr [ebp - 04H]
    add     ecx, offset ext_var
    mov     eax, ecx
        leave
    ret 04H
        abc     endp
        code32  ends
    end
```

Figure 7-5 ASM386 Expansion of Assembler Source for Accessing the Address of an External Variable

3. This example is an implementation of the `strcmp` function for ASM. This example demonstrates special techniques for source code that can be compiled with different assemblers. Registers preceded with a percent sign (%) expand to the expanded register for ASM386. Different instructions are generated for the different processors in the `%if(%i386_asm) - then - else` statement. The assembler invocations are as follows:

```
C:> asm86 strcmp.asm %define (controls)()

C:> asm286 strcmp.asm %define (controls)(asm286)

C:> asm386 strcmp.asm %define (controls)(asm386)
```

Figure 7-6 shows the assembler source code, and Figures 7-7 through 7-9 show the expanded source code for ASM86, ASM286, and ASM386, respectively, for the default small memory model.

```
; strcmp - compare 2 strings
; Copyright (C) 1988 Intel Corporation, ALL RIGHTS RESERVED

%define(controls) (module=cq_strcmp fp %controls)

$include (:include:util.ah)

%function(strcmp)
%param(%ptr, str2)            ; Second string
%param(%ptr, str1)            ; First string

        %prolog(si di es ds)
;
;  Determine the length of the first string:
;
        cld                   ;
%if (not %far_data) then (
        mov di, ds            ; Ensure that extra segment selector
        mov es, di            ;  is the set correctly.
) fi
        %mov|les %di, %str1   ; Load the source address.
        xor %ax, %ax          ; Clear register (E)AX for the string
                              ;   scan instructions that follow.
        xor %cx, %cx          ; Set the count register to its
        dec %cx               ;   maximum value.
        repnz scasb           ; Scan to the end of str2 one
                              ;   byte at a time.
        not %cx               ; Maximum number of bytes to compare.
```

Figure 7-6 Assembler Source Code for strcmp Function

```
;
;  Compare the two strings:
;
        %mov|les %di, %str1     ; Load the address of string 1.
        %mov|lds %si, %str2     ; Load the address of string 2.

        mov %dx, %cx            ; Register (E)DX will contain the
                                ;   number of bytes left over after
                                ;   the word or dword compares.
%if(%i386_asm)
then (
        and %dx, 03H            ; Calculate the number of bytes that
                                ;   remain.
        shr %cx, 2              ; Divide the count by 4 for the number
                                ;   of dword transfers.
        repe cmpsd              ; Compare the dwords.
        jz left_over            ; If zero, then strings are equal
                                ;   so far.
        sub %si, 4              ; Set the string pointers to the dword
        sub %di, 4              ;   that contained the differences.
        mov %dx, 04H            ;
) else (
        and %dx, 01H            ; Calculate the number of bytes that
                                ;   remain.
        shr %cx, 1              ; Divide the count by 2 for the number
                                ;   of word transfers.
        repe cmpsw              ; Compare the words.
        jz left_over            ; If zero, then strings are equal
                                ;   so far.
        sub %si, 2              ; Set the string pointers to the word
        sub %di, 2              ;   that contained the differences.
        mov %dx, 02H            ;
) fi

left_over:
        mov %cx, %dx            ; Compare the left-over bytes (if any).
        repe cmpsb              ;
        jz alldone              ; Strings are equal, so return to the
                                ;   caller.
        sub %si, 1              ; Set the string pointers to the byte
        sub %di, 1              ;   that contained the differences.
different:
        xor %bx, %bx
        mov al, byte ptr %if_sel(ds:)[%si]
                                ; Subtract the character of
        mov bl, byte ptr %if_sel(es:)[%di]
                                ;   the destination string from
        sub %ax, %bx            ;   the source string.
alldone:
        %epilog
        %ret
%endf(strcmp)
        end
```

Figure 7-6 Assembler Source Code for strcmp Function (continued)

```
; strcmp - compare 2 strings
$include (:include:util.ah)
; macros and defines for assembly language code
        name    cq_strcmp
        code    segment para public 'code'
        code    ends
         data   segment para public 'data'
         data   ends
        memory  segment para memory 'memory'
        memory  ends
        stack   segment para stack 'stack'
                cq_strcmp_tos   label word
        stack   ends
                const   segment para public 'const'
                const   ends
                cgroup  group   code
                dgroup  group    data          , const         , stack, memory
                assume  cs:cgroup
                assume  ds:dgroup
                assume  es:nothing
                assume  ss:dgroup
        code    segment
        strcmp  proc    near
                public  strcmp
                        ; Second string
                        ; First string
                push    ds
                push    bp
                mov     bp, sp
;
;  Determine the length of the first string:
;
        cld                     ;
        mov di, ds              ; Ensure that extra segment selector
        mov es, di              ;  is the set correctly.
        mov di, word ptr [bp + 06H + 04H - 04H] ; Load the source address.
        xor ax, ax              ; Clear register (E)AX for the string
                                ;   scan instructions that follow.
        xor cx, cx              ; Set the count register to its
        dec cx                  ;   maximum value.
        repnz scasb             ; Scan to the end of str2 one
                                ;   byte at a time.
        not cx                  ; Maximum number of bytes to compare.
;
;  Compare the two strings:
;
      mov di, word ptr [bp + 06H + 04H - 04H] ; Load the address of string 1.
      mov si, word ptr [bp + 06H + 04H - 02H] ; Load the address of string 2.
       mov dx, cx               ; Register (E)DX will contain the
                                ;   number of bytes left over after
                                ;   the word or dword compares.
```

Figure 7-7 ASM86 Expansion of Assembler Source Code for strcmp Function

```
        and dx, 01H             ; Calculate the number of bytes that
                                ;   remain.
        shr cx, 1               ; Divide the count by 2 for the number
                                ;   of word transfers.
        repe cmpsw              ; Compare the words.
        jz left_over            ; If zero, then strings are equal
                                ;   so far.
        sub si, 2               ; Set the string pointers to the word
        sub di, 2               ;   that contained the differences.
        mov dx, 02H             ;
left_over:
        mov cx, dx              ; Compare the left-over bytes (if any).
        repe cmpsb              ;
        jz alldone              ; Strings are equal, so return to the
                                ;   caller.
        sub si, 1               ; Set the string pointers to the byte
        sub di, 1               ;   that contained the differences.
different:
        xor bx, bx
        mov al, byte ptr [si]   ; Subtract the character of
        mov bl, byte ptr [di]   ;   the destination string from
        sub ax, bx              ;   the source string.
alldone:
                mov     sp, bp
                pop     bp
                pop     ds
        ret 04H
        strcmp  endp
        code    ends
        end
```

Figure 7-7 ASM86 Expansion of Assembler Source Code for strcmp Function (continued)

```
; strcmp - compare 2 strings
$include (:include:util.ah)
; macros and defines for assembly language code
        name    cq_strcmp
        code    segment er public
        code    ends
         data   segment rw public
         data   ends
        data    stackseg        00H
                assume  ds: data
                assume  es:nothing
                assume  ss:data
        code    segment
        strcmp  proc    near
                public  strcmp
```

Figure 7-8 ASM286 Expansion of Assembler Source Code for strcmp Function

```
                        ; Second string
                        ; First string
                push    ds
        enter   00H, 0
;
;  Determine the length of the first string:
;
        cld                             ;
        mov di, ds                      ; Ensure that extra segment selector
        mov es, di                      ;  is the set correctly.
        mov di, word ptr [bp + 06H + 04H - 04H] ; Load the source address.
        xor ax, ax                      ; Clear register (E)AX for the string
                                        ;   scan instructions that follow.
        xor cx, cx                      ; Set the count register to its
        dec cx                          ;   maximum value.
        repnz scasb                     ; Scan to the end of str2 one
                                        ;   byte at a time.
        not cx                          ; Maximum number of bytes to compare.
;
;  Compare the two strings:
;
        mov di, word ptr [bp + 06H + 04H - 04H] ; Load the address of string
1.
        mov si, word ptr [bp + 06H + 04H - 02H] ; Load the address of string
2.
        mov dx, cx                      ; Register (E)DX will contain the
                                        ;   number of bytes left over after
                                        ;   the word or dword compares.
        and dx, 01H                     ; Calculate the number of bytes that
                                        ;   remain.
        shr cx, 1                       ; Divide the count by 2 for the number
                                        ;   of word transfers.
        repe cmpsw                      ; Compare the words.
        jz left_over                    ; If zero, then strings are equal
                                        ;   so far.
        sub si, 2                       ; Set the string pointers to the word
        sub di, 2                       ;   that contained the differences.
        mov dx, 02H                     ;
```

Figure 7-8 ASM286 Expansion of Assembler Source Code for strcmp Function (continued)

```
left_over:
        mov cx, dx              ; Compare the left-over bytes (if any).
        repe cmpsb              ;
        jz alldone              ; Strings are equal, so return to the
                                ;   caller.
        sub si, 1               ; Set the string pointers to the byte
        sub di, 1               ;   that contained the differences.
different:
        xor bx, bx
        mov al, byte ptr [si]   ; Subtract the character of
        mov bl, byte ptr [di]   ;   the destination string from
        sub ax, bx              ;   the source string.
alldone:
        leave
                pop     ds
        ret 04H
        strcmp  endp
        code    ends
        end
```

Figure 7-8 ASM286 Expansion of Assembler Source Code for strcmp Function (continued)

```
; strcmp - compare 2 strings
$include (:include:util.ah)
; macros and defines for assembly language code
        name    cq_strcmp
        code32  segment er public
        code32  ends
         data   segment rw public
         data   ends
        data    stackseg        ØØH
                assume  ds: data
                assume  es:nothing
                assume  ss:data
        code32  segment
        strcmp  proc    near
                public  strcmp
                        ; Second string
                        ; First string
                push    es
                push    ds
        enter   ØØH, Ø
;
;  Determine the length of the first string:
;
        cld                     ;
        mov di, ds              ; Ensure that extra segment selector
        mov es, di              ;  is the set correctly.
        mov edi, dword ptr [ebp + 1ØH + ØØH - ØØH]      ; Load the source
address.
        xor eax, eax            ; Clear register (E)AX for the string
                                ;   scan instructions that follow.
        xor ecx, ecx            ; Set the count register to its
        dec ecx                 ;   maximum value.
        repnz scasb             ; Scan to the end of str2 one
                                ;   byte at a time.
        not ecx                 ; Maximum number of bytes to compare.
;
;  Compare the two strings:
;
        mov edi, dword ptr [ebp + 1ØH + ØØH - ØØH]      ; Load the address of
string 1.
        mov esi, dword ptr [ebp + 1ØH + ØØH - Ø4H]      ; Load the address of
string 2.
        mov edx, ecx            ; Register (E)DX will contain the
                                ;   number of bytes left over after
                                ;   the word or dword compares.
        and edx, Ø3H            ; Calculate the number of bytes that
                                ;   remain.
        shr ecx, 2              ; Divide the count by 4 for the number
                                ;   of dword transfers.
        repe cmpsd              ; Compare the dwords.
        jz left_over            ; If zero, then strings are equal
                                ;   so far.
```

Figure 7-9 ASM386 Expansion of Assembler Source Code for strcmp Function

```
        sub esi, 4              ; Set the string pointers to the dword
        sub edi, 4              ;   that contained the differences.
        mov edx, 04H            ;
left_over:
        mov ecx, edx            ; Compare the left-over bytes (if any).
        repe cmpsb              ;
        jz alldone              ; Strings are equal, so return to the
                                ;   caller.
        sub esi, 1              ; Set the string pointers to the byte
        sub edi, 1              ;   that contained the differences.
different:
        xor ebx, ebx
        mov al, byte ptr [esi]  ; Subtract the character of
        mov bl, byte ptr [edi]  ;   the destination string from
        sub eax, ebx            ;   the source string.
alldone:
        leave
                pop     ds
                pop     es
        ret 08H
        strcmp  endp
        code32  ends
        end
```

Figure 7-9 ASM386 Expansion of Assembler Source Code for strcmp Function (continued)

4. This example is an implementation of the memcpy function for ASM386. The DOS assembler invocation is as follows:

```
C:> asm386 memcpy.asm %define (controls)(asm386)
```

Figure 7-10 shows the assembler source code for the memcpy function, and Figure 7-11 shows the expanded source code for the default small memory model.

```
; memcpy - copy a block of memory
; Copyright (C) 1988, 89 Intel Corporation, ALL RIGHTS RESERVED.

%define(controls) (module=cq_memcpy fp %controls)
$include(:include:util.ah)
%function(memcpy)
%param(%ptr, dst)
%param(%ptr, src)
%param(%int, count)
        %prolog(si di es ds)
        mov ecx, %count         ; Fetch the number of bytes to move.
        or ecx, ecx             ; If this number is zero, then
        jz done                 ;  there is no work to do.
;
; Set up necessary registers in order to use the 86 string instructions:
;
        cld                     ;
%if (not %far_data) then (
        mov dx, ds              ; Ensure that extra segment selector
        mov es, dx              ;  is the set correctly.
) fi
        %mov|lds esi, %src      ; Load the source address.
        %mov|les edi, %dst      ; Load the destination address.
;
; Move the block of memory:
;
        mov edx, ecx            ; Register (E)DX will contain the
                                ;  number of bytes left over after
                                ;  the word or dword transfers.
        shr ecx, 2              ; Divide the count by 4 for the number
                                ;  of dword transfers.
        rep movsd               ; Transfer the dwords.
        and edx, Ø3H            ; Calculate the number of bytes that
                                ;   remain.
        mov ecx, edx            ; Transfer the left-over bytes (if any).
        rep movsb               ;
done:
        mov %retoff, %reg_size %dst   ; return the destination address
        %if_sel(mov %retsel, %reg_size %dst + %int_size)
        %epilog
        %ret
%endf(memcpy)
        end
```

Figure 7-10 ASM386 Assembler Source for memcpy Function

```
; memcpy - copy a block of memory
$include(:include:util.ah)
; macros and defines for assembly language code
        name    cq_memcpy
        code32  segment er public
        code32  ends
         data   segment rw public
         data   ends
        data    stackseg        00H
                assume  ds: data
                assume  es:nothing
                assume  ss:data
        code32  segment
        memcpy  proc    near
                public  memcpy
                push    es
                push    ds
        enter   00H, 0
        mov ecx, dword ptr [ebp + 10H + 0CH - 0CH  ; Fetch the number of bytes
                                                   ;  to move.
        or ecx, ecx             ; If this number is zero, then
        jz done                 ;  there is no work to do.
;
; Set up necessary registers in order to use the 86 string instructions:
;
        cld                     ;
        mov dx, ds              ; Ensure that extra segment selector
        mov es, dx              ;  is the set correctly.
        mov esi, dword ptr [ebp + 10H + 0CH - 08H] ; Load the source address.
        mov edi, dword ptr [ebp + 10H + 0CH - 04H] ; Load the destination
                                                   ;  address.
;
; Move the block of memory:
;
        mov edx, ecx            ; Register (E)DX will contain the
                                ;  number of bytes left over after
                                ;  the word or dword transfers.
        shr ecx, 2              ; Divide the count by 4 for the number
                                ;  of dword transfers.
        rep movsd               ; Transfer the dwords.
        and edx, 03H            ; Calculate the number of bytes that
                                ;   remain.
        mov ecx, edx            ; Transfer the left-over bytes (if any).
        rep movsb               ;
done:
        mov eax, dword ptr dword ptr [ebp + 10H + 0CH - 04H]    ; return the
                                                                ; destination address
        leave
                pop     ds
                pop     es
        ret 0CH
        memcpy  endp
        code32  ends
        end
```

Figure 7-11 ASM386 Expansion of Assembler Source for memcpy Function

Contents

8

Function-calling Conventions

Function-calling Conventions 8

To interface functions in different languages, a programmer must know the calling convention, data types, and segmentation model used by the different translators. This chapter discusses calling conventions for interfacing iC-86/286/386 functions with functions written in other Intel programming languages. See Chapter 4 for information on segmentation memory models. See Chapter 10 for information on data types.

This chapter contains information on how iC-86/286/386 generates object code for a function call, and how the fixed parameter list and variable parameter list conventions differ.

See Chapter 10 for information on the following related topics:

- conformance to the ANSI C standard
- implementation-dependent compiler features
- data types and reserved words

A large application can consist of many separately compiled modules. The binding process combines the modules before execution to satisfy references to external symbols. Use Intel translators and binding tools to ensure compatibility with the segmentation model of the microprocessor.

A function-calling convention establishes rules and responsibilities for the following activities:

- passing arguments to the called function
- returning a value from the called function to the calling function
- saving registers
- cleaning up the stack

The compiler generates four sections of object code for a function call. These sections contain the code that handles the function-calling convention. Figure 8-1 shows these four sections of code. The sections are as follows:

setup	code in the calling function that the processor executes just before control transfers to the called function
prolog	code in the called function that the processor executes first when control has transferred from the calling function
epilog	code in the called function that the processor executes just before control returns to the calling function
cleanup	code in the calling function that the processor executes just after control returns from the called function

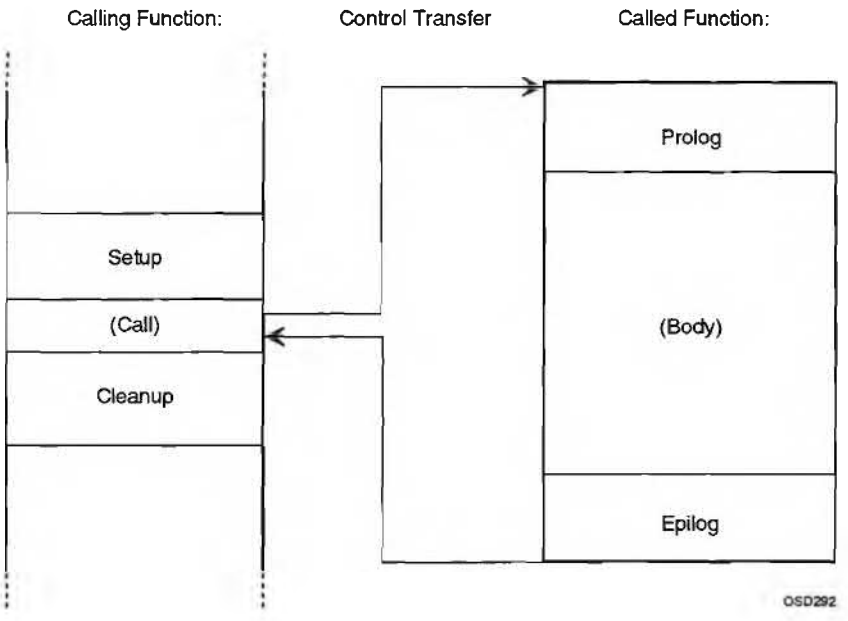

Figure 8-1 Four Sections of Code for a Function Call

The iC-86/286/386 compilers support two calling conventions: fixed parameter list (FPL) and variable parameter list (VPL). The FPL calling convention is the default for the iC-86/286/386 compilers and for most non-C compilers or translators. Ensure that the object code for the calling function and for the called function use the same convention. For iC-86/286/386, use the `fixedparams` control for the FPL convention and the `varparams` control for the VPL convention. See Chapter 3 for more information about these controls.

NOTE

The iC-86/286/386 compilers use the fixed parameter list (FPL) calling convention as its default. This feature produces more compact code. Intel C compilers for *n*86 processors before Version 4.1 use the variable parameter list (VPL) calling convention. If the calling function and the called function do not use the same calling convention, the result is unpredictable.

8.1 Passing Arguments

A calling function passes some or all of its arguments to the called function on the processor stack. The following points differ in calling conventions:

- position that arguments occupy on the stack, or order in which arguments are pushed onto the stack
- whether the calling function passes an argument by value (the actual value of the argument appears on the stack) or passes an argument by reference (a pointer to the argument appears on the stack)
- the format of pass-by-value arguments on the stack

The iC-86/286/386 compilers always use pass-by-reference for passing arrays and pass-by-value for other objects. The calling function's setup code pushes arguments onto the stack.

8.1.1 FPL Argument Passing

In the FPL convention, the calling function pushes all non-floating-point arguments onto the processor stack, and the first seven (left-to-right) floating-point arguments onto the numeric coprocessor (or numeric coprocessor emulator) stack. The calling function pushes all remaining floating-point arguments onto the processor stack.

The FPL convention pushes the leftmost argument in the function call first and the rightmost argument last. Therefore, the first argument in the list occupies the highest memory location of all the arguments on the stack for this function call, and the last argument in the list is on the top of the stack.

Aggregate objects occupy memory on the stack in the same way that they exist in the data segment: bytes match from low-order memory to high-order memory.

Each argument on the processor stack occupies a multiple of four bytes. If the size of the argument is less than four bytes, the compiler pads the argument to four bytes with undefined bits. The compiler pads aggregate arguments to a multiple of four bytes with undefined bits.

The floating-point arguments on the numeric coprocessor stack occupy 80 bits each (extended precision). In conformance to the ANSI C standard, the parameter prototype declaration determines the size of any floating-point arguments on the processor stack. In the absence of a prototype, or if the parameter is the eight or subsequent floating-point value, the calling function pushes floating-point arguments in `double` format (64 bits).

When the calling function expects a structure or union as a return value, the calling function pushes last an argument that is an address where the called function places the structure or union.

NOTE

A non-prototyped FPL function risks using incorrect offsets for all parameters following the eighth floating-point parameter if the eighth or subsequent floating-point parameter is declared within the function as `float` instead of `double`, as follows:

1. Under the FPL calling convention, the first seven floating-point arguments are passed in the numeric coprocessor registers, and all subsequent floating-point arguments are passed on the CPU stack.
2. In the absence of a prototype for the called function, the calling function always promotes an argument of type `float` to type `double` before passing the argument on the CPU stack to the called function.
3. If the called function declares the eighth or subsequent floating-point parameter as type `float` (instead of type `double`, as passed), the called function uses incorrect offsets to access the ninth and subsequent parameters, and the stack is not adjusted correctly upon return to the calling function.

To avoid such errors, always provide prototypes for all FPL functions that include floating-point parameters.

8.1.2 VPL Argument Passing

In the VPL convention, the calling function pushes all arguments, including floating-point arguments, onto the processor stack.

The VPL convention pushes the rightmost argument in the function call first and the leftmost argument last. Therefore, the last argument in the list occupies the highest memory location of all the arguments on the stack for this function call, and the first argument in the list is on the top of the stack.

Aggregate objects occupy memory on the stack in the same way that they exist in the data segment: bytes match from low-order memory to high-order memory.

Each argument on the processor stack occupies a multiple of four bytes. If the size of the argument is less than four bytes, the compiler zero-extends or sign-extends to four bytes depending on the argument's data type. The compiler pads aggregate arguments to a multiple of four bytes with undefined bytes.

In conformance to the ANSI C standard, the parameter prototype declaration determines the size of a floating-point argument on the processor stack. In the absence of a prototype, or if the parameter is beyond the ellipsis, the calling function pushes a floating-point argument in `double` format (64 bits).

When the calling function expects a structure or union as a return value, the calling function pushes last an argument that is an address where the called function places the structure or union.

NOTE

Variables declared with the `register` storage class are candidates for storage in registers only under the VPL calling convention. The `register` storage class is ignored under the FPL calling convention. See *C: A Reference Manual*, listed in Chapter 1, for more information on the `register` storage class.

8.2 Returning a Value

Both the FPL and VPL calling conventions return scalar values in a register and a floating-point value on the top of the numeric coprocessor stack.

The called function copies a returned union or structure starting at the memory location pointed to by the last argument on the stack. The called function also loads the address of the structure or union into a register, as if returning a pointer to the return object.

Loading the register and copying a returned union or structure occurs in the called function's epilog code.

In iC-86 and iC-286, FPL and VPL conventions use different registers to return different scalar objects. Tables 8-1 and 8-2 show the registers used for different scalar objects for iC-86/286 and iC-386, respectively.

Table 8-1 iC-86 and iC-286 FPL and VPL Return Register Use

Data Type	FPL	VPL
8-bit result	AL	AL
16-bit result	AX	AX
32-bit result	DX:AX	DX:AX
near (short) pointer	BX	AX
far (long) pointer	ES:BX	DX:AX
real	top of coprocessor or emulator stack	top of coprocessor or emulator stack

Table 8-2 iC-386 FPL and VPL Return Register Use

Data Type	FPL or VPL
8-bit result	AL
16-bit result	AX
32-bit result	EAX
64-bit result	EDX:EAX
near (short) pointer	EAX
far (long) pointer	EDX:EAX
real	top of coprocessor or emulator stack

8.3 Saving and Restoring Registers

The FPL and VPL calling conventions preserve different sets of registers. The VPL calling convention preserves the (E)DI, (E)SI, and (E)BX registers. Tables 8-3 and 8-4 show the register preservation scheme of iC-86/286 and iC-386, respectively, for the FPL and VPL conventions.

In the FPL convention, if the calling function uses register variables, the calling function is responsible for saving their values in the setup code. The balance of register preservation occurs in the called function's prolog code.

Table 8-3 iC-86 and iC-286 FPL and VPL Register Preservation

Reg.	FPL Preserved	FPL not Preserved	VPL Preserved	VPL not Preserved
AX		X		X
BX		X		X
CX		X		X
DX		X		X
SP	X		X	
BP	X		X	
DI		X	X	
SI		X	X	
CS	X		X	
DS	X		X	
SS	X		X	
ES		X		X

Table 8-4 iC-386 FPL and VPL Register Preservation

Reg.	FPL Preserved	FPL not Preserved	VPL Preserved	VPL not Preserved
EAX		X		X
EBX		X	X	
ECX		X		X
EDX		X		X
ESP	X		X	
EBP	X		X	
EDI		X	X	
ESI		X	X	
CS	X		X	
DS	X		X	
SS	X		X	
ES	X		X	
FS		X		X
GS		X		X

8.4 Cleaning Up the Stack

In the FPL calling convention, the called function pops all the arguments off the processor stack in its epilog before it returns control to the calling function.

In the VPL calling convention, the calling function pops all the arguments off the processor stack in its cleanup code after the called function returns control.

In both conventions, the called function's prolog code pops any floating-point arguments off the numeric coprocessor stack and saves them as local variables. If the called function returns a floating-point value, it is left on the top of the numerics coprocessor stack and is overwritten by the next floating-point operand.

Contents

9

Subsystems

9 Subsystems

This chapter tells you how to use subsystems to create extended segmentation models and contains the following topics:

- when to use subsystems
- how subsystems combine to form extended segmentation models
- syntax for defining subsystems
- example definitions

Segmentation is the term for the division of code, data, and stacks in the 86, 286, Intel386™, and Intel486™ architectures. The small, compact, medium, large, and flat segmentation memory models described in Chapter 4 are the standard ways that iC-86/286/386 creates code, data, and stack segments. When your program contains large amounts of data or code, the standard segmentation memory models do not offer a way to group code and data references and to structure your program into more segments to take advantage of segmentation protection mechanisms.

Subsystems extend the efficiency and protection of the small, compact, and large segmentation memory models described in Chapter 4. A subsystem is a collection of program modules that uses the same standard model of segmentation. If you use only the standard segmentation controls (and not the `subsys` control) to compile your program modules, then your program consists of one subsystem with all modules using the same model of segmentation. The term "extended segmentation model" refers to the memory model used by any program that consists of more than one subsystem.

Extended segmentation models offer the following advantages:

- When a program contains multiple subsystems, each subsystem can use a different segmentation model.
- Each program subsystem can execute at a different protection level.
- Each subsystem enjoys the segmentation protection mechanisms of the processor architecture, such as restricted entry points and protection from segment overruns.

The iC-86 and iC-286 compilers support three extended segmentation models: the small, compact, and large models, and the iC-386 compiler supports two extended segmentation models: the small model and the compact model. A program can contain subsystems in the same or different models.

A subsystem uses either the RAM or the ROM submodel, with constants in the data segment or code segment, respectively. A program can contain subsystems that use different submodels.

To compile a module that is part of a subsystem, place the definitions for the subsystems in a special text file and use the `subsys` compiler control in the invocation or in a `#pragma` preprocessor directive to include the special file in each compilation. If you use `subsys` in a `#pragma` directive, the directive must precede any data definitions or executable statements.

9.1 Dividing a Program into Subsystems

Using subsystems is an efficient way to structure programs that have large amounts of data or code. For example, consider a program consisting of 10 modules, `mod1` through `mod10`. Modules `mod1` through `mod3` deal with input and initial processing. Modules `mod4` through `mod8` do the main data processing. Modules `mod9` and `mod10` output the data. Figure 9-1 illustrates the program structure and data flow.

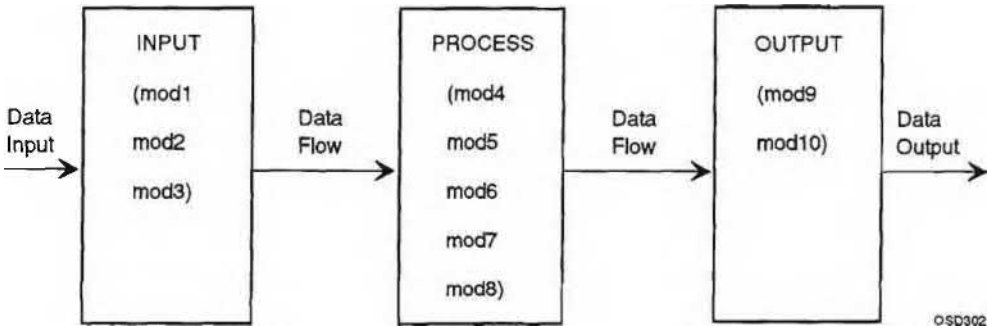

Figure 9-1 Subsystems Example Program Structure

Under the compact segmentation memory model described in Chapter 4, the binder combines the segments for this program into one code segment containing all the code from `mod1` through `mod10`, one data segment containing all the data from `mod1` through `mod10`, and one stack segment, as shown in Figure 9-2.

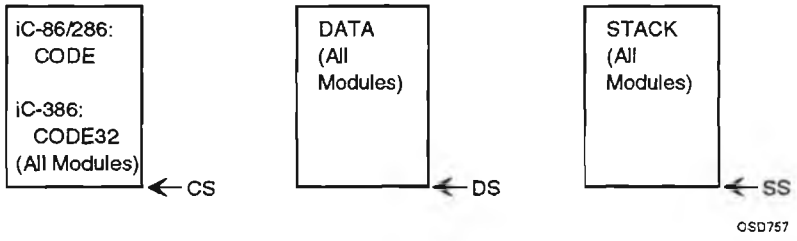

Figure 9-2 Subsystems Example Program in Regular Compact Segmentation Memory Model

Suppose the program is restructured using an extended segmentation model composed of three small-model subsystems. Each subsystem is given a name indicating its function, as follows:

Subsystem Name	Modules in Subsystem
`SUBINPUT`	`mod1` through `mod3`
`SUBPROCESS`	`mod4` through `mod8`
`SUBOUTPUT`	`mod9` and `mod10`

In a program composed of small-model subsystems, modules are combined by the linker or binder so that:

- Each subsystem has one code segment.
- All subsystems share one data-stack segment.

Figure 9-3 shows the segments for the example if the modules are grouped into three small-model subsystems.

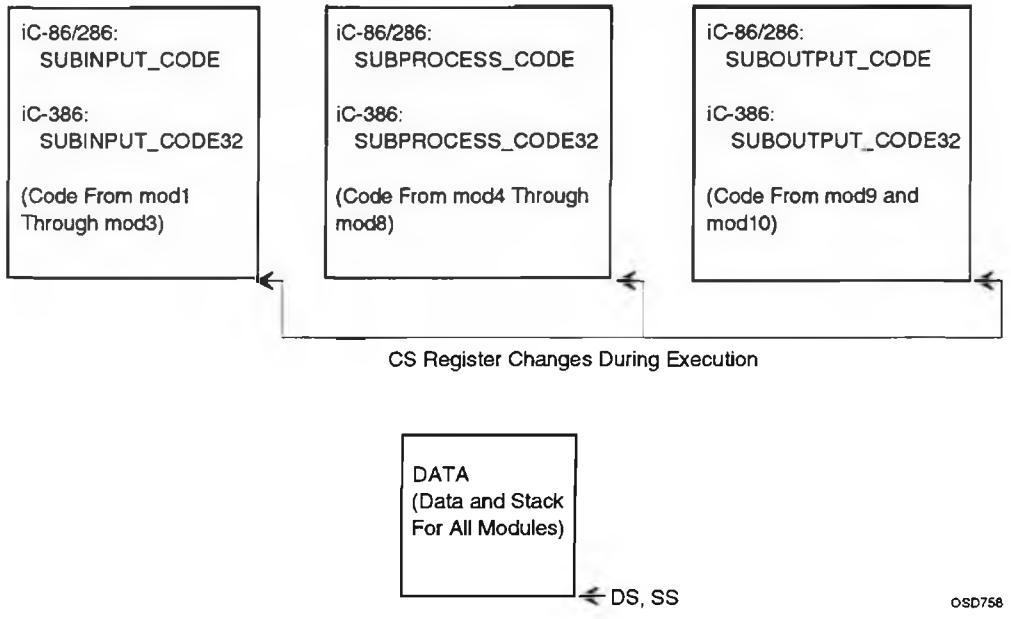

Figure 9-3 Subsystems Example Program Using Small-model Subsystems

The program is efficient because most of the calls and references are near and take place within a subsystem, and each subsystem enjoys segmentation protection. Far calls are needed only between the subsystems. Far data references are needed only if data is referenced between subsystems, or if constants are in code. The compiler implicitly modifies the declarations of symbols referred to by other subsystems by inserting the `far` keyword in the appropriate place in the declarations even if the `extend` control is not in effect.

One further refinement might be to create subsystems that use different segmentation models. Suppose that the SUBPROCESS subsystem is stack-intensive, and you wish to separate the SUBPROCESS data from the processor stack, leaving more space in the data-stack segment. You can place the SUBPROCESS data into a separate segment by using a compact-model subsystem for SUBPROCESS and small-model subsystems for SUBINPUT and SUBOUTPUT. Figure 9-4 shows the segments for this example if the modules are grouped into two small-model subsystems and one compact-model subsystem.

NOTES

All code in small-model subsystems assumes that the DS and SS registers contain identical selectors, which occurs only if the function where program execution begins is in a small-model subsystem. Therefore, if a small-model subsystem is mixed with one or more compact-model subsystems, the `main()` function, where program execution begins, must be in a small-model subsystem, ensuring proper access to the processor stack from every subsystem.

The stack segment resulting from any compact-model subsystems is not used when small-model and compact-model subsystems are mixed in a program.

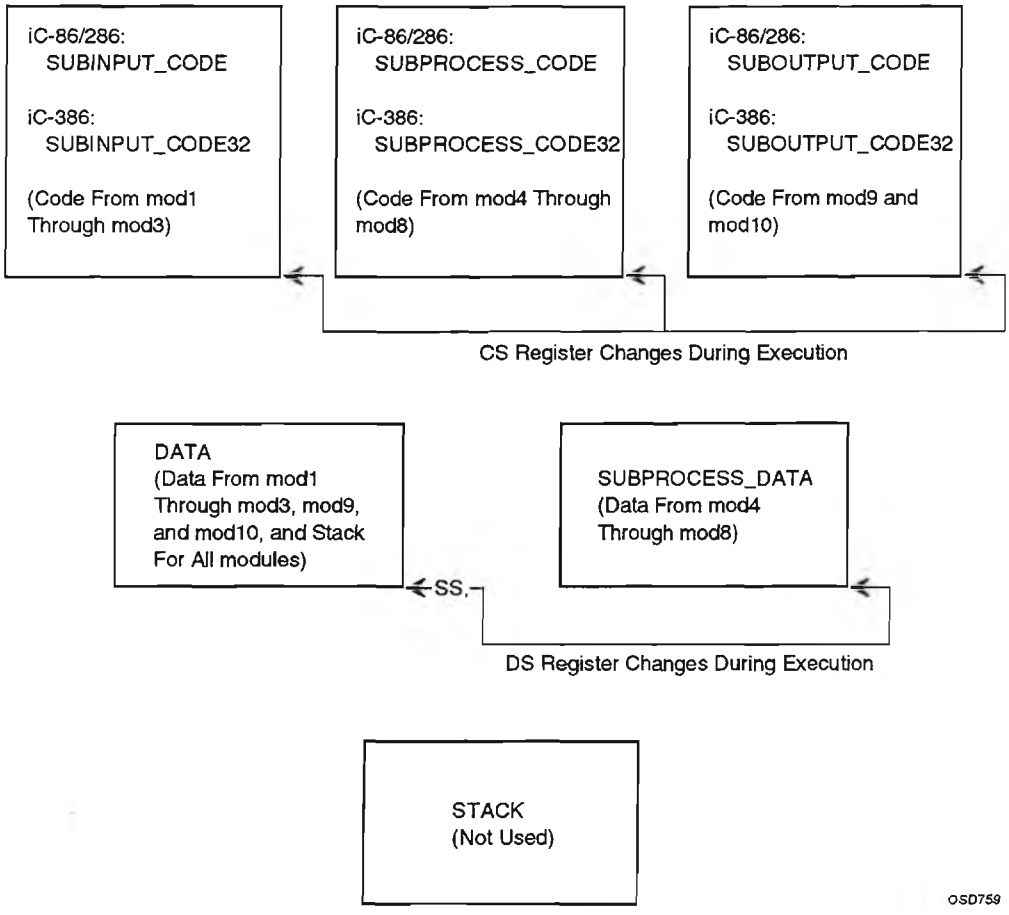

Figure 9-4 Subsystems Example Program Using Two Small-model Subsystems and One Compact-model Subsystem

You do not increase efficiency or protection by merely dividing a program into subsystems. If all the even-numbered modules are placed in one subsystem, for instance, and all the odd-numbered ones into another, the program becomes less efficient due to the greater number of far calls and far data references between subsystems. A program is most efficient and takes best advantage of segmentation protection when you place data accessed by a collection of modules and the functions that refer to that data into a

subsystem. Data and code in another subsystem are protected and can be accessed only if explicitly declared in the subsystem definition. All code references within a subsystem are near calls. If you choose the member modules for your subsystem carefully, you ensure few far calls.

9.2 Segment Combination in Subsystems

Chapter 4 describes the way that the binder combines segments under the standard segmentation memory models. To understand the combination of segments for programs structured with subsystems, you must understand the distinction between compiling modules with iC-86/286/386 and combining modules into a program with LINK86, BND286, or BND386.

The compiler compiles only one module at a time. During these separate compilations, the compiler generates many code, data, and stack segment definitions. Then, the linker or binder creates an executable program by combining the segments that have compatible attributes. See Chapter 4 for more information on the segment attributes that the binder uses, such as like names.

Both the standard segmentation controls (`small`, `compact`, `medium`, `large`, and `flat`) and the extended segmentation control (`subsys`) determine the way segments are combined by controlling the way segments are named.

9.2.1 Small-model Subsystems

Recall that the linker or binder combines compiler-generated segments that have the same name, and compatible characteristics. A linked small-model subsystem named `SMALLSUB` contains two segments: `SMALLSUB_CODE` for iC-86/286 or `SMALLSUB_CODE32` for iC-386, and `DATA`. When code in the subsystem is executing, the CS register contains the selector for `SMALLSUB_CODE` or `SMALLSUB_CODE32`, and the DS and SS registers contain the selector for `DATA`.

Tables 9-1 through 9-3 show the compiler segment definitions for a module compiled with the `subsys` control and a definition for a small-model subsystem. When you specify `-const in code-` in the subsystem definition, the compiler places the constants in the module's code segment, which is like

specifying the `rom` control when you are not using subsystems. When you specify `-const in data-` in the subsystem definition, the compiler places the constants in the module's data-stack segment, which is like specifying the `ram` control when you are not using subsystems. If the subsystem definition contains a *`subsystem-id`*, making a closed subsystem as defined in Section 9.3.1, the identifier and an underscore (_) prefix the `CODE` or `CODE32` segment name.

For iC-86, the DGROUP compiler segments link together to become `DATA`, and the CGROUP compiler segments link together to become `CODE`.

Table 9-1 iC-86 Segment Definitions for Small-model Subsystems

Description	Name	Combine-type	Group
code segment	[*subsystem-id*_]CODE	concatenate	CGROUP
data segment	DATA	concatenate	DGROUP
stack segment	STACK	overlay additively	DGROUP
constant segment (only with -const in data-)	CONST	concatenate	DGROUP

Table 9-2 iC-286 Segment Definitions for Small-model Subsystems

Description	Name	Combine-type	Access
code segment	[*subsystem-id*_]CODE	normal	execute-read
data segment	DATA	normal	read-write
stack segment	DATA	stack	read-write

Table 9-3 iC-386 Segment Definitions for Small-model Subsystems

Description	Name	Combine-type	Access
code segment	[*subsystem-id*_]CODE32	normal	execute-read
data segment	DATA	normal	read-write
stack segment	DATA	stack	read-write

The linker or binder combines segments with the same name when linking the modules for the program. Thus, each small-model subsystem contains its

own code segment up to 64 kilobytes for iC-86/286 or 4 gigabytes for iC-386. All data-stack segments from all small-model subsystems are combined into one data-stack segment up to 64 kilobytes for iC-86/286 or 4 gigabytes for iC-386.

Function pointers are near by default (the offset-only address format). If you specify `-const in data-` in the subsystem definition, all variables, temporary variables, and constants fall within one segment `DATA`, and data pointers are near by default. If you specify `-const in code-`, which places constants in the code segment, data pointers are far (the segment-eselector-and-offset address format). See Section 4.3 for an explanation of near and far address formats.

Keep the following limitations in mind when using a small-model subsystem:

The program must begin execution in a small-model subsystem.	All code in small-model subsystems assumes that the DS and SS registers contain identical selectors, which only occurs if the function where program execution begins is in a small-model subsystem. Therefore, if a small-model subsystem is mixed with one or more compact-model subsystems, the `main()` function, where program execution begins, must be in a small-model subsystem, ensuring proper access to the processor stack from every subsystem.
The `far` keyword is required when mixing a small-model RAM subsystem with any other model subsystem.	The default near pointers generated under the small model limit small-model RAM subsystems. A function in a small-model RAM subsystem can accept a pointer argument from a subsystem under another model, such as small-model ROM or any compact- or large-model subsystem, only if the pointer parameter is declared with the `far` keyword. A small-model RAM subsystem must also use the `far` keyword in a prototype, declaration, or cast to pass a data pointer to a function in a subsystem that is not small-model RAM.

Small-model subsystems offer limited data protection.	Because small-model subsystems contain one data-stack segment, data is not protected from stack overruns.

9.2.2 Compact-model Subsystems

Recall that the linker or binder combines compiler-generated segments that have the same name, and compatible characteristics. A linked compact-model subsystem named `COMPSUB` contains three segments: `COMPSUB_CODE` for iC-86/286 or `COMPSUB_CODE32` for iC-386, `COMPSUB_DATA`, and `STACK`. When code in the subsystem is executing, the CS register contains the selector for `COMPSUB_CODE` or `COMPSUB_CODE32`, the DS register contains the selector for `COMPSUB_DATA`, and the SS register contains the selector for `STACK`.

Tables 9-4 through 9-6 show the compiler segment definitions for a module compiled with the `subsys` control and a definition for a compact-model subsystem. When you specify `-const in code-` in the subsystem definition, the compiler places the constants in the module's code segment, which is like specifying the `rom` control when you are not using subsystems. When you specify `-const in data-` in the subsystem definition, the compiler places the constants in the module's data segment, which is like specifying the `ram` control when you are not using subsystems. If the subsystem definition contains a *`subsystem-id`*, making a closed subsystem as defined in Section 9.3.1, the identifier and an underscore (_) prefix the `CODE` or `CODE32` and `DATA` segment names.

For iC-86, the DGROUP compiler segments link together to become `DATA`, the CGROUP compiler segments link together to become `CODE`, and the stack compiler segments link together to become `STACK`.

Table 9-4 iC-86 Segment Definitions for Compact-model Subsystems

Description	Name	Combine-type	Group
code segment	[*subsystem-id*_]CODE	concatenate	CGROUP
data segment	[*subsystem-id*_]DATA	concatenate	DGROUP
stack segment	STACK	overlay additively	
constant segment (only with -const in data-)	CONST	concatenate	DGROUP

Table 9-5 iC-286 Segment Definitions for Compact-model Subsystems

Description	Name	Combine-type	Access
code segment	[*subsystem-id*_]CODE	normal	execute-read
data segment	[*subsystem-id*_]DATA	normal	read-write
stack segment	STACK	stack	read-write

Table 9-6 iC-386 Segment Definitions for Compact-model Subsystems

Description	Name	Combine-type	Access
code segment	[*subsystem-id*_]CODE32	normal	execute-read
data segment	[*subsystem-id*_]DATA	normal	read-write
stack segment	STACK	stack	read-write

The linker or binder combines segments with the same name when linking the modules for the program. Thus, each compact-model subsystem contains its own code segment up to 64 kilobytes for iC-86/286 or 4 gigabytes for iC-386 and its own data segment up to 64 kilobytes for iC-86/286 or 4 gigabytes for iC-386. All stack segments from all compact-model subsystems are combined into one stack segment up to 64 kilobytes for iC-86/286 or 4 gigabytes for iC-386.

Function pointers are near by default (the offset-only address format). Data pointers are far by default (the segment-selector-and-offset format). Compact-model subsystems can pass pointer arguments between compact-model RAM, compact-model ROM, small-model ROM, and large-model modules without specifying the `far` keyword because data pointers are always far pointers. See Section 4.3 for an explanation of near and far address formats.

If a function in a compact-model subsystem accepts a pointer parameter exported from a small-model RAM subsystem, the small-model RAM subsystem must explicitly use the `far` keyword in a prototype, declaration, or cast to pass the data pointer.

9.2.3 Large-model Subsystems (iC-86 and iC-286 Only)

Modules in a large-model subsystem are equivalent to the same modules compiled with the `large` segmentation control, because the segments are named identically. Using all large-model subsystems has the same effect as using the `large` segmentation control without subsystems. However, using a mixture of large-model and other subsystems may be useful. See Section 4.2.4 for information on segment names and characteristics under the `large` segmentation control.

9.2.4 Efficient Data and Code References

The most efficient and compact code contains few far calls and few far data references. A call from any subsystem to another subsystem is always a far call. Only small-model RAM subsystems have data, constants, and stack in the same segment. Therefore, a data reference between a small-model RAM subsystem and another small-model RAM subsystem is a near reference. Data references to and from other model subsystems are far references.

The `near` and `far` keywords are type qualifiers that allow programs to override the default address size generated for a data or code reference. You must use the `extend` control when you compile programs that use the `near` and `far` keywords. Table 9-7 shows the default address sizes for code and data references in all subsystem models. See Section 4.3 for information on how to use the `near` and `far` keywords. See Chapter 3 for a description of the `extend` control.

Table 9-7 Subsystems and Default Address Sizes

Subsystem Model	Code Reference	Data Reference
small RAM	offset	offset
small ROM	offset	selector and offset
compact RAM	offset	selector and offset
compact ROM	offset	selector and offset
large RAM	selector and offset	selector and offset
large ROM	selector and offset	selector and offset

9.3 Creating Subsystem Definitions

A text file contains the definition for a subsystem. To compile a module as part of a subsystem, use the `subsys` compiler control in the invocation or in a `#pragma` preprocessor directive to include the definition file in the compilation. See Chapter 3 for a description of the `subsys` control. The `subsys` control is a primary control and must appear in the invocation line or in a `#pragma` preprocessor directive before the first line of data declaration or executable source text. A `#pragma` preprocessor directive containing the `modulename` control cannot follow any `#pragma` containing the `subsys` control.

NOTE

When a module from a small-model subsystem calls a function that is exported from a compact-model or large-model subsystem, the linker or binder does not automatically compute the stack requirement because the segments containing them have different names. To get the proper stack size, use the `segsize` control during linking or binding to increase the size of the data-stack segment by the sum of the stack requirements for both the small-model subsystem and the compact-model or large-model subsystem.

9.3.1 Open and Closed Subsystems

The subsystems that make up an iC-86/286/386 program can be either open or closed. The definition for a closed subsystem must list every program module within it. An open subsystem contains all modules not specified as part of another subsystem by default. A program can use open and closed subsystems, according to one of the following options:

- All subsystems in a program are closed.
- A program can have many closed subsystems and a single open subsystem.
- By default, a program has one open subsystem and no closed subsystems.

The syntax for a subsystem definition is shown in Section 9.3.2. For a closed subsystem, the compiler must know the name of the subsystem, the *`subsystem-id`*, and the modules belonging to it, the `has` list. For an open subsystem, the definition cannot have a *`subsystem-id`*. By omitting the subsystem name in one subsystem definition, you automatically create an open subsystem that contains all modules not claimed in another subsystem's `has` list. You can add modules not named in a closed subsystem definition to your program at any time, and the modules automatically become part of this open subsystem without changing any subsystem definition.

9.3.2 Syntax

Defining subsystems tells the compiler the following:

- the memory model that each subsystem uses
- whether to place the constants in the code segment or data segment for the subsystem
- the modules that belong to each subsystem
- the functions and data that are accessible from outside the subsystem

Making all functions and data available to all subsystems defeats the purpose of subsystems and decreases the efficiency of the program. For example, if a a subsystem definition declares a function to be accessible from another subsystem, the function is a far function, making all calls far calls, even if the function actually is never accessed from outside its subsystem.

A function or data that is accessible to another subsystem must have external linkage. In the C programming language, public and external symbols are functions or variables with external linkage. The linker or binder resolves the addresses for such symbols. The following definitions identify public and external symbols. See *C: A Reference Manual*, listed in Chapter 1, for more information on external linkage.

A public variable	is defined at the file level, not within a function, and without the `static` keyword. By default, a public variable is globally accessible within its subsystem. Other subsystems can refer to a public variable if the definition for the containing subsystem exports the variable.
A public function	is defined without the `static` keyword. The public definition includes the function code. By default, a public function is globally accessible within its subsystem. Other subsystems can call a public function if the definition for the containing subsystem exports the function name.
An external variable	is declared with the `extern` keyword. The external declaration refers to a corresponding public definition for the variable in another module within the same or another subsystem.

An external function is declared with the `extern` keyword. The external declaration can take on the form of a function prototype. The external declaration does not contain the function code but refers to a corresponding public definition for the function in another module within the same or another subsystem.

Each subsystem in a program must have a subsystem definition. In the following subsystem definition syntax, items in brackets ([ ]) are optional, items in braces ({ }) are a list from which to choose, and [; ...] indicates you can choose another item from the previous list, separating adjacent list items with a semicolon (;). Enter the dollar sign ($) and parentheses (( )) as shown:

```
$ model ([subsystem-id] [submodel] [ { has module-list      } [; ...] ] )
                                   { exports public-list }
```

Where:

model specifies the segmentation model for the subsystem. Use `small`, `compact`, or `large`. Case is not significant in the `small`, `compact` and `large` keywords. All modules in a subsystem must be compiled with the same model of segmentation.

subsystem-id specifies a unique name for a closed subsystem. This name can be up to 31 characters long and must not conflict with any module name. The compiler forces this identifier to all uppercase. The identifier can contain dollar signs ($), which the compiler ignores.

submodel specifies the submodel, which defines the placement of constants. Use `-const in code-` for placing constants in the code segment or `-const in data-` (default) for placing constants in the data segment. Case is not significant in the `-const in code-` and `-const in data-` keywords. All modules in one subsystem are compiled with the same submodel.

`has` *`module-list`*

specifies the modules that make up the subsystem. Case is not significant in the `has` keyword. A `has` specification is required for a closed subsystem, and the *`module-list`* must contain all the closed subsystem modules. A `has` specification is optional for an open subsystem, and the *`module-list`* does not have to contain all of the open subsystem modules. Identifiers in the *`module-list`* can be up to 31 characters long and are forced to all uppercase.

`has` *`module-list`*

Each identifier in the *`module-list`* must match a module name to be included in the subsystem. A module name is the module's source file name without extension, unless specified differently by the `modulename` control. A particular module name can appear in only one *`module-list`* (i.e., a module can belong to only one subsystem). Any module whose name does not appear in a *`module-list`* becomes part of the open subsystem. Module names can appear in any order in the *`module-list`*.

`exports` *`public-list`*

lists the functions and variables exported by the subsystem, which are the functions and variables that the subsystem wishes to make accessible to other subsystems. Case is not significant in the `exports` keyword. Any symbol named in the *`public-list`* must be a public symbol in one of the subsystem modules. Each symbol must be declared as an external symbol in all modules accessing the identified function or variable, whether or not these modules are within the same subsystem. Case is significant in symbols in the *`public-list`*. Every subsystem definition, with the possible exception of the subsystem that contains the `main()` function, must have an `exports` list that contains at least the public symbol for the entry point to the subsystem.

The *public-list* must list all symbols referred to by other subsystems. Public symbols not in the *public-list* are accessible only from within the subsystem itself. Non-public symbols do not appear in the *public-list*. Public symbols can appear in any order in the *public-list*.

Exported functions have the following characteristics:

- They use the far form of call and return.
- They save and restore the caller's DS register upon entry and exit.
- They reload the DS register with their associated data segment selector upon entry.

The compiler implicitly modifies the declarations of exported symbols, if necessary, by inserting the `far` keyword in the appropriate place in the declarations. The modifications occur even if the `extend` control is not in effect.

Export a function only if it is referenced outside the defining subsystem, because accessing exported functions requires more code and more execution time than accessing functions within the same subsystem.

Within a program, the *subsystem-id* name must be distinct from all module names because both share the same name space. Within a program (across all subsystems), exported symbols must also be unique. However, *subsystem-id* names and module names do not share name space with public symbols.

The `has` and `exports` lists often have several dozen entries each. To accommodate lists of this length, a subsystem definition can be continued over more than one line. The continuation lines must be contiguous, each must begin with a dollar sign (`$`) in the first column, and the next non-whitespace character cannot be a comma (`,`), a right parenthesis (`)`), or a semicolon (`;`). You can specify any number of `has` and `exports` lists in a definition, in any order, which allows you to format your subsystem specification file so it can be easily read and maintained.

Compile all modules in your program with the same set of subsystem definitions, so that the compiler makes consistent assumptions about the location of external symbols. To avoid conflicting definitions, place all of the subsystem definitions into one file and use the `subsys` control in the invocation line or in a `#pragma` preprocessor directive for every compilation. Inconsistent subsystem definitions cause the linker or binder to issue an error.

NOTES

Do not use the `codesegment` or `datasegment` control in an invocation that specifies the `subsys` control, or when the source text contains the `subsys` control in a `#pragma` preprocessor directive. The compiler issues an error or a warning, depending on whether the `subsys` control is found in the invocation line or in a `#pragma` preprocessor directive, respectively.

A `#pragma` preprocessor directive specifying the `modulename` control must precede any `#pragma` directives that specify the `subsys` control.

The definition for an open subsystem without submodel, `has` list, or `exports` list can be placed on the invocation line. Place all definitions of closed subsystems inside the subsystem definitions file.

Programs written in iC-86/286/386 and in PL/M-86/286/386 can share subsystem definitions because the syntax for the definitions is identical for both languages. Symbol names in the `exports` list must match the case used in the C program because C is a case-sensitive language.

The compiler preserves case distinction in identifiers in `exports` lists. The compiler always ignores dollar signs (`$`) in identifiers, even if the `extend` control is not in affect. The compiler ignores valid PL/M controls unrelated to segmentation, such as `$IF` and `$INCLUDE`. The compiler ignores lines whose first character is not a dollar sign (`$`).

9.4 Example Definitions

Recall the example program in Section 9.1. The following examples guide you through creating subsystem definitions for the small-model subsystems in Figure 9-3 and the mixed-model subsystems in Figure 9-4.

9.4.1 Creating Three Small-model RAM Subsystems

The following subsystem definitions define three small-model RAM subsystems for the program, which are closed subsystems by definition. The `SUBPROCESS` and `SUBOUTPUT` subsystems export their entry-point functions. No other symbols are exported. The definitions default to the `-const in data-` submodel specification.

```
$ small (SUBINPUT
$        has mod1, mod2, mod3)
$ small (SUBPROCESS
$        has mod4, mod5, mod6, mod7, mod8;
$        exports process_entry)
$ small (SUBOUTPUT
$        has mod9, mod10;
$        exports output_entry)
```

The program does not contain calls or references that require the `far` keyword, because all three subsystems share one single `DATA` segment, which contains constants. Assuming that the `mod3_fn` function in the `mod3` module calls the `process_entry` function defined in the `mod4` module and passes a pointer to some data called `data_object`, the definitions of `mod3_fn` and `process_entry` have the following general form:

```
/* in SUBINPUT                          */

int data_object;

int mod3_fn ()
{
   extern int process_entry (int * );

...

/* calling a function in another      */
/* subsystem causes a load to a       */
/* segment register                   */
```

```
   process_entry ( &data_object );

...

}

/*-------------------------------------------------------*/

/* in SUBPROCESS                    */

int process_entry (int * data)
{
   int mod4int;

...

/* de-referencing the pointer causes */
/* a load to a segment register      */

   mod4int = *data + 1;

...

}
```

If the subsystem definitions are in a file named `smallss.def`, the compilation of `mod3.c` is as follows, where `ic`*n*`86` is `ic86`, `ic286`, or `ic386`:

```
C:> icn86 mod3.c subsys(smallss.def)
```

9.4.2 Two Small-model ROM Subsystems and One Compact-model ROM Subsystem

The following subsystem definitions define two small-model ROM subsystems and one compact-model ROM subsystem for the program, all closed subsystems by definition. The definitions list the entry points for the SUBPROCESS and SUBOUTPUT subsystems. No other symbols are exported.

```
$ small   (SUBINPUT
$           -const in code-
$           has mod1, mod2, mod3)
$ compact (SUBPROCESS
$           -const in code-
$           has mod4, mod5, mod6, mod7, mod8;
$           exports process_entry)
$ small   (SUBOUTPUT
$           -const in code-
$           has mod9, mod10;
$           exports output_entry)
```

All pointers to data in all three subsystems are far pointers, because data can be in different segments within any of these subsystems. However, all of the subsystems use near function calls within a subsystem. The definitions of mod3_fn and process_entry have the same form as in Section 9.4.1.

If the subsystem definitions are in a file named compss.def, the compilation of mod3.c is as follows, where ic*n*86 is ic86, ic286, or ic386:

```
C:> icn86 mod3.c subsys(compss.def)
```

9.4.3 Example Using an Open Subsystem

Recall that if a program uses both small-model and compact-model subsystems, the `main()` function must be in a small-model subsystem. Assume the `main()` function is in the `mod1` module. If the `mod1`, `mod2`, and `mod3` modules are part of an open small-model RAM subsystem, the program can use the following subsystem definitions:

```
$ compact (SUBPROCESS
$          -const in code-
$          has mod4, mod5, mod6, mod7, mod8;
$          exports process_entry)
$ small   (SUBOUTPUT
$          -const in code-
$          has mod9, mod10;
$          exports output_entry)
$ small   (-const in data-)
```

Because the program passes a data pointer from a small-model RAM subsystem to a different model subsystem, the calling function (in the small-model RAM subsystem) must explicitly use the `far` keyword in a prototype, declaration, or cast to pass the pointer. Use one of these three options within the `mod3_fn` function as follows:

- Casting the address of `data_object` to `far` uses the selector-and-offset address for this call only:

  ```
  process_entry (( int far * ) &data_object);
  ```

- Declaring `data_object` as a far integer results in all references to `data_object` using the selector-and-offset address, unless overridden by the `near` keyword:

  ```
  int far data_object;
  ```

- Changing the prototype for the `process_entry` external function results in all calls from within `mod3_fn` (where the prototype is declared) to `process_entry` to use the selector-and-offset address for whatever pointer is passed:

  ```
  extern int process_entry (int far *);
  ```

If the subsystem definitions are in a file named `subs.def`, the compilation of `mod1.c` is as follows, where `ic`*n*`86` is `ic86`, `ic286`, or `ic386`:

```
C:> icn86 mod1.c extend subsys(subs.def)
```

Figure 9-5 shows the names of the segments. The code segment for the open subsystem has the name `CODE` or `CODE32` instead of `SUBINPUT_CODE` or `SUBINPUT_CODE32` as in Figure 9-4, because an open subsystem by definition does not have a *`subsystem-id`*.

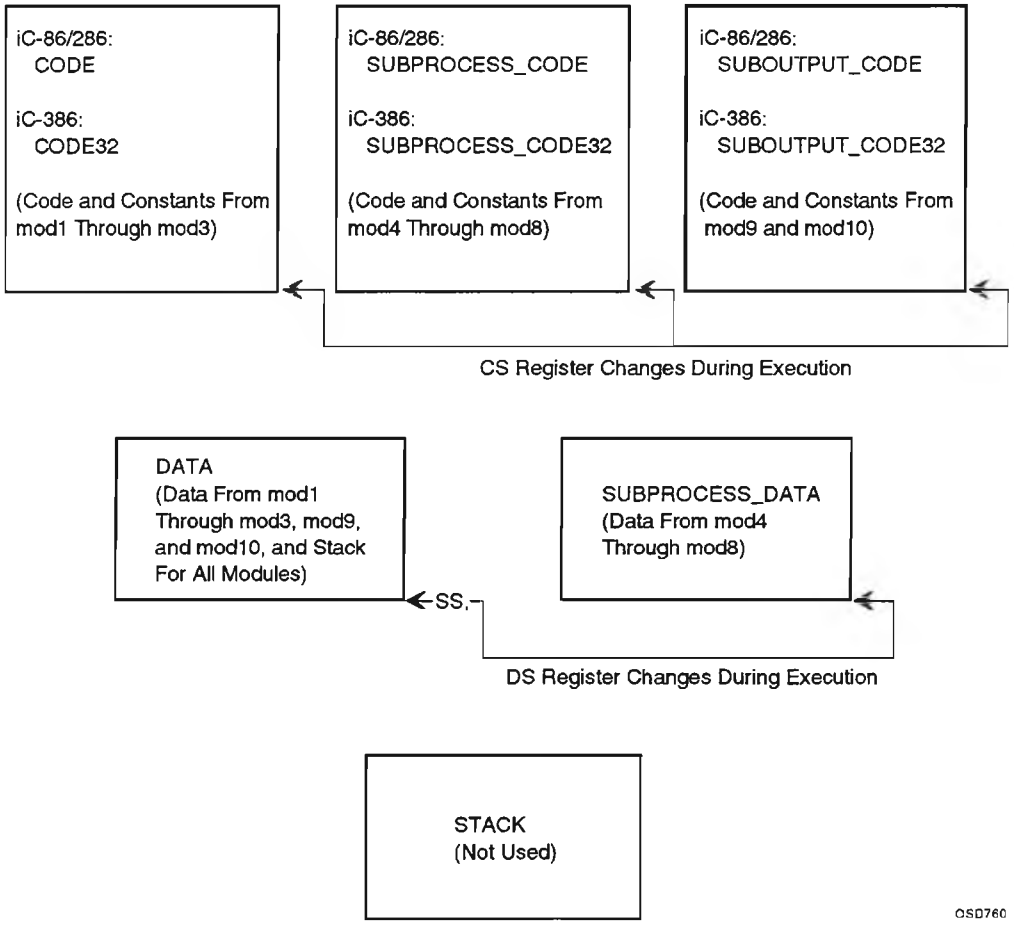

Figure 9-5 Subsystems Example Program Using One Open and Two Closed Subsystems

Contents

10

Language Implementation

Language Implementation 10

This chapter contains information on the iC-86/286/386 implementation of the C programming language. This information is more specific than the information found in *C: A Reference Manual*, listed in Chapter 1. The implementation of the language is divided into the following topics:

- data types and keywords
- conformance to the ANSI C standard
- implementation-dependent compiler features

Where applicable throughout the chapter, conformance to the ANSI C standard is noted.

10.1 Data Types

The iC-86/286/386 compilers recognize three classes of data types: scalar, aggregate, and `void`. This section describes the iC-86/286/386 implementation of the data types. See *C: A Reference Manual*, listed in Chapter 1, for more general information about data types.

Objects of a data type longer than one byte occupy consecutive bytes in memory. Objects reside in memory from low-order to high-order bytes within a word and from low address to high address across multiple bytes. The address of an object is the address of the low-order byte of the object.

Many names of the data types serve as keywords in the source text. The following words are keywords in iC-86/286/386:

```
auto        do          goto        signed      unsigned
break       double      if          sizeof      void
case        else        int         static      volatile
char        enum        long        struct      while
const       extern      register    switch
continue    float       return      typedef
default     for         short       union
```

The following additional keywords are supported by iC-86/286/386 if the `extend` control is in effect:

`alien`	is a storage-class specifier that indicates a function uses the fixed parameter list calling convention.
`far`	is a type qualifier that indicates a segment-selector-and-offset address.
`near`	is a type qualifier that indicates an offset-only address.
`readonly`	is a type qualifier that is equivalent to the `const` keyword.

See Chapter 4 for information on where to use the `near` and `far` qualifiers.

10.1.1 Scalar Types

A scalar object is a single value, such as the integer value 42 or the bit field 10011. Most scalar objects occupy 1, 2, 4, or 8 bytes of memory. Bit fields occupy as many bits as assigned and need not be a multiple of one byte long (8 bits). A bit field cannot be longer than one word (2 bytes for iC-86 and iC-286, or 4 bytes for iC-386).

Tables 10-1 and 10-2 show the scalar data types for iC-86/286 and iC-386, the amount of memory occupied by the data type's object, the arithmetic format, and the range of accepted values.

The iC-86/286/386 compilers support the declaration of:

- a `char` to explicitly be declared `signed` or `unsigned`
- an integer constant to be declared `long`, `unsigned`, or `unsigned long`
- enumerated types

Table 10-1 86 and 286 Processor Scalar Data Types

Data Type	Size in Bytes	Format	Range
char[1]	1	integer or two's-complement integer	0 to 255 or -128 to 127
unsigned char	1	integer	0 to 255
signed char	1	two's-complement integer	-128 to 127
enum	2	two's-complement integer	-32,768 to 32,767
unsigned short	2	integer	0 to 65,535
signed short	2	two's-complement integer	-32,768 to 32,767
unsigned int	2	integer	0 to 65,535
signed int	2	two's-complement integer	-32,768 to 32,767
unsigned long	4	integer	0 to 4,294,967,295
signed long	4	two's complement integer	-2,147,483,658 to 2,147,483,647
float	4	single-precision floating-point number	8.43×10^{-37} to 3.37×10^{38} (approx. absolute value)
double	8	double-precision floating-point number	4.19×10^{-307} to 1.67×10^{308} (approx. absolute value)
long double	8	double-precision floating-point number	4.19×10^{-307} to 1.67×10^{308} (approx. absolute value)
bit field	1 to 16 bits	integer	depends on number of bits
near pointer	2	offset-only address	64K bytes
far pointer	4	2-byte offset and 2-byte selector	1 megabyte for 86 processor 1 gigabyte for 286 processor

[1] Integer (unsigned) if the nosignedchar control is in effect, or two's complement integer (signed) if the signedchar control is in effect (default).

Table 10-2 Intel386™ Processor Scalar Data Types

Data Type	Size in Bytes	Format	Range
char[1]	1	integer or two's-complement integer	0 to 255 or -128 to 127
unsigned char	1	integer	0 to 255
signed char	1	two's-complement integer	-128 to 127
enum	4	two's-complement integer	-2,147,483,648 to 2,147,483,647
unsigned short	2	integer	0 to 65,535
signed short	2	two's-complement integer	-32,768 to 32,767
unsigned int	4	integer	0 to 4,294,967,295
signed int	4	two's-complement integer	-2,147,483,648 to 2,147,483,647
unsigned long[2]	4 or 8	integer	0 to 4,294,967,295 or 0 to 2^{64}-1
signed long[3]	4 or 8	two's-complement integer	-2,147,483,648 to 2,147,483,647 or -2^{63} to 2^{63}-1
float	4	single precision floating-point	8.43×10^{-37} to 3.37×10^{38} (approximate absolute value)
double or long double	8	double precision floating-point	4.19×10^{-307} to 1.67×10^{308} (approximate absolute value)
bit field	1 to 32 bits	integer	depends on number of bits
near pointer	4	offset-only address	4 gigabytes
far pointer	6	4-byte offset and 2-byte selector	64 terabytes

[1] Integer (unsigned) if the nosignedchar control is in effect, or two's complement integer (signed) if the signedchar control is in effect (default).
[2] If long64 control is specified, size is 8 bytes and range is 0 to 2^{64}-1.
[3] If long64 control is specified, size is 8 bytes and range is -2^{63} to 2^{63}-1

The iC-86/286/386 compilers support two precisions for floating-point numbers: `float` and `double`. The compiler treats the `double` and `long double` formats as `double`. The numeric coprocessor automatically promotes `float` and `double` objects to extended precision for arithmetic operations.

10.1.2 Aggregate Types

An object of an aggregate type is a group of one or more scalar objects. The iC-86/286/386 aggregate data types are as follows:

array	has one or more scalar or aggregate elements. All elements in an array are the same data type. The elements reside in contiguous locations from first to last. Multi-dimensional arrays reside in memory in row-major order.
structure	has one or more scalar or aggregate components. The different components of a structure can be different data types. The components of a structure reside in memory in the order that they appear in the structure definition, but may have unused memory between components. See Chapter 3 for more information on the `align` control and the allocation of structures.
union	has one piece of contiguous memory that can hold one of a fixed set of components of different data types. The amount of memory for a union is sufficient to contain the largest of its components. A union holds only one component at a time, and the union's data type is the data type of the component most recently assigned.

10.1.3 Void Type

The `void` data type has no values and no operations. Use the `void` keyword for a function that returns no value or for a function that takes no arguments. Use `void *` to denote a pointer to an unspecified data type or a pointer to a function that returns no value. Cast to `void` to explicitly discard a value. The following are sample declarations for these uses:

```
void retnothing (int a);  /* function returns no value   */
int intfunc (void);       /* function takes no arguments */
void * genericptr();      /* pointer to unspecified type */
(void) intfunc();         /* discard the return value    */
```

10.2 iC-86/286/386 Support for ANSI C Features

This section provides information about features in the ANSI C standard that are not discussed elsewhere in this chapter. The iC-86/286/386 compilers support these features unless otherwise noted.

10.2.1 Lexical Elements and Identifiers

Trigraphs allow C programs to be written without using characters reserved by ISO (International Standards Organization) as alphabet extensions. See *C: A Reference Manual*, listed in Chapter 1, for more information about trigraphs.

Character constants and string literals can contain numeric escape codes in hexadecimal format. See *C: A Reference Manual*, listed in Chapter 1, for more information about numeric escape codes.

Wide characters support very large character sets, such as pictographic alphabets. The iC-86/286/386 compilers recognize the ANSI wide-character syntax but implements wide characters the same as ASCII characters by truncation.

At least 31 characters of non-external names must be significant. The compiler supports 40-character significance in internal and external names. Case is significant in internal names.

10.2.2 Preprocessing

The ## operator concatenates adjacent tokens in macro definitions, forming a single token. See *C: A Reference Manual*, listed in Chapter 1, for more information about the ## operator.

The compiler concatenates adjacent string literals.

Preprocessor directives in the source text do not have to begin in column one; the # character must be the first nonblank character of a preprocessor directive line.

The # operator, followed by the name of a macro parameter, expands to the actual argument enclosed in quotation marks ("). When creating the string, the preprocessing facility precedes quotation marks (") and backslashes (\) within the argument with a backslash.

The ANSI C standard specifies the new `#elif` preprocessor directive and the `defined` preprocessor operator. See *C: A Reference Manual*, listed in Chapter 1, for more information about these additions.

A single-character character constant in an `#if` or `#elif` conditional preprocessor directive has the same value as the same character in the execution character set.

The `#pragma` preprocessor directive allows communication of implementation-specific information to the compiler. Most of the iC-86/286/386 compiler controls can be used in a `#pragma` preprocessor directive. For more information about using `#pragma` and the syntax of compiler controls, see Chapter 3.

The maximum length of a `#pragma` preprocessor directive is 1 kilobyte characters. All compiler controls except `define` and `include` can be specified in a `#pragma` preprocessor directive. Where *`control`* is a single compiler control and an optional argument list a `#pragma` has the following form:

```
#pragma control
```

An `#include` preprocessor directive can use a macro to identify the file or header file.

The arguments to a `#line` preprocessor directive may result from macro expansion.

The `#error` preprocessor directive reports user-defined diagnostics.

The maximum nesting level of conditional compilation directives is 16. The maximum nesting level of macro invocations is 64.

The maximum number of arguments in macro invocation is 31.

See Chapter 5 for a list of predefined macros.

10.3 Implementation-dependent iC-86/286/386 Features

This section provides additional information about how iC-86/286/386 implements the implementation-dependent characteristics of the C language as specified by the ANSI C standard.

The compiler's word size is 2 bytes for iC-86/286 and 4 bytes for iC-386. By default, memory read and write operations in the *n*86 processors occur from low-order address to high-order address ("little endian"). Objects over 32 kilobytes do not conform to ANSI standards for pointer arithemtic.

10.3.1 Characters

The source character set is 7-bit ASCII, except in comments and strings, where it is 8-bit ASCII. The execution character set is 8-bit ASCII. The compiler maps characters one-to-one from the source to the execution character set. You can represent all character constants in the execution character set. The iC-86/286/386 compilers recognize the wide-character ANSI syntax. Wide characters are implemented the same as ASCII characters.

The `signedchar | nosignedchar` control determines whether the compiler considers a `char` that is declared without the `signed` or `unsigned` keywords to be `signed` or `unsigned`. The default control is `signedchar`. A character value occupies a single byte. Each character is made up of 8 bits, ordered from right to left, or least significant to most significant.

In a character constant, the compiler assigns up to two characters for iC-86/286 or four characters for iC-386 to a word, with the first character in the low-order byte. In words containing at least one character, when any byte does not contain a character, the compiler fills the byte with the sign of the highest-order byte that does contain a character. An unused byte is sign-extended if the `signedchar` control is in effect (default), or zero-extended if the `nosignedchar` control is in effect.

The encoding of multi-byte characters does not depend on any shift state.

10.3.2 Integers

When a signed or unsigned integer is converted to a narrower signed integer, or an unsigned integer is converted to a signed integer of equal width, overflow is ignored and high-order bits are truncated; a sign change can occur.

The compiler treats signed integers as bit strings in bitwise operations.

The sign of the remainder on integer division is the same as the sign of the dividend.

A right shift of a signed integral type is arithmetic.

See Table 10-1 for types and sizes of integers.

10.3.3 Floating-point Numbers

When the compiler converts:

- an integral number to a floating-point number, any truncation is controlled by the numeric coprocessor or emulator.
- a floating-point number to a narrower floating-point number, the direction of rounding is controlled by the numeric coprocessor or emulator.

See Table 10-1 for types and sizes of floating-point numbers.

10.3.4 Arrays and Pointers

Character string initializers within a character array are not null-terminated.

An unsigned integer is large enough to hold the maximum size of an array. An integer is large enough to hold the difference between two pointers to members of the same array.

When you cast:

- a near pointer to `int`, the compiler preserves the bit representation.
- a near pointer to `long`, the iC-86/286 compilers zero-extend the offset. The iC-386 compiler sign-extends the offset if the `long64` control is in effect. If the `nolong64` control is in effect, the result is the same as casting a near pointer to `int`.
- a far pointer to `int`, the compiler yields the offset-only part of the pointer value and discards the selector.
- a far pointer to `long`, the iC-86/286 compilers preserve the bit representation. The iC-386 compiler sign-extends the high-order 16 bits if the `long64` control is in effect. If the `nolong64` control is in effect, the result is the same as casting a far pointer to `int`.
- an `int` constant to a near pointer, the compiler preserves the bit representation.
- an `int` constant expression to a far pointer, the compiler uses zero bits for the selector. Casting any other `int` expression to a far pointer uses the current value of the DS register for the selector.
- a `long` integer to a near pointer, the iC-86/286 compilers discard the high-order 16 bits. The iC-386 compiler discards the high-order 32 bits if the `long64` control is in effect. If the `nolong64` control is in effect, the result is the same as casting an `int` to a near pointer.
- a `long` integer to a `far` pointer, the iC-86/286 compilers preserve the bit representation. The iC-386 compiler discards the high-order 16 bits if the `long64` control is in effect. If the `nolong64` control is in effect, the result is the same as casting an `int` to a far pointer.

The compiler can initialize arrays with storage class `auto`.

See Table 10-1 for the types and sizes of pointers.

10.3.5 Register Variables

The (E)SI and (E)DI registers can contain objects of the `register` storage class. The `register` storage class is effective only for `enum`, `signed short`, `signed char`, `int`, `unsigned int`, and `near` pointer objects. Register storage is honored only under the variable parameter list (VPL) function calling convention.

The iC-86/286/386 compilers allocate registers for register objects in the following order (only under VPL):

1. parameters, in the order that they appear in the function declaration
2. local variables, in the order that the code references them

When a local variable assigned to a register goes out of scope, its register becomes available again.

10.3.6 Structures, Unions, Enumerations, and Bit Fields

Each of the sets of structure, union, and enumeration tags has its own name space. Each function has a name space for its labels. Each structure or union has a name space for its members. Identical names in different name spaces do not conflict. See Section 10.3.9 for information on virtual symbol table capacity.

Assignment expressions can assign to structures or unions. A function can have structures and unions as parameters. The function call passes structures and unions by value. A function can return a structure or a union.

The compiler can initialize unions and structures of storage class `auto`.

When the program accesses a member of a union object using a member of a different type than was last assigned, the result is undefined.

The first member in a union declaration determines the map of the union's initializer.

The compiler represents enumeration types as `int`.

Bit fields are not necessarily allocated on word boundaries; if a bit field is short enough, it occupies the space between the end of the previous bit field and the end of the word the previous bit field occupies. See Chapter 3 for information on the `align` control to allocate bit fields on word boundaries.

The compiler treats a bit field that is declared without the `signed` or `unsigned` keywords as `signed`.

The allocation of bit fields in an integer is low-order to high-order.

10.3.7 Declarators and Qualifiers

Objects can be declared as being `const` or `volatile`. Pointers can point to `const` or `volatile` objects. A `const` object cannot be modified by assignment. The compiler does not remove references to `volatile` objects during optimization.

Access to a `volatile` object constitutes two references, a load and a store, when an object qualified with the `volatile` keyword occurs as any of the following:

- an operand of a pre-increment operator
- an operand of a pre-decrement operator
- an operand of a post-increment operator
- an operand of a post-decrement operator
- a left operand of a compound assignment operator

Every other occurrence of a `volatile` object constitutes one reference.

The iC-86/286/386 compilers allow attribute specifiers to follow a left parenthesis (( ) or comma (,). In the ANSI C standard, attribute specifiers are valid in declarators only when subordinate to an asterisk (*). For example, the following line is invalid in the ANSI C standard:

```
int (const i), volatile j;
```

However, the iC-86/286/386 compilers recognize the line above as equivalent to these lines:

```
int const i;
int volatile j;
```

This extended syntax does not affect the semantics of any source text that conforms fully to the rules of the ANSI C standard. The extension causes an asymmetry. For example, the first of the following two declarations causes `x`, `y`, and `z` all to be read-only variables. The second declaration causes only `y` to be read-only; `x` and `z` are both modifiable:

```
int const x, y, z;                  /* valid for ANSI C */
int x, const y, z;                  /*  extended syntax */
```

See Section 10.1 for information on the `alien`, `far`, and `near` type qualifiers. See Chapter 4 for information on where to use the `near` and `far` type qualifiers.

10.3.8 Statements, Expressions, and References

The maximum number of:

- `case` values in a `switch` statement is 512.
- functions defined in a module is 1,022.
- external references in a module is 511.
- arguments in a function call is 31.

The maximum nesting level of:

- statements is 32.
- functions specified in function argument lists is 20.

The iC-86/286/386 `optimize` control governs association of subexpressions in evaluation.

10.3.9 Virtual Symbol Table

The maximum virtual symbol table size is 512 kilobytes. This size is large enough to hold over 8,000 C symbols or over 16,000 macros. The virtual symbol table also stores identifiers and macro bodies. In addition, the compiler generates a symbol for each string literal, floating-point constant, and temporary variable.

The type table can contain a maximum of 2,048 entries. Each distinct type takes up one entry in the type table. The compiler does not duplicate identical pointer, array, function, or qualified types, except that every prototype has a unique entry, even if an identical prototype entry exists.

Contents

11

Messages

11 Messages

The iC-86/286/386 compilers can issue the following types of messages:

- fatal errors
- errors (syntax and semantic)
- warnings
- remarks
- subsystem diagnostics
- internal errors

All messages, except fatal and internal error messages, are reported in the print file. Fatal and internal errors appear on the screen, abort compilation, and no object module is produced. Other errors do not abort compilation but no object module is produced. Warnings and remarks usually provide information only and do not necessarily indicate a condition affecting the object module.

iC-86/286/386 messages relating to syntax are interspersed in the listing at the point of error. Messages relating to semantics are interspersed in the listing or displayed at the end of the source program listing; they refer to the statement number on which the error occurred.

11.1 Fatal Error Messages

Fatal error messages have the following syntax:

```
iC-n86 FATAL ERROR
message
```

Where:

n	is empty, 2, or 3 for the iC-86, iC-286, or iC-386 compiler, respectively.

Following is an alphabetic list of fatal error messages.

`argument expected for` *`control`* `control`

A compiler control is specified without the argument required by context. Not having a required argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if it occurs in a `#pragma` directive. See Chapter 3 for more information on compiler control syntax.

`argument length limit exceeded for` *`control`* `control`

The length of the argument to the control exceeds the maximum allowable by the compiler. For example, an argument to `modulename` exceeds 40 characters.

`compiler error`

This message follows internal compiler error messages. If you receive this message, contact Intel customer service. See the Service Information on the inside back cover.

`control` `control cannot be negated`

You cannot use the `no` prefix with this compiler control. See Chapter 3 for information on which compiler controls can be negated. Improper negating is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if it occurs in a `#pragma` directive.

`duplicate` *`control`* `control`

A control that must not be specified more than once was specified more than once. Only the following controls can be specified more than once:

```
align         include           subsys
define        interrupt         varparams
fixedparams   searchinclude
```

See Chapter 3 for more information on these controls. If you specify a compiler control both in the compiler invocation and in a `#pragma` preprocessor directive, the compiler invocation specification takes precedence. A duplicate control is a fatal error if it occurs in the compiler invocation but the preprocessor only issues a warning if it occurs in a `#pragma` directive.

`duplicate interrupt number:` *`interrupt_number`*

Indicates *`interrupt_number`* was used more than once in `interrupt` controls. A duplicate interrupt number is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if it occurs in a `#pragma` directive.

`expression too complex`

A complex expression exhausted an internal structure in the compiler. Break the expression down into simpler components, or try a lower optimization level.

`illegal macro definition:` *`macro_name`*

An invalid macro was defined on the command line with the `define` control.

`input pathname is missing`

A primary source file pathname was not specified in the compiler invocation.

`insufficient memory`

There is not enough memory available for the compiler to run. Check the available system memory.

`insufficient memory for macro expansion`

An internal structure was exhausted during macro expansion. Two causes of this error are: the macro or the actual arguments are too complex, or the macro's expansion is too deeply nested. See Chapter 10 for information on the applicable limits. Also see the related error message, `macro expansion too nested`.

`invalid control: control`

A control not supported by the compiler was specified. Check the spelling of the control. An invalid control is a fatal error if it occurs in the compiler invocation but the preprocessor only issues a warning if the invalid control occurs in a `#pragma` directive. See Chapter 3 for a list of the iC-86/286/386 controls.

`invalid control syntax`

The compiler control contained a syntax error. See Chapter 3 for more information on the syntax of the compiler controls. Invalid control syntax is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the invalid syntax occurs in a `#pragma` directive.

`invalid decimal parameter: value`

Non-decimal characters were found in an argument that must be a decimal value. See Chapter 3 for more information on the syntax of the compiler controls. An improper non-decimal argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive.

`invalid identifier: identifier`

An identifier does not follow the rules for forming identifiers in C. An invalid identifier is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the invalid identifier occurs in a `#pragma` directive.

`invalid syntax for control control`

Invalid syntax is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper control syntax occurs in a `#pragma` directive. See Chapter 3 for more information on the syntax of the compiler controls.

`missing or misplaced right parenthesis`

A right parenthesis is required to delimit arguments to a compiler control. See Chapter 3 for more information on the syntax of the compiler controls. An improper right parenthesis is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the misplaced or missing parenthesis occurs in a `#pragma` directive.

`no more free space`

The internal structure used to hold macros is exhausted. Use fewer macros in your program. See Chapter 10 for information on the applicable limits.

`null argument for control control`

Null arguments for compiler controls are not allowed. For example, the following is illegal:

```
ALIGN(siga=2,,sigb=2)
```

A null argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the null argument occurs in a `#pragma` directive.

`parameter not allowed for control control`

This message indicates an attempt to pass arguments to a control that accepts none. See Chapter 3 for more information on the syntax of compiler controls. Improper argument passing is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive.

`parameter not allowed for negated` *`control`* `control`

Negated controls generally do not accept arguments. The `noalign` control is the only exception. See Chapter 3 for more information on the syntax of compiler controls. An improper argument for a negated control is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive.

`parameter out of range for` *`control`* `control:` *`parameter`*

This message indicates an attempt to use an argument value that is out of the valid range. See Chapter 3 for more information on the range of argument values accepted by compiler controls. An out-of-range argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the improper argument occurs in a `#pragma` directive.

`parameter required for` *`control`* `control`

A missing required argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the missing argument occurs in a `#pragma` directive.

`previous errors prevent further compilation`

The compiler was unable to recover from previous errors in the compilation. Correct the errors reported thus far, then recompile.

`subsys control conflicts with codeseg/dataseg control`

A `subsys` control cannot occur while the `codesegment` or `datasegment` control is in effect, and vice versa.

`switch table overflow`

Too many active `cases` exist in a `switch` statement that has not yet been completed. See Chapter 10 for information on the applicable limits.

`too many directories are specified for search - pathname`

Too many directories are specified in the compiler invocation with the control `searchinclude`. The *`pathname`* is the directory at which the error occurred, that is, the first directory over the limit. See Chapter 10 for information on the applicable limits.

`type table full`

Too many symbols with non-standard data types are defined in the module. Remove unused definitions, or break down the module.

`unable to recover from syntax error`

A syntax error has put the compiler in a state that would lead to spurious error messages or internal error messages if the compiler continues to process the program; for example, using the `far` or `near` keywords in a program compiled without the `extend` control, or omitting a semicolon from a function declaration.

`whiles, fors, etc. too deeply nested`

The statement nesting structure of the module exhausted an internal structure in the compiler. See Chapter 10 for information on the applicable limits.

11.2 Error Messages

Syntax error messages have the following format:

`*** ERROR AT LINE` *`number`* `OF` *`file`*`: syntax error near` *`token`*

Where:

`number`	is the line number of the offending source line.
`file`	is the name of the source file.
`token`	is the token in the source text near where the error occurred.

Semantic error messages have the following syntax:

*** ERROR AT LINE *nn* OF *filename*: *message*

Where:

filename	is the name of the primary source file or include file in which the error occurred.
nn	is the source line number where the error is detected.
message	is the explanation.

Following is an alphabetic list of error messages.

`# operator missing macro parameter operand`

The # operator must be followed by a macro parameter.

`## operator occurs at beginning or end of macro body`

The ## (token concatenation) operator is used to paste together adjacent preprocessing tokens, so it cannot be used at the beginning or end of a macro body.

`a semantic token cannot precede subsys control`

Text that constitutes a semantic token cannot occur before a `#pragma subsys`.

`align/noalign control not allowed with union/enum tag`

A union or enumeration tag cannot be used as an argument to the `align` or `noalign` control. Use a structure tag only.

`an attempt to undefine a non-existent macro`

The name in the `#undef` preprocessor directive is not recognized as a macro.

`anonymous parameter`

A parameter in a function definition is prototyped but not named.

`arguments not allowed`

Arguments were passed to a function that does not accept arguments.

`array too large`

This error occurs when the size of an array exceeds 64 kilobytes for iC-86 and iC-286, or 4 gigabytes for iC-386.

`attempt to use Ø as divisor in division/modulo`

A divide-by-0 was detected in a divide or modulo operation.

`basic block too complex`

This error is caused by a function with a long list of statements without any statements such as `label`, `case`, `if`, `goto`, or `return`. Break the function into several smaller functions, or add labels to some statements.

`call not to a function`

A call is made to a symbol which is not a function.

`call to interrupt handler`

An interrupt handler can be activated only by an interrupt.

`cannot initialize`

The type or number of initializers does not match the initialized variable.

`cannot initialize extern in block scope`

An external declaration cannot be initialized in any scope other than file scope. The following example is an invalid external declaration:

```
f()
{ extern int i = 1;
}
```

`case not in switch`

A case was specified, but not within a `switch` statement.

`code segment too large`

The size of the code segment exceeds 64 kilobytes for iC-86 and iC-286, or 4 gigabytes for iC-386. Break the module into two or more separately compiled modules, or use subsystem definitions. See Chapter 9 for information on defining subsystems.

`conditional compilation directive is too nested`

The module contains more than the maximum number of conditional statements. See Chapter 10 for information on the applicable limits.

`constant expected`

A non-constant expression appears when a constant expression is expected (e.g., a non-constant expression as array bounds or as the width of a bit field).

`constant value must be an int`

The constant specified must be representable as the data type `int`.

`data segment too large`

The size of the data segment exceeds 64 kilobytes for iC-86 and iC-286, or 4 gigabytes for iC-386. Break the module into two or more separately compiled modules, or use subsystem definitions. See Chapter 9 for information on defining subsystems.

`default not inside switch`

A `default` label was specified outside of a `switch` statement.

`duplicate case in switch, `*`number`*

The same value, *`number`*, was specified in more than one `case` in the same `switch` statement.

`duplicate default in switch`

More than one `default` label was specified within the same `switch` statement.

`duplicate label`

A label was defined more than once within the same function.

`duplicate parameter name`

The same identifier was found more than once in the identifier list of a function declarator. For example, the following code contains a duplicate `a` identifier:

```
int f(a, a) {}
```

`duplicate tag`

A tag was defined more than once within the same scope.

`empty character constant`

A character constant should include at least one character or escape sequence.

`floating point operand not allowed`

An operand is non-integral, but the operator requires integral operands. That is, `~`, `&`, `|`, `^`, `%`, `>>`, and `<<` all require integral operands.

`function body for non-function`

A function body was supplied for an identifier that does not have function type, as in this example:

```
int i {}
```

`function declaration in bad context`

A function is defined (i.e., appears with a formal parameter list), but not at module-level. Or, a function declarator with an identifier list, which is legal only for function definitions, was encountered within a function, as in this example:

```
int main(void)
{
  int f(a);
}
```

`function redefinition`

More than one function body has been found for a single function, as in this example:

```
int f() {}
int f() {}
```

`illegal assignment to const object`

Constants cannot be modified.

`illegal break`

A `break` statement appears outside of any `switch`, `for`, `do`, or `while` statement.

`illegal constant expression`

The expression within an `#if` or `#elif` is not built correctly.

`illegal constant suffix`

The suffix of a number is not `L`, `U`, or a legal combination of the two.

`illegal continue`

A `continue` statement appears, but not within any `for`, `do`, or `while` statement.

`illegal #elif directive`

An `#elif` directive is encountered after an `#else` directive.

`illegal #else directive`

An `#else` directive is encountered after an initial `#else` directive.

`illegal field size`

Legal field sizes are 0-32 for unnamed fields, and 1-32 for named fields. See *C: A Reference Manual* for more information on bit fields.

`illegal floating point constant in exponent`

A floating-point constant cannot be an exponent.

`illegal function declaration`

Internal error; may be caused by an earlier syntax error.

`illegal hex constant`

A hexadecimal constant contains non-hex characters or is without a 0 prefix.

`illegal macro redefinition`

A macro can be redefined only if the body of the redefined macro is exactly the same as the body of the originally defined macro.

`illegal nesting of blocks, ends not balanced`

Braces delimiting a block of code are unbalanced.

`illegal syntax - left parenthesis is expected`

The name of a macro that accepts arguments is specified with no argument list, or the argument list is not properly delimited with parentheses.

`illegal syntax in a directive line`

A syntax error is encountered in a preprocessor directive.

`illegal syntax in a directive line - newline expected`

A preprocessor directive line is not terminated with a newline character.

`illegal syntax in an argument list`

An argument list in a macro contains misplaced or illegal characters.

`incompatible types`

The two operands of a binary operator have incompatible types, for example, assigning a non-zero integer to a pointer.

`incomplete type`

The compiler detected a variable whose type is incomplete, such as the following example declaration where the type of `s` is not complete if the program contains no previous declaration defining the tag `S`.

```
int f(struct S s)
{ ... }
```

`invalid argument for builtin function`

For example, the built-in function `causeinterrupt` appears with a non-constant argument. Built-in functions are the functions that provide direct access to various processor features. See Chapter 6 for the syntax of the built-in function calls.

`invalid attribute for:` *`function_name`*

The source program attempted to set multiple and conflicting attributes for a function. For example, a `varparams` or `fixedparams` control appears for a function whose calling convention has already been established by use, definition, declaration, or a previous calling-convention control. For another example, a function identifier appears as an argument to an `interrupt` control which appeared in a previous calling-convention or `interrupt` control, or the function identifier has been previously used, defined, or declared.

`invalid built-in function`

Use i486™-specific built-in functions only with the `mod486` control. Use i386™-specific built-in functions only with the iC-386 compiler. See Chapter 6 for more information on built-in functions.

`invalid cast`

The following are examples of invalid casts:

- casting to or from `struct` or `union`
- casting a `void` expression to any type other than `void`

`invalid field definition`

A field definition appears outside a structure definition or is attached to an invalid type.

`invalid interrupt handler`

Interrupt handlers take no arguments and return no value (`void`).

`invalid interrupt number`

An interrupt number argument to the function `causeinterrupt` or to the control `interrupt` must be an integer constant in the range 0 to 255 for the iC-86 compiler. Only the iC-86 compiler generates this message. See Chapter 3 for more information on the syntax of compiler controls. See Chapter 6 for more information on interrupt functions.

`invalid member name`

The member name (that is, the right operand of a `.` or a `->`) is not a member of the corresponding structure or union.

`invalid number of parameters`

The number of actual arguments passed to a function does not match the number defined in the prototype of that function.

`invalid object type`

An invalid object type has been detected in a declaration, for example `void array[5];`.

`invalid pointer arithmetic`

The only arithmetic allowed on pointers is to add or subtract an integral value from a pointer, or to subtract two pointers of the same type. Any other arithmetic operation is illegal.

`invalid redeclaration` *`name`*

An object is being redeclared, but not with the same type. For example, a function reference implicitly declares the function as a function returning an `int`. If the actual definition follows, and it is different, it is an error.

`invalid register number`

Only certain of the 386 or i486 processor special registers are available for use in built-in functions. The register number specified must be a numeric constant. See Chapter 6 for more information on the 386 and i486 processor special registers.

`invalid storage class`

The storage class is invalid for the object declared. For example, `alien` can be used only for external procedures, or a module-level object cannot be `auto` or `register`.

`invalid storage class combination`

You cannot have more than one storage class specifier in a declaration.

`invalid structure reference`

The left operand of a `.` is not a structure or a union; or the left operand of a `->` is not a pointer to a structure or a pointer to a union. This error message also occurs if an assignment is made from one structure to another of a different type.

`invalid type`

An invalid combination of type modifiers was specified.

`invalid type combination`

An invalid combination of type specifiers was specified.

`invalid use of void expression`

An expression of data type `void` was used in an expression.

`left operand must be lvalue`

The left operand of an assignment operator, and of the `++` and `--` operators, must be an "lvalue;" that is, it must have an address.

`limit exceeded: number of externals`

The number of external declarations has exceeded the compiler limit. See Chapter 10 for information on the applicable limits.

`macro expansion buffer overflow`

Insufficient memory exists for expansion of a macro; the macro is not expanded.

`macro expansion too nested`

The maximum nesting level of macro expansion has been exceeded. See Chapter 10 for information on the applicable limits. Macro recursion, direct or indirect, can also cause this error.

`member of unknown size`

The data type of a member of a structure is not sufficiently specified.

`missing left brace`

The initialization data for an aggregate object (array, structure, or union) must be enclosed by at least one pair of braces.

`multiple parameters for a macro`

Two parameters in the definition of a macro are identical. Every parameter must be unique in its macro definition.

`nesting too deep`

See Chapter 10 for information on nesting level limits.

`newline in string or char constant`

The new-line character can appear in a string or character constant only when it is preceded by a backslash (\).

`no more room for macro body`

Parameter substitution in the macro has increased the number of characters to more than maximum allowed. See Chapter 10 for information on the applicable limits.

`non addressable operand`

The & operator is used illegally, such as, to take an address of a register or of an expression.

`non-constant case expression`

The expression in a `case` is not a constant.

`nothing declared`

A data type without an associated object or function name is specified.

`number of arguments does not match number of parameters`

The number of arguments specified for the macro expansion does not match the number of parameters specified in the macro definition.

`operand stack overflow`

An illegal constant expression exists in a preprocessor directive line.

`operand stack underflow`

An illegal constant expression exists in a preprocessor directive line.

`operator not allowed on pointer`

An operand is a pointer, but the operator requires non-pointer integral operands (e.g., &, |, ^, *, /, %, >>, <<).

`operator stack overflow`

An illegal constant expression exists in a preprocessor directive line.

`operator stack underflow`

An illegal constant expression exists in a preprocessor directive line.

`parameter list can not be inherited from typedef`

A function body was supplied for an identifier that has function type, but whose type was specified via a `typedef` identifier, as in the following example:

```
typedef void func(void);
func f {}
```

`parameters can't be initialized`

An attempt was made to initialize the parameters in a function definition.

`procedure too complex for optimize (2)`

The combined complexity of statements, user-defined labels, and compiler-generated labels is too great. Simplify as much as possible, breaking the function into several smaller functions, or specify a lower level of optimization. See the `optimize` entry in Chapter 3 for more information on optimization.

`program too complex`

The program has too many complex functions, expressions, and cases. Break it into smaller modules.

`real expression too complex`

The real stack has eight registers. Heavily nested use of real functions with real expressions as arguments is excessively complex. Simplify as much as possible.

`respecified storage class`

A storage class specifier is duplicated in a declaration.

`respecified type`

A type specifier is duplicated in a declaration.

`respecified type qualifier`

A type qualifier is duplicated in a declaration.

`sizeof invalid object`

An implicit or explicit `sizeof` operation is needed on an object with an unknown size. Examples of invalid implicit `sizeof` operations are *`*p++`*, where *`p`* is a pointer to a function, or `struct` *`sigtype siga`*, when *`sigtype`* is not yet completely defined.

`statement is too large`

A statement is too large for the compiler. Break it into several smaller statements.

`string too long`

A string of over 1024 characters is being defined.

`syntax error near 'string'`

A syntax error occurred in the program. The `near` *`string`* information attempts to identify the error more precisely.

`too many active cases`

The limit of active cases in an uncompleted `switch` statement was exceeded. See Chapter 10 for information on the applicable limits.

`too many active functions`

The number of function calls within a single expression has exceeded the compiler limit. See Chapter 10 for information on the applicable limits.

`too many characters in a character constant`

A character constant can include one to four characters. The effect of this error on the object code is that the character constant value remains undefined. See Chapter 10 for information on character constant size for your target processor.

`too many cross-references, data truncated`

The cumulative number of cross-references exceeded the compiler's internal limit. Cross-references appear in the symbol table listing when the `xref` control is active.

`too many externals`

Too many external identifiers were declared. See Chapter 10 for information on the applicable limits.

`too many functions`

Too many functions were declared. See Chapter 10 for information on the applicable limits.

`too many initializers`

An array is initialized with more items than the number of elements specified in the array definition.

`too many macro arguments`

The maximum number of arguments specified for a macro was exceeded. See Chapter 10 for information on the applicable limits.

`too many nested calls`

The nesting limit for functions called in function argument lists has been exceeded. See Chapter 10 for information on the applicable limits.

`too many nested struct/unions`

The lexical nesting of `struct` and `union` member lists is limited to a depth of 32.

`too many parameters for one function`

The maximum number of parameters specified for one function was exceeded. See Chapter 10 for information on the applicable limits.

`too many parameters for one macro`

The maximum number of parameters specified for one macro was exceeded. See Chapter 10 for information on the applicable limits.

`unbalanced conditional compilation directive`

Conditional compilation directives are improperly formed. For example, the program contains too many `#endif` preprocessor directives, or an `#else` preprocessor directive without a matching `#if` preprocessor directive.

`undefined identifier: identifier`

The program contains a reference to an identifier that has not been previously declared.

`undefined label: label`

A label has been referenced in the function, but has never been defined.

`undefined or not a label`

An identifier following a `goto` must be a label; the identifier was declared otherwise, or the identifier was declared as a label but was not defined.

`undefined parameter`

The argument being defined did not appear in the formal parameter list of the function.

`unexpected EOF`

The input source file or files ended in the middle of a token, such as a character constant, string literal, or comment.

`unit string literal too long; truncated`

The maximum length of a string is 1024 characters.

`variable reinitialization`

An initializer for this variable was already processed.

`void function cannot return value`

A return with an expression is encountered in a function that is declared as type `void`. In such functions, all returns must be without a value.

11.3 Warnings

Warnings have the following syntax:

*** WARNING AT LINE *nn* OF *filename*: *message*

Where:

filename	is the name of the file in which the warning occurred.
nn	is the source line number where the warning is detected.
message	is the explanation.

Following is an alphabetic list of warnings.

`a #endif directive is missing`

At least one `#endif` preprocessor directive is missing at the end of the input source file(s). The `#if`, `#elif`, and `#endif` preprocessor directives are not balanced.

`an old builtin header file has been used`

A built-in header file from a previous release of the compiler has been used. Obtain the built-in header file provided with this release and use it.

`argument expected for` *`control`* `control`

A compiler control is specified without the argument required by context. A missing required argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if it occurs in a `#pragma` directive.

`bad octal digit:` *`hex_value`* `(hex)`

An octal number contains a non-octal character. The *`hex_value`* is the ASCII value of the illegal character.

`comment extends across the end of a file`

A comment that is started in a file is not closed before the end of the file.

`control` `control cannot be negated`

The prefix `no` cannot be specified for this compiler control. Improper negating is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if it occurs in a `#pragma` directive. See Chapter 3 for a list of compiler controls that can be negated.

`control` `control not allowed in pragma`

The compiler encountered either a `define` or an `include` control in a `#pragma` preprocessor directive.

`different enum types`

An attempt was made to assign one `enum` type to a different `enum` type.

`directive line too long`

The line length limit for `#pragma` preprocessor directives was exceeded. See Chapter 10 for information on the applicable limits.

`division by 0`

A division by the constant `0` was specified.

`escape sequence value overflow`

The escape sequence is undefined.

`export ignored: identifier`

An identifier that is an enumeration constant appeared in the `EXPORTS` list of a subsystem specification. An enumeration constant cannot be `far`. See Chapter 9 for information on subsystems.

`exported identifier: identifier`

An identifier that is either a built-in or appears as an argument to the `interrupt` control, appears also in the `EXPORTS` list of a subsystem specification.

`extra characters in pragma ignored: string`

The *`string`* represents characters that the compiler cannot process as part of the current `#pragma`.

`filename too long; truncated`

The filename length exceeded the limit of the operating system.

`illegal character in header name: hex_value (hex)`

An illegal character was found in the header name of an `#include < >` preprocessor directive.

`illegal character: hex_value (hex)`

The character with the ASCII value *hex_value* is not part of the iC-86/286/386 character set.

`illegal escape sequence`

The sequence following the backslash is not a legal escape sequence. The compiler ignores the backslash and prints the sequence.

`illegal syntax in a directive line`

A preprocessor directive line is not terminated with a new-line character.

`illegal syntax in a directive line - newline expected`

A preprocessor directive line is not terminated with a new-line character.

`indirection to different types`

A pointer to one data type was used to reference a different data type.

`initializing with ROM option in effect`

When a program is placed in ROM, initialization of a variable that does not have the `const` type qualifier has no effect. See Chapter 3 for more information on the compiler controls `ram` and `rom`.

`invalid control syntax`

Invalid control syntax is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if it occurs in a `#pragma` directive. See Chapter 3 for more information on the syntax of compiler controls.

`invalid decimal parameter: value`

Non-decimal characters were found in an argument that requires a decimal value. See Chapter 3 for more information on the syntax of compiler controls. Invalid non-decimal argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the invalid argument occurs in a `#pragma` directive.

`invalid identifier: identifier`

An identifier does not follow the rules for forming identifiers in C. An invalid identifier is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the invalid identifier occurs in a `#pragma` directive.

`invalid syntax for control control`

Invalid syntax is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the invalid syntax occurs in a `#pragma` directive. See Chapter 3 for more information on the syntax of compiler controls.

`missing or misplaced right parenthesis`

A right parenthesis is required to delimit arguments to a compiler control. See Chapter 3 for more information on the syntax of compiler controls. Improper right parenthesis is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the missing or misplaced parenthesis occurs in a `#pragma` directive.

`null argument for control control`

Null arguments for compiler controls are not allowed. See Chapter 3 for more information on the syntax of compiler controls. For example, the following is illegal:

```
align(siga=2,,sigb=2)
```

A null argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the null argument occurs in a `#pragma` directive.

`parameter not allowed for control control`

An argument or arguments were passed to a control that accepts none. See Chapter 3 for more information on the syntax of compiler controls. Improper argument passing is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the argument occurs in a `#pragma` directive.

`parameter not allowed for negated` *`control`* `control`

Negated controls generally do not accept arguments (`noalign` is the only exception). See Chapter 3 for more information on the syntax of compiler controls. Improper argument for negated control is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the argument occurs in a `#pragma` directive.

`parameter out of range for` *`control`* `control:` *`parm`*

An argument or arguments were passed that were out of the specified range for the parameter. See Chapter 3 for more information on the range of values accepted by various compiler controls. An out of range argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the argument occurs in a `#pragma` directive.

`parameter required for` *`control`* `control`

A missing required argument is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if the argument occurs in a `#pragma` directive. See Chapter 3 for more information on the syntax of compiler controls.

`pointer extension`

An integral expression is being converted to a far pointer type, and the current value of `DS` is being inserted as the selector part. Later operations using this value, particularly comparison against the `NULL` constant, may not give correct results.

`pointer truncation`

A far pointer expression is being converted to a narrower type, which cannot represent the value of the selector part of the pointer. Later indirection using this value can give incorrect results.

`pragma ignored`

An entire `#pragma` preprocessor directive was ignored as a result of an error. Whenever an error is found in a `#pragma` preprocessor directive, the diagnostic is followed by either this message or `remainder of pragma ignored`, whichever is appropriate. This message is usually paired with one of several other messages.

`predefined macros cannot be deleted/redefined`

The predefined macros (e.g., `__LINE__` or `__FILE__`) cannot be deleted or redefined by the preprocessor directives `#define` or `#undefine`.

`remainder of pragma ignored`

This message indicates that a `#pragma` preprocessor directive is partially ignored as a result of an error. Whenever an error is found in a `#pragma` preprocessor directive, the message is followed by either this message or `pragma ignored`, whichever is appropriate. This message is usually paired with one of several other messages.

`subsys control conflicts with codeseg/dataseg control`

A `subsys` control cannot occur while the `codesegment` or `datasegment` control is in effect, and vice versa. The preprocessor detected both controls in `#pragma` preprocessing directives.

`token too long; ignored from character:` *`hex_value`* `(hex)`

A character sequence was too long; such as an identifier or a macro argument.

`too many alignment specifiers for this tag:` *`structure_tag`*

Alignment has already been specified for this *`structure_tag`*, either in the current or in a previous align control. Redundant alignment specification is a fatal error if it occurs in the compiler invocation, but the preprocessor only issues a warning if it occurs in a `#pragma` directive.

`zero or negative subscript`

In an array declaration, the value of an array subscript must be a positive integer.

11.4 Remarks

Remarks have the following syntax:

```
*** REMARK AT LINE nn OF filename: message
```

Where:

filename	is the name of the file in which the remark occurred.
nn	is the source line number where the remark is detected.
message	is the explanation.

Following is an alphabetic list of remark messages.

`a constant in a selection statement`

A constant is encountered in the expression of a selection statement such as an `if`, `else`, or `switch` statement.

`implicit function declaration`

The function is used without any previous declarations.

`invalid number of parameters`

The actual number of arguments in a function call do not agree with the number of parameters in a function definition that is not a prototype.

`return statement has no expression`

A return statement with no return expression is encountered in a function definition which returns an expression other than `void`.

`statement has no apparent effect`

A statement that does not have any effect in the source code is encountered, as in the following example:

```
var + 1;
```

`the characters /* are found in a comment`

A comment-start delimiter (`/*`) occurs between a comment-start delimiter and a comment-end delimiter (`*/`).

11.5 Subsystem Diagnostics

Subsystem diagnostic messages have the following syntax:

```
*** ERROR AT LINE nn OF filename: message
```

Where:

filename	is the name of the primary source file or include file in which the error occurred.
nn	is the source line number where the error is detected.
message	is the explanation.

Following is an alphabetic list of subsystem diagnostic messages.

`conflicting segmentation controls`

More than one segmentation control affecting the module being compiled was encountered. One common cause is specifying both `-const in code-` in a subsystem definition and the `rom` control.

`illegal identifier in subsystem specification`

An identifier was encountered that does not follow rules for PL/M identifiers. See Chapter 9 for information on subsystem identifiers.

`invalid control`

An unrecognized control is in the subsystem definition. See Chapter 9 for more information on subsystem definitions.

`subsystem already defined`

The subsystem name has already been defined.

`symbol exists in more than one has list`

A module name can occur in only one `HAS` list.

`unexpected end of control`

A subsystem definition was expecting a continuation line or a right parenthesis.

11.6 Internal Error Messages

Internal error messages have the following syntax:

```
internal error: message
```

If your compilation consistently produces any of these errors, contact your Intel representative.

Installation

This section provides the information you need to install the iC-86, iC-286, or iC-386 compiler and libraries on a DOS host system.

Hardware

All of Intel's development tools for DOS require an IBM compatible PC (XT- or AT- class) or an Intel386™ or Intel 486™ processor-based workstation with a recommended minimum 512 kilobytes of RAM and DOS V3.1 or later. Use your PC host system with Intel development tools as a program development workstation.

Use `chkdsk` to determine whether your system has enough available memory and disk space, as follows:

```
C:> chkdsk
Volume name    created date time

 33435648 bytes total disk space
    53248 bytes in 3 hidden files
   227328 bytes in 106 directories
 27154432 bytes in 1448 user files
   133120 bytes in bad sectors
  5867520 bytes available on disk

   655360 bytes total memory
   554864 bytes free
```

If your system contains expanded memory and the expanded memory manager LIM 3.2 (or higher) is present, iC-86/286/386 uses available expanded memory prior to spilling its tables to disk.

Installation on DOS Systems

The iC-86/286/386 compiler and libraries product is supplied on 5-1/4" diskettes and 3-1/2" diskettes. Before installation, make a backup copy of the product diskettes using the DOS `diskcopy` command.

The installation program `install.exe`, found on the first distribution disk, installs the compiler and libraries on your DOS host system. The program creates directories if needed and copies files into them. At certain points during the installation, you can:

- name a base subdirectory under which all files will be copied (some into subdirectories), or choose the default.
- choose to install all or selected parts of the product, such as the header files. You can rerun the installation program later to install additional parts of iC-86/286/386 if you need them.
- create files named `autoexec.new` and `config.new` that contain the modified `path`, `files`, and `buffers` commands. These changes simplify invocation and ensure efficient operation of the compiler. You can rename the files `autoexec.bat` and `config.sys` if you choose to use the new files, or copy the values into your own files.

NOTE

The installation program used to install the iC-86/286/386 product, *INSTALL*, is licensed software provided by Knowledge Dynamics Corporation, Highway Contract 4 Box 185-H, Canyon Lake, Texas 78133-3508 (USA). *INSTALL* is provided to you exclusively for installing the iC-86/286/386 software.

Software

After compiling your source code, you need the appropriate Intel utilities to link or bind the object modules into an application:

- For iC-86, you need the 86,88 utilities.
- For iC-286, you need the 286 utilities.
- For iC-386, you need the Intel386™ family utilities.

To execute applications that use floating-point arithmetic, you need an *n*87 numeric coprocessor (or i486 with on-chip FPU) or a true software emulator.

Glossary

Absolute address	An address in memory relative to the beginning of memory.
Access attributes 286 and higher processors	Characteristics which define the type of segment access allowed: read-only data, read-write data, execute-read code, or execute-only code. These attributes are represented by bits 41 (Writable/Readable) and 43 (Executable) in the segment descriptor.
Aggregate data type	A data type that is a collection of scalar and sometimes aggregate data types, treated either as a unit, or as individual scalar or aggregate data types.
Alignment (of an object)	The allocation of an object in memory relative to byte, even-byte, or 4-byte addresses and boundaries.
Alignment (of a segment) 86 processors	The allocation of a segment in memory relative to byte, word, paragraph, or page addresses and boundaries.
Big-endian	A processor that stores multi-byte objects starting with the high-order byte at the lowest address.
Binder, BND286 and BND386	The utility that performs linking. The binder combines segments with like names and resolves symbolic addressing.

Build file 286 and higher processors	A file of system implementation definitions used by the system builder, BLD286 or BLD386, to create an absolutely-located system. The definitions describe system data structures, initial values for the system, and memory configuration.
Builder, BLD286 and BLD386	The utility that creates an absolutely-located system from linkable input modules and system definitions in a build file.
Calling convention	The set of instructions that the compiler inserts in object code to handle parameter passing, stack and register use, and return values in a function call.
Code segment	A memory segment containing instructions and sometimes constants.
Compiler control	A directive you can specify in the compiler invocation.
Compiler invocation	The command that causes the compiler to begin execution.
Conditional compilation	Compiling only part of the source code, depending on the preprocessor's evaluation of conditions in the source code.
Cross-referenced symbol table	A symbol table containing source line-number reference information.
Current segment	The segment pointed to by a segment register at any particular time during execution.
D bit Intel386™ and Intel486™ processors	Bit 54 (B/D) in a segment descriptor. The D bit refers to the default operand size of a code segment. If the bit is 1, the default operand size is 32 bits. If the bit is 0, the default operand size is 16 bits.

Data register	One of four 16-bit registers (AX, BX, CX, or DX for 86 and 286 processors), or four 32-bit registers (EAX, EBX, ECX, or EDX for Intel386 and Intel486 processors); the processor usually uses data registers in arithmetic and logical operations.
Data segment	A segment containing data (e.g., variables and constants).
Data type	The format for representing a value.
Debugger	A development tool that enables you to observe and manipulate the step-by-step execution of your program.
Descriptor 286 and higher processors	An eight-byte data structure containing the base, limit, and access attributes for a given region of linear address space such as a segment, table, or task state segment.
Descriptor privilege level 286 and higher processors	Bits 29 and 30 in a segment descriptor. The segmentation hardware checks descriptor privilege levels on accesses to code and data segments to ensure that the referring code has sufficient privilege.
Development tool	Any product used for application development.
EFLAGS register Intel386 and Intel486 processors	The processor register containing indicators of the current state of the processor and of the result of the just-completed instruction.
Error	An exception that does not immediately terminate compilation but can cause an invalid object module.
EXE file iC-86 only	A DOS-executable file with a filename extension of `.EXE`.

Expand-down 286 and higher processors	A special kind of data segment useful for stacks. The expand-down attribute is in bit 42 of the segment descriptor. A software system can dynamically increase the expand-down segment size by lowering the limit in the segment descriptor.
External reference	A reference to a location in a different object module via a data pointer or function call.
Far	A reference from a location in one segment to a location in a different segment; an address with both the segment selector and offset specified.
Fatal error	An exception that terminates compilation; no object module is produced.
File type	The characteristics of a file reflected in the characters of the filename following the dot character.
Filename	The name of a file, including the device and directory path, if necessary.
Filename base	The part of a filename that is left of the ".".
Filename extension	The part of a filename that is right of the ".".
FLAGS register	The processor register containing indicators of the current state of the processor and of the result of the just-completed instruction. The low-order 16 bits of the EFLAGS register in Intel386 and Intel486 processors.
Gate 286 and higher processors	An eight-byte data structure used to regulate transfer of control to another code segment. A gate is sometimes called a descriptor because it has a layout similar to a segment descriptor. Gates provide indirection that allows the processor to perform protection checks.
General control	A compiler control that you can specify on the command line and in a `#pragma` preprocessor directive anywhere in the source code as often as necessary.

General register	Any of the data, pointer, or index registers.
Global descriptor table (GDT) 286 and higher processors	An array of descriptors defining segments and gates available for use by all tasks in the system. A software system contains only one global descriptor table.
Global descriptor table register (GDTR) 286 and higher processors	The system register that contains the base address and limit of the global descriptor table.
Group iC-86 only	Two or more segments concatenated and constrained to occupy together up to 64 kilobytes of memory.
Hardware flags	See FLAGS register and EFLAGS register.
Host system	The system on which the compiler executes. (See also target system.)
Identifier	The name you specify in your source code to refer to an object or function.
In-circuit emulator	A system of hardware and software that emulates the operation of a microprocessor or microcontroller within a target system.
Include files	The source files other than the primary source file; specified in the `include` compiler control or in the `#include` preprocessor directive.
Index register	One of two registers (SI or DI for 86 and 286 processors, or ESI or EDI for Intel386 and Intel486 processors) that you use for addressing operands during execution.
Instruction set	The executable elements of the object code.
Interrupt descriptor table (IDT) 286 and higher processors	An array of task, interrupt, and trap gates that act as interrupt vectors. A software system contains only one interrupt descriptor table.

Interrupt descriptor table register (IDTR) 286 and higher processors	The system register that contains the base address and limit of the interrupt descriptor table.
Interrupt handler	The function called when an interrupt occurs.
Listing controls	Controls which specify the names, locations, and contents of the output listing files.
Little-endian	A processor that stores multi-byte objects starting with the low-order byte at the lowest address.
Local descriptor table (LDT) 286 and higher processors	An array of descriptors defining segments and gates protected from use by all but specified tasks in the system. Tasks that have a pointer to a local descriptor in their task state segment can access that table. The global descriptor table can hold descriptors for local descriptor tables. A software system can contain many local descriptor tables.
Local descriptor table register (LDTR) 286 and higher processors	The system register that contains the selector for the descriptor of the currently active local descriptor table.
Lowercase	For ASCII characters a through z, the hexadecimal values 61 through 7A.
Machine status word (MSW) 286 and higher processors	A 16-bit register whose value indicates the configuration and status of the processor. In Intel386 and higher processors, the MSW is the low-order 16 bits of control register 0 (CR0).
Macro	A string that the preprocessor replaces with text you specify.
Module	A file of code in some stage of translation. An object module refers to the output of a translator, linker, binder, or system builder. An input module refers to a file in the form accepted by translating, binding, or building software.

Near	A reference from one location to another within the same segment; an offset-only address.
Numeric coprocessor	An 8087, 80C187, i287™, or i387™ coprocessor, the Intel486 processor on-chip floating-point unit, or a true software emulator.
Object	A variable, temporary variable, constant, literal, or macro. (See also object module.)
Object code	Executable instructions and associated data in binary format.
Object file	The file containing the object module that the compiler generates.
Object-file content controls	Controls which determine the internal configuration of the object file.
Object module	The formatted object code that the compiler generates.
Offset	The displacement; the number of units (usually bytes) away from the zeroth location in memory, or the number of units away from the base address of the enclosing segment or data structure.
Output listing	The print file and preprint file that the compiler generates.
Pathname	The name of a directory or file relative to a given directory.
Pointer registers	The base pointer (BP for 86 and 286 processors, or EBP for Intel386 and Intel486 processors) and stack pointer (SP for 86 and 286 processors, or ESP for Intel386 and Intel486 processors) registers.
Preprint file	A text file that the compiler generates, containing the intermediate source code after macro expansion, files included using the `include` control or the `#include` preprocessor directive, and conditional compilation.

Primary control	A compiler control that can only be specified once. When you specify it in a preprocessor directive, you must specify it before the first line of data definition or executable source code.
Primary source file	The file specified as the source file in a compiler invocation.
Primary source text	The contents of the primary source file.
Print file	A compiler-generated text file containing code listings, symbolic information, and information about the compilation.
Privilege level 286 and higher processors	One of four values in bits 45 and 46 of a segment or special descriptor: 0 (most privileged), 1, 2, or 3 (least privileged). The descriptor privilege level (DPL) of the currently executing code segment is also called the current privilege level (CPL).
Privileged instructions 286 and higher processors	Instructions that affect system registers or halt the processor. These instructions can only be executed when the current privilege level is 0.
Program	A set of compiled modules ready to be linked or located, or the complete associated source text.
Protected mode 286 and higher processors	A mode of execution where the protection-enable bit (PE) is on in the machine status word. The first far jump has been executed. This mode uses selectors and descriptors to calculate addresses.
Protection 286 and higher processors	The mechanisms implemented by the hardware of the processor, especially when the protection-enable bit (PE) is on and the first far jump has been executed. There are five basic kinds of protection available: type checking, limit checking, restricting addressable domain, restricting entry points, and restricting instruction set.

Protection-enable bit (PE) 286 and higher processors	Bit 0 in the machine status word. If PE is 1, the processor executes in protected mode. If PE is 0, the processor executes in real mode.
Qualifier	Invocation command element that controls the result of the invocation.
Real mode	The mode of execution of the 86 processor, or of higher processors with the protection-enable bit (PE) off. The 286 and higher processors execute in this mode upon reset, except the 376 processor executes in protected mode on reset.
Relative address	An offset into a segment, before the segment loads into memory.
Scalar data type	A data type treated as a single value.
Search path	A list of strings that the debugger uses as default prefixes of possible pathnames to a file.
Segment 286 and higher processors	A continuous piece of memory defined by a base address and a limit.
Segment register	One of the CS, SS, DS, and ES registers (or FS and GS registers in Intel386 and higher processors) containing a segment selector.
Segmentation model	The format used to combine object modules into individual or contiguous blocks of memory addressable by the processor determines the placement of constants and the number and names of segments generated by the compiler.
Selector 286 and higher processors	A system data structure used in computing an address that identifies a descriptor by specifying a descriptor table and an index to a descriptor within that table. A selector also contains a requested privilege level (RPL), which is the descriptor privilege level (DPL) of the referring segment.

Separately-compiled code	Individual object modules each resulting from its own compilation.
Source directory	The directory containing your primary source file.
Source-processing controls	Controls which specify the names and locations of input files or define macros at compile time.
Source text	Text you write in a programming language such as C.
Stack segment	A segment reserved for dynamic memory allocation for objects such as temporary variables and function activation records.
Symbol table	A chart in the print file containing symbolic information.
Symbolic debugger	See debugger.
Symbolic information	Information about the format, location, and identifier of an object or function.
System data structures 286 and higher processors	Descriptors, tables, gates, selectors, and task state segments.
Target system	The system on which your compiled program is intended to execute. (See also host system.)
Task 286 and higher processors	The code, data, and system data structures which collectively define a sequential thread of execution.
Uppercase	For ASCII characters `A` through `Z`, the hexadecimal values 41 through 5A.
Warning	A message indicating a situation that is probably unusual but that does not terminate compilation and probably does not invalidate the object module.

Word	Two bytes on all *n*86 processors. In C programming, a word is the amount of storage reserved for an integer, which is 16 bits for iC-86 and iC-286 and 32 bits for iC-386. The Intel386 and Intel486 processor documentation and ASM386 instruction sets refer to a 16-bit word and a 32-bit double word.
Work file	A file that the compiler creates, uses, and deletes during compilation.

Index

B

C

G

H

I

K-L

M

N

O

P

Q

R

S

T

U

V

W

X