IC-386 Compiler User's Guide

RadiSys Corporation
5445 NE Dawson Creek Drive
Hillsboro, OR 97124
(503) 615-1100
FAX: (503) 615-1150
www.radisys.com
07-0577-01
December 1999

December 1999

Quick Contents

Chapter 1. Overview

Chapter 2. Compiling and Binding

Chapter 3. Compiler Controls

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

Glossary

Index

Notational Conventions

The *iC-386 Compiler User's Guide* uses the following notational conventions:

italics	Italics indicate a symbol that is replaced with an identifier, an expression, or a value.
monospace type	Type of this style represents syntax, filenames, program examples, or computer output.

Contents

1 Overview

Software Development With iC-386 1
- Using the Run-time Libraries 2
- Debugging 2
- Optimizing 4
- Using the Utilities 4
- Programming for Embedded ROM Systems 5

Compiler Capabilities 5
Compatibility With Other Development Tools 6
About This Manual 7
- Related Publications 7

2 Compiling and Binding

Using Files and Directories 9
Invoking the iC-386 Compiler 10
- Invocation Syntax on iRMX Systems 10
- Invocation Syntax on DOS Systems 11
- Sign-on and Sign-off Messages 12

Files That the Compiler Uses 13
- Work Files 14
- Object File 14
- Listing Files 14

Using Submit, Batch and Command Files 17
- Using iRMX Submit Files 17
- Using DOS Batch Files for DOSRMX Systems 18
- Using DOS Command Files in DOSRMX Systems 20

Binding Object Files 22
- Choosing the Files to Bind 22
- Examples of Binding 24
  - BND386 Example on DOS Systems 24

In-line Functions 25
Compiling at Different Optimization Levels 27

Results at Optimization Level 0 27
Results at Optimization Level 1 31
Results at Optimization Level 2 33
Results at Optimization Level 3 35

3 Compiler Controls

How Controls Affect the Compilation 39
Where to Use Controls 40
Alphabetical Reference of Controls 44
align | noalign 45
code | nocode 51
codesegment 53
compact 54
cond | nocond 56
datasegment 57
debug | nodebug 58
define 60
diagnostic 62
eject 64
extend | noextend 65
fixedparams 66
include 69
interrupt 71
line | noline 72
list | nolist 73
listexpand | nolistexpand 75
listinclude | nolistinclude 76
long64 | nolong64 78
mod486 | nomod486 79
modulename 81
object | noobject 82
optimize 84
pagelength 88
pagewidth 89
preprint | nopreprint 90
print | noprint 92
ram | rom 94
searchinclude | nosearchinclude 96
signedchar | nosignedchar 98
srclines | nosrclines 99
subsys 100
symbols | nosymbols 102

tabwidth 103
title 104
translate | notranslate 105
type | notype 106
varparams 108
xref | noxref 111

4 Segmentation Memory Models

How the Binder Combines Segments 113
Combining iC-386 Segments With BND386 114
How Subsystems Extend Segmentation 114
Compact Segmentation Memory Model 115
Compact Model 116
Using near and far 118
Addressing Under the Segmentation Models 119
Using far and near in Declarations 120
Examples Using far 121

5 Listing Files

Preprint File 125
Macros 126
Include Files 127
Conditional Compilation 128
Propagated Directives 128
Print File 129
Print File Contents 129
Page Header 130
Compilation Heading 130
Source Text Listing 131
Remarks, Warnings, and Errors 132
Pseudo-assembly Listing 132
Symbol Table and Cross-reference 133
Compilation Summary 133

6 Processor-specific Facilities

Making Selectors, Far Pointers, and Near Pointers 138
Using Special Control Functions 139
Examining and Modifying the FLAGS Register 140
Examining and Modifying the Input/Output Ports 144
Enabling and Causing Interrupts 146

Interrupt Handlers 146
Protected Mode Features of Intel386 and Higher Processors 148
Manipulating System Address Registers 148
Manipulating the Machine Status Word 150
Accessing Descriptor Information 152
Adjusting Requested Privilege Level 158
Manipulating the Control, Test, and Debug Registers of Intel386™, Intel486™, and Pentium® Processors 159
Managing the Features of the Intel486 and Pentium Processors 162
Manipulating the Numeric Coprocessor 163
Tag Word 165
Control Word 166
Status Word 168
Intel387™ Numeric Coprocessor, and Intel486 or Pentium Processor FPU Data Pointer and Instruction Pointer 172
Saving and Restoring the Numeric Coprocessor State 173

7 Assembler Header File

Macro Selection 175
Flag Macros 181
Register Macros 182
Segment Macros 183
Type Macros 185
Operation Macros 186
External Declaration Macros 186
Instruction Macros 187
Conditional Macros 188
Function Definition Macros 188
%function 189
%param 190
%param_flt 191
%auto 192
%prolog 193
%epilog 194
%ret 195
%endf 196

8 Function-calling Conventions

Passing Arguments 199
FPL Argument Passing 200
VPL Argument Passing 201

Returning a Value 202
Saving and Restoring Registers 203
Cleaning Up the Stack 204

9 Subsystems

Dividing a Program into Subsystems 206
Segment Combination in Subsystems 209
Compact-model Subsystems 209
Efficient Data and Code References 210
Creating Subsystem Definitions 211
Open and Closed Subsystems 211
Syntax 212
Example Definitions 217
Creating Three Compact-model RAM Subsystems 217

10 Language Implementation

Data Types 221
Scalar Types 222
Aggregate Types 224
Void Type 224
iC-386 Support for ANSI C Features 225
Lexical Elements and Identifiers 225
Preprocessing 225
Implementation-dependent iC-386 Features 227
Characters 227
Integers 227
Floating-point Numbers 228
Arrays and Pointers 228
Register Variables 229
Structures, Unions, Enumerations, and Bit Fields 229
Declarators and Qualifiers 230
Statements, Expressions, and References 231
Virtual Symbol Table 231

11 Messages

Fatal Error Messages 234
Error Messages 239
Warnings 249
Remarks 254
Subsystem Diagnostics 255
Internal Error Messages 256
iRMX Condition Codes in Error Messages 256

Glossary 271

Index 279

Tables

1-1. Assemblers, Compilers, Debuggers, and Utilities 19
1-2. Intel386, Intel486, or Pentium Processor and Tool Publications..... 20
2-1. In-line Functions..... 38
3-1. Compiler Controls Summary 53
3-2. Compiler Exit Status..... 75
4-1. iC-386 Segment Definitions for Compact-model Modules 128
4-2. Segmentation Models and Default Address Sizes 131
5-1. iC-386 Predefined Macros..... 138
5-2. Controls That Affect the Print File Format..... 141
5-3. Controls That Affect the Source Text Listing..... 144
6-1. Built-in Functions in i86.h..... 148
6-2. Built-in Functions in i186.h..... 148
6-3. Built-in Functions in i286.h..... 149
6-4. Built-in Functions in i386.h..... 149
6-5. Built-in Functions in i486.h..... 149
6-6. Flag Macros..... 154
6-7. Machine Status Word Macros 163
6-8. General Descriptor Access Rights Macros 167
6-9. Segment Descriptor Access Rights Macros..... 168
6-10. Special Descriptor Access Rights Macros..... 169
6-11. Control Register 0 Macros for Intel386, Intel486, and Pentium Processors 174
6-12. Numeric Coprocessor Tag Word Macros..... 178
6-13. Numeric Coprocessor Control Word Macros..... 180
6-14. Intel387 Numeric Coprocessor, and Intel486 or Pentium Processor FPU Condition Codes 183
6-15. Numeric Coprocessor Status Word Macros..... 185
7-1. Assembler Header Controls for Macro Selection 190
7-2. Assembler Flag Macros Set by Header Controls..... 195
7-3. Assembler Register Macros..... 196
7-4. ASM386 Segment Macro Expansion for Compact Memory Model..... 197
7-5. ASM386 Type Macro Expansion for Compact Memory Model 199
7-6. ASM386 External Declaration Macro Expansion for Compact Memory Model..... 201
8-1. iC-386 FPL and VPL Return Register Use..... 216
8-2. iC-386 FPL and VPL Register Preservation..... 217
9-1. iC-386 Segment Definitions for Compact-model Subsystems..... 224
9-2. Subsystems and Default Address Sizes 225
10-1. Intel386 Processor Scalar Data Types..... 237

Figures

1-1. 32-bit Protected Mode iRMX Application Development 15
2-1. Input and Output Files 25
2-2. Controls That Create or Suppress Files 27
2-3. Redirecting Input to a DOS Batch File 31
2-4. Choosing Libraries to Bind with iC-386 Modules 35
2-5. Pseudo-assembly Code at Optimization Level 0 40
2-6. Part of the Pseudo-assembly Code at Optimization Level 1 43
2-7. Part of the Pseudo-assembly Code at Optimization Level 2 45
2-8. Part of the Pseudo-assembly Code at Optimization Level 3 47
3-1. Effect of iC-386 align Control on Example Structure Type 61
3-2. Effect of iC-386 noalign Control on Example Structure Type 62
4-1. Creating a Compact RAM Program 129
4-2. Creating a Compact ROM Program 130
6-1. FLAGS and EFLAGS Register 153
6-2. Gate Descriptor 159
6-3. Machine Status Word 162
6-4. Segment Descriptor 164
6-5. Special Descriptor 167
6-6. Selector 170
6-7. Control, Test, and Debug Registers of Intel386, Intel486, and Pentium Processors 172
6-8. Control Register 0 of Intel386, Intel486, and Pentium Processors 174
6-9. Numeric Coprocessor Stack of Numeric Data Registers 176
6-10. Intel387 Numeric Coprocessor or Intel486 and Pentium Processor FPU Environment Registers 177
6-11. Numeric Coprocessor Tag Word 178
6-12. Numeric Coprocessor Control Word 179
6-13. Numeric Coprocessor Status Word 182
6-14. Intel387 Numeric Coprocessor, and Intel486 or Pentium Processor FPU Data Pointer and Instruction Pointer 186
7-1. Precedence Levels of Assembler Header Controls 193
8-1. Four Sections of Code for a Function Call 212
9-1. Subsystems Example Program Structure 220
9-2. Subsystems Example Program in Regular Compact Segmentation Memory Model 221
9-3. Subsystems Example Program Using Small-model Subsystems 222

Overview 1

This chapter provides an overview of the iC-386 compiler and run-time libraries (referred to as iC-386) and their role in developing applications. References throughout the chapter direct you to more detailed information. This chapter contains information on:

- Development of an application using an iC-386 compiler and related RadiSys development tools
- Compiler capabilities
- Compatibility with other translators and utilities
- This manual and related publications

Software Development With iC-386

The iC-386 compiler supports modular, structured development of applications. Figure 1-1 shows the development paths using the iC-386 compiler. Some of the tasks in developing a modular, structured iC-386 application are:

- Compile and debug application modules separately.
- Select appropriate optimizations for the code.
- Use BND386 to bind the compiled modules and libraries to create a loadable file. Use BLD386 to create a bootloadable file for ICU-configurable iRMX® systems.

  See also: Examples of binding, in Chapter 2
- Use OH386 to prepare the code for programming into ROM.
- For ICU-configurable systems, use the interactive configuration utility (ICU) to combine an application with the first level or I/O layer of an iRMX system.
- Use the Soft-Scope debugger to debug your application. You can also use an ICE in-circuit emulator or the iRMX Bootstrap Loader and the iRMX System Debugger.

Using the Run-time Libraries

The iRMX Operating System (OS) C library and interface library support the entire ANSI C library definition and provide 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.

See also: *C Library Reference* for description of the iC-386 libraries, supplementary functions and macros
Library file names, binding, in Chapter 2

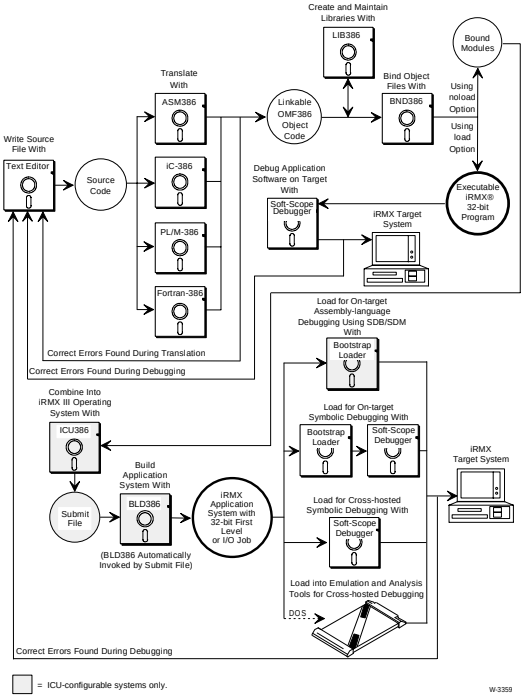

Figure 1-1. 32-bit Protected Mode iRMX Application Development

Debugging

At logical stages in the application development, use a source-level symbolic debugger such as Soft-Scope or an in-circuit emulator to debug and test the

application. iC-386 supports debugging by enabling you to specify the amount of symbolic information in the object code and to customize the output listing. Use these 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 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-386 compiler is compatible with current versions of Soft-Scope and in-circuit emulators capable of loading the object module format (OMF).

See also: Detailed descriptions of each control, in Chapter 3

Optimizing

Optimized code is more compact and efficient than unoptimized code. The iC-386 compiler has several controls to adjust the level of optimization performed on your code. These controls adjust optimization:

- The `align` | `noalign` control specifies whether to generate aligned STRUCTS or non-aligned STRUCTS.
- The `optimize` control specifies the level of optimization the compiler performs when generating object code. The iC-386 compiler provides 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.
- The `compact` control sets the memory segmentation model.

See also: Memory segmentation model in Chapter 4, examples of code generated at each optimization level in Chapter 2, and detailed descriptions of each control in Chapter 3

Using the Utilities

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

- LIB386 organizes frequently used object modules into libraries.
- BND386 binds together object modules from the translators. The binder produces a relocatable loadable module or a module for incremental binding.
- For ICU-configurable systems, BLD386 locates or builds an executable, bootloadable system.
- OH386 converts object code into hexadecimal form for programming into ROM.
- For ICU-configurable systems, use the Interactive Configuration Utility (ICU) to generate a submit file that builds the final application system. In iRMX applications, the submit file automatically invokes BLD386 to assign the absolute addresses to the application.

See also: LIB386, BND386, and OH386, *Intel386 ™ Family Utilities User's Guide*
BND386, *Intel386 Family System Builder User's Guide*

Programming for Embedded ROM Systems

This section only applies to ICU-configurable systems.

Use the `rom` compiler control to locate constants with code in the object module. Bind your object modules with startup code. Use the 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 OH386 utilities convert absolute or OMF386 code into hexadecimal form for use with non-Intel PROM programming utilities.

See also: `ram | rom` control description in Chapter 3

Compiler Capabilities

The iC-386 compiler translates C source files and produces code for the Intel386, Intel486™ or Pentium® processors.

The executable programs can be targeted for these environments:

- An Intel386/Intel486/Pentium processor-based system running the iRMX OS
- A custom-designed Intel386/Intel486/Pentium processor-based system running the iRMX OS

The iC-386 compiler generates floating-point instructions for the Intel387™ numeric coprocessor, and the Intel486 or Pentium microprocessor floating-point unit.

The iC-386 compiler conforms 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 also: `extend` control description in Chapter 3

Compatibility With Other Development Tools

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

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

Tool	Tool Name for Each Intel386, Intel486, or Pentium Processor
assembler	ASM386
C compiler	iC-386
FORTRAN compiler	Fortran-386
PL/M compiler	PL/M-386
Soft-Scope debugger	
binder	BND386
absolute locator	BLD386*
librarian	LIB386
cross-reference utility	MAP386
object-to-hex converter	OH386

* For ICU-configurable systems only

The iC-386 compiler is largely compatible with previous Intel C compilers. The `extend` control enables the compilers to recognize the `alien`, `far`, and `near` keywords.

See also: `extend` control description in Chapter 3, `far` and `near` keywords in Chapter 4, `alien` keyword in Chapter 10

Modules compiled by the iC-386 compiler can refer to object modules created with RadiSys assemblers and other RadiSys compilers. Use only RadiSys compilers or translators to ensure compatibility with the memory segmentation model of the application.

See also: Memory segmentation models in Chapter 4, facilities that aid interfacing with assembler modules in Chapter 7, function-calling conventions of iC-386 in Chapter 8

About This Manual

The *iC-386 Compiler User's Guide* describes how to use the iC-386 compiler in the iRMX and DOS environments. This manual applies to Versions 4.5 and later of the iC-386 compiler.

This manual does not teach either programming techniques or the C language.

Related Publications

Table 1-2 identifies additional publications that describe the other development tools you are most likely to use when programming with iC-386. The table also identifies the programmer's reference manuals for the processors for which the iC-386 compiler generates object code.

Table 1-2. Intel386, Intel486, or Pentium Processor and Tool Publications

Title	Contents
ASM386 Macro Assembler Operating Instructions	assembler operation
ASM386 Assembly Language Reference Manual	assembly language for the Intel386 and Intel486 processors
Intel386 Family System Builder User's Guide	utility for building complete systems
Intel386 Family Utilities User's Guide	utilities for binding, mapping, and maintaining libraries
80386 System Software Writer's Guide	advanced programming guidelines
386 DX Microprocessor Programmer's Reference Manual	Intel386 DX architecture and assembly language
387 DX Microprocessor Programmer's Reference Manual	Intel387 DX coprocessor architecture and numerics assembly instructions
Pentium Processor User's Manual	Intel Pentium processor operation and use (3 volume set)

See also: The *Customer Literature Guide* for part numbers and to identify other appropriate user's guides and manuals

❑❑❑

Compiling and Binding 2

This chapter provides the information you need to compile and bind an iC-386 program. If you are an experienced iRMX user and have used other Intel development tools, the most important information you need is in Invoking the iC-386 Compiler, and in Binding Object Files. Less experienced developers can obtain information on all of these topics:

- Invoking the compiler - syntax, compiler messages, and the files that the compiler uses
- Using iRMX submit files
- Using DOS batch and command files
- Binding object files - general syntax, how to choose the libraries you need, and examples
- Compiling an example at different optimization levels

See also: Various sample programs in the *rmx386\demo\c\intro* compiler directory

Using Files and Directories

The iRMX OS arranges files and directories in a hierarchical structure. You can reference a file or directory literally, by specifying the entire pathname, or indirectly, by specifying a logical name. A logical name has the format:

`:logicalname:`

The *logicalname* is a short name that represents a full pathname.

See also: Logical names, *Command Reference*

Invoking the iC-386 Compiler

This section describes the syntax for invoking the iC-386 compiler, the messages that the compiler displays on the screen, and the files that the compiler uses.

Invocation Syntax on iRMX Systems

On iRMX systems, the iC-386 compiler invocation command has the format:

```
ic386 sfile [controls]
```

Where:

`ic386` is an alias used to invoke the compiler. Case is not significant. The alias is:

```
run86 :lang:ic386
```

`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.

`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 also: Syntax of individual controls in Chapter 3

If you do not specify a logical name or pathname for the directory containing the compiler, the iRMX system searches through a list of directories. The search path is set at system configuration time. The `:lang:` directory is included in the default search path.

See also: iRMX directory structure, *Installation and Startup*
search path, *Command Reference*

This invocation line causes the iRMX system to expand the iC-386 alias and find the compiler in the directory specified by the iC-386 alias:

```
- ic386 demo.c
```

To continue an invocation command on another screen line, type the ampersand continuation character (`&`) at the end of each line, press <Enter>, and continue typing on the next screen line.

iRMX limits the invocation line to 80 characters. If your screen width is less than 80 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 <Enter>, and continue typing at the `**` prompt on the next screen line.

For example, this command on an iRMX system invokes the iC-386 compiler to compile the contents of the file `myprog.c` in the current directory (`:$:`) and print the title `Example Program` on each page of the listing:

```
- ic386 myprog.c &
** title("Example Program")
```

Invocation Syntax on DOS Systems

On DOS, the iC-386 compiler invocation has the format:

```
ic386 sfile [controls]
```

Where:

`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.

`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 also: Syntax of individual controls in Chapter 3

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.

Names of DOS directories and files are limited to eight characters preceding the optional period, plus a three-character extension. DOS truncates longer names from the right.

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 a message similar to this:

```
system-id iC-386 COMPILER Vx.y
Intel Corporation Proprietary Software
```

Where:

system-id
: identifies your host system.

V*x.y*
: identifies the version of the compiler.

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

```
iC-386 COMPILATION COMPLETE. x REMARKS, y WARNINGS, z ERRORS
```

Where:

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 also: `diagnostic` and `notranslate` control descriptions in Chapter 3

On abnormal termination, the compiler displays the message:

```
iC-386 FATAL ERROR --
message
COMPILATION TERMINATED
```

Where:

message
: describes the condition causing the fatal error.

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.

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 the compiler uses. The compiler also uses temporary work files during the compilation process. For DOSRMX systems, the DOS `config.sys` file, `files` specification controls the maximum number of files that DOS allows open at the same time.

See also: `preprint` and `include` control descriptions in Chapter 3, for information on how many files the compiler has open at one time

The installation utility for the compiler identifies necessary changes to your system configuration.

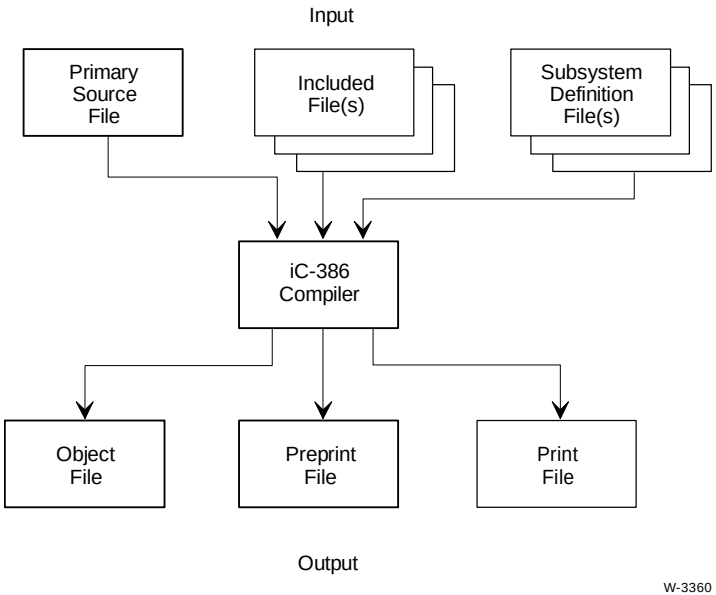

Figure 2-1. Input and Output Files

Work Files

The compiler creates and deletes 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, this 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. 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.

See also: Your DOS documentation for information on RAM disks and environment variables

Object File

By default, the compiler produces an object file. Use the `noobject` control or the `notranslate` control to suppress creation of an object file.

See also: `noobject` and `notranslate` control descriptions in Chapter 3

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

The object file contains the compiled object module, which is the relocatable code and data resulting from successful compilation. Compiler controls and preprocessor directives specify the information content and configuration of the object module.

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 also: Preprint and print files in Chapter 6;
`preprint` and `print` control descriptions in Chapter 3

Figure 2-2 summarizes the controls that create or suppress files.

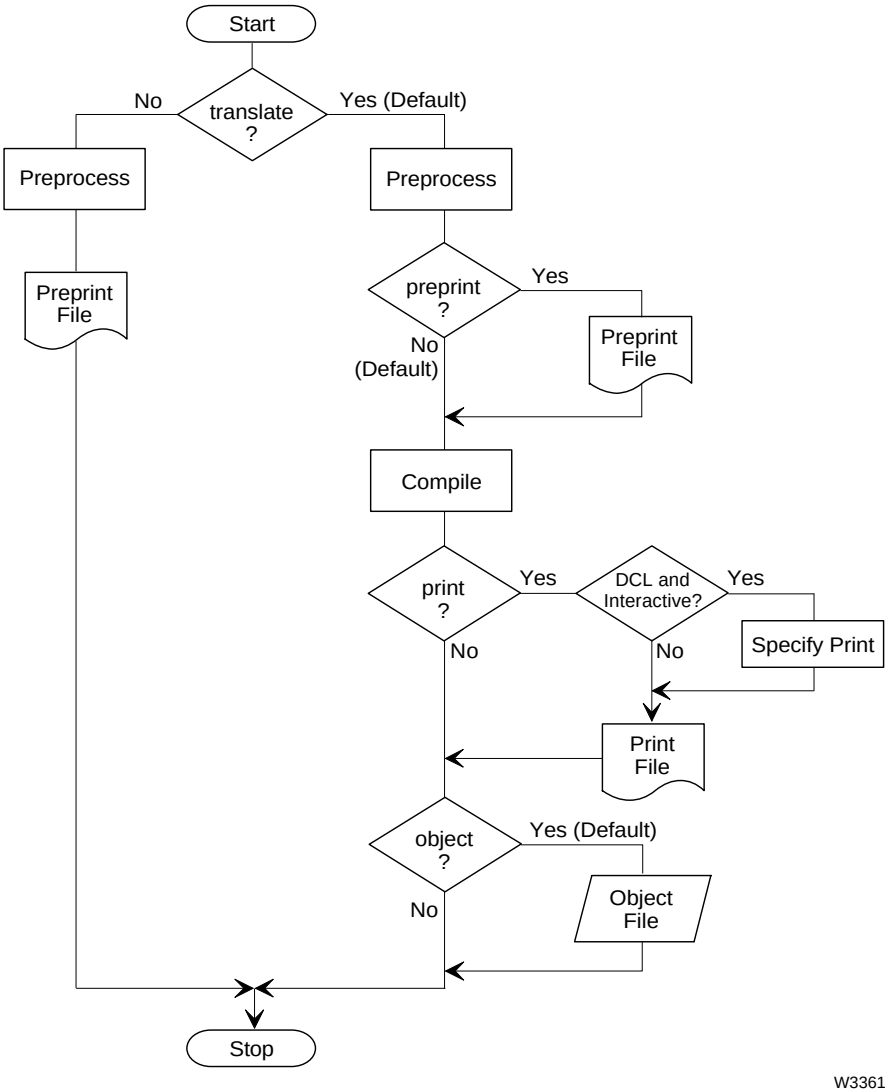

Figure 2-2. Controls That Create or Suppress Files

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. By default, the compiler places the preprint file in the directory containing the source file. If a file with the same name already exists, the compiler writes over it. To override the 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. By default, the compiler places the print file in the directory containing the source file. If a file with the same name already exists, the compiler writes over it. To override the defaults, use the `print` control.

Using Submit, Batch and Command Files

An iRMX submit file contains one or more commands that the iRMX system executes sequentially. On iRMX systems, use a submit file to invoke the compiler.

DOS offers two ways to invoke a series of commands automatically: batch files and command files.

Using iRMX Submit Files

Using submit files lets you consistently repeat complex commands without having to retype the entire command sequence each time. You can create a submit file with any text editor.

To invoke a submit file, use the **submit** command as follows:

```
submit filename
```

The *filename* can be a simple name for a submit file in the current directory, or it can be a pathname to a submit file in a different directory.

To save the console output of the submit file to a file named `csave.out`, enter:

```
- submit filename over csave.out echo
```

Commands in a submit file can contain continuation lines. To continue a command over two or more lines in a submit file, place an ampersand (&) at the end of each line to be continued, the same as when typing the command at the system prompt.

You can pass arguments to a command in a submit file by putting parameters as arguments to the command in the submit file. A parameter in a submit file takes the form:

```
%number
```

Where *number* indicates the position of the argument in the **submit** command invoking the submit file.

In this iRMX example, the parameter `%0` contains the value `hello`.

```
- submit /intel/gen/bind (hello)
```

Using DOS Batch Files for DOSRMX Systems

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 file. In this 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\bin\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\bin\ic386 prog1.c
```

DOS batch files have several other useful features, such as `if`, `goto`, `for`, and `call` commands.

See also: Your DOS documentation for explanation of these and other batch file commands

Consider these characteristics when developing a batch file for the iC-386 compiler:

- 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.

  In earlier versions of DOS, 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.

- Batch files can contain command labels and control flow commands such as `if` and `goto`. For example, in this command the result of program execution from the previously executed batch file determines 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. 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.

See also: `diagnostic` control description in Chapter 3 for more information on `errorlevel` values

- 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.

This 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 `386c1.bat` batch file, the `386c1.1tx` file of filenames, and the `make_map.bat` batch file. The 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.

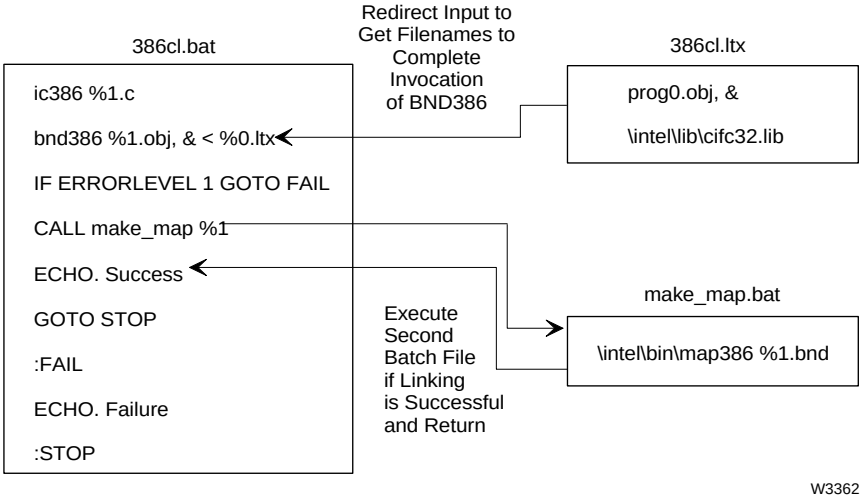

Figure 2-3. Redirecting Input to a DOS Batch File

The DOS batch file `%0` parameter always represents the name of the batch file itself (without the `.bat` extension). In the preceding example, since `386c1.bat` and `386c1.1tx` have identical names except for the extension, `386c1.bat` can refer to `386c1.1tx` as `0%.1tx`.

To execute the `386c1.bat` batch file and pass `prog1` as an argument, at the DOS command prompt type:

```
C:> 386c1 prog1
```

When `386c1.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 `386c1.1tx`. If the binding is successful, the `make_map.bat` file produces a map file named `prog1.map`.

Using DOS Command Files in DOSRMX Systems

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 <CR> follows the `exit` command, not an end-of-file character.

See also: DOS `command` and `exit` commands, in your DOS documentation

For example, the `exemakec.cmd` command file contains these commands (not a functional example of how to compile and bind an iC-386 program):

```
ic386 prog0.c
ic386 prog1.c
bnd386 prog0.obj, prog1.obj, &
progxs.lib
exit
```

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

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

Consider these characteristics when developing a command file for the iC-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 <CR>. 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, this 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
```

Binding Object Files

The iC-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 BND386 binder utility for iC-386 modules.

The binder can perform type checking and resolve intermodule references. The binder can automatically select modules from specified libraries to resolve references.

This is the general syntax (without device and path designations) for BND386:

```
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.

controls are the binder controls separated by spaces.

See also: BND386, *Intel386 Family System Builder User's Guide*

Choosing the Files to Bind

An iC-386 application can consist of many separately translated modules. The application can call functions from libraries. To create an executable file, you must use a binder to bind all translated code and libraries together. The iRMX OS includes the `cifc32.lib` C interface library; you can include other libraries.

See also: *C Library Reference* for more information on the C interface library

The iRMX C interface library supports only the compact memory segmentation model.

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 also: `compact`, `ram`, and `rom` compiler control descriptions in Chapter 3; segmentation model for iC-386 in Chapter 4

Figure 2-4 shows how to select libraries for binding with iC-386 modules.

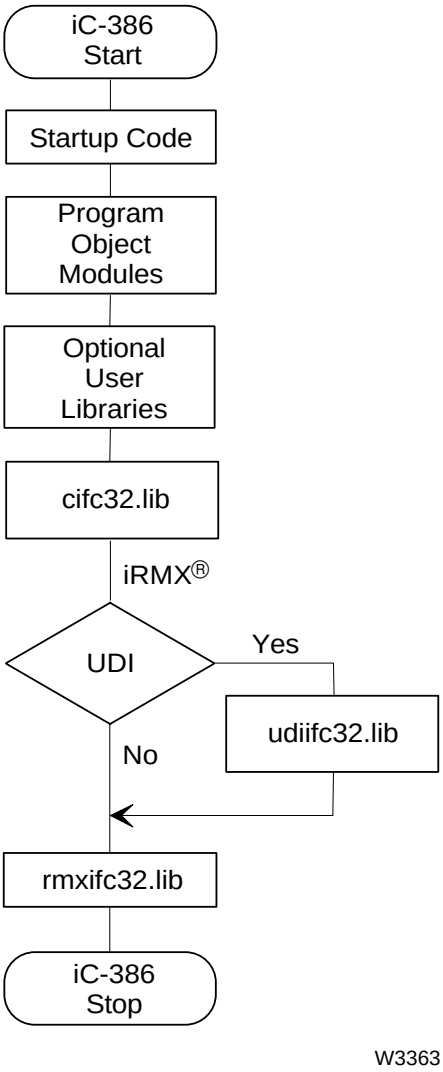

Figure 2-4. Choosing Libraries to Bind with iC-386 Modules

Examples of Binding

You can bind applications for iRMX systems in several different ways to accomplish several different objectives. This section lists examples of binding modules for different purposes.

See also: Various sample programs in the *rmx386\demo\c\intro* compiler directory

BND386 Example on DOS Systems

The *demo.c* example is cross-compiled to run under the iRMX OS.

See also: makefile sample code in *rmx386\demo\c\intro* compiler directory for *demo.c* example, invocation and binder parameters

The BND386 invocation links the object modules with the startup code and libraries and creates a loadable file named *demo*.

First, the binder invocation list must specify the object module for the C startup code and the application routines, in that order. Next, the binder links in the C interface library. Last, the binder links in the iRMX OS interface library.

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 loader. The `object` control names the executable file `demo` instead of the default `demo.bnd`.

The iRMX C interface library is included with iC-386 for use with applications written for the iRMX OS. The iRMX system interface library is part of the iRMX OS.

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 also: `compact` control description in Chapter 3 and segmentation memory models in Chapter 4 of this manual
C Library Reference for more information on `cstart` startup code

In-line Functions

The compiler generates in-line machine code by default for several run-time library functions. The 1989 ANSI C standard specifies that the header file containing the function declaration can additionally contain a macro definition; the compiler uses this feature to define in-line versions of some functions. Using the in-line versions of the functions produces more efficient code. To use the in-line functions, simply include the appropriate header file.

For example, the `stdlib.h` header file contains this declaration for the `abs` absolute value function:

```
int      abs(int value);          /* function prototype declaration */
#pragma  _builtin_("_abs_"=33)    /* tell compiler about the in-line version */
int      _abs_(int value);        /* prototype for the in-line version */
#define  abs(x) _abs_(x)          /* use the in-line version when the abs() */
                                  /* function is called */
```

Taking advantage of the in-line versions of the functions is transparent within the program. A fragment of code such as this uses the in-line `abs` function:

```
#include <stdlib.h>  /* including the appropriate header */
int main (int argc, char * argv[])
{
   int i,j;
                  /* assume that j holds an appropriate value */
   i = abs(j);       /* uses the in-line function */
}
```

You can use either of two methods to override the in-line version of the function, and call the actual function instead: enclose the function name in parentheses when it is called, or use the `#undef` preprocessing directive to remove the macro definition that maps the function to the in-line version. This example calls the function but allows other calls to use the in-line version:

```
#include <stdlib.h>
int main (int argc, char * argv[])
{
   int i,j;
                  /* assume that j holds an appropriate value */
   i = (abs)(j);    /* function call */
}
```

This example un-defines the macro and thus disables the in-line version for the remainder of the module:

```
#include <stdlib.h>
#undef abs
int main (int argc, char * argv[])
{
   int i,j;
              /* assume that j holds an appropriate value */
   i = abs(j);   /* function call */
}
```

Table 2-1 lists the iC-386 in-line functions, the header file in which each is defined, and a brief description of each.

Table 2-1. In-line Functions

Header File	Function	Description
<string.h>	memcpy	copies specified number of bytes
	memcmp	compares specified number of bytes
	memset	fills memory area with a byte value
	strcpy[1]	copies a constant string
	strcmp[1]	compares to a constant string
<stdlib.h>	abs	absolute value of integer
	labs	absolute value of long integer
<math.h>	fabs	absolute value of floating-point
	sqrt	non-negative square root
	log[2]	natural logarithm
	log10[2]	base 10 logarithm
	cos[2]	cosine of angle in radians
	sin[2]	sine of angle in radians
	tan[2]	tangent of angle in radians
	acos[2]	arc cosine of angle in radians
	asin[2]	arc sine of angle in radians
	atan[2]	arc tangent of angle in radians
	atan2[2]	principal value of arc tangent of angle in radians

[1] The compiler issues in-line instructions for these functions only if the appropriate arguments are constant values.
[2] This in-line function is provided by the iC-386 compiler only.

Note

In-line functions perform no range or domain checking; this checking is particularly important for floating-point functions. Use the library function if your application needs such checking.

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.

The *optimiz.c* example provides source text that demonstrates optimization at each level. Figures 2-5 through 2-8 show the significant results of compiling with iC-386 at different optimization levels.

See also: `optimize` control description in Chapter 3, which includes an explanation of each type of optimization
Sample code in *rmx386\demo\c\intro* compiler directory for *optimiz.c* example

Results at Optimization Level 0

Figure 2-5 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.

```
iC-386  COMPILER   Optimization Level 0      mm/dd/yy hh:mm:ss  PAGE 2
                   ASSEMBLY LISTING OF OBJECT CODE

                                                        ; 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  C705080000003000000
                                     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:
```

Figure 2-5. Pseudo-assembly Code at Optimization Level 0

```
                                                ; 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  C705080000064000000
                          MOV     k,64H
                @5:
                                                ; STATEMENT # 24
                l1:
00000099  E900000000      JMP     l2
                                                ; STATEMENT # 25
                l2:
0000009E  C705040000064000000
                          MOV     j,64H
                                                ; STATEMENT # 26

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  C70508000000C8000000
                          MOV     k,0C8H
                                                ; STATEMENT # 31
                main      ENDP
                                                ; STATEMENT # 31
```

Figure 2-5 Pseudo-assembly Code at Optimization Level 0 (continued)

```
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 2-5. Pseudo-assembly Code at Optimization Level 0 (continued)

Results at Optimization Level 1

Figure 2-6 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.

```
iC-386  COMPILER   Optimization Level 1     mm/dd/yy hh:mm:ss  PAGE  2
                   ASSEMBLY LISTING OF OBJECT CODE
.
.
.
                                                        ; 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:
```

Figure 2-6. Part of the Pseudo-assembly Code at Optimization Level 1

```
                                                    ; STATEMENT # 22
        00000078  C7050800000064000000
                                MOV     k,64H
                          @5:
                                                    ; STATEMENT # 24
                          l1:
        00000082  E900000000    JMP     l2
                                                    ; STATEMENT # 25
                          l2:
        00000087  C7050400000064000000
                                MOV     j,64H
.
.
.
```

Figure 2-6. Part of the Pseudo-assembly Code at Optimization Level 1 (continued)

Results at Optimization Level 2

Figure 2-7 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
                                                        ; STATEMENT # 24
                            l1:
                                                        ; STATEMENT # 25
                            l2:
```

Figure 2-7. Part of the Pseudo-assembly Code at Optimization Level 2

```
          0000005A  C705040000064000000
                                    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
                                                    ; 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 2-7. Part of the Pseudo-assembly Code at Optimization Level 2 (continued)

Results at Optimization Level 3

Figure 2-8 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.

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

Figure 2-8. Part of the Pseudo-assembly Code at Optimization Level 3

```
                                                ; STATEMENT # 20
                                                ; STATEMENT # 21
                                                ; STATEMENT # 22
        00000050  C7050800000064000000
                                MOV     k,64H
                                                ; STATEMENT # 24
                          l1:
                                                ; STATEMENT # 25
                          l2:
        0000005A  C7050400000064000000
                                MOV     j,64H
                                                ; STATEMENT # 26
        00000064  A10C000000    MOV     EAX,a
        00000069  C700C8000000  MOV     [EAX],0C8H
                                                ; STATEMENT # 27
        0000006F  C7050000000064000000
                                MOV     i,64H
                                                ; STATEMENT # 28
        00000079  5D            POP     EBP
        0000007A  C20800        RET     8H
                                                ; STATEMENT # 30
                                                ; STATEMENT # 31
                          main      ENDP

                                                ; 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 2-8. Part of the Pseudo-assembly Code at Optimization Level 3 (continued)

When you cast a floating point number to an integer, the compiler rounds the result at Optimization level 3, instead of truncating it as it does at levels 0, 1, and 2. For example, this code produces different results at different levels:

```
void main()
{
float f=3.67;
int i;
i = (int)f;
}
```

Under optimization levels 0, 1, and 2, the compiler truncates the variable `i` and sets it equal to 3. At optimization level 3, the compiler rounds it and sets it to 4.

If you want floating point variables to be truncated when they are cast to an integer, use an optimization level other than 3.

□□□

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 these topics:

- How controls affect the compilation
- Where to use controls
- Alphabetical reference of controls

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.

Where to Use Controls

You can use a compiler control once, multiple times, or only on invocation, depending on which kind of control it is:

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. The iRMX system preserves the case of arguments to controls. DOS requires 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 of a type to specified byte boundaries. Default: 4-byte boundaries	Object	General
code nocode	Generates or suppresses pseudo-assembly object code in the print file. Default: nocode.	Listing content	General
codesegment *a*	Names the iC-386 code segment. Default: CODE32.	Object	Primary
compact	Specifies segment allocation and segment register addressing in object module. Default: small.	Object	Primary
cond nocond	Includes or suppresses uncompiled conditional code in the print file. Default: nocond.	Listing content	General
datasegment *a*	Names the iC-386 data segment. Default: DATA.	Object	Primary
debug nodebug	Includes or suppresses debug information in the object module. Default: nodebug. nodebug overrides line.	Object	Primary
define *a*	Defines a macro.	Source	Invocation
diagnostic *a*	Specifies the level of diagnostic messages. Default: diagnostic level 1.	Listing content	Primary
eject	Inserts form feed in print file.	Listing format	General
extend noextend	Recognizes or suppresses Intel extensions. Default: noextend.	Source	General
fixedparams [*a*] varparams [*a*]	Specifies the FPL or VPL function-calling convention. Default:fixedparams for all functions.	Object	General

continued

Table 3-1. Compiler Controls Summary (continued)

Control	Description, Default, and Precedence	Effect	Usage
include *a*	Specifies a file to process before the primary	Source	Invocation source file
interrupt *a*	Specifies a function to be an interrupt handler.	Object	General
line noline	Generates or suppresses source line number debug information in the object file. Default: line if debug or noline if nodebug.	Object	Primary
list nolist	Includes or suppresses source code in the print file. Default: list. nolist overrides cond, listexpand, listinclude.	Listing content	General
listexpand nolistexpand	Includes or suppresses macro expansion in the print file. Default: nolistexpand.	Listing content	General
listinclude nolistinclude	Includes or suppresses text of include files in the print file. Default: nolistinclude. nolistinclude overrides listexpand and cond for include files.	Listing content	General
long64 nolong64	Sets the size for objects declared with the long data type. Default: nolong64.	Object	Primary
mod486 nomod486	Uses the Intel486 processor instructions, or restricts to the Intel386 processor instruction set. Default: nomod486.	Object	Primary
modulename *a*	Names object module. Default: *sourcename*.	Object	Primary
object [*a*] noobject	Generates and names or suppresses the object file. Default: object named *sourcename*.obj. noobject overrides all object controls except as affects the print file.	Object	Primary
optimize *a*	Specifies the level of optimization. Default: optimization level 1.	Object	Primary

continued

Table 3-1. Compiler Controls Summary (continued)

Control	Description, Default, and Precedence	Effect	Usage
pagelength *a*	Specifies the number of lines per page in the print file. Default: 60	Listing format	Primary
pagewidth *a*	Specifies the number of characters per line in the print file. Default: 120	Listing format	Primary
preprint [*a*] nopreprint	Generates and names or suppresses the preprint file. Default: nopreprint if translate or preprint *sourcename* if notranslate.	Listing content	Invocation
print [*a*] noprint	Generates and names or suppresses the print file. Default: print file named *sourcename*.lst. noprint overrides all listing controls except preprint.	Listing content	Primary
ram rom	Puts constants in the data segment or in the code segment. Default: ram (constants in data segment).	Object	Primary
searchinclude *a* nosearchinclude	Specifies a path to prepend to include files or limits the path to the source directory plus the :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 the identifier list in the print file. Default: nosymbols.	Listing content	Primary
tabwidth *a*	Specifies the number of characters between tabstops in the print file. Default: 4.	Listing format	Primary
title "*a*"	Places a title on each page of the print file. Default: "*modulename*".	Listing format	Primary

continued

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 the object module. Default: type.	Object	Primary
xref noxref	Adds or suppresses identifier cross-reference information in the print file. Default: noxref xref overrides nosymbols	Listing content	Primary

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 either.

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 these characters:

```
,  =  #  !  %  '  \  ~  +  -  ;  &  |  [ ]  < >
```

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

align | noalign

Aligns structures on a specified boundary.

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

For iRMX applications, use `noalign`. The default is `align`. Data structures supplied for the iRMX OSs are all unaligned. Use the `noalign` control for each structure individually, instead of globally.

The default value for *size* is 4 bytes for iC-386. The compiler attempts to place structure components so that they do not cross 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 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.

Examples

These 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 4-byte boundaries for iC-386. Use this in the compiler invocation:

   ```
   noalign (argument_list)
   ```

   Or use this 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 this in the compiler invocation:

   ```
   noalign align (argument_list)
   ```

   Or, use this 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:

```
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-386 compiler allocates a `std_struct` structure, assuming the `nolong64` control is in effect.

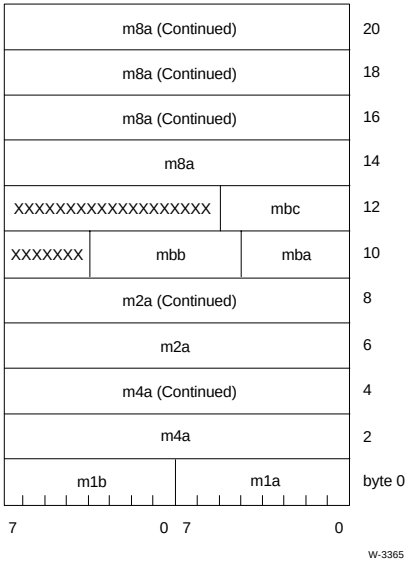

Figure 3-1. Effect of iC-386 align Control on Example Structure Type

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

```
noalign (std_struct)
```

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

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

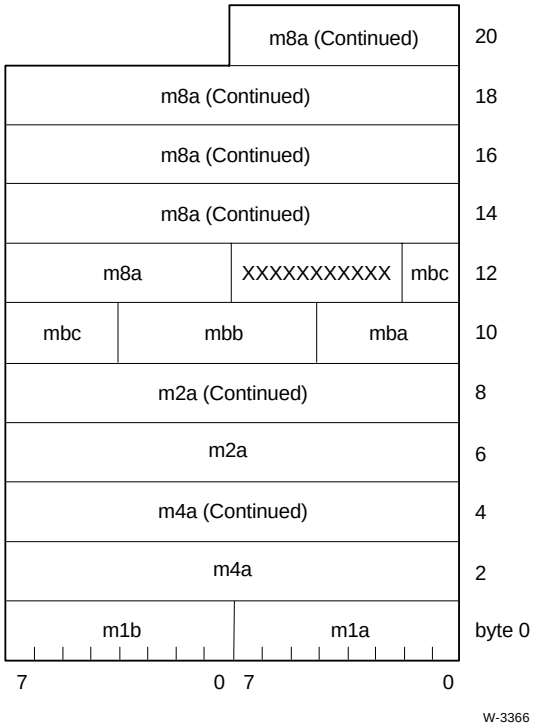

Figure 3-2. Effect of iC-386 noalign Control on Example Structure Type

Cross-references

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

code | nocode

Generates or suppresses pseudo-assembly language code in a listing.

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 `code` control produces a pseudo-assembly listing even if the `noobject` control is specified (suppressing the object file) but not if the `notranslate` control is specified (suppressing code generation). The `noprint` control causes the compiler to suppress all of the print file, including the pseudo-assembly listing, even if `code` is specified.

Use the `code` control:

- To view the effects of different levels of optimization set by the `optimize` control
- To view the difference in code the compiler generates under 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 also: Chapter 5 for more information on the print file

Cross-references

```
extend | noextend
mod486 | nomod486
object | noobject
optimize
print | noprint
translate | notranslate
```

codesegment

Names the code segment.

Syntax

`codesegment (`*`code_segment_name`*`)`

`#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 MAP386.

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`

compact

Specifies the compact segmentation memory model.

Syntax

```
compact

#pragma compact
```

Abbreviation

```
cp
```

Default

For iRMX applications use `compact`. 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 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 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 also: `extend|noextend` control description in Chapter 3 for more information about the `far` and `near` keywords

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

See also: Segmentation and the `compact` memory model in Chapter 4

Cross-references

```
extend | noextend
object | noobject
ram | rom
translate | notranslate
```

cond | nocond

Includes or suppresses uncompiled conditional code in listing.

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 also: Preprint and print files in Chapter 5

Cross-references

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

datasegment

Names the data segment.

Syntax

`datasegment (`*`data_segment_name`*`)`

`#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.

This control is provided for compatibility with Intel's previous compiler for the Intel386 processor.

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`

debug | nodebug

Includes or suppresses debug information in the object module.

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 these 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 using the Soft-Scope source-level debugger.

`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 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 puts source file and object file line-number 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

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

define

Defines a macro.

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.

Examples

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

```
define (SYS)
```

The source text contains the 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`.

diagnostic

Specifies the level of diagnostic messages.

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:

`diagnostic (0)` 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 0 if the program contains no errors but could contain remarks or warnings. At level 2, the compiler's exit status is non-0 only if the program contains errors, as shown in Table 3-2.

Table 3-2. Compiler Exit Status

Diagnostic Level	Fatal Errors	Errors	Warnings	Remarks	Exit Status
2	no	no	not used	not used	zero
	no	yes	not used	not used	nonzero
	yes	yes or no	not used	not used	nonzero
1 (default)	no	no	no	not used	zero
	no	no	yes	not used	nonzero
	no	yes	yes or no	not used	nonzero
	yes	yes or no	yes or no	not used	nonzero
0	no	no	no	no	zero
	no	no	no	yes	nonzero
	no	no	yes	yes or no	nonzero
	no	yes	yes or no	yes or no	nonzero
	yes	yes or no	yes or no	yes or no	nonzero

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

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 also: Chapter 5 for a description of the print file

The `eject` control is a general control. Use it as often as you like in the compiler invocation or in `#pragma` preprocessor directives.

Cross-references

```
pagelength
pagewidth
tabwidth
title
```

extend | noextend

Recognizes or suppresses Intel C extensions.

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 also: `fixedparams` and `varparams` control descriptions in Chapter 3 for information on calling convention compatibility with earlier versions of Intel C;
`alien, far` and `near` keywords in Chapter 10

Cross-references

```
fixedparams
ram | rom
varparams
```

fixedparams

Specifies fixed parameter list calling convention.

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:

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.

The FPL calling convention uses fewer instructions and therefore occupies less space in memory and executes more quickly than 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 this 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 also: FPL and VPL calling conventions in Chapter 8,
`extend|noextend` control description for other information on code compatibility with previous versions of Intel C,
`varparams` control description for information on the variable parameter list calling convention

Examples

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:

   ```
   #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:

   ```
   varparams (argument_list)
   ```

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

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

Cross-references

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

include

Inserts text from specified file.

Syntax

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

Where:

`filename` is the file specification (including a 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-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 also: Example of using the `include` control on DOS in Chapter 3, Chapter 5 for a description of the print file

Cross-references

```
listinclude
preprint | nopreprint
searchinclude
```

interrupt

Specifies a function to be an interrupt handler.

Syntax

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

Where:

function is the name of a function defined in the source text.

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.

Use the **rq_set_interrupt** iRMX system call to associate an interrupt function with an interrupt number. The **rq_set_interrupt** call puts the address of the function into the Interrupt Descriptor Table (IDT) for you; do not manipulate this table directly from your code.

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

See also: `interrupt` control description, in Chapter 3 of this manual
Interrupts, and **rq_set_interrupt**, *System Call Reference*

Cross-references

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

line | noline

Generates or suppresses source line number debug information.

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

Cross-references

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

list | nolist

Specifies source text listing in the print file.

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:

- 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 also: Chapter 5 for a description of the print file

Cross-references

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

listexpand | nolistexpand

Includes or suppresses macro expansion in listing.

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 also: Chapter 5 for a description of the print file

Cross-references

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

listinclude | nolistinclude

Includes or suppresses text from include files in listing.

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:

- 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 also: Chapter 5 for a description of the print file

Cross-references

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

long64 | nolong64

Specifies the size of long objects.

Syntax

```
[no]long64
```

```
#pragma [no]long64
```

Abbreviation

```
[no]l64
```

Default

For iRMX applications, use the default `nolong64` unless you are using iRMK calls that require `long64`.

Discussion

The `nolong64` control (default) specifies that objects declared with the `long` type qualifier are 32 bits in length.

The `long64` control specifies that objects declared as `long` are 64 bits in length. For compatibility, change any `longs` that need to stay 32 bits to `long32`. Header files are independent and not affected by the `long64` control.

The `long64` compiler switch may be used with C modules that make iRMK system calls. Under certain circumstances, however, the compiler may hang when compiling programs with `long64` set. C library and POSIX functions do not support `long64`.

If `notranslate` is specified, the compiler does not generate object code and the `long64` and `nolong64` controls have no effect. If `noobject` is specified, the effect of the `long64` and `nolong64` controls on the object code can be seen in the print file, although the compiler does not produce a final object file.

See also: iC-386 data types in Chapter 10

Cross-references

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

mod486 | nomod486

Generates Intel486 processor code or Intel386 processor code.

Syntax

```
[no]mod486
```

```
#pragma [no]mod486
```

Abbreviation

(none)

Default

`nomod486`

Discussion

Use the iC-386 `mod486` control to cause the compiler to generate code for the Intel486 processor. This code is particularly suited for fast execution on Intel486 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 Intel386 processor, which also executes on the Intel486 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.

Note

An object module compiled with the `mod486` control can execute on an Intel386 processor, but may execute more slowly than if compiled with the `nomod486` control.

Do not execute a `mod486`-compiled object module that contains Intel486 processor built-in functions on an Intel386 processor. The behavior of such code on an Intel386 processor is unpredictable.

Cross-references

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

modulename

Names the 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 binder, and builder. BND386 can rename object modules. 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`
`translate | notranslate`

object | noobject

Generates and names or suppresses object file.

Syntax

```
object [(filename)]
noobject

#pragma object [(filename)]

#pragma noobject
```

Where:

filename is the file specification (including a 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

Specifies the level of optimization.

Syntax

(*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.

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 also: compact, debug|nodebug, line|noline, and type|notype control descriptions for other ways to optimize code size

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 this 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 that 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, this `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 this 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! */
```

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

Unsafe conversions of floating-point types to integral types can occur at optimization level 3. The 1989 ANSI C standard specifies that these conversions must use truncation. At optimization level 3, the numeric 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
compact
debug | nodebug
type | notype
```

pagelength

Specifies lines per page in the print file.

Syntax

```
pagelength control (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.

See also: Chapter 5 for a description of the print file

Cross-references

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

pagewidth

Specifies line length in the print file.

Syntax

```
pagewidth control (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.

See also: Chapter 5 for a description of the print file

Cross-references

```
pagelength
print | noprint
tabwidth
translate | notranslate
```

preprint | nopreprint

Generates or suppresses a preprocessed source text listing file.

Syntax

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

Where:

filename is the file specification (including a 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 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 . 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 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 also: Chapter 5 for a description of the print and preprint files

Cross-reference

`print | noprint`

print | noprint

Generates or suppresses the print file.

Syntax

```
print [(filename)]
noprint

#pragma print (filename)

#pragma noprint
```

Where:

filename is the file specification (including a 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 `listexpand|nolistexpand` and `listinclude|nolistinclude` qualifiers and the `code|nocode`, `cond|nocond`, `diagnostic`, `list|nolist`, `listexpand|nolistexpand`, `listinclude|nolistinclude`, `symbols|nosymbols`, and `xref|noxref` controls affect the contents of the print file. The `pagewidth`, `pagelength`, `tabwidth`, and `title` controls affect the format of the print file.

See also: Chapter 5 for a description of 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
```

ram | rom

Specifies the placement of constants in the object module.

Syntax

```
ram control
rom control

#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 `rom` or `ram` control does affect the value of the `_ROM_` predefined macro.

See also: Predefined macros in Chapter 5

The `compact` control determines 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 also: Segmentation in Chapter 4

Cross-references

```
compact
object | noobject
translate | notranslate
```

searchinclude | nosearchinclude

Specifies search paths for include files.

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:` logical name from the iRMX OS, or the `:include:` environment variable from DOS, and the null prefix (current directory). The compiler always uses the path prefix in the `:include:` logical name from the iRMX OS or the `:include:` environment variable from DOS after the list specified by the `searchinclude` control.

The `:include:` logical name is `/intel/gen/inc` on iRMX systems. The submit file is `/intel/gen/inc/bind.csd`. Attach the library as `:include:` explicitly using the iRMX **attachfile** command.

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.

When the compiler searches for a file specified in the `include` control, or when it searches for a source file specified in an `#include` preprocessor directive, the compiler tests the prefixes in this order:

1. The source directory prefix
2. The directories specified by the `searchinclude` list
3. The directory or directories specified by the `:include:` logical name for the iRMX OS or environment variable for DOS, if defined
4. The null prefix, that is, the current directory

The iC-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.

Cross-references

`include`
`subsys`

signedchar | nosignedchar

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.

Cross-references

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

srclines | nosrclines

Generates or suppresses debug information (iC-386 only).

Syntax

```
[no]srclines
```

```
#pragma [no]srclines
```

Abbreviation

```
[no]sl
```

Default

`srclines`	when the debug and line controls are in effect
`nosrclines`	when the nodebug or noline control is in effect

Discussion

Use the iC-386 `srclines` control (default) to cause the compiler to add source file name and source line offset information to the object file. Use the `nosrclines` control to suppress this information, reducing the object file size by as much as 80%. This source file name and offset information is used by some symbolic debuggers for source-level debugging. Other debuggers, such as Soft-Scope III, do not require this information.

This control also modifies the amount of object code line offset information generated by the `line` control. When `srclines` is in effect, object code offset information is generated for every line in the source file (and include files), beginning with the first line of the source file. When `nosrclines` is in effect, the compiler starts emitting object code offset information only when the first executable statement is encountered; non-executable statements preceding the first executable statement, such as the definitions and declarations typically contained in header files, do not cause object code offsets to be emitted.

The `noline` control, the `nodebug` control, the `noobject` control, and the `notranslate` control override the `srclines` control.

Cross-references

```
debug | nodebug
line | noline
object | noobject
translate | notranslate
```

subsys

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 also: `searchinclude` control description for the search method, defining subsystems in Chapter 9

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 also: Subsystems in Chapter 9, `extend|noextend` control in Chapter 3, segmentation memory models and the `far` keyword in Chapter 4

Cross-references

```
codesegment
datasegment
extend | noextend
modulename
object | noobject
searchinclude | nosearchinclude
translate | notranslate
```

symbols | nosymbols

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 also: Chapter 5 for a description of the print file

Cross-references

```
print | noprint
translate | notranslate
xref | noxref
```

tabwidth

Specifies characters per tab stop in the print file.

Syntax

`tabwidth control (`*`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.

Cross-references

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

title

Specifies the print file title.

Syntax

`title control ("`*`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.

See also: Chapter 5 for a description of the print file

Cross-references

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

translate | notranslate

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
```

type | notype

Generates or suppresses type information in the object module.

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 also: `debug` control description 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 puts source line number information into 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

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

varparams

Specifies variable parameter list calling convention.

Syntax

```
varparams control [(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

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. This is the VPL calling convention:

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.

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 also: FPL and VPL calling conventions, `fixedparams` control description

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 this 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 also: `extend|noextend` control for other information on code compatibility with previous versions of Intel C; `fixedparams` control for information on the fixed parameter list calling convention

Examples

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

   ```
   varparams fixedparams (argument_list)
   ```

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

   ```
   #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:

   ```
   varparams (argument_list)
   ```

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

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

Cross-references

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

xref | noxref

Specifies symbol table cross-reference in listing.

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 also: Symbol table and print file in Chapter 5

Cross-references

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

❑❑❑

Segmentation Memory Models 4

This chapter discusses how segmentation memory models manage code, data, and stacks for the Intel386 segmented architecture. This chapter contains these topics:

- How the binder combines the compiler-created segments
- Characteristics of the compact memory model
- How to use and interpret the `far` and `near` keywords

Use the compact segmentation memory model for iRMX applications.

How the Binder Combines 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 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 BLD386 system builder or operating system loader assigns a fixed address to a segment and establishes its size. The maximum size of an Intel386 processor segment is 4 gigabytes.

Combining iC-386 Segments With BND386

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

- The same segment name
- The same kind of contents, i.e., code or data
- The same privilege level
- Compatible granularity, 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 these 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 writeable
- The combine-type
- The size of the segment

See also: Intel386 processor segment characteristics in Chapter 6

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.

See also: Use and syntax of subsystems in Chapter 9

Compact Segmentation Memory Model

The segmentation memory model determines the number of segments and the contents of those segments in the compiler-created object module. The binder uses the segments from each compiled object module to create the bound object module. The `compact` compiler control determines the segmentation model that the compiler uses to create an object module.

> **Note**
> The iRMX OS supports 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.

Compact Model

The BND386 binder combines 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: `CODE32` (iC-386), `DATA`, and `STACK`. The CS, DS, and SS registers contain the selectors for the `CODE32`, `DATA`, and `STACK` segments, respectively. For iC-386, the ES register contains the same value as the DS register.

Table 4-1 shows 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-386 compiler places the constants in the module's data segment.

Table 4-1. 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 bound compact model module contains one code segment up to 4 gigabytes long, one data segment up to 4 gigabytes long, and one stack segment up to 4 gigabytes 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 12 gigabytes of memory, but has a full 4 gigabytes 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 also: Near and far address formats in Chapter 4

Because all data pointers are `far` pointers by default, a compact model program can dynamically allocate one or more additional data segments up to 4 gigabytes long.

Figures 4-1 and 4-2 show the process of 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.

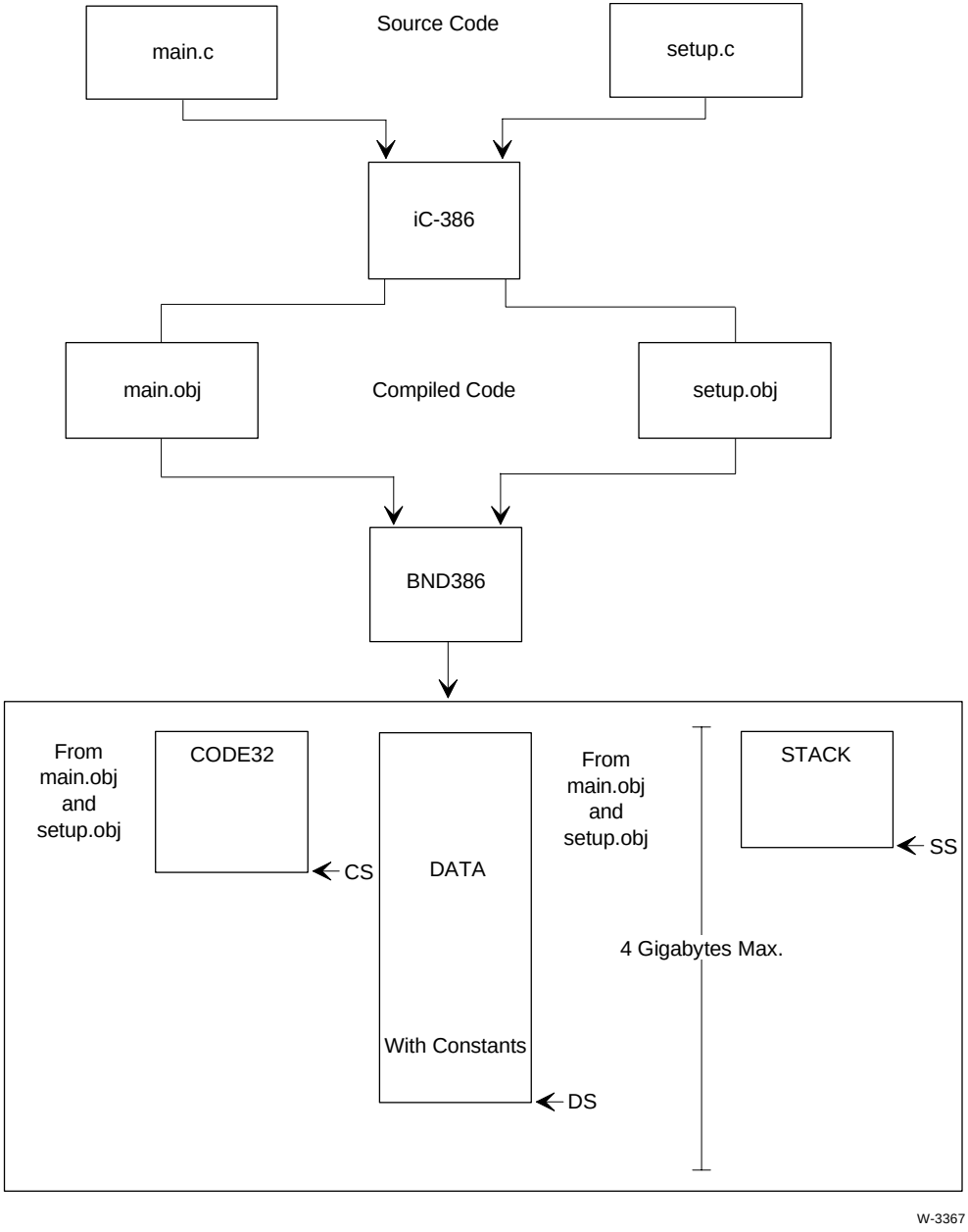

Figure 4-1. Creating a Compact RAM Program

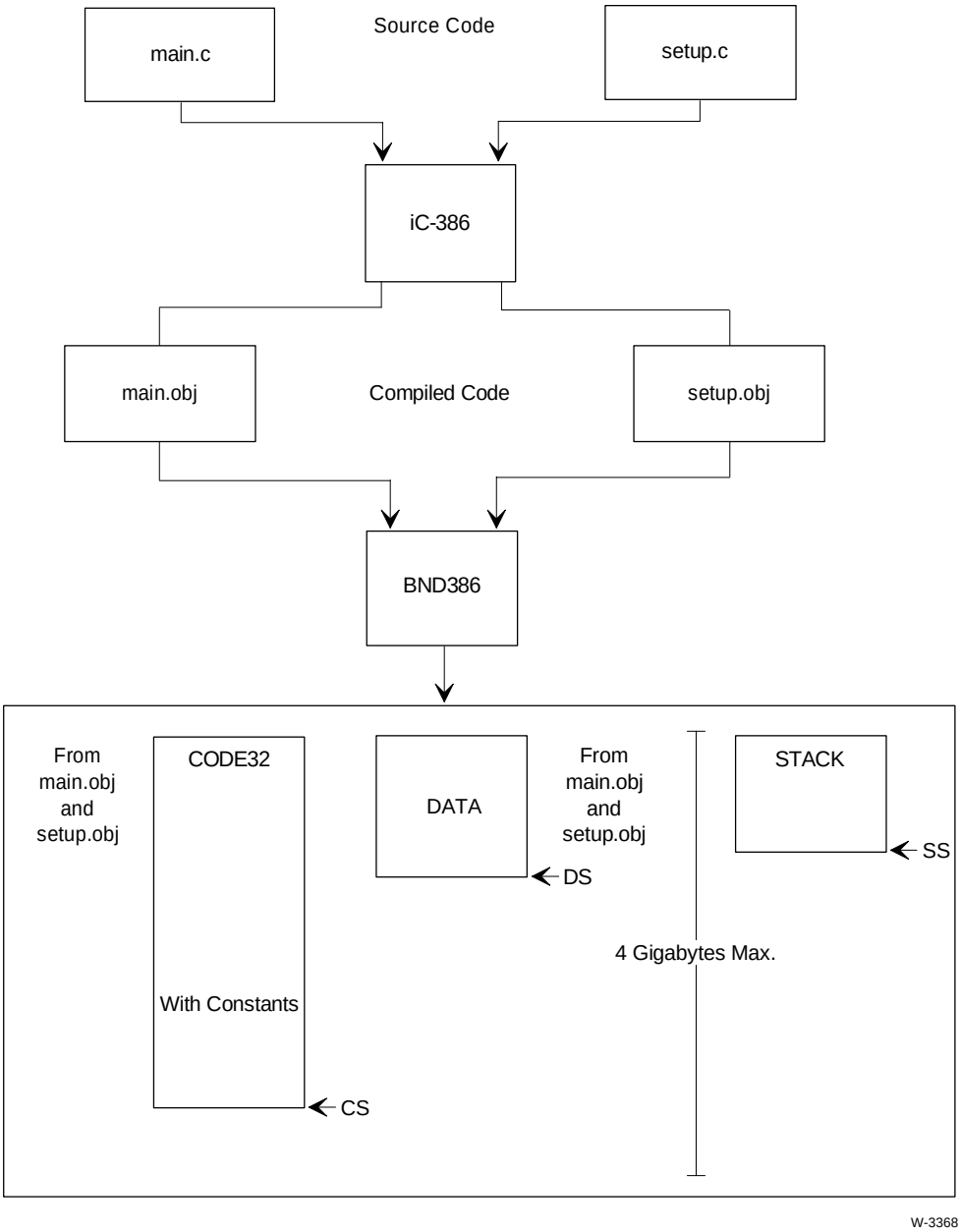

Figure 4-2. Creating a Compact ROM Program

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 also: `extend` control in Chapter 3

Table 4-2 shows the default address sizes for the `compact` memory model.

Table 4-2. Segmentation Models and Default Address Sizes

Segmentation Model	Code Reference	Data Reference
compact RAM	offset	selector and offset
compact ROM	offset	selector and 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:

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	In multiple subsystem applications, non-local references can require the far type qualifier. See also: Using multiple subsystems within an application in Chapter 9
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.

Use the `near` type qualifier:

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.

Addressing Under the Segmentation Models

In compact 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.

A reference to a selector-and-offset object requires a load to a segment register. 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 FS or GS register (Intel386 CPU) if neither contains the necessary selector. Then the processor uses either the FS or GS register with the offset portion of the variable's address to form the address of the variable.

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 also: Chapter 10 for the way iC-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, these 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                  */
```

Examples Using far

All of the examples that follow assume the compilation uses the `compact` 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". These are the identifiers and types in the examples:

```
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)();
```

□□□

Listing Files 5

The iC-386 compiler provides 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.

See also: `preprint` and `noprint` controls in Chapter 3

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:

- 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.

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.

The iC-386 compiler provides several predefined macros for your convenience. Table 5-1 shows these macros and their values.

See also: Using the `define` control to define macros; `long64 | nolong64`, `nomod287`, `mod486 | nomod486`, `optimize`, `rom`, and `ram` control descriptions in Chapter 3; segmentation memory models and addressing formats in Chapter 4

Table 5-1. iC-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)
ARCHITECTURE	Intel386 for iC-386 compiler and nomod486 control (default) Intel486 for iC-386 compiler and mod486 control
_FAR_CODE_	Default address size for function pointers and default range for function calls: 0 (near) for the compact segmentation model

continued

Table 5-1. iC-386 Predefined Macros (continued)

Name	Value
FAR_DATA_	Default address size for data pointers: 1 (far) for all ROM and compact RAM 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
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

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 also: `include` control in Chapter 3

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.

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 also: `cond`|`nocond` control in Chapter 3

The conditional directives are `#if`, `#else`, `#elif`, `#endif`, `#ifdef`, and `#ifndef`. The `#if` directive can take a special `defined` operator.

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 also: Individual directive descriptions in Chapter 11, list of controls that a `#pragma` directive can use in Chapter 3

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. These 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

Print File Contents

The print file contains these sections:

page header — identifies the compiler and the object module name and gives the date and time of compilation.

compilation heading — identifies the host OS, 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.

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.

This 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.

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, to invoke the compiler on a DOS host system:

```
C:\CEXAMPLE> ic386 main.c compact define(NPAPER) &
>> include(prags.h) &
>> searchinclude(\intel\include\,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 OS, the compiler version, the module name, and the controls used on invocation:

```
system-id       iC-386 COMPILER Vx.y, COMPILATION OF MODULE MAIN
OBJECT MODULE PLACED IN main.obj
COMPILER INVOKED BY: \INTEL\bin\IC386.EXE main.c compact
define(NPAPER)
                 include(prags.h)
searchinclude(\intel\include\,includes\)
```

If the invocation includes the `modulename` control and uses the `noobject` control to suppress the object file, the invocation looks like:

```
C:\CEXAMPLE> ic386 main.c compact define(NPAPER) &
>> include(prags.h) &
>> searchinclude(\intel\include\,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:

```
system-id       iC-386 COMPILER Vx.y, COMPILATION OF MODULE NEWNAME
NO OBJECT MODULE PRODUCED
COMPILER INVOKED BY: \INTEL\bin\IC386.EXE main.c compact
define(NPAPER)
                 include(prags.h)
searchinclude(\intel\include\,includes\)
                 modulename(NewName) noobject
```

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 OS.

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 also: Limitations on the number of nested include files in Chapter 11, control descriptions in Chapter 3

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.

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 also: `diagnostic` control in Chapter 3

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 also: `code | nocode` control in Chapter 3

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 also: Control descriptions in Chapter 3

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:

```
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, this line is displayed on the console:

```
COMPILATION TERMINATED
```

❑❑❑

Processor-specific Facilities 6

This chapter describes the functions, macros, and data types available in the `i86.h`, `i186.h`, `i286.h`, `i386.h`, and `i486.h` header files. These facilities enable the program to manipulate the unique characteristics of the Intel386, Intel486, and Pentium family of processors. This chapter contains these 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 Intel386, Intel486, and Pentium processors
- Manipulating the special control, test, and debug registers in the Intel386, Intel486, and Pentium processors
- Managing the data cache and paging translation lookaside buffer using special Intel486 and Pentium processor instructions
- Manipulating the Intel387 numeric coprocessor, and the Intel486 and Pentium floating-point units

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.

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 6-5 list the function names in the header files. All the functions are discussed in this chapter. 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 Intel386/Intel486/Pentium processors, the Intel387 coprocessor, and the Intel486/Pentium processor floating-point unit. Two functions are not defined for Intel386, Intel486, and Pentium processors, as noted.

Table 6-1. Built-in Functions in i86.h

Function	Function	Function
buildptr	halt	outword
causeinterrupt	inbyte	restorerealstatus[1]
disable	initrealmathunit	saverealstatus[1]
enable	inword	setflags
getflags	lockset	setrealmode
getrealerror	outbyte	

[1] Not for Intel386, Intel486, or Pentium processors. See the i386.h header file for substitute definitions.

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 6-2. Built-in Functions in i186.h

Function	Function	Function
blockinbyte	blockoutbyte	blockinword
blockoutword		

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.

Table 6-3. Built-in Functions in i286.h

Function	Function	Function
adjustrpl	gettaskregister	segmentwritable
cleartaskswitchedflag	restoreglobaltable	setlocaltable
getaccessrights	restoreinterrupttable	setmachinestatus
getlocaltable	saveglobaltable	settaskregister
getmachinestatus	saveinterrupttable	waitforinterrupt
getsegmentlimit	segmentreadable	

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 `i86.h` header file, as well. The `i386.h` header file contains functions and macros that apply to the Intel386, Intel486, and Pentium processors in protected mode.

Table 6-4. Built-in Functions in i386.h

Function	Function	Function
blockinhword	gettestregister	saverealstatus[1]
blockouthword	inhword	setcontrolregister
getcontrolregister	outhword	setdebugregister
getdebugregister	restorerealstatus[1]	settestregister

[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`, and `i86.h` header files, as well. The `i486.h` header file contains functions and macros that apply to Intel486 and Pentium processors in protected mode.

Table 6-5. Built-in Functions in i486.h

Function	Function	Function
byteswap	invalidatetlbentry	wbinvalidatedatacache
invalidatedatacache		

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.

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. This is the prototype for `buildptr`:

```
void far * buildptr (selector    sel,
                     void near * offset);
```

The offset argument can be 0, and the value that `buildptr` returns is equivalent to casting a selector to a far pointer type, as these 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 this expression shows:

```
(void near *) farptr
```

Using Special Control Functions

The `lockset` and `halt` functions in the `i86.h` header file provide special control over processing.

See also: Enabling and Causing Interrupts in this chapter 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. This is the prototype for `lockset`:

```
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). This is the prototype for `halt`:

```
void halt (void);
```

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 processors, or the EFLAGS register for Intel386, Intel486, and Pentium processors. In Intel386, Intel486 and Pentium processors, the EFLAGS register contains the FLAGS register in its low-order 16 bits. Table 6-6 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 word, according to other Intel386, Intel486 and Pentium processor documentation. In C programming literature, a word is the amount of storage reserved for an integer, which is 32 bits for iC-386.

The `getflags` function takes no arguments, and returns a 32-bit unsigned integer for iC-386. Use it to retrieve the value of the EFLAGS register. This is the prototype for `getflags`:

```
unsigned int getflags (void);
```

The `setflags` function takes as an argument a 32-bit unsigned integer for iC-386. Use it to set the value of the EFLAGS register. This is the prototype for `setflags`:

```
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 FLAGS and Intel386, Intel486, or Pentium EFLAGS register.

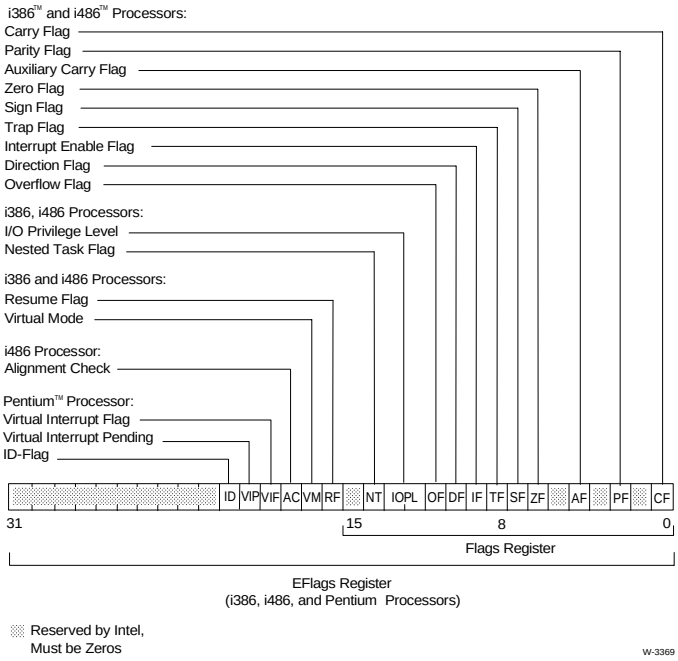

Figure 6-1. FLAGS and EFLAGS Register

Table 6-6 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 6-6. 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).
FLAG_IOPL	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	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.

continued

Table 6-6. Flag Macros (continued)

Name	Value	Meaning
FLAG_RESUME	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	0x20000	This flag, when set, indicates that the current task is a virtual 86 program.
FLAG_ALIGNCHECK[1]	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 Intel486 and Pentium processors only.

Use the functions and flag macros to set or clear particular flags.

See also: Sample code in *rmx386\demo\c\intro* compiler directory for example programs that test the carry bit, and disable and restore interrupts; Enabling and Causing Interrupts in this chapter

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 word, according to Intel386, Intel486, and Pentium processor documentation. In C programming literature, a word is the amount of storage reserved for an integer, which is 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. The `inword` function returns a 32-bit word for Intel386, Intel486, and Pentium processors. The `inhword` function returns a 16-bit word for Intel386, Intel486, and Pentium processors. These are the function prototypes:

```
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. The `outword` function sends a 32-bit word for Intel386, Intel486, and Pentium processors. The `outhword` function sends a 16-bit word for Intel386, Intel486, and Pentium processors. These are the function prototypes:

```
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. The `blockinword` function reads 32-bit words for Intel386, Intel486, and Pentium processors. The `blockinhword` function reads 16-bit words for Intel386, Intel486, and Pentium processors. These are the function prototypes:

```
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`, `blockoutword`, and `blockouthword` 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. The `blockoutword` function copies 32-bit words for Intel386, Intel486, and Pentium processors. The `blockouthword` function copies 16-bit words for Intel386 and Intel486 processors. These are the function prototypes:

```
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);
```

Enabling and Causing Interrupts

The `enable`, `disable`, `causeinterrupt`, and `halt` functions in the `i86.h` header file provide control over the interrupt process.

The `enable` function generates a set interrupt instruction (STI). STI sets the interrupt enable flag. This is the prototype for `enable`:

```
void enable (void);
```

The `disable` function generates a clear interrupt instruction (CLI). CLI clears the interrupt enable flag. This is the prototype for `disable`:

```
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. This is the prototype for `causeinterrupt`:

```
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). This is the prototype for `halt`:

```
void halt (void);
```

Interrupt Handlers

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 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 6-2 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 also: Descriptors, in *System Concepts*

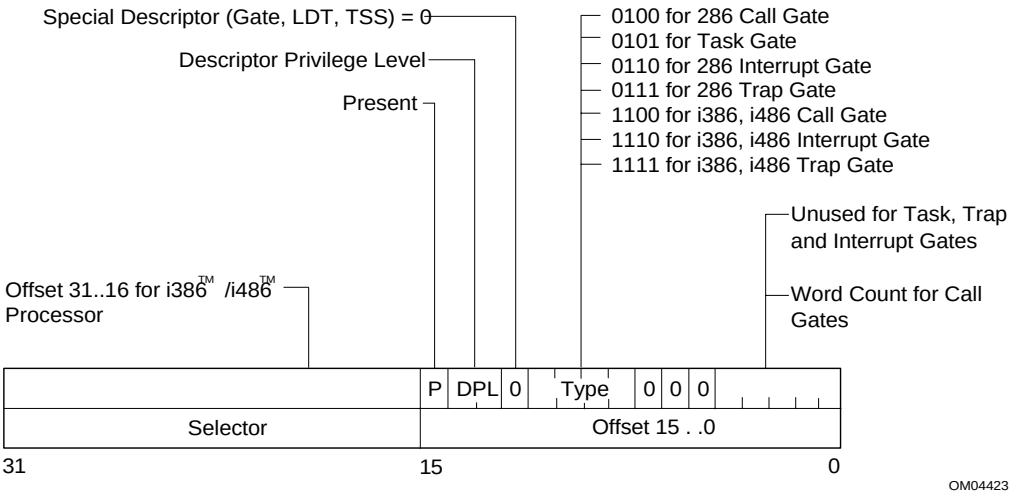

Figure 6-2. Gate Descriptor

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-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.

The easiest way to associate an iC-386 interrupt handler with a processor interrupt is to use the Nucleus system call **rq_set_interrupt**.

See also: `interrupt` control description, in Chapter 3 of this manual;
rq_set_interrupt, *System Call Reference*

Protected Mode Features of Intel386 and Higher Processors

See also: The *System Concepts* manual for a description of the protected mode features of the Intel386, Intel486, and Pentium processors available to iRMX applications

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 TR. This is the prototype for `gettaskregister`:

```
selector gettaskregister (void);
```

The `settaskregister` function loads a selector into the TR. Only protected mode code at privilege level 0 can execute this function. It takes the selector value as its argument. This is the prototype for `settaskregister`:

```
void settaskregister (selector sel);
```

The `descriptor_table_reg` structure type describes the register value returned by the `saveglobaltable` and `saveinterrupttable` functions. This is the structure definition:

```
#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 GDTR into a specific 6-byte location of type `descriptor_table_reg`. The function takes a pointer to this destination as an argument. This is the prototype for `saveglobaltable`:

```
void saveglobaltable
        (struct descriptor_table_reg * destinationptr);
```

The `restoreglobaltable` function loads a value of type `descriptor_table_reg` into the 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. This is the prototype for `restoreglobaltable`:

```
void restoreglobaltable
       (struct descriptor_table_reg const * sourceptr);
```

The `saveinterrupttable` function copies the contents of the IDTR into a specific 6-byte location of type `descriptor_table_reg`. The function takes a pointer to this destination as an argument. This is the prototype for `saveinterrupttable`:

```
void saveinterrupttable
       (struct descriptor_table_reg * destinationptr);
```

The `restoreinterrupttable` function loads a value of type `descriptor_table_reg` into the 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. This is the prototype for `restoreinterrupttable`:

```
void restoreinterrupttable
       (struct descriptor_table_reg const * sourceptr);
```

The `getlocaltable` function returns the contents of the LDTR. This is the prototype for `getlocaltable`:

```
selector getlocaltable (void);
```

The `setlocaltable` function loads a value of type `selector` into the LDTR. Only protected mode code at privilege level 0 can execute this function. It takes the selector value as an argument. This is the prototype for `setlocaltable`:

```
void setlocaltable (selector sel);
```

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, Intel486, and Pentium processors, the machine status word is the lower word in control register 0 (CR0). Figure 6-3 shows the format of the machine status word.

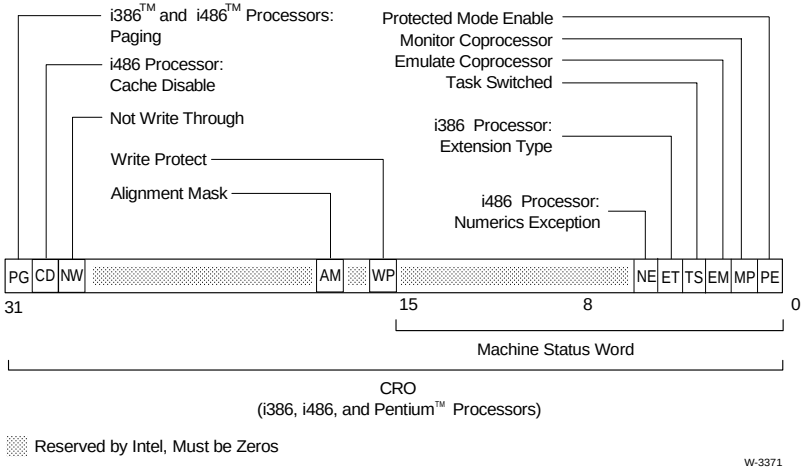

Figure 6-3. Machine Status Word

The `getmachinestatus` function returns the contents of the machine status word. This is the prototype for `getmachinestatus`:

```
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. This is the prototype for `setmachinestatus`:

```
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. This is the prototype for `cleartaskswitchedflag`:

```
void cleartaskswitchedflag (void);
```

Four macros isolate particular fields in the machine status word. Table 6-7 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 6-7. Machine Status Word Macros

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 or Pentium processors.

Accessing Descriptor Information

A segment descriptor contains several attributes in its access rights byte. Figure 6-4 shows the format of an Intel386 and Intel486 segment descriptor.

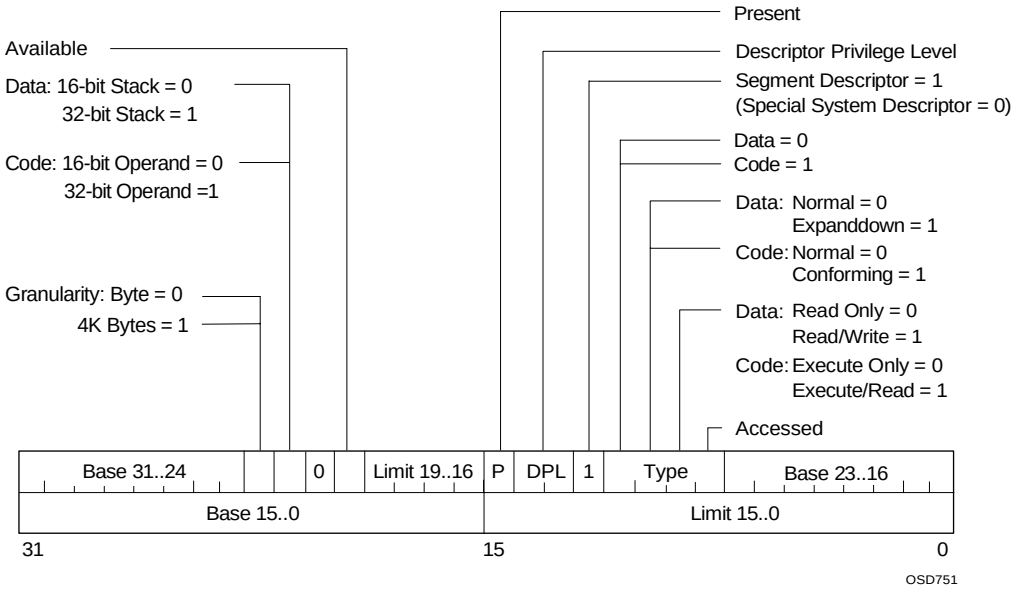

Figure 6-4. Segment Descriptor

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 a s 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 confirming 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 6-5 shows the format of a special descriptor, such as a gate, local descriptor table (LDT), or task state segment (TSS).

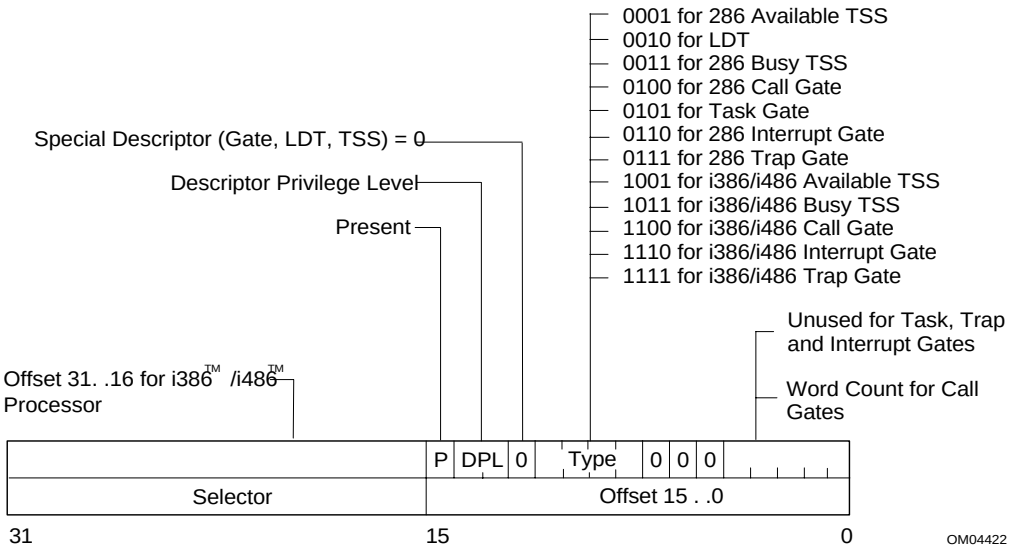

Figure 6-5. Special Descriptor

Table 6-8 lists the name s 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 Figure 6-4 for the format of a segment descriptor. These macro names must be uppercase in the source text.

Table 6-8. General Descriptor Access Rights Macros

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 6-9 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 Figure 6-4 for the format of a segment descriptor. These macro names must be uppercase in the source text.

Table 6-9. Segment Descriptor Access Rights Macros

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 21 and the AR_EXECUTABLE bit is 1, this bit is 1 for a readable code segment and 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 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 - for a data segment.

Table 6-10 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 6-10. Special Descriptor Access Rights Macros

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(TM) 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)

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 6-6 shows the format of a selector.

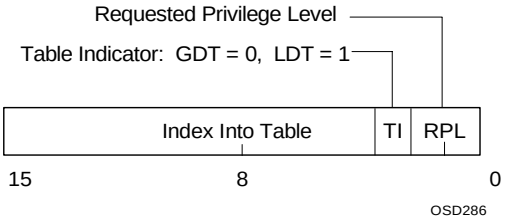

Figure 6-6. Selector

Adjusting the RPL field of the selector of a called segment ensures that nested code segment access occur at a level no more privileged than the level of the original calling segment.

The `adjustrpl` function is the 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 routing 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.

Manipulating the Control, Test, and Debug Registers of Intel386™, Intel486™, and Pentium® 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. Only code executing at privilege level 0 can access these registers. Figure 6-7 shows the special registers accessible in the Intel386, Intel486, and Pentium processors.

Note

Applications accessing these registers cannot be debugged using the Soft-Scope or iRMX SDM debuggers.

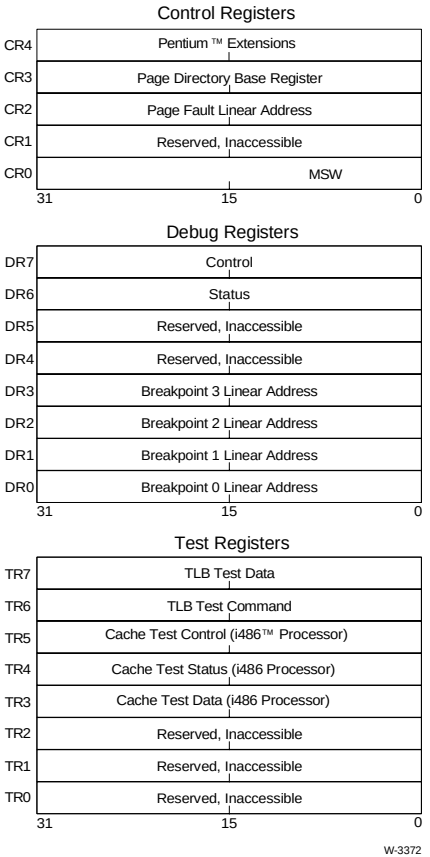

Figure 6-7. Control, Test, and Debug Registers of Intel386, Intel486, and Pentium Processors

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. The functions' prototypes are:

```
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. These are their prototypes:

```
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. Figure 6-8 shows the format of control register 0.

See also: Manipulating the Machine Status Word in this chapter

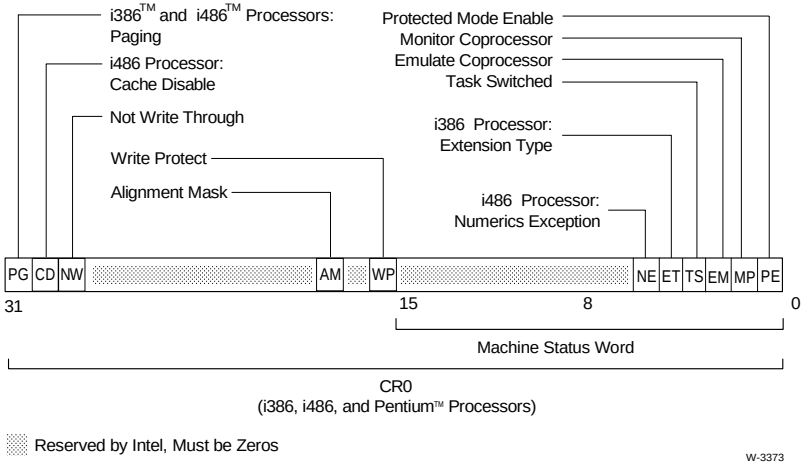

Figure 6-8. Control Register 0 of Intel386, Intel486, and Pentium Processors

Table 6-11 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 6-11. Control Register 0 Macros for Intel386, Intel486, and Pentium Processors

Name	Value	Meaning
CR0_EXTENSION_TYPE	0x0010	This bit is 1 if the Intel387 coprocessor, Intel486 processor, or the Pentium processor is present, and 0 if the Intel287 coprocessor is present.
CR0_PAGING_ENABLED	0x8000	This bit is 1 if paging is enabled, or 0 if paging is disabled.

Managing the Features of the Intel486 and Pentium Processors

The `i486.h` header file contains functions that enable iC-386 programs to manipulate the unique features of the Intel486 and Pentium processors.

The Intel386, Intel486, and Pentium 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 word, converting little endian format to big endian format. This feature is useful for transferring data between the Intel486 or Pentium processor and foreign processors or peripherals. The function takes a 32-bit word as its argument, and returns the swapped 32-bit value. This is the function prototype:

```
unsigned int byteswap (unsigned int value);
```

The Intel486 and Pentium processors also contain on-chip caches and provide instructions to manipulate those caches. The `invalidatedatacache` function flushes the internal data cache. Its prototype is:

```
void invalidatedatacache (void);
```

The `wbinvalidatedatacache` function flushes the internal data cache and directs any external cache to write back its contents and flush itself. This is the function prototype:

```
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. This is the function prototype:

```
void invalidatetlbentry (void far * memoryaddress);
```

Manipulating the Numeric Coprocessor

The `i86.h` header file contains several functions, macros, and data types that enable iC-386 programs to manipulate a numeric coprocessor, a true software emulator, or the Intel486 or Pentium processors floating-point unit.

See also: *80387 Programmer's Reference Manual* or *ASM386 Assembly Language Reference*

This section uses the term numeric coprocessor to indicate a coprocessor, emulator, or on-chip unit.

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 6-9 shows the numeric data register set. Figure 6-10 shows the environment registers for the Intel387 coprocessor, and the Intel486 and Pentium processor FPU.

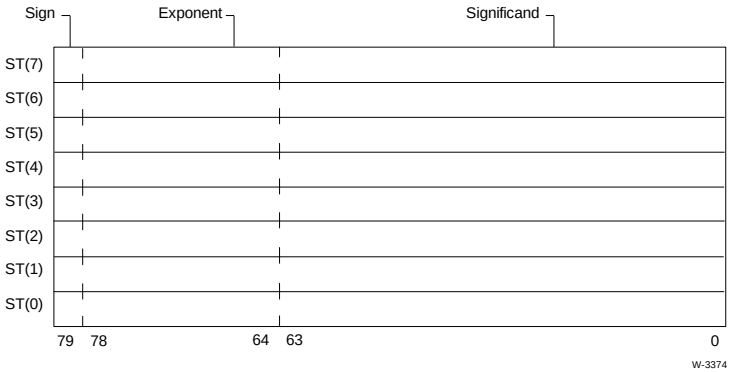

Figure 6-9. Numeric Coprocessor Stack of Numeric Data Registers

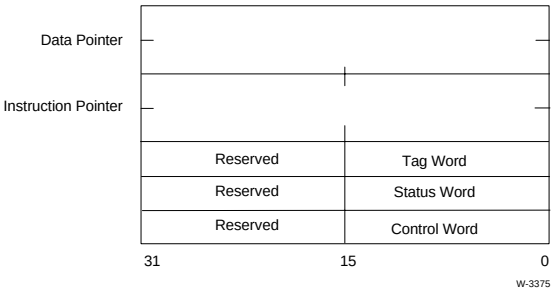

Figure 6-10. Intel387 Numeric Coprocessor or Intel486 and Pentium Processor FPU Environment Registers

The `setrealmode` function sets the fields of the control word.

The `getrealerror` function retrieves the value of the status word.

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 also: Control word and the `setrealmode` function; status word and the `getrealerror` function; Saving and Restoring the Numeric Coprocessor State for data types and functions relative to the numeric data registers, environment, and state, in this chapter

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 6-11 shows the tag word and the possible values for each tag.

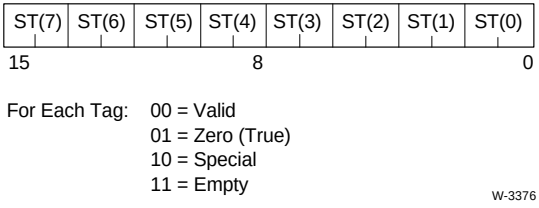

Figure 6-11. Numeric Coprocessor Tag Word

Table 6-12 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 6-12. 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] This is the macro definition:

```
#define I87_TAG(x,y) (((x).tag >>  (I87_TAG_SHIFT * (y))) &
I87_TAG_MASK)
```

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 6-13 for the format of the status word). Figure 6-12 shows the format of the control word.

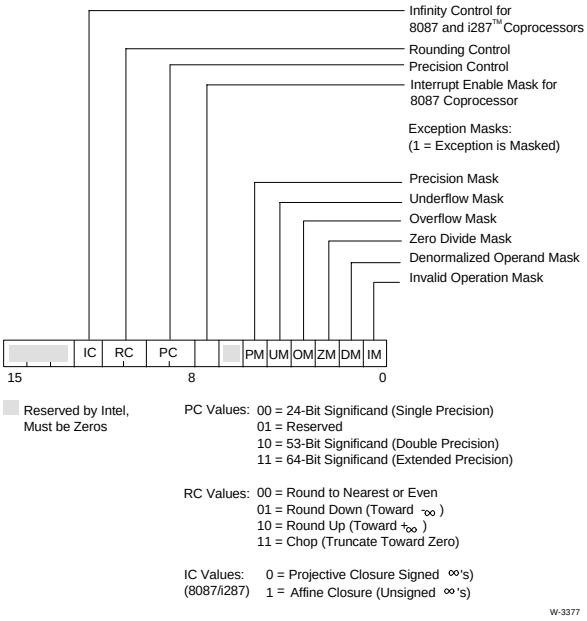

Figure 6-12. Numeric Coprocessor Control Word

The `setrealmode` function loads a value into the control word. The function takes the value as its argument. This is the prototype for `setrealmode`:

```
void setrealmode (unsigned short mode);
```

Table 6-13 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 6-13. 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.
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.

continued

Table 6-13. Numeric Coprocessor Control Word Macros (continued)

Name	Value	Meaning
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.

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 6-12 for the format of the control word). Figure 6-13 shows the format of the status word. Table 6-14 shows the values of the condition codes for the Intel387 numeric coprocessor or Intel487 FPU.

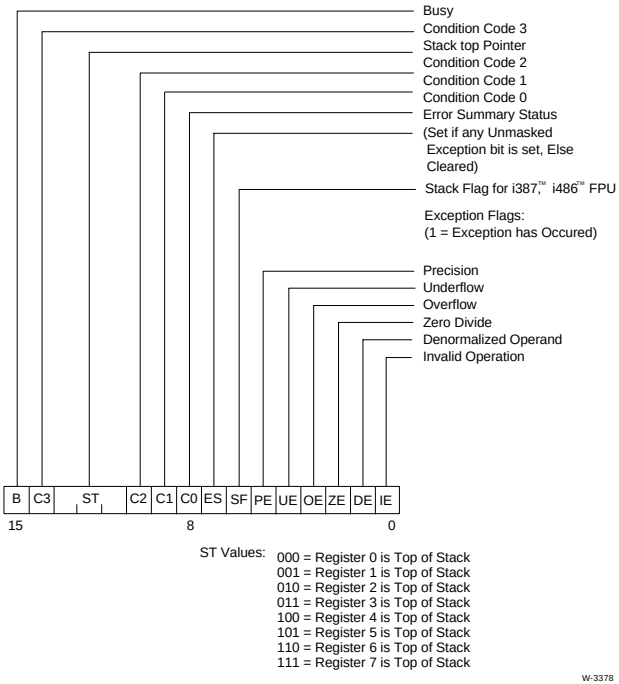

Figure 6-13. Numeric Coprocessor Status Word

Table Error! Reference source not found.**-14. Intel387 Numeric Coprocessor, and Intel486 or Pentium Processor FPU Condition Codes**

Instructions	C_3	C_2	C_1	C_0	Interpretation
FCOM, FCOMP,	0	0	0 or O/U	0	stack top > operand
FCOMPP, FTST,	0	0	0 or O/U	1	stack top < operand
FUCOM, FUCOMP,	1	0	0 or O/U	0	stack top = operand 1
FUCOMPP, FICOM, FICOMP	1	1	0 or O/U	1	unordered
FPREM, FPREM1	Q1	0	Q0	Q2	complete reduction with 3 low bits of quotient in C0, C3, and C1 U
	U	1	U	U	incomplete reduction
FXAM	0	0	Sign	0	unsupported 0
	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

continued

Table Error! Reference source not found.**-14. Intel387 Numeric Coprocessor, and Intel486 or Pentium Processor FPU Condition Codes (continued)**

Instructions	C_3	C_2	C_1	C_0	Interpretation
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. This is the prototype for `getrealerror`:

```
unsigned short getrealerror (void);
```

Table **Error! Reference source not found.**-15 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 Error! Reference source not found.**-15. 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.
I87_CONDITION_C0	0x0100	This bit is condition code bit 0 (see Table 6-14).
I87_CONDITION_C1	0x0200	This bit is condition code bit 1 (see Table 6-14).
I87_CONDITION_C2	0x0400	This bit is condition code bit 2 (see Table 6-14).
I87_CONDITION_C3	0x4000	This bit is condition code bit 3 (see Table 6-14).

[1] This is the macro definition:

```
#define I87_STACKTOP(env) (((env).status &
I87_STATUS_STACKTOP_MASK)
>> \ I87_STATUS_STACKTOP_SHIFT)
```

Intel387™ Numeric Coprocessor, and Intel486 or Pentium Processor FPU Data Pointer and Instruction Pointer

Figure 6-14 shows the protected mode format of the data pointer and instruction pointer for the Intel387 numeric coprocessor, and the Intel486 or Pentium processor FPU.

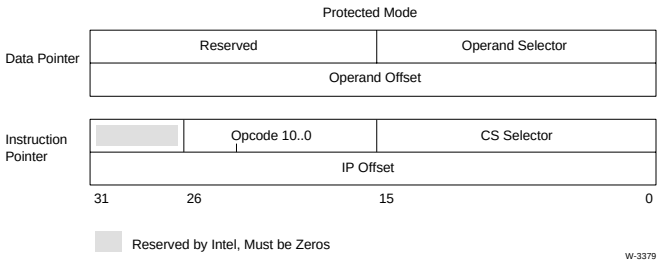

Figure 6-14. Intel387 Numeric Coprocessor, and Intel486 or Pentium Processor FPU Data Pointer and Instruction Pointer

The `i387_protected_addr` data type defines the structure of the information in the data pointer or instruction pointer for the Intel387 numeric coprocessor, and the Intel486 or Pentium processor FPU.

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. This is the structure definition:

```
#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;
};
```

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 6-9 and 6-10 for the general format of these registers.

The `i387_environment` data type defines the environment for the Intel387 coprocessor, and the Intel486 or Pentium processor FPU. The `i87_tempreal` data type and the `tempreal_t` typedef define the format of one numeric register. The `i387_state` data type defines the structure of all the registers for the Intel387 coprocessor, and the Intel486 or Pentium processor FPU. The `saverealstatus` and `restorerealstatus` functions manipulate the entire state of the numeric coprocessor.

The `i387_environment` structure type defines the Intel387 numeric coprocessor, and the Intel486 or Pentium processor FPU environment. This is the structure definition:

```
#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. These are the definitions for `i87_tempreal` and `tempreal_t`:

```
#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 `i387_state` structure defines the state of the Intel387 numeric coprocessor, and the Intel486 or Pentium processor FPU. This is the structure definition:

```
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 `i387_state` for the Intel387 coprocessor, and the Intel486 or Pentium processor FPU. The function takes a pointer to this destination as an argument.

The prototype for `saverealstatus` for the Intel387 coprocessor, and the Intel486 or Pentium processor FPU is:

```
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 `i387_state` save area for the Intel387 coprocessor, and the Intel486 or Pentium processor FPU.

The prototype for `restorerealstatus` for the Intel387 coprocessor, and the Intel486 or Pentium processor FPU is:

```
void restorerealstatus (struct i387_state const * sourceptr);
```

❑❑❑

7 Assembler Header File

The `util.ah` header file contains macros that help interface assembly routines to iC-386 programs. To use these facilities, include the header file in your assembly routines. The `util.ah` assembler header file provides these facilities:

- Segmentation and linkage directives and generic data type specifiers for any standard memory model; for iRMX applications, use compact model
- 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 also: Sample code in *rmx386\demo\c\intro* compiler directory for examples of code using macros, source files, expanded source code for ASM386 for the compact memory model, and implementations of the `strcmp` and `memcpy` functions.

Macro Selection

The macros defined in `util.ah` fall into five groups:

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\lib\
```

Use this 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 the header controls to use for iRMX applications.

Table 7-1. Assembler Header Controls for Macro Selection

Header Control	Abbr.	Description	Default
asm386		generate code for ASM386	asm86
compact	cp	generate code for compact memory model	small l
fixedparams	fp	generate prolog/epilog for FPL calling convention	fixedparams
varparams	vp	generate prolog/epilog for VPL calling convention	fixedparams
'module=*name*'[1]		set module name	module=anonymous
ram		generate code for RAM sub-model	ram
rom		generate code for ROM sub-model	ram
'stacksize=*size*'[1]		set size of the stack segment	stacksize=0

[1] 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.

You can define the `controls` macro in the assembler invocation, or in the source text, or both places:

- 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.

This is the DOS syntax for the assembler invocation:

```
asm386 file [asm_controls] %define(controls)([header_controls])
```

Where:

file	is the source file to assemble.
asm_controls	are controls for the assembly.
	See also: ASM controls, *ASM386 Macro Assembler Operating Instructions*
header_controls	are header controls from Table 7-1, separated by spaces.

Within the source text, this is the syntax for defining the `controls` macro and including the `util.ah` header file:

```
%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. These are the precedence levels of the header controls:

- 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.

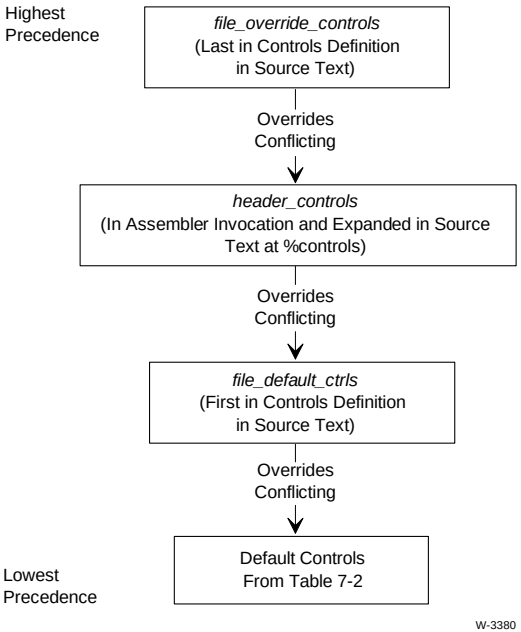

Figure 7-1. Precedence Levels of Assembler Header Controls

These examples demonstrate invoking the assembler with header controls to select macros.

1. This example invokes the ASM386 assembler with non-default assembler settings and header controls. The assembler processes the source text in the file `utest.asm` using the compact model, and produces an object module with variable parameter list (VPL) calling convention.

```
C:> asm386 utest.asm %define(controls)(cp vp)
```

2. This example defines `controls` in the assembly source text. The header control settings specify ASM386, the compact model, and the ROM submodel.

```
%define(controls)(asm386 cp rom)
$include(:include:util.ah)
```

3. This example defines header control defaults partly different from the global default controls. The assembly source text contains:

```
%define(controls)
       (cp vp 'stacksize=50' %controls 'module=ut1')
```

   This definition of the `controls` macro sets these 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 `ut1` instead of `anonymous` and cannot be overridden; its position after `%controls` indicates that it is a file override control.

   This is the assembler invocation for ASM386 on DOS:

```
C:> asm386 utest.asm %define(controls)(asm386 rom)
```

   The `controls` defined in the assembler invocation override only the file default controls that specify the memory model:

   - The object module is ROM model rather than RAM.
   - The calling convention is VPL and the stack size is 50, as specified in the file default controls.

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 also: Macro programming language, *ASM386 Macro Assembler Operating Instructions*

Use the flag macros to test these 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.

`%far_data` indicates that data pointers are far; set by the `compact`, 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` header control.

`%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 `asm386` header controls.

`%i386_asm` indicates code specific to a particular architecture when code is common between products targeted for 86, 286, or Intel386 processors; set by `asm386` header control.

Table 7-2 lists which flag macros are set when you specify various header controls.

Table 7-2. Assembler Flag Macros Set by Header Controls

Header Control	Flag Macros Set
asm386	%i386_asm %i186_instrs
compact	%far_data %far_stack
fixedparams	%fpl
rom	%const_in_code %far_data

Register Macros

You can use a register macro as an instruction operand in place of the register name. Table 7-3 shows macros useful in specifying operands to instructions.

Table 7-3. Assembler Register Macros

Macro	ASM386 Expansion
%ax	eax
%bx	ebx
%cx	ecx
%dx	edx
%bp	ebp
%sp	esp
%si	esi
%di	edi

These are the register macros and the registers they reference:

`%retoff` is the register that holds the offset portion of a pointer return value. The `%retoff` macro expands to `eax` for ASM386.

`%retsel` is the register that holds the selector portion of a pointer return value. The `%retsel` macro expands to `edx` for ASM386.

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 are the segment names and what each macro expands to:

`%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

Table 7-4 shows the segment macro expansion for the compact memory model for ASM386.

Table 7-4. ASM386 Segment Macro Expansion for Compact Memory Model

Macro	Model	Sub-model	Expansion
%code	compact	RAM or ROM	CODE32
%cgroup	compact	RAM or ROM	%code
%data	compact	RAM or ROM	DATA
%dgroup	compact	RAM or ROM	%data
%stack	compact	RAM or ROM	STACK
%sgroup	compact	RAM or ROM	%stack
%const	compact compact	RAM ROM	%data %code

This example uses %DATA to bracket static variable data:

```
%data segment
;assembler commands, e.g.,
  var dw 0
%data ends
```

This example expands to:

```
DATA segment
;assembler commands, e.g.,
  var dw 0
DATA ends
```

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:

`%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

Table 7-5 shows the type macro expansion for the compact memory model.

Table 7-5. ASM386 Type Macro Expansion for Compact Memory Model

Macro	Model	Sub-model	Expansion
%fnc	compact	RAM or ROM	near
%fnc_ptr	compact	RAM or ROM	dword
%ptr	compact	RAM or ROM	pword
%reg_size	compact	RAM or ROM	dword ptr
%int	compact	RAM or ROM	dword
%dint	compact	RAM or ROM	dd

Operation Macros

The operation macros are grouped in four different classes according to their function:

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.

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.

Table 7-6 shows the external definition macro expansion for the compact memory model for ASM386.

Table 7-6. ASM386 External Declaration Macro Expansion for Compact Memory Model

Macro	Model	Sub-model	Expansion
%extern	compact	RAM or ROM	DATA segment extrn *vname*:*type* DATA ends
%extern_const	compact	RAM	CONST segment extrn *aconst:type* CONST ends
	compact	ROM	CODE32 segment extrn *aconst:type* CODE32 ends
%extern_fnc	compact	RAM or ROM	CODE32 segment extrn *fname*:near CODE32 ends

Instruction Macros

The instruction macros provide compatibility between 86 and higher processor instruction sets.

`%enter` expands to the `enter` instruction.

`%leave` expands to `leave` instruction for 186 and higher instruction sets.

`%pusha` expands to the `pushad` instruction for the Intel386 instruction set.

`%popa` expands to the `popad` instruction for the Intel386 instruction set.

`%pushf` expands to `pushfd` for the Intel386 instruction set.

`%movsx` expands to `movsx` for the Intel386 instruction set.

`%movzx` expands to `movzx` for the Intel386 instruction set.

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:

```
%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:

```
%if (not %far_data) then (text) fi
```

Function Definition Macros

These entries describe the function macros in detail in their order of use:

`%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

%function

Open 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 ASM386 compact model, the `%function` macro expands to:

```
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.

%param_flt

Define a floating-point parameter name

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:

```
%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`*.

%prolog

Generate a function prolog

Syntax

`%prolog(`*`registers`*`)`

Where:

`registers` is a list of segment registers and general registers. However, the macro ignores all but the 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 these tasks:

- Pushes registers
- Pushes EBP for ASM386 (the base pointer register) and initializes it for use as a local frame pointer using the `ENTER` assembler instruction
- Sets ESP for ASM386 using the `ENTER` assembler instruction
- Allocates space for automatic variables

%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 EBP register for ASM386; and the ESP register for ASM386.

%ret

Generate a return instruction

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:

```
ret
```

Under the `fixedparams` header control, `%ret` expands to:

```
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:

```
fname endp
```

❑❑❑

8 Function-calling Conventions

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-386 functions with functions written in other Intel programming languages.

This chapter contains information on how iC-386 generates object code for a function call, and how the fixed parameter list and variable parameter list conventions differ.

See also: Segmentation memory models in Chapter 4;
data types, reserved words, conformance to the ANSI C standard, implementation-dependent compiler features, in Chapter 10

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 these 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:

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

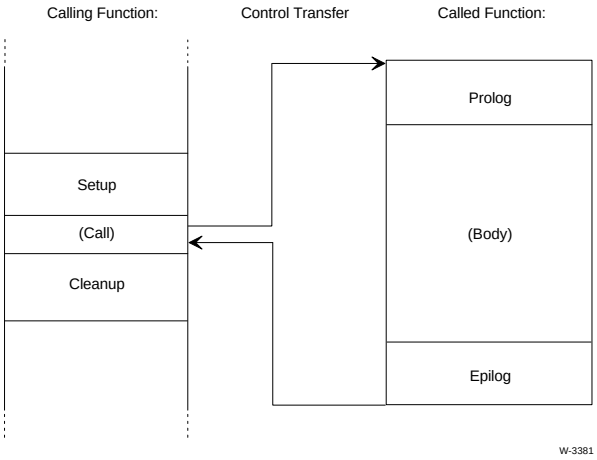

Figure 8-1. Four Sections of Code for a Function Call

The iC-386 compiler supports two calling conventions: fixed parameter list (FPL) and variable parameter list (VPL). The FPL calling convention is the default for the iC-386 compiler 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-386, use the `fixedparams` control for the FPL convention and the `varparams` control for the VPL convention.

See also: Individual control descriptions in Chapter 3

Note

The iC-386 compiler uses the fixed parameter list (FPL) calling convention as its default. This feature produces more compact code. Intel C compilers for Intel386 and Intel486 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.

Passing Arguments

A calling function passes some or all of its arguments to the called function on the processor stack. These 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-386 compiler always uses pass-by-reference for passing arrays and pass-by-value for other objects. The calling function's setup code pushes arguments onto the stack.

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.

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.

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.

Table 8-1 shows the registers used for different scalar objects for iC-386.

Table 8-1. 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

Saving and Restoring Registers

The FPL and VPL calling conventions preserve different sets of registers. The VPL calling convention preserves the EDI, ESI, and EBX registers. Table 8-2 shows the register preservation scheme of iC-386 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 8-2. iC-386 FPL and VPL Register Preservation

Register	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	X	
ESI		X	X	
CS	X		X	
DS	X		X	
SS	X		X	
ES	X		X	
FS		X		X
GS		X		X

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 numeric coprocessor stack and is overwritten by the next floating-point operand.

❑❑❑

9 Subsystems

This chapter tells how to use subsystems to create extended segmentation models, and contains these 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 Intel386, Intel486, and Pentium architectures. The compact segmentation memory model described in Chapter 4 is the standard way that iC-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 compact segmentation memory model 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 these advantages:

- 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.

Use compact subsystems for iRMX applications.

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.

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.

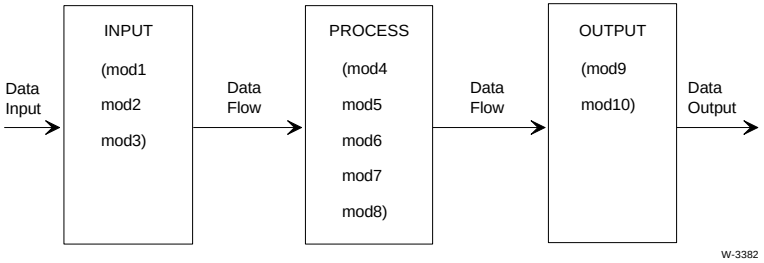

Figure 9-1. Subsystems Example Program Structure

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.

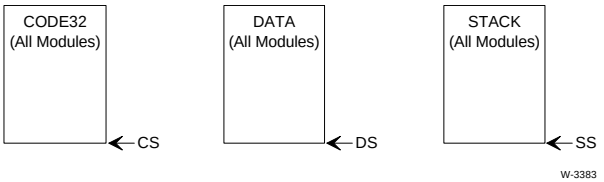

Figure 9-2. Subsystems Example Program in Regular Compact Segmentation Memory Model

Suppose the program is restructured using an extended segmentation model composed of three compact-model subsystems. Each subsystem is given a name indicating its function:

Subsystem Name	Modules in Subsystem
SUBINPUT	mod1 through mod3
SUBPROCESS	mod4 through mod8
SUBOUTPUT	mod9 and mod10

In a program composed of compact-model subsystems, modules are combined by the binder so that:

- Each subsystem has one code segment.
- Each subsystem has one data segment.
- All subsystems share one stack segment.

Figure 9-3 shows the segments for the example if the modules are grouped into three small-model subsystems.

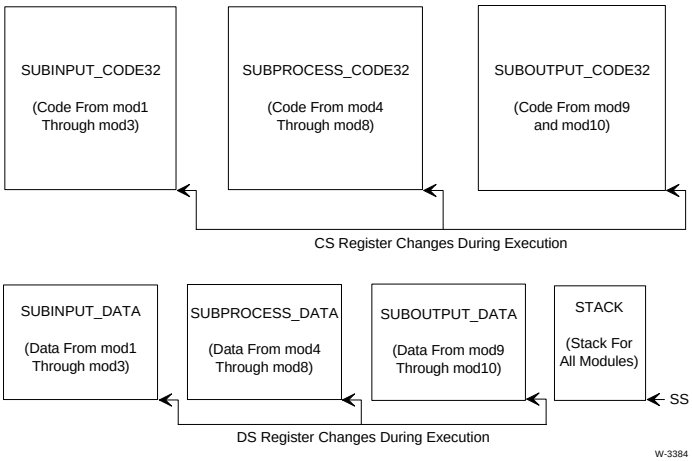

Figure 9-3. Subsystems Example Program Using Small-model Subsystems

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.

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 in 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.

Segment Combination in Subsystems

Chapter 4 describes the way 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-386 and combining modules into a program with 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 binder creates an executable program by combining the segments that have compatible attributes.

See also: Chapter 4 for more information on segment attributes that the binder uses, such as like names

Both the standard segmentation control `compact` and the extended segmentation control `subsys` determine the way segments are combined by controlling the way segments are named.

Compact-model Subsystems

Recall that the binder combines compiler-generated segments that have the same name and compatible characteristics. A linked compact-model subsystem named `COMPSUB` contains three segments: `COMPSUB_CODE32` for iC-386, `COMPSUB_DATA`, and `STACK`. When code in the subsystem is executing, the CS register contains the selector for `COMPSUB_CODE32`, the DS register contains the selector for `COMPSUB_DATA`, and the SS register contains the selector for `STACK`.

Table 9-1 shows 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 Open and Closed Subsystems, the identifier and an underscore (_) prefix the `CODE32` and `DATA` segment names.

Table 9-1. 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 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 4 gigabytes for iC-386 and its own data segment up to 4 gigabytes for iC-386. All stack segments from all compact-model subsystems are combined into one stack segment up to 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 also: near and far address formats in Chapter 4

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.

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. Data references to and from other 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 9-2 shows the default address sizes for code and data references in all subsystem models.

See also: `near` and `far` keywords in Chapter 4,
`extend` control description in Chapter 3

Table 9-2. Subsystems and Default Address Sizes

Subsystem Model	Code Reference	Data Reference
compact RAM	offset	selector and offset
compact ROM	offset	selector and offset

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. 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.

See also: `subsys` control description in Chapter 3

Open and Closed Subsystems

The subsystems that make up an iC-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 these 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 the Syntax section. 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.

Syntax

Defining subsystems tells the compiler:

- 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 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 binder resolves the addresses for such symbols. These definitions identify public and external symbols:

Public variable	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.
Public function	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.
External function	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.
External function	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 this 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. Case is not significant in the compact 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.

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 these :

- 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 in 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 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-386 and in PL/M-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 `$INCLUDE`. The compiler ignores lines whose first character is not a dollar sign (`$`).

Example Definitions

Recall the example program in Dividing a Program into Subsystems. This example guides you through creating subsystem definitions for the compact model subsystems in Figure 9-3.

Creating Three Compact-model RAM Subsystems

These subsystem definitions define three compact 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.

```
$ compact (SUBINPUT
$          has mod1, mod2, mod3)
$ compact (SUBPROCESS
$          has mod4, mod5, mod6, mod7, mod8;
$          exports process_entry)
$ compact (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 general form:

```
/* in SUBINPUT                        */

int data_object;

int mod3_fn ()
{
   extern int process_entry ((int far *)int far * );

...

/* calling a function in another     */
/* subsystem causes a load to a      */
/* segment register                  */

   process_entry ( &data_object );
...

}

/*-------------------------------------------------*/

/* in SUBPROCESS                     */

int process_entry (int far * 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 `compss.def`, the compilation of `mod3.c` is:

```
C:> ic386 mod3.c cp subsys(compss.def)
```

□□□

Language Implementation 10

This chapter contains information on the iC-386 implementation of the C programming language, and is divided into these 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.

Data Types

The iC-386 compiler recognizes three classes of data types: scalar, aggregate, and `void`. This section describes the iC-386 implementation of the 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. These are keywords in iC-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
```

These additional keywords are supported by iC-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 also: Using the `near` and `far` qualifiers, in Chapter 4

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 (4 bytes for iC-386).

Table 10-1 shows the scalar data types for iC-386, the amount of memory occupied by the data type's object, the arithmetic format, and the range of accepted values.

The iC-386 compiler supports 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. 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	4 or 8	integer	0 to 4,294,967,295 or 0 to 2^{64}-1
signed long	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).

The iC-386 compiler supports 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.

Aggregate Types

An object of an aggregate type is a group of one or more scalar objects. These are the iC-386 aggregate data types:

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 also: `align` control and the allocation of structures in Chapter 3

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.

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. These 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    */
```

iC-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-386 compiler supports these features unless otherwise noted.

Lexical Elements and Identifiers

Trigraphs allow C programs to be written without using characters reserved by ISO (International Standards Organization) as alphabet extensions.

Character constants and string literals can contain numeric escape codes in hexadecimal format.

Wide characters support very large character sets, such as pictographic alphabets. The iC-386 compiler recognizes 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.

Preprocessing

The operator concatenates adjacent tokens in macro definitions, forming a single token.

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.

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-386 compiler controls can be used in a `#pragma` preprocessor directive.

See also: Using `#pragma` and compiler control syntax in 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 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 also: List of predefined macros in Chapter 5

Implementation-dependent iC-386 Features

This section provides additional information about how iC-386 implements the implementation-dependent characteristics of the C language as specified by the ANSI C standard.

The compiler's word size is 4 bytes for iC-386. By default, memory read and write operations in the Intel386, Intel486, and Pentium processors occur from low-order address to high-order address (little endian). Objects over 32 kilobytes do not conform to ANSI standards for pointer arithmetic.

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-386 compiler recognizes 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 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.

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.

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.

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-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-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-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-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.

Register Variables

The ESI and EDI 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-386 compiler allocates registers for register objects in this 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.

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 also: Virtual symbol table capacity in this chapter

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 also: Using the `align` control to allocate bit fields on word boundaries in Chapter 3

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.

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 these:

- 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-386 compiler allows 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, this line is invalid in the ANSI C standard:

```
int (const i), volatile j;
```

However, the iC-386 compiler recognizes 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 these 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 also: `alien`, `far`, and `near` type qualifiers in Chapter 4

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-386 `optimize` control governs association of subexpressions in evaluation.

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.

□□□

Messages 11

The iC-386 compiler can issue these types of messages:

- Fatal errors
- Errors (syntax and semantic)
- Warnings
- Remarks
- Subsystem diagnostics
- Internal errors
- I/O 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-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.

Fatal Error Messages

Fatal error messages have the syntax:

```
iC-386 FATAL ERROR
message
```

These are the fatal error messages, in alphabetic order:

`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 also: Compiler control syntax in Chapter 3

`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 RadiSys customer service.

`control``control cannot be negated`

You cannot use the `no` prefix with 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 also: Negating compiler controls in Chapter 3

`duplicate` *`control`* `control`

A control that must not be specified more than once was specified more than once. Only these controls can be specified more than once:

```
align         include             subsys
define        interrupt           varparams
fixedparams   searchinclude
```

See also: Individual control descriptions in Chapter 3

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 also: Macro limits in Chapter 10; and the related error message, `macro expansion too nested`

`internal error: invalid dictionary access, case 3`

This error occurs when the compiler is used in a DOS window for Windows 3.0 or 3.1.

- You may not have enough expanded memory for the compiler. Try compiling the source file outside of Windows, and if this is successful, make more expanded memory available to the compiler.
- If you are using `emm386`, do not set the `noems` switch.

`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 also: List of iC-386 controls in Chapter 3

`invalid control syntax`

The compiler control contained a syntax error. 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.

See also: Compiler control syntax in Chapter 3

`invalid decimal parameter: value`

Non-decimal characters were found in an argument that must be a decimal value. 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.

See also: Compiler control syntax in Chapter 3

`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 also: Compiler control syntax in Chapter 3

`missing or misplaced right parenthesis`

A right parenthesis is required to delimit arguments to a compiler control. 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.

See also: Compiler control syntax in Chapter 3

`no more free space`

The internal structure used to hold macros is exhausted. Use fewer macros in your program.

See also: Macro limits in Chapter 10

`null argument for control control`

Null arguments for compiler controls are not allowed. For example, this 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. 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. 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. 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.

See also: Argument values accepted by compiler controls, in Chapter 3

`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 also: Switch statement limits in Chapter 10

`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 also: Search limits in Chapter 10

`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
- Omitting a semicolon from a function declaration, as in this code:

```
struct a { int c} s /* No semicolon after s */
main () {s.c = 0;}
```

  You can correct the problem by adding a semicolon after the declaration of the structure `s`:

```
struct a { int c} s; /* Semicolon added after s */
main () {s.c = 0;}
```

- Not defining a macro for a user-defined name for a standard data type, as in this code:

```
INT i;
main()
{ i = 0; }
```

  You can eliminate the error by using:

```
#define INT int
```

`whiles, fors, etc. too deeply nested`

The statement nesting structure of the module exhausted an internal structure in the compiler.

See also: Nesting limits in Chapter 10

Error Messages

Syntax error messages have the 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 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 4 gigabytes for iC-386.

`attempt to use 0 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. This 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 4 gigabytes for iC-386. Break the module into two or more separately compiled modules, or use subsystem definitions.

  See also: Defining subsystems in Chapter 9

`conditional compilation directive is too nested`
: The module contains more than the maximum number of conditional statements.

  See also: Nesting limits in Chapter 10

`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 4 gigabytes for iC-386. Break the module into two or more separately compiled modules, or use subsystem definitions.

  See also: Defining subsystems in Chapter 9

`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, this 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.

`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 this 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 also: Syntax of the built-in function calls in Chapter 6

`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 Intel486- and Pentium-specific built-in functions only with the `mod486` control. Use Intel386-specific built-in functions only with the iC-386 compiler.

See also: Built-in functions in Chapter 6

`invalid cast`

These 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 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 Intel386, Intel486, or Pentium processor special registers are available for use in built-in functions. The register number specified must be a numeric constant.

  See also: Intel386, Intel486, and Pentium processor special registers in Chapter 6

`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 also: External declaration limits in Chapter 10

`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. Macro recursion, direct or indirect, can also cause this error.

  See also: Nesting limits in Chapter 10

`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 nesting level limits in Chapter 10.

`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 the maximum allowed.

  See also: Macro limits in Chapter 10

`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 cannot 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 this 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 also: Optimization in Chapter 3

`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 also: Switch statement limits in Chapter 10

`too many active functions`
: The number of function calls within a single expression has exceeded the compiler limit.

  See also: Function call limits in Chapter 10

`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 also: Character constant size for your target processor in Chapter 10

`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 also: External identifier limits in Chapter 10

`too many functions`
: Too many functions were declared.

  See also: Function limits in Chapter 10

`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 also: Macro limits in Chapter 10

`too many nested calls`
: The nesting limit for functions called in function argument lists has been exceeded.

  See also: Nesting limits in Chapter 10

`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 also: Function parameter limits in Chapter 10

`too many parameters for one macro`
: The maximum number of parameters specified for one macro was exceeded.

  See also: Macro parameter limits in Chapter 10

`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.

Warnings

Warnings have the 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 also: List of compiler controls that can be negated in Chapter 3

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 also: Line length limit in Chapter 10

`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 also: Subsystems in Chapter 9

`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 OS.

`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-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 also: `ram` and `rom` compiler controls in Chapter 3

`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 also: Compiler control syntax in Chapter 3

`invalid decimal parameter: `*`value`*

Non-decimal characters were found in an argument that requires a decimal value. 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 also: Compiler control syntax in Chapter 3

`missing or misplaced right parenthesis`

A right parenthesis is required to delimit arguments to a compiler control. 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. For example, this 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. 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.

See also: Compiler control syntax in Chapter 3

`parameter not allowed for negated` *`control`* `control`

Negated controls generally do not accept arguments (`noalign` is the only exception). 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 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. 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.

See also: Values accepted by compiler controls in Chapter 3

`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 also: Compiler control syntax in Chapter 3

`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.

Remarks

Remarks have the 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 this 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 (`*/`).

Subsystem Diagnostics

Subsystem diagnostic messages have the 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 also: Subsystem identifiers in Chapter 9

`invalid control`
An unrecognized control is in the subsystem definition.

See also: Subsystem definitions in Chapter 9

`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.

Internal Error Messages

Internal error messages have the syntax:

```
internal error:  message
```

If your compilation consistently produces any of these errors, contact your RadiSys representative.

iRMX Condition Codes in Error Messages

In some cases, such as a reference to a non-existent file, the compiler passes iRMX condition codes, as shown below.

See also: List of iRMX condition codes, *System Call Reference*

```
ERROR: EXCEPTION: 0021H FILE DOES NOT EXIST
```

Check to see that the specified file exists.

If you are using a submit file and it uses library files, check that the path to the library files is fully expanded.

If you have just installed DOSRMX, verify that the *autoexec.bat* file sets the path to the C header files, then reboot your system.

□□□

Glossary

Absolute address	An address in memory relative to the beginning of memory.
Access attributes	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)	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, BND386	The utility that performs linking. The binder combines segments with like names and resolves symbolic addressing.
Build file	A file of system implementation definitions used by BLD386, to create an absolutely-located system. The definitions describe system data structures, initial values for the system, and memory configuration.
Builder, 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, Intel486, and Pentium 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 32-bit registers (EAX, EBX, ECX, or EDX for Intel386, Intel486, and Pentium 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	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	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, Intel486, and Pentium 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.
Expand-down	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;far, definition G-	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 dot character.
Filename extension	The part of a filename that is right of the dot character.
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, Intel486, and Pentium processors.
Gate	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)	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)	The system register that contains the base address and limit of the global descriptor table.
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, ESI or EDI (for Intel386, Intel486, and Pentium processors) that you use for addressing operands during execution.
Instruction set	The executable elements of the object code.
Interrupt descriptor table (IDT)	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)	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)	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)	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)	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 Intel387 coprocessor, or the Intel486 or Pentium processor on-chip floating-point unit.
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 zero 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 (EBP for Intel386, Intel486, and Pentium processors) and stack pointer (ESP for Intel386, Intel486, and Pentium 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	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	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	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	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)	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	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	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	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	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 Intel family processors. In C programming, a word is the amount of storage reserved for an integer, which is 32 bits for iC-386. The Intel386, Intel486, and Pentium processor documentation and ASM386 instruction sets refer to a 16-bit word and a 32-bit word.
Work file	A file that the compiler creates, uses, and deletes during compilation.

□□□

Index

operator, 225
operator, 225
#define preprocessor directive, 60, 61
#DELETE#__DATE__ macro, 126
#DELETE#__FILE__ macro, 126
#DELETE#__LINE__ macro, 126
#DELETE#__STDC__ macro, 126
#DELETE#__TIME__ macro, 126
#DELETE#_ARCHITECTURE_ macro, 127
#DELETE#_FAR_CODE_ macro, 127
#DELETE#_FAR_DATA_ macro, 127
#DELETE#_LONG64_ macro, 127
#DELETE#_NPX_ macro, 127
#DELETE#_OPTIMIZE_ macro, 127
#DELETE#_ROM_ macro, 127
#elif preprocessor directive, 225
#error preprocessor directive, 125, 226
#include preprocessor directive, 10, 70, 76, 97, 125, 127, 131, 226
#line preprocessor directive, 125, 127, 226
#pragma preprocessor directive, 40, 206, 211, 215, 225
#undef preprocessing directive, 25
#undef preprocessor directive, 60
$ dollar sign in identifiers
    extend control, 65
%auto assembler macro, 192
%cgroup assembler macro, 183
%code assembler macro, 183
%const assembler macro, 183
%const_in_code assembler macro, 181
%data assembler macro, 183
%dgroup assembler macro, 183
%dint assembler macro, 185
%endf assembler macro, 196
%epilog assembler macro, 194
%extern assembler macro, 186
%extern_const assembler macro, 186
%extern_fnc assembler macro, 186
%far_code assembler macro, 181
%far_data assembler macro, 181
%far_stack assembler macro, 181
%fnc assembler macro, 185
%fnc_ptr assembler macro, 185
%fpl assembler macro, 181
%function assembler macro, 189
%i186_instrs assembler macro, 181
%i386_asm assembler macro, 181
%if_nsel assembler macro, 188
%if_sel assembler macro, 188
%int assembler macro, 185
%leave assembler macro, 187
%mov|lsr assembler macro, 188
%movsx assembler macro, 187
%movzx assembler macro, 187
%param assembler macro, 190
%param_flt assembler macro, 191
%popa assembler macro, 187
%prolog assembler macro, 193
%ptr assembler macro, 185
%pusha assembler macro, 187
%pushf assembler macro, 187
%reg_size assembler macro, 185
%ret assembler macro, 195
%sgroup assembler macro, 183
%stack assembler macro, 183
(E)DI register, used for register variables, 203
(E)SI register, used for register variables, 203
.i extension, 16
.lst extension, 16
.obj extension, 14
/lang directory, 9

A

abnormal termination, 12
access rights
    compact-model subsystem, 210
    iC-386 compact model, 116
access rights (iC-386), 114
activation records, 115
address
    of an object, 221
    size, 127
aggregate types, 221, 224
aliasing variables, 86
alien keyword
    extend control, 65
align | noalign control, 45, 46, 47
    examples, 47, 48, 50
ANSI C standard, 5, 7, 25, 65, 221, 225, 227
    conformance, 126, 201
application development, 2
    examples, 24
    modular, 4, 22
    tasks, 1
application system, 5
arguments
    maximum number, 231
array, 224, 228
assembler invocation, 177
auto storage class specifier, 228, 229
automatic variables, 115

B

big endian, 162
binder, 207, 215
    combining segments, 209, 210
binding
    compact model, 116
binding (iC-386), 114
bit fields, 222, 229
BLD386, 4, 113
    interrupt gate, 71
block nesting level, 131
blockinbyte function, 145
blockinhword function, 145
blockinword function, 145
blockoutbyte function, 145
blockoutword function, 145
BND286/386
    syntax, 22
BND386, 4, 22
    example, 24
    object control, 24
    rconfigure control, 24
    renameseg control, 24
    using libraries, 24
Bootstrap Loader, 1
buildptr function, 138
built-in functions, 135
byteswap function, 162

C

C libraries, 24
C-386 compatibility, 53, 57
CALL instruction for Intel386 and Intel486 processors, 79
calling convention, 66, 67, 108
calling convention, see also function-calling convention, 198
case significance, 225
    control arguments, 40
    controls, 40
case values
    maximum, 231
casting
    pointer to near, 119
    to and from pointers, 228
causeinterrupt function, 146
char data type, 98
character
    constant, 227
    set, 227
    strings, 115
cleaning up the stack, see fixed parameter list and variable parameter list
cleanup code, 198, 203
cleartaskswitchedflag function, 151
CODE
    compact-model subsystem, 210
    iC-86/286 compact model, 116
code | nocode control, 51
code access
    efficiency, 113, 119

code segment, 53, 54, 94, 115
compact model, 116
compact-model subsystem, 210
CODE32, 209
compact-model subsystem, 210
iC-386 compact model, 116
CODE32 segment name (iC-386), 53
codesegment control, 53, 216
combine-type
compact model, 116
compact-model subsystem, 210
combine-type (iC-386), 114
combining application with iRMX, 1
command line
preserving case, 40
preserving special characters, 44
compact control, 54, 55, 115, 116, 209
compact model, 113, 205, 206, 210
default address size, 116
dynamic data segments, 116
efficiency, 116
maximum program size, 116
number of segments, 116
segment definitions, 116
segments, 116
selector register use, 116
compact-model subsystems, 207
example, 207
far keyword, 210
segment definition, 209
segment definitions, 210
selector, 209
compatibility
function calling conventions, 65
iC-386 with C-386, 53, 57
non-C translators, 4
other intel compilers, 65, 66
with Intel tools, 6
compilation heading, 129, 130
example, 130
compilation summary, 130, 133
compiler capabilities, 5
compiler version, 7, 130
compiling, 125
cond | nocond control, 56, 132
conditional assembler macros, 188
conditional code, 132
in source listing, 56
conditional compilation, 91, 125, 128
example, 61
macros, 60
maximum nesting, 226
conditional directives, 128
const attribute specifier, 94, 230
constants, 94
code or data segment, 115
compact model, 116
compact-model subsystem, 210
definition, 115
continued lines
in source text listing, 131
control arguments
case significance, 40
special characters, 44
control register 0 (CR0), 150, 160
control registers, 159
control word macros
numeric coprocessor, 167, 168
controls, 39, 40, 41, 44
arguments, 10
case significance, 10
debugging, 2
for print file, 73, 74
optimizing, 4
converting
char objects, 98
floating-point to integer, 87
cross-reference listing, 102, 106, 111, 130, 133
CS register
compact model, 55, 116
far function, 119
near variable, 119

D

data
definition, 115
compact model, 116
compact-model subsystem, 210
data access
efficiency, 113, 119
data pointers, 210
compact model, 116
data segment, 54, 57, 94, 115

allocating dynamically, 116
compact model, 116
compact-model subsystem, 210
data types, 221
char, 98
iC-386, 78
void *, 138
datasegment control, 216
iC-386, 57
iC-386, and subsys control, 57
debug | nodebug control, 58
debug information, 58, 106
debug registers, 159
debugging, 84
line control, 72
source file information, 99
using print file, 51
debugging information
compatibility, 3
declaration syntax, 120
default address size, 210
compact model, 116
overriding, 118, 119
examples, 120
segmentation models, 118
default address size (iC-386), 114
define control, 60
example, 61
defined preprocessor operator, 225
descriptor:, see special descriptor. see general descriptor. see gate descriptor
descriptor_table_reg structure, 148
diagnostic control, 12, 62, 63, 132
diagnostic messages, 76, 92, 94, 105, 129, 233
disable function, 146
dollar sign ($), 100, 215
in identifiers
extend control, 65
DOS applications
iC-86 compiler controls, 5
numeric coprocessor, 5
DS register
compact model, 55, 116
near variable, 119

E

eject control, 64
embedded applications, 5
enable function, 146
enumeration types, 229
epilog code, 198
interrupt handlers, 71
error messages, 62, 63, 233, 239
errors, 129, 132
ES register
compact model, 116
de-referencing, 119
far variable, 119
exit status, 63
extend | noextend control, 65, 222
extend control, 118, 208, 210
extended segmentation models, 205
definition, 205
extended syntax, 230
extensions to ANSI C, 65
extern keyword, 212
extern storage class specifier
examples with far type qualifier, 121
external
linkage, definition, 212
External
function, definition, 212
variable, definition, 212
external declaration assembler macros, 186
external references
maximum per module, 231
external symbols, 106
definition, 212
type information, 58

F

far address
compact model, 116
far function, 119
far keyword, 208, 210
extend control, 65
far pointers, 127, 138
compact model, 55
converting to near pointer, 138
converting to selector, 138

far type qualifier, 118, 119
effect, 118
examples, 120, 121, 122, 123
when to use, 118
where to use, 120
far variable, 119
fatal error messages, 233, 234
file use, 13
fixed parameter list (FPL), 66, 108, 198, 222
argument passing, 199
cleaning up the stack, 204
order of arguments on the stack, 199
returning values in registers, 202
saving and restoring registers, 202, 203
fixedparams control, 66, 68, 198
examples, 67
flag assembler macros, 181, 182
flag macros, 142, 143
flags
examples manipulating, 143
FLAGS register, 140
floating-point, 228
in-line functions, 27
libraries, 24
precisions, 223
unit, 135
unit, special functions, 163
using special libraries, 24
floating-point literals, 115
floating-point unit:, see numeric coprocessor
form feed in print file, 64
FS register
de-referencing, 119
FS register (Intel386)
far variable, 119
function
far, 119
near, 119
function activation records, 115
function call
conventions, 66
four sections of code for, 198
maximum arguments, 231
function calling conventions, 67
function definition assembler macros, 188
%auto, 192
%endf, 196
%epilog, 194
%function, 189
%param, 190
%param_flt, 191
%prolog, 193
%ret, 195
function pointers, 210
compact model, 116
function-calling convention
calling function and called function, 199
passing arguments, 199
returning a value, 202
saving and restoring registers, 203
stack use, 204
functions
interfacing, 197
maximum in argument list, 231
maximum per module, 231

G

gate
descriptor, 147
GDTR (global descriptor table register), 149
general controls, 40
getcontrolregister function, 159
getdebugregister function, 159
getflags function, 140
getlocaltable function, 149
getmachinestatus function, 150
getrealerror function, 170
gettaskregister function, 148
gettestregister function, 159
global descriptor table register (GDTR), 148, 149
global functions, 212
global variables, 115, 212
granularity (iC-386), 114
group definition
compact-model subsystem, 210
GS register
de-referencing, 119
GS register (Intel386)
far variable, 119

H

halt function, 139, 146
header controls, 176, 177, 178, 179
    controls assembler macro, 175, 176, 177, 178, 179
        syntax, 177
    defaults, 176
    flag assembler macros, 181, 182
    operation assembler macros, 186
    precedence, 177, 178, 179
    register assembler macros, 182
    segment assembler macros, 183, 184
    type assembler macros, 184
header files, 97
    in-line functions, 25

I

I/O layer, 1
I/O ports
    reading and writing, 144
i186.h header file, 135, 136
i286.h header file, 135
i386.h header file, 135, 137
i387_environment structure type, 173
i387_protected_addr structure type, 172
i387_state structure type, 174
i486.h header file, 135
i8086.h header file, 135
i86.h header file, 135, 136
i87_tempreal structure type, 173
ICU, 1, 5
identifiers, 225
    with dollar signs, 100
IDTR (interrupt descriptor table register), 149
inbyte function, 144
in-circuit emulator, 1, 2
include control, 13, 69, 70, 76, 97, 125, 127, 131
include files, 69, 76, 97, 132
    nesting, 70, 131
inhword function, 144
instruction assembler macros, 187
instruction set, 127
    Intel386 and Intel486, 79
    seeing effect in print file, 51
integers, 227
integral type
    converting to selector type, 138
Intel C
    VPL calling convention, 199
Intel development tools
    application development, 4
    experience with, 9
    host systems, 1
Intel publications
    ordering, 7
Intel486 processor, 79, 135
interactive configuration utility, see ICU
internal error messages, 256
interrupt
    task switch, 146
interrupt control, 71, 147
interrupt descriptor table (IDT), 71
interrupt descriptor table register (IDTR), 148, 149
interrupt gate, 147
    vs. trap gate and task gate, 147
interrupt gate (iC-386), 71
interrupt handlers, 71, 147
    286 and higher processors, 146
interrupt number (iC-386), 71
interrupts
    manipulating, 146
invalidatedatacache function, 162
invalidatetlbentry function, 162
invocation
    example, 130
    messages, 12
    syntax, 10
invocation line
    continuing, 10
    length, 10
invocation-only controls, 40
inword function, 144
iPPS PROM programming software, 5
iRMX memory models, 115

K

keywords, 222

L

language directory, 9
language implementation, 221
large segmentation model, 205, 210
LDTR (local descriptor table register), 149
LIBn86, 4
libraries, 2, 22
    binding, 22
    choosing for binding, 24
    choosing for iC-386, 24
    far calls, 118
    floating-point, 24
    operating system interface, 24
line | noline control, 72
LINK86, 4
    syntax, 22
linker, 207
    combining segments, 209, 210
linking
    compact model, 116
list | nolist control, 73, 74, 132
listexpand | nolistexpand control, 75, 132
listinclude | nolistinclude control, 76, 132
listing, see print file
listing files, 13
little endian, 162, 227
LOC86, 4
local descriptor table register (LDTR), 148, 149
location counter, 132
lockset function, 139
logical names
    language directory, 9
long data type (iC-386), 127
long type qualifier (iC-386), 78
long64 | nolong64 control, 223
long64 | nolong64 control (iC-386), 78
    aligning structures, 48, 50

M

machine status word (MSW), 150
machine status word macros, 151
macro, 91
    defining with define control, 60
    expansion
        In print file, 75
    scope, 69
Macro
    example, 61
macro definition, 25
macro expansion, 132
macro invocation
    maximum arguments, 226
    maximum nesting, 226
macros, 126
    predefined, 126
manual scope, 7
memory model
    compact, 54, 55
memory model:, see also segmentation memory model
messages, 132, 233
    console, 12
    diagnostic, 62
    Diagnostic, 63
    print file, 12
mod486 | nomod486 control, 79
modulename control, 81, 211, 214, 216
    and subsys control, 81

N

name space, 229
near address
    compact model, 116
near function, 119
near keyword, 210
    extend control, 65
near pointers, 127
    compact model, 55
    converting to far pointer, 138
near type qualifier, 118, 119
    effect, 118
    when to use, 119
    where to use, 120
near variable, 119
normal completion, 12
notational conventions, 44
numeric coprocessor, 24, 87, 135
    control word, 164, 166
        macros, 167, 168
    data pointer, 164
    environment, 164, 173

flags, 168
instruction pointer, 164
Intel387, Intel486, and Pentium condition codes, 169
numeric registers, 163
stack top, 168
registers, 164
special functions, 163
state, 164, 173
status word, 164, 168
macros, 171
tag word, 164, 165
macros, 165
numerics libraries for iC-386, 24

O

object | noobject control, 14, 82, 83
object code
components, 115
offset information, limiting, 99
object file, 13
defaults, 14
reducing size, 99
Object file
name, 83
pseudo-assembly listing, 83
object module
name, 81
reducing the size of, 84, 106
size, 133
object module format (OMF), 3
offset-only address, 222
format, 210
OH386, 5
OHn86, 4
operation assembler macros, 186
classes, 186
conditional assembler macros, 188
external declaration assembler macros, 186
function definition assembler macros, 188
instruction assembler macros, 187
optimization, 4, 84, 87
at different levels, 27
reducing debug information, 58, 72
run-time performance, 135
structure aligning, 46
using FPL calling convention, 66
optimization example, 27
level 0, 27, 30
pseudo-assembly code, 30
level 1, 31
pseudo-assembly code, 31
level 2, 33
pseudo-assembly code, 33
level 3, 35
source code, 27
optimize control, 27, 84, 127, 231
order of arguments on the stack, see fixed parameter list and variable parameter list
outbyte function, 144
outhword function, 144
outword function, 144

P

page break in print file, 64
page header, 129, 130
pagewidth control, 89
pass-by-reference arguments, 199
pass-by-value arguments, 199
path prefix, 96, 97
Pentium processor, 135
pointer, 228
compact model, 55
seeing size in print file, 51
pointer indirection, 86
precedence of controls, 40
preprint | nopreprint control, 13, 14, 16, 90, 125
preprint file, 90, 93
contents of, 14, 125
defaults, 14, 16
Preprint file
Defaults, 16
preprocessing, 16, 90, 105, 125
conditional compilation directives, 56
diagnostic messages, 63
macro expansion, 75
preprocessing directives, 128
preprocessor directives, 125, 225
primary controls, 40
primary source file, 10, 69, 90, 91, 92, 96
print | noprint control, 12, 14, 16, 92, 125

print file, 12, 91, 92, 93, 98, 101, 102, 106, 109, 131, 233
assembly code, 51
characters per line, 89
characters per tab stop, 103
contents, 14
contents of, 125
controls that affect contents, 129
defaults, 14, 16
form feed, 64
lines per page, 88
page heading, 88
page numbers, 130
source listing, 56, 73, 74, 75
include files, 69
title in, 104
Print file
source listing
Include files, 76
privilege level, 118
privilege level (iC-386), 114
processor
I/O ports
reading and writing, 144
program
efficiency, 208
programming for ROM, 5
prolog code, 198
interrupt handlers, 71
protected mode
interrupt handlers, 146
protection, 205, 208
levels, 205
prototype, 109
pseudo-assembly code
example, 30, 31, 33
pseudo-assembly language listing, 51
pseudo-assembly listing, 129, 132
Public function
definition, 212
public symbols, 106
definition, 212
name space, 215
type information, 58
Public variable
definition, 212
punctuation in control syntax, 44

Q

quotation marks around control arguments, 40, 44

R

ram control, 94, 115, 127, 209
compact model, 116
reading and writing I/O ports, 144
register assembler macros, 182
register storage class, 202
register variables, 229
registers, 108
related publications, 7
remarks, 62, 63, 129, 132, 233, 254
reserved words, see keywords, 222
restoreglobaltable function, 149
restoreinterrupttable function, 149
restorerealstatus function, 174
rom control, 94, 115, 127, 209
compact model, 116
ROM, programming for, 4
run-time libraries, 2

S

saveglobaltable function, 149
saveinterrupttable function, 149
saverealstatus function, 174
SBITFIELD macro, 173
scalar data types, 221, 222, 223
searchinclude | nosearchinclude control, 69, 96
segment
address in memory, 113
attributes, 113
binding, 113, 116
binding iC-386, 114
compact model, 116
iC-386 characteristics, 114
segment assembler macros, 183
example, 184
segmentation
definition, 205
protection mechanisms, 205
see also memory model, 39
segmentation control, 209

segmentation memory model, 209
choosing for iC-86/286, 113
efficiency, 113
extending with subsystems, 114
implementation, 113
iRMX operating systems, 55
number of segments, 115
segmentation protection mechanisms, 205
segments
attributes, 209
compact-model subsystem, 210
name, 209
segment-selector-and-offset address, 222
segment-selector-and-offset format, 210
selector register
compact model, 55, 116
selector type, 138
converting to far pointer, 138
converting to integral type, 138
setcontrolregister function, 160
setdebugregister function, 160
setflags function, 140
setlocaltable function, 149
setmachinestatus function, 150
setrealmode function, 166
settaskregister function, 148
settestregister function, 160
setup code, 198
signedchar | nosignedchar control, 98, 227
sign-off message, 12
sign-on message, 12
small segmentation model, 208, 210
Soft-Scope debugger, 1
source text
filename, 126
line number, 126
listing, 111, 129, 131
source text listing, 131
special characters in control arguments, 44
srclines | nosrclines control, 99
SS register
compact model, 55, 116
stack, 108
definition, 115
STACK
compact model, 116
compact-model subsystem, 210
stack segment, 115
compact model, 54, 116
compact-model subsystem, 210
statement numbers, 131
statements
maximum nesting level, 231
static keyword, 212
static variables, 94, 115
initializing, 94
status word macros
numeric coprocessor, 171
storage-class specifier, 222
string literals
preprocessing, 225
structure, 224, 229
structure aligning, 45, 46
by structure tag, 46
with typedef, 47
structures
passing and returning, see fixed parameter list and variable parameter list
submit files, 5
subsys control, 97, 100, 205, 206, 209, 211, 216
and modulename control, 101
subsystem definitions, 211
constants, 212, 213
examples, 217, 219
exports keyword, 214
functions and data, 212, 213
has keyword, 214
memory model, 212, 213
modules, 212, 213, 214
syntax, 211, 213
continuation lines, 215
sharing with PL/M, 216
subsystem error messages, 255
subsystems, 100, 205
closed, 209, 211, 214, 216
code segment, 209
compact keyword, 213
compact-model, 207
example, 207
compiling, 206
consistent definitions, 215
-const in code-, 209, 213
-const in data-, 209, 213
constants, 205

data segment, 209
definition, 114, 205
efficiency, 208, 210
example, 206
exported functions, 215
characteristics, 215
exported symbols
name space, 215
exports list, 214, 215
far calls, 118, 208
far data references, 208
far keyword, 215
has list, 211, 215
has specification, 214
implicit declaration modification, 208, 215
module name
name space, 215
near calls, 208
open, 211, 214, 216
RAM and ROM submodels, 205
subsystem-id, 209, 211, 213
name space, 215
switch statement
maximum case values, 231
symbol attributes, 106
symbol tables, 229, 231
symbolic debugger, 2, 106
Symbolic debugger, 58
symbols | nosymbols control, 102, 133
symbols listing, 102, 106, 111, 130, 133
syntax conventions, 44
system address registers, 148
system calls, 71
System Debugger, 1

T

tabwidth control, 103
tag word macros
numeric coprocessor, 165
target environments, 5
task gate, 147
vs. interrupt gate and trap gate, 147
task register (TR), 148
task switch in nested interrupt task, 146
tempreal_t typedef, 174
test registers, 159
title control, 104
translate | notranslate control, 12, 14, 16, 105
translation, 105
trap gate, 147
vs. interrupt gate and task gate, 147
trigraphs, 225
type | notype control, 106
and debug control, 58
type assembler macros, 184
type checking, 58, 106
type information, 106
type qualifiers, 222
interpreting, 121
near and far keywords, 118, 210
type table, 231
typedef, 3
aligning structures, 47

U

union, 224, 229
util.ah header file, 175
assembling with, 177
controls assembler macro, 175, 176, 177, 178, 179
header controls, 176, 177, 178, 179
including in assembly text, 175
syntax, 177
macro groups, 175
utilities, 4

V

variable parameter list (VPL), 66, 108, 198
argument passing, 201
cleaning up the stack, 204
example, 110
order of arguments on the stack, 199
returning values in registers, 202
saving and restoring registers, 202, 203
variables
aliasing, 86
far, 119
near, 119
static, 94
varparams control, 108, 198
examples, 68

version of compiler, 7
void * data type, 138
void data type, 221, 224
void type specifier
    interrupt handlers, 71
volatile attribute specifier, 230

W

warnings, 62, 63, 129, 132, 233, 249
wbinvalidatedatacache function, 162
wide characters, 225
word size, 227
work files, 13

X

xref | noxref control, 111, 133