



Microtec® C/C++ Compiler User's Guide and Reference Manual for the PowerPC Family

Software Version 3.8

Part Number 227464

January 2015

**© 1996-2015 Mentor Graphics Corporation
All rights reserved.**

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

U.S. GOVERNMENT LICENSE RIGHTS: The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

TRADEMARKS: The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: www.mentor.com/trademarks.

The registered trademark Linux[®] is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

Mentor Graphics Corporation
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777
Telephone: 503.685.7000
Toll-Free Telephone: 800.592.2210
Website: www.mentor.com
SupportNet: supportnet.mentor.com/

Send Feedback on Documentation: supportnet.mentor.com/doc_feedback_form

Table of Contents

Chapter 1

Introduction.....	19
Components of the Microtec Toolkit.....	19
MCCPPC C Compiler.....	19
CCCPPC C++ Compiler.....	19
Compiler Features.....	19
ASMPPC Assembler.....	20
LNKPPC Linker.....	20
LIBPPC Object Module Librarian.....	21
EDGE Debugger.....	21
Embedded System Support.....	21
Data Flow.....	22

Chapter 2

Using the Compilation Driver.....	23
Invoking the Compilation Driver.....	23
Command Line Syntax.....	23
Filename Extensions.....	24
Input and Output File Locations.....	25

Chapter 3

Using Command Line Options.....	27
Command Line Options.....	27
Command Line Option Summary.....	28
Command Line Options - Expanded Descriptions.....	37
Set ANSI-Compliant Mode (C Compiler Only).....	37
Control ROM Data Addressing.....	37
Save Comments.....	38
Produce Object File Only (Driver Option).....	38
Define a Preprocessor Macro Name.....	38
Read Options From File.....	38
Display Preprocessor Output.....	39
Pass Command File to Linker (Driver Option).....	39
Redirect Diagnostic Messages.....	39
Specify Format of Listing Files.....	39
Control Floating Point Code Generation.....	40
Produce Debugging Information.....	40
Run-Time Checks.....	41
Run-Time Error Reporting Function.....	43
Save Assembly File (Driver Option).....	44
Add Search Path for Nonstandard Include Files.....	44
Add Search Path for Standard Include Files.....	45

Use Precompiled Headers	45
Produce Minor Code Generation Variations.	46
Generate Listing File.	52
Generate Position-Independent Code and Data	52
Rename Compiler Predefined Sections	55
Optimize Code	56
Name the Output File (Driver Option)	58
Send Preprocessor Output to File (Driver Option)	59
Produce Code for Specified Processor	59
Control Diagnostic Messages	60
Use Default Libraries for Linking.	62
Produce Assembler Source File (Driver Option)	62
Control Template Instantiation	62
Undefine a Preprocessor Macro Name	63
Modify Naming Conventions	63
Control Verbose Output Information (Driver Option)	64
Pass Options to Tools (Driver Option)	64
Initialize Uninitialized Global Data	65
Enable Microtec Extensions	66
Accept Out-of-range Floating Point Constants.	67
Enable GNU Extensions	67
Check Program Syntax (Driver Option)	67
Produce Minor Code Generation Variations (-Zx options)	67
Enable C++ Features.	67
Using Options	68
Customizing the Compilation Driver	68
Pragmas and Options	69
Locating Header Files	70
Determining Option Defaults	71
Producing Listing Files	71
Command Line Examples	72
Suggested Option Combinations	73

Chapter 4

C/C++ Extensions	75
Microtec C/C++ Extensions.	75
Predefined Mirror Symbols.	75
Utilities	79
_STR_CMP/_STR_CMP	81
_STR_NOCASE_CMP/_STR_NOCASE_CMP	82
_SUBSTR/_SUBSTR	83
_STRIP_QUOTES/_STRIP_QUOTES	84
_TOLOWER/_TOLOWER	85
_TOUPPER/_TOUPPER	86
Preprocessor Directives	88
#error — Issues an Error Message	89
#informing — Issues an Informational Message	90
#pragma asm — Begins Embedded Assembly Instructions	91

Table of Contents

#pragma do_not_instantiate — Disables Instantiation of Specified Template	92
#pragma eject — Controls Page Listing	93
#pragma endasm — Ends Embedded Assembly Instructions	94
#pragma error — Issues an Error Message and Halts Compilation	95
#pragma info — Issues an Informational Message	96
#pragma instantiate — Instantiates Specified Template Manually	97
#pragma list — Controls Source Line Listings	98
#pragma macro — Lists Predefined Processor Symbols	99
#pragma options — Specifies Command Line Options	100
#pragma warn — Issues a Warning Message	101
#warning — Issues a Warning Message	102
Type Cast on Left	103
Void Pointer Arithmetic	103
Type Information	104
Keywords	105
Include File Optimization	107
asm Support	108
Predefined Identifier __func__	108
Variadic Macros	109
Loop-Scoped Variables	110
Compound Literals	111
C++ Style Comments	111
inline Keyword	112
Dispersed Statements and Declarations	112
Variable Length Arrays	112
_Bool Keyword	112
Supported GNU C/C++ Extensions	112
GNU Extensions	112
GNU C Mode	115
GNU C++ Mode	117
Unsupported GNU Features	121

Chapter 5

Using Libraries 123

C/C++ Compiler Libraries	123
Library Use	124
C Libraries	125
Non-Reentrant Functions	126
C++ Library	129
Language Support	130
Diagnostics	130
General Utilities	130
Strings	131
Localization	131
Numerics	131
Input/Output	131
Standard Template Library	132
Intrinsic System Functions	133

Library Extensions	134
C Function Groups and Include Files	134
Non-Reentrant Extensions	136
I/O System Functions	136
Single Precision Math Functions	137
Library Function Definitions	138
_cxxfini — Calls Static Destructors to Destroy Static Objects	139
eprintf — Provides Formatted Print to Standard Error	140
fileno — Gets File Descriptor	141
ftoa — Converts Floating Point Number to ASCII String	142
getl — Reads a Long Integer From a Stream	143
getw — Reads a Short Integer From a Stream	144
isascii — Tests for an ASCII Character	145
itoa — Converts Integer to ASCII String	146
itostr — Converts Unsigned Integer to ASCII String	147
ltoa — Converts Long Integer to ASCII String	148
ltostr — Converts Unsigned Long Integer to ASCII String	149
_main — Calls Static Constructors to Create Static Objects	150
max — Returns the Greater of Two Values	151
memccpy — Copies Characters in Memory	152
memclr — Clears Memory Bytes	153
min — Returns the Lesser of Two Values	154
open — Opens a Specified File	155
__pure_virtual_function_called — Handles Abnormal Conditions With Pure Virtual Functions	157
putl — Writes a Long Integer to a Stream	158
putw — Writes a Short Integer to a Stream	159
sbrk — Allocates Memory Space	160
swab — Swaps Odd and Even Bytes in Memory	161
toascii — Converts a Byte to ASCII Format	162
_tolower — Converts Characters to Lowercase Without Argument Checking	163
_toupper — Converts Characters to Uppercase Without Argument Checking	164
unlink — Unlinks a Filename	165
write — Writes Bytes to a Specified File	166
zalloc — Allocates Data Space Dynamically	167
Chapter 6	
C Library Customizer	169
When to Create a Custom Library	169
Directories in the C Library Customizer	169
Windows System Requirements	170
Using the C Library Customizer	170
Step 1: Creating a Custom Configuration File	171
Step 2: Building a Custom Library	173
Step 3: Testing a Custom Library	174
Using a Custom Library	176

Table of Contents

Using Custom Libraries	176
Chapter 7	
Optimizations	177
General Optimizations	177
Algebraic Simplification	177
Redundant Code Elimination	177
Strength Reduction	178
Global Optimizations	178
Dead Code Elimination	178
Factorization	180
Constant Propagation	180
Copy Propagation	181
Register Allocation	181
Unused Definition Elimination	181
Loop Optimizations	182
Local Optimizations	184
Common Subexpression Elimination	184
Constant Folding	184
Redundant Load and Store Elimination	184
Jump Optimizations	185
Branch Tail Merging	185
Code Hoisting	185
Multiple Jump Optimization	186
Redundant Jump Elimination	186
Function Inlining	187
inline Keyword	187
Instruction Scheduling	188
Chapter 8	
Template Instantiation	189
Templates Versus Macros	189
Inline Macro	189
Macro Function	189
Function Template	190
Compile-Time Template Instantiation	190
Declare the Class Template Stack in stack.h	191
Define the Class Template in stack.def.	191
Use Template Class in Source File user1.cc	192
Automatic Instantiation	192
Scope of Template Instances	192
Manual Template Instantiation	193
Specialization	193
Chapter 9	
Precompiled Header Files	195
Precompiled Header File Processing	195
Using Options With Precompiled Header Files	197

Using the Preprocessor With Precompiled Headers	197
Precompiled Header File Usage Guide	198
Chapter 10	
Interlanguage Calling.....	201
Parameter Passing	201
Setting Parameters	201
Integer Parameters	201
Floating Point Parameters.....	202
Structure Parameters.....	202
Implicit Parameter for Structure/Union Return Value	202
Function Return Values	202
Register and Stack Usage With Functions.....	203
Stack Frames.....	204
Prologue	204
Epilogue	205
Assembly Language Interfacing	205
Assembly Language Routine Example.....	206
Variable References from Assembly Routines.....	206
Defining a Function	206
Calling C Functions From C++	207
Passing Arguments	207
Calling C++ Functions From C	209
Overloaded Functions.....	209
Member Functions	209
Chapter 11	
Internal Data Representation	213
Storage Layout.....	213
Data Type Summary	214
Structure Operations	215
Structure Alignment	215
Bit Fields.....	216
Allocation of Variables	219
Register.....	220
Local Variables.....	220
Static Variables.....	220
Memory Allocation	220
Big-Endian	220
Little-Endian	220
Chapter 12	
Run-Time Organization.....	221
Code Organization.....	221
Compiler-Generated Sections.....	221

Chapter 13

Embedded Environments..... 225

Embedded Applications.....	225
Considerations for Embedded Systems.....	225
Inline Assembly.....	226
Assembler Inlining Features.....	228
Variables in asm and Optimization Levels.....	232
#pragma asm or asm.....	233
Considerations for Assembler Inlining.....	234
Addressing.....	235
Direct Memory and I/O Port Addressing.....	235
Calling a Function at an Absolute Address.....	235
Interrupt Handlers.....	236
Built-in Exception Processing.....	238
Reentrant Code.....	238
Position-Independent Code and Data.....	240
Position-Independent Versus Position-Dependent.....	240
Programming Considerations.....	241
Assembler and Linker Considerations.....	242
Absolute Versus Register-Relative Addressing.....	242
User-Modified Routines.....	242
User-Modified Routines for Embedded Systems.....	242
Removing Unneeded I/O Support.....	244
Removing Unneeded Floating Point Support.....	245
Data Initialization.....	245
Saving Initialized Variables in ROM.....	245
Using the Linker.....	246
Default Sections.....	246
Libraries.....	246
INITDATA Command.....	246
Linker Command Example.....	247
Linker Command Example (ROM-Based System).....	248
Static Initialization and Termination.....	250
Static Constructor.....	251
Static Destructor.....	251
init Section.....	251
Startup and Exit Routines.....	251
I/O Static Initialization and Termination.....	252

Chapter 14

Compiler Output Files..... 253

Assembly Source File.....	253
Advantages of Producing an Assembly File.....	253
Variable Names.....	253
Code and Data Sections.....	253
Compiler Output Listing.....	253
Generating an Executable Program.....	254
Invoking the Linker.....	255

Appendix A

Compiler Messages	257
Message Severity Levels	257
Controlling Severity	258
Error Position Marker	258
Suppressing Error Messages	259
Reporting Problems	261
Preparing a Test Case	261
Calling Technical Support	261
C and C++ Compiler Messages	262
Expanded Descriptions	314

Appendix B

C and C++ Differences	325
C++ Program Structure	325
void*	325
Global and Local Scope Issues	325
Declarations	326
Keywords	329
C++ Grammar Rules	329
Functions	330
Comments	332
Internal Data Representation	332
Enumeration Types	332
Arrays	333
Nested Structures	333
Enumerators Within Structures	334

Appendix C

Migration Guide	337
Optimization Option Changes	337
Option Specification Format Changes	338
Removed Options	339
Added Behaviors	339

Appendix D

XRAY Commands and Functionality	341
I/O System Functions	341
Functions Implemented for Use With SysHost	341
User-Modified Routines for Embedded Systems	341
In Character Routine	341
Out Character Routine	342
System Functions and the SysHost Feature	342
SysHost Functions	343
chdir — Changes Working Directory	344
close — Closes a Specified File	345
connect — Initiates a Socket Connection on the Host System	346
creat — Creates a Specified File	347

Table of Contents

_exit — Terminates a Program	348
getcwd — Returns Current Working Directory	349
lseek — Sets the Current Location in a File	350
_pclose — Closes a Pipe from a Process	351
_popen — Opens a Pipe to a Process	352
read — Reads Bytes From a Specified File	353
socket — Creates a Communication Endpoint on the Host System.	354
stat — Gets Information About File	355
sys_in — Reads Characters From Standard Input Window	356
sys_out — Sends Characters to Standard Output Window	357
sys_stat — Checks for Input Characters Waiting.	358

Glossary

Index

Third-Party Information

Embedded Software and Hardware License Agreement

List of Examples

Example 3-1. Negating Options With -n	27
Example 3-2. Position-Independent Code	53
Example 3-3. Position-Independent Data	54
Example 3-4. Overriding Command Line Options	68
Example 3-5. Overriding Command Line Options in a Script	68
Example 3-6. Options File	69
Example 3-7. Multiple #pragma options Directives	69
Example 3-8. #pragma options Directives Across Modules	70
Example 3-9. #pragma options push and pop	70
Example 3-10. Locating Header Files	71
Example 3-11. Turning a Listing File Off and On.	72
Example 3-12. Simple C++ Source Compilation	72
Example 3-13. C++ Compilation From Multiple Sources.	72
Example 3-14. Producing an Object File.	73
Example 3-15. Compiling and Assembling Multiple Files With a Library.	73
Example 4-1. Using Compiler Mirror Symbols.	76
Example 4-2. Using the _VARIANT Symbol.	78
Example 4-3. Using the _BUS_WIDTH Symbol	79
Example 4-4. Using Section Information Mirrors.	79
Example 4-5. __OPTION_VALUE Macros	87
Example 4-6. Penalties in lvalue Type Casting	103
Example 4-7. Void Pointer Arithmetic	104
Example 4-8. Using the typeof Operator	105
Example 4-9. Using a C++ Interrupt Handler	105
Example 4-10. __func__ Identifier	108
Example 4-11. Variadic Macros	109
Example 4-12. Loop-Scoped Variables	110
Example 4-13. Compound Literals	111
Example 7-1. Algebraic Simplification	177
Example 7-2. Redundant Code Elimination	178
Example 7-3. Strength Reduction	178
Example 7-4. Dead Code Elimination	179
Example 7-5. Dead Code Elimination With an if Statement.	180
Example 7-6. Factorization	180
Example 7-7. Constant Propagation	181
Example 7-8. Copy Propagation	181
Example 7-9. Unused Definition Elimination	182
Example 7-10. Loop Invariant Code Optimization	182
Example 7-11. Loop Rotation	183
Example 7-12. Strength Reduction and Index Simplification	184

List of Examples

Example 7-13. Common Subexpression Elimination	184
Example 7-14. Constant Folding	184
Example 7-15. Redundant Load and Store Elimination	184
Example 7-16. Branch Tail Merging	185
Example 7-17. Code Hoisting	185
Example 7-18. Multiple Jump Optimization	186
Example 7-19. Redundant Jump Elimination	186
Example 7-20. Function In-Lining	187
Example 7-21. The inline Keyword	188
Example 9-1. Precompiled Header Processing	195
Example 10-1. Passing Arguments Between C and C++	208
Example 11-1. Structure Alignment	216
Example 11-2. Declaring Bit Fields	216
Example 11-3. Bit Field Allocation	217
Example 13-1. Using Variables Within asm()	229
Example 13-2. #pragma asm	233
Example 13-3. Interrupt Assembly Code	237
Example B-1. void Pointers	325
Example B-2. Global and Local Declarations	326
Example B-3. Declaring const for C++	327
Example B-4. Conversion Operator	329
Example B-5. Function Declarations	330
Example B-6. Prototype Declarations	331
Example B-7. Enumeration Types	333
Example B-8. Nested Structures	334
Example B-9. Processing of Enumerators	334

List of Figures

Figure 1-1. Data Flow Through the Microtec Toolkit	22
Figure 10-1. Parameter Area and Short Integers	202
Figure 10-2. C/C++ Interlanguage Calling	210
Figure 11-1. Memory Layout.	213
Figure 11-2. Loading Dependent on Processor Type	214
Figure 11-3. Byte Ordering in Words	214
Figure 11-4. Sample Little-Endian Bit Field Allocation (Over 8-Bytes)	218
Figure 11-5. Sample Big-Endian Bit Field Allocation (Over 8-Bytes)	218
Figure 11-6. Sample Bit Field Allocation (Over 2-Bytes)	219
Figure 11-7. Sample Packed Bit Field Allocation	219
Figure 13-1. Memory Configuration	248
Figure 13-2. Memory Configuration (ROM-Based)	250

List of Tables

Table 2-1. Default Input Filename Extensions	24
Table 2-2. Default Output Filename Extensions	25
Table 3-1. Command Line Option Summary	28
Table 4-1. Standard Mirrors	75
Table 4-2. Compiler Mirrors	76
Table 4-3. Language Mirrors	76
Table 4-4. Host Mirrors	77
Table 4-5. Type Mirrors	77
Table 4-6. Underlying Types	78
Table 4-7. Target Mirrors	78
Table 4-8. String Manipulation Mirrors	80
Table 4-9. Option Mirrors	86
Table 4-10. Option Mirror Shorthand Forms	87
Table 4-11. Microtec Preprocessor Directives	88
Table 5-1. Libraries Distributed With MCCPPC and CCCPPC	123
Table 5-2. UNIX-Style System Functions	125
Table 5-3. Non-Reentrant Functions	126
Table 5-4. Intrinsic System Functions	134
Table 5-5. mriext.h Functions	134
Table 5-6. mriext.h Macros	135
Table 5-7. Non-Reentrant Extensions	136
Table 5-8. System Functions	136
Table 5-9. Single-Precision Versions of the Standard Math Libraries	137
Table 5-10. Mode Values for the open Function	155
Table 6-1. C Library Customizer Preprocessor Symbols	171
Table 10-1. Register Usage	203
Table 10-2. Stack Frame Layout	204
Table 11-1. Scalar Data Types	214
Table 12-1. Sections Generated by the Compiler	221
Table 12-2. Example Modules	222
Table 12-3. module1.o and module2.o Sections	222
Table 12-4. Executable File Sections	222
Table 13-1. Variables in asm and Optimization level	232
Table 13-2. Position-independent Code Options	240
Table 13-3. Position-independent Data Options	240
Table A-1. Message Severity Levels	258
Table A-2. Using Compiler Options to Suppress Diagnostic Messages	259
Table A-3. Compiler Error Messages	262
Table B-1. Function Declarations	330
Table C-1. Deprecated Optimization Options	337

Table C-2. Options Removed in Version 3.0	339
Table C-3. New Options in Version 3.0	339
Table D-1. Functions Implemented for Use With SysHost	341

Revision History

Revision	Changes	Date
001	Initial release. Document ID 101945.	1/96
002	Updated for 1.3 release.	11/96
003	Updated for Windows NT release.	6/97
004	Updated for Windows 95 release.	12/97
005	Product name changes and minor edits. Supports software version 1.4.	8/98
006	Updated for software version 1.6.	11/98
007	Updated for software version 1.7.	4/99
008	Updated for software version 1.8.	4/99
009	Updated for software version 1.9.	8/00
010	Updated default environment path names.	8/02
011	Updated for software version 2.0. Templates updated for Frame 6.0.	5/03
012	Updated for software version 2.1.	11/03
013	Updated for software version 3.0.	08/04
014	Updated for software version 3.1.	12/04
015	Updated to new template style. Minor edits.	3/05
016	Updated for software version 3.2.	6/05
017	Updated for software version 3.2A.	12/05
018	Added third-party licensing information.	7/06
019	Updated for software version 3.3.	2/07
020	Further updates for software version 3.3	8/07
021	Updates for software version 3.3a	11/07
022	Updates for software version 3.4	3/08
023	Updates for software version 3.5	10/08
024	Updates for software version 3.6	7/09
025	Updates for software version 3.7	8/12

Revision History

Revision	Changes (cont.)	Date
026	Updates for software version 3.7.7	12/12
027	Updates for software version 3.8	1/15

Components of the Microtec Toolkit

The MCCPPC C Compiler and the CCCPPC C++ Compiler are parts of the Microtec© software development toolkit for the Motorola PowerPC family of microprocessors. Other components include a relocatable cross assembler, a linker, and an object module librarian. The EDGE™ Debugger is available as an option.

The components of the Microtec Toolkit for the PowerPC family are described in more detail in the following sections.

MCCPPC C Compiler

The MCCPPC optimizing cross-compiler converts both ANSI C and traditional C source programs into tight, efficient assembly language code for the ASMPPC Assembler.

You can use C compiler options to create ROMable programs, suppress warnings, and redirect listings. The MCCPPC C Compiler contains a standard ANSI C preprocessor. The C compiler can produce information necessary for debugging with the EDGE Debugger. Executables compiled with MCCPPC use the C libraries.

CCCPPC C++ Compiler

The CCCPPC optimizing cross-compiler converts C++ source programs into tight, efficient assembly language code for the ASMPPC Assembler. The C++ compiler can also produce information necessary for debugging with the EDGE PPC Debugger.

The CCCPPC compiler supports all features of the MCCPPC C compiler, as well as C++ language features described in *The Annotated C++ Reference Manual*, such as templates, multiple inheritance, overloading resolution, memberwise assignment and initialization, static member functions, abstract classes with pure virtual functions, pointers to members, and type-safe linkage. By default, an executable built with CCCPPC uses the C++ libraries, even if the source code is composed entirely of C files.

Compiler Features

The Microtec C and C++ compilers:

- Provide a compiler driver that invokes the C or C++ compiler and, as needed, the assembler and linker (the correct libraries are specified to the linker)

- Provide a fully ANSI-compatible C and C++ preprocessor
- Add Microtec extensions to the C and C++ languages
- Generate standard or optimized code for supported microprocessors
- Issue comprehensive diagnostic messages (most warning and error messages can be suppressed individually)
- Provide a simple interface to assembly language
- Contain a complete run-time library with many standard C and C++ functions
- Support inline assembly instructions
- Provide options to specify search paths for standard and nonstandard **#include** files
- Generate fully reentrant code
- Include a linker with capabilities suitable for cross development
- Locate code and constants in ROM and data in RAM
- Support features described in *The Annotated C++ Reference Manual*, such as templates, multiple inheritance, overloading resolution, memberwise assignment and initialization, static member functions, abstract classes with pure virtual functions, pointers to members, and type-safe linkage
- Include C++ I/O class libraries
- Integrate C++ name demangling support with the Microtec assembler and linker
- Generate C++ symbol debugging information for use with the EDGE Debugger

ASMPPC Assembler

The ASMPPC Assembler converts assembly language programs into relocatable object modules conforming to the Executable Loader Format (ELF) standard. Object modules are suitable for linking with other object modules or with libraries.

The ASMPPC Assembler accepts source program statements that are syntactically compatible with those accepted by assemblers for the PowerPC family of microprocessors. It also processes macros and conditional assembly statements. The assembler generates a cross-reference table as well as a standard symbol table. Code and data can be placed in multiple named or numbered sections.

LNKPPC Linker

The LNKPPC Linker combines relocatable object modules into a single absolute object module in ELF. Additionally, the linker combines multiple relocatable object modules into a single

relocatable module, which subsequently can be relinked with other modules. This feature is called incremental linking.

If one of the input files is a library of object modules, the linker automatically loads any modules from that library that are referenced by the other named object modules. The linker can also produce a link map that shows the final location of all modules and sections and the final absolute values of all symbols. A cross-reference listing shows which modules refer to each global symbol.

The linker can run from the command line or in batch mode using a command file. The linker reports unresolved external symbols as well as errors that occur at link time and prints these messages on the link map or on the screen.

LIBPPC Object Module Librarian

The LIBPPC Librarian creates and maintains program libraries. A program library is a file that contains relocatable object modules. The LIBPPC Librarian program lets you format and organize library files that are subsequently used by the LNKPPC Linker.

EDGE Debugger

The EDGE™ Debugger monitors and controls program execution at the source and assembly level. You can examine or modify the value of program variables using the same source-level terms, definitions, and structures that were defined in the original source code. To display information, the EDGE Debugger employs a window-oriented user interface that divides debugging information into windows.

The debugger provides complete interactive control of the program by directing an execution environment including a simulator, an in-circuit emulator, a single-board computer, or a target monitor. The debugger executes your program while allowing access to variables, procedures, source line addresses, and other program entities.

The powerful EDGE Debugger command language allows simple and complex breakpoint setting, single-stepping, and variable monitoring. Input and output can be directed to or from files, buffers, or windows. In addition, a sophisticated macro facility lets you automate repetitive tasks and associate complex command sequences with events such as the execution of a specific statement, or the accessing of a specific data location. These features allow you to isolate errors and patch your source code.

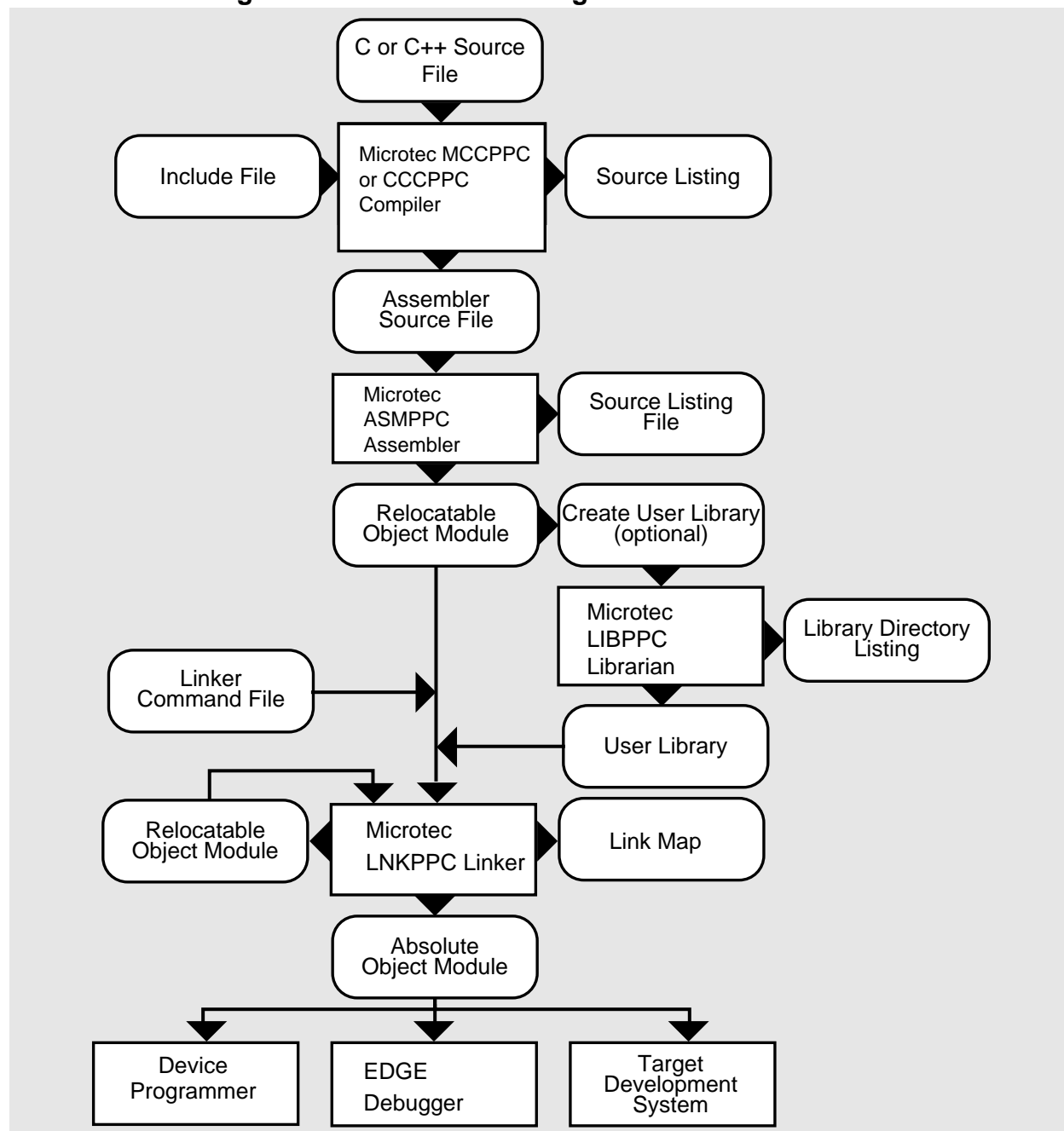
Embedded System Support

The Microtec C and C++ compilers provide many features that make them particularly suitable for generating code in an embedded systems environment. These features include the ability to produce ROMable and reentrant code.

Data Flow

Figure 1-1 illustrates the components of the Microtec Toolkit and the flow of data among them.

Figure 1-1. Data Flow Through the Microtec Toolkit



Chapter 2

Using the Compilation Driver

Invoking the Compilation Driver

The MCCPPC ANSI C Compiler and the CCCPPC C++ Compiler are distributed with a compilation driver that invokes either the C or the C++ compiler, and the ASMPPC Assembler, and LNKPPC Linker, as needed.

The compilation driver accepts multiple input files and automatically invokes the appropriate sequence of compiling, assembling, and linking commands to produce an executable file.

This chapter describes the MCCPPC and CCCPPC compilation driver, input and output filename extensions, and default file locations on host computers running a UNIX or UNIX-like operating system or a DOS shell under Windows. The syntax is identical for these systems, unless specifically noted. The MCCPPC compilation driver can be used to compile C or C++ programs. Based on the filename extension (refer to the “[Filename Extensions](#)” section in this chapter for more details) of the input source file, the driver determines whether it is a C or C++ compilation and uses the appropriate run-time library for linking. However, when the input contains only assembly source or object files, the MCCPPC driver cannot determine whether the input files originate from C or C++ source. In such a situation, it assumes that the input files are the result of a C compilation. For this reason, the CCCPPC driver should be used instead of MCCPPC to link object files or assembly source created from C++ code.

Command Line Syntax

The following commands invoke the compilation driver:

```
mccppc [option] . . . source_filename. . .
```

or

```
cccppc [option] . . . source_filename. . .
```

Description

- **mccppc**
Indicates the name of the C compilation driver.
- **cccppc**
Indicates the name of the C++ compilation driver.
- *option*

Indicates any options described in Chapter 3, “[Using Command Line Options](#)”. You can specify options and source filenames in any order. The default options are used if no options are specified.

- *source_filename*

Specifies the name of a file containing one of the following:

- C program source file
- C++ program source file
- PowerPC family assembly language file
- Object file
- Object library

Possible extensions for *source_filename* are listed in the section “[Filename Extensions](#)” in this chapter.

Filename Extensions

The C and C++ compilers identify file types by their extensions. The compiler driver will determine how a file should be processed based upon the file’s extension.

Input Filename Extensions

The compiler determines the type of each input file based on its extension. [Table 2-1](#) lists the extensions for compiler input files.

Table 2-1. Default Input Filename Extensions

File	UNIX Extension	Windows Extension
C source file	<i>.c</i> or <i>.i</i>	<i>.c</i> or <i>.i</i>
C++ source file	<i>.cc</i> , <i>.cxx</i> , <i>.cpp</i> , or <i>.ixx</i>	<i>.cc</i> , <i>.cxx</i> , <i>.cpp</i> , or <i>.ixx</i>
Assembly language file	<i>.s</i> , <i>.src</i> , or <i>.asm</i>	<i>.src</i> , <i>.s</i> , or <i>.asm</i>
Relocatable object file	<i>.o</i> or <i>.obj</i>	<i>.obj</i> or <i>.o</i>
Object file library	<i>.lib</i>	<i>.lib</i>
Linker command file	<i>.cmd</i>	<i>.cmd</i>

The compiler uses the extension to determine how to process each file. The compiler uses the following rules:

- If a file’s extension is *.c* or *.i*, it is compiled with the C Compiler.
- If a file’s extension is *.cc*, *.cxx*, *.cpp*, or *.ixx*, it is compiled with the C++ Compiler.
- If a file’s extension is *.s*, *.src*, or *.asm*, it is assembled.

- If a file's extension is *.cmd*, it is passed to the linker as a command file.
- If a file has any other extension or no extension, it is assumed to be an object file or a library and is given to the LNKPPC Linker. A warning message is displayed if the file does not have a *.o*, *.obj*, or *.lib* extension.

Output Filename Extensions

When an output file is created by the compiler, assembler, or linker, it is given a default extension based on its type. [Table 2-2](#) lists the extensions.

Table 2-2. Default Output Filename Extensions

File	UNIX Extension	Windows Extension
Preprocessor output file	<i>.i</i> (C) or <i>.ixx</i> (C++)	<i>.i</i> (C) or <i>.ixx</i> (C++)
Assembly language file	<i>.s</i>	<i>.src</i>
Relocatable object file	<i>.o</i>	<i>.obj</i>
Executable file	<i>.x</i>	<i>.abs</i>

Input and Output File Locations

If an input file does not contain a directory specification, the current working directory is assumed.

If an output filename is not specified or if an output file has no directory specification, the output file is placed in the current working directory.

Chapter 3

Using Command Line Options

This chapter describes the syntax of the options for the Microtec MCCPPC C and CCCPPC C++ compilation driver. These options are supported on host computers running a UNIX or UNIX-like operating system, or a DOS shell under Windows. The syntax is identical for these systems, unless specifically noted. MCCPPC accepts a subset of the options accepted by CCCPPC, ignoring or rejecting options specific to C++.

The chapter includes:

- Quick-reference tables summarizing the command line options
- Expanded descriptions of command line options and defaults
- Tips on using combinations of options
- Example command lines making use of various options

Command Line Options

Command line options specify the names of output files and turn features on and off. Each option begins with a leading dash (-) that distinguishes it from filenames.

Options that begin with the same character sequence can be combined into a single grouping on the command line, unless the resultant grouping represents a different option. For example, you can specify the **-Og** and **-Oi** options together as **-Ogi**. If one member of the option grouping is a value, that value must be placed at the end of the grouping; for example, to combine the **-Qe** and **-Q20** options, you must specify **-Qe20**; **-Q20e** is not a valid option.

A single-letter option without arguments can be followed immediately by another option letter. For example, the combination of options **-c** and **-g** can be written as **-cg**.

Some options have a negative form, which disables or turns off the option. You can specify the negative form by entering **n** before the name of the option.

Example 3-1. Negating Options With -n

- | | |
|-------------|---|
| -Og | Enable a standard set of global optimizations. |
| -nOg | Disable a standard set of global optimizations. |

If conflicting options are specified in the same command line, the rightmost specified option takes precedence. For example, if a command line includes two filenames specified with the **-o** option, the second name is used for the output file.

All of the following command line options can be given to the driver program. Some of the options control the behavior of the driver itself. However, most of the options do not pertain directly to the driver program and are passed on to the compiler.

Command Line Option Summary

Table 3-1 summarizes the positive form of MCCPPC and CCCPPC command line options.

Table 3-1. Command Line Option Summary

Option	Meaning
Set ANSI-Compliant Mode (C Compiler Only)	
-A	Sets ANSI-compliant mode (see -E and -P) (default: -A).
Control ROM Data Addressing	
-ar{a d}	Controls how ROM data is addressed (default: -ara).
Save Comments	
-C	Saves comments in preprocessor output (see -E and -P) (default: -nC).
Produce Object File Only (Driver Option)	
-c	Produces an object file but not an executable file (default: an executable file is produced).
Define a Preprocessor Macro Name	
-Dname[=string]	Defines the value of a preprocessor macro (default: no preprocessor macro values supplied).
Read Options From File	
-doption_file	Reads options from the specified file (default: options supplied by command line).
Display Preprocessor Output	
-E[s]	Invokes only the preprocessor without removing #line or #pragma directives. Sends output to standard output (see -C , -P) (default: preprocessor output is not displayed).
Pass Command File to Linker (Driver Option)	
-ecommand_file	Passes the command file to the linker

Table 3-1. Command Line Option Summary (cont.)

Option	Meaning
Control Floating Point Code Generation	
-f	Instructs the backend to generate PowerPC floating point opcodes (default: -f).
Redirect Diagnostic Messages	
-Fee	Writes diagnostic messages to standard error. (default: -Fee)
-Feo	Writes diagnostic messages to standard output. (default: -nFeo)
Specify Format of Listing Files	
-Fli	Shows the contents of #include files in the listing file (default: -nFli).
-Flp <i>number</i>	Sets the page length of the listing file (default: -Flp55).
-Flt <i>string</i>	Specifies the title for the listing file (default: no title string appears on the listing file).
-Fsm	Includes high-level source code as comments in assembly output file (-S enables -Fsm) (default: -nFsm).
Produce Debugging Information	
-g	Generates debugging information (default: -ng).
Run-Time Checks	
-GP	Enable pointer checks (null pointers and array bounds) (default -nGP)
-GS	Enable check on stack overflow (default -nGS)
-GW	Enable switch statement check (default -nGW)
-GZ	Enable check on division-by-zero (default -nGZ)
Save Assembly File (Driver Option)	
-H	Saves assembly file (default: -nH).
-h	Displays usage information

Table 3-1. Command Line Option Summary (cont.)

Option	Meaning
Add Search Path for Nonstandard Include Files	
<i>-Idir</i>	Specifies the search path for nonstandard #include files (default: see the full description of this option later in this chapter for a list of the directories searched).
Add Search Path for Standard Include Files	
<i>-Jdir</i>	Specifies the search path for standard #include files (default: see the full description of this option later in this chapter for a list of the directories searched).
Use Precompiled Headers	
<i>-jH</i>	Directs the compiler to automatically use and create (if necessary) a precompiled header file (default: -njH).
<i>-jHcfile_name</i>	Creates a precompiled header file with the specified name (default: no precompiled header file is created).
<i>-jHddir</i>	Specifies the directory to search for and store precompiled header files in (default: -jHd. [that is, current directory]).
<i>-jHufile_name</i>	Directs the compiler to use the specified precompiled header file during the current compilation (default: no precompiled header file is used).
Produce Minor Code Generation Variations	
<i>-KE</i>	Specifies big-endian processor type (default: -KE).
<i>-Ken</i>	Calls the interrupt function code using the standard function call instruction (default: -nKen).
<i>-Kepprefix</i>	Specifies names for the context save and restore routines (default: -Kep_interrupt_).
<i>-Kes</i>	Saves the link registers and R12 at the location pointed to by special register SPRG0 (default: -nKes).
<i>-Ket</i>	Specifies the interrupt handler and body to be placed into the .text section (default: -nKet).
<i>-Kf</i>	Directs the compiler to generate a stack frame for all functions (default: -nKf).

Table 3-1. Command Line Option Summary (cont.)

Option	Meaning
-KgM	Directs the compiler to generate debugging information for preprocessor macros (default: -nKgM).
-Khreg[,reg]...	Directs the compiler to reserve the specified register(s) (default: no registers reserved).
-Kia	Functions containing “asm()” statements may be inlined. (See – Oi)
-Kkargs	Specifies support for full arguments in functions generated by the -Kkfe , -Kkfx , -Kknp , and -Kkz options (default: -nKkargs).
-Kkfefunc	Specifies a function to be called at procedure entry (default: -Kkfe0).
-Kkfxfunc	Specifies a function to be called at procedure exit (default: -Kkfx0).
-Kknp	Instructs the compiler to include run-time checks for dereferencing null pointers (default: -nKknp).
-Kkz	Instructs the compiler to include run-time checks for zero divisors (default: -nKkz).
-KLfn	Specifies support for long calls (default: -KLf0).
-KLin	Specifies text or data static initializers (default: -KLi1).
-KP	Specifies structs , unions , and enums to be packed by default (default: -nKP).
-Kq	Specifies double variables or constants as float (32-bits) (default: -nKq).
-KR	Directs the compiler to use symbolic names for registers in the assembly file (default: -nKR).
-Ksr	Use save/restore routines to save/restore registers used in functions. (default: -nKsr)
-Kst	Generates jump tables for dense switch cases (default: -Kst).
-Ku	Treats plain char variables as unsigned (default: -Ku).

Table 3-1. Command Line Option Summary (cont.)

Option	Meaning
-Kv	Treats plain bit fields as unsigned (default: -Kv).
Generate Listing File	
-l[filename]	Generates a source listing, including diagnostics (default: no listing).
-m[filename]	Generates a linker map file (default: no map file).
Generate Position-Independent Code and Data	
-Mcnumber	Directs the compiler to use register <i>number</i> as a location for the amount of the current code offset (default: -Mcr).
-Mcp	Directs the compiler to generate code in each function prologue to automatically calculate the current code offset (default: -Mcr).
-Mcr	Directs the compiler to use relative branches for known targets and absolute addresses for indirect calls (default: -Mcr).
-Mdnumber	Directs the compiler to use register-relative addressing for all data references (default: -Mda).
-Mda	Directs the compiler to use absolute addressing for all data references (default: -Mda).
Rename Compiler Predefined Sections	
-NCname	Sets the constant variables section name (default: the constant variables section is <code>.rodata</code>).
-NDname	Sets the data section name (default: the data section is <code>.data</code>).
-NIname	Sets the initialization section name (default: the initialization section is <code>.init</code>).
-NSname	Sets the strings section name (default: the strings section is <code>.rodata1</code>).
-NTname	Sets the code section name (default: the code section is <code>.text</code>).
-NZname	Sets the weak external section name (default: the weak external section is <code>.bss</code>).

Table 3-1. Command Line Option Summary (cont.)

Option	Meaning
Negate an Option	
<i>-nooption</i>	Negates the specified option(s).
Optimize Code	
Safe Optimizations	
-O	Enables optimizations (same as default: -Ot).
-OD	Optimizes based on interprocedural data flow analysis (default: -OD).
-Og	Enables global-flow optimizations (default: -Og).
-Oi	Inlines function calls within a module (default: -nOi).
-Oj	Enables run-time library function in-lining (default: -nOj).
-Ol	Enables local optimizations (default: -Ol).
-Or	Enables instruction scheduling (default: -nOr).
-Os	Optimizes for space (same as -ODglr) (default: -ODglrU)
-Ot	Optimizes for time. (same as -ODgijlrU) (default: -ODglrU).
-OU	Enables loop unrolling (default: -OU).
-Ox	Enables maximum optimization (default: -ODglrU).
Potentially Unsafe Optimizations	
-Of	Enables floating point expression rearrangement (default: -nOf).
-OJ	Disables intrinsic function error checking (default: -nOJ).
Name the Output File (Driver Option)	
<i>-ofilename</i>	Specifies the name of the output file (default: see full entry for more information).

Table 3-1. Command Line Option Summary (cont.)

Option	Meaning
Send Preprocessor Output to File (Driver Option)	
-P[s]	Executes the preprocessor only, sending output to the .i (C) or .ixx (C++) file (see -C , -E) (default: compilation continues and the preprocessed source file is not saved).
Produce Code for Specified Processor	
-pprocessor	Produces code for a specified processor (default: -p603).
Control Diagnostic Messages	
-Qnumber	Specifies the maximum number of error messages before quitting (default: -Q20).
-QA	Suppresses all messages (default: no messages suppressed).
-Qe	Suppresses error, warning, and informational messages (default: no messages suppressed).
-Qfn	Suppresses the display of the message number when the diagnostic is displayed (default: message number displayed).
-Qfs	Suppresses the display of the source line when the diagnostic is displayed (default: source line number displayed).
-Qi	Suppresses informational messages (default: -Qi).
-Qmsgid [,msgid2,] . . .	Makes diagnostic IDs error messages (default: diagnostics have standard severity levels).
-Qmmsgid [,msgid2,] . . .	Makes diagnostic IDs informational messages (default: diagnostics have standard severity levels).
-Qsmmsgid [,msgid2,] . . .	Prevents the compiler from issuing the specified diagnostics (default: all diagnostics occur as normal).
-Qwmmsgid [,msgid2,] . . .	Makes diagnostic IDs warnings (default: diagnostics have standard severity levels).
-Qo	Suppresses the listing of currently active options (default: -nQo).
-Qs	Suppresses the summary message (default: no messages suppressed).

Table 3-1. Command Line Option Summary (cont.)

Option	Meaning
-Qw	Suppresses warning and informational messages (default: no messages suppressed).
-Qwo	Suppresses code generator warnings (default: no messages are suppressed).
-nQ	Permits the display of all messages (default: -nQo -Qi).
Use Default Libraries for Linking	
-q	Instructs the linker to use the default libraries for linking (default: -q).
Produce Assembler Source File (Driver Option)	
-S	Generates code for the assembler (default: no assembly language file saved).
Control Template Instantiation	
-tl	Instantiates all used templates as local (default: -tu).
-tm	Does not instantiate templates, unless manually created with a #pragma (default: -tu).
-tu	Instantiates all used templates as external unless declared as static (default: -tu).
Undefine a Preprocessor Macro Name	
-Uname	Undefines a preprocessor macro (default: no preprocessor macro is undefined).
Modify Naming Conventions	
-ui[<i>char</i>]	Changes the insert character for asm pseudofunction calls to <i>char</i> (default: Back quote (`) character used).
Control Verbose Output Information (Driver Option)	
-Vb	Displays the banner before compiling (default: banner not displayed).
-Vd	Displays commands for invoking components. Commands are not executed (default: commands executed but not displayed).
-Vi	Displays commands for invoking components. Commands are executed (default: commands executed but not displayed).

Table 3-1. Command Line Option Summary (cont.)

Option	Meaning
-Vt	Displays a time stamp for compilation stages (default: time stamps not displayed).
-V	Displays the tool components' banners and exits (default: compilation occurs without displaying banner).
Pass Options to Tools (Driver Option)	
-Wa, <i>option1</i> [, <i>option2</i> ,] . . .	Passes options to the assembler (default: no options passed to assembler).
-Wl, <i>option1</i> [, <i>option2</i> ,] . . .	Passes options to the linker (default: no options passed to linker).
Initialize Uninitialized Global Data	
-Xc	Treats uninitialized global variables as "weak externals" (default: -Xp).
-Xp	Allocates space for global variables that have not been explicitly initialized (default: -Xp).
Enable Microtec Extensions	
-x	Enables Microtec extensions to the C language (default: -x).
Accept Out-of-range Floating Point Constants	
-xf	Accept out-of-range floating point constants (default: -nxf).
Enable GNU Extensions	
-xG	Enable GNU extensions (default -nxG)
Check Program Syntax (Driver Option)	
-y	Checks source syntax but does not compile (default: compiler checks syntax and generates code).
Produce Minor Code Generation Variations (-Zx options) (Part II)	
-Zan	Sets bus alignment to size <i>n</i> .
-Zcn	Sets code alignment to size <i>n</i> .
-Zdn	Sets data alignment to size <i>n</i> .
-Zin	Sets maximum size of asm() statements (default: -1).
-Zmn	Sets size of packed structures to a multiple of <i>n</i> .

Table 3-1. Command Line Option Summary (cont.)

Option	Meaning
-Znn	Sets sizes of unpacked structures to a multiple of <i>n</i> .
-Zssize	Allocates data items smaller than <i>size</i> to the small data area. (default: <i>size</i> is 0).
Enable C++ Features	
-ze	Enables C++ exception handling (default: -nze).
-zr	Enables run-time type information (default: -nzh).

Command Line Options - Expanded Descriptions

The following pages describe the command line options allowed on UNIX and Windows operating systems for the Microtec C/C++ compiler and the compilation driver, in alphabetical order.

Set ANSI-Compliant Mode (C Compiler Only)

- A (default)
- nA

The **-A** option enables additional features provided by ANSI C. The `__STDC__` preprocessor symbol is predefined.

The **-nA** option disables ANSI features, so the compiler will accept pre-ANSI programs. For example, if **-nA** is used, `volatile` will not be recognized as a keyword.

The **-nA** option does not guarantee compatibility with earlier versions of Microtec compilers or with other pre-ANSI compilers.

Control ROM Data Addressing

- ara (default)
- ard

The **-arx** option specifies how ROM data is addressed. The **-ara** option causes ROM data to be addressed using absolute addresses. The **-ard** option causes ROM data to be addressed according to the specification of the **-Mdx** option.

Save Comments

-C

-nC (default)

The **-C** option keeps high-level source comments in the preprocessor output. (See also the **-E** and **-P** options.)

The **-nC** option excludes comments from the preprocessor output.

Produce Object File Only (Driver Option)

-c

The **-c** option produces only an object file having either a *.o* extension for UNIX or *.obj* for Windows. The option tells the driver to produce an object file but not to call the linker to produce an executable file having either a *.x* extension for UNIX or *.abs* for Windows.

By default, the driver produces an object file and then calls the linker to produce an executable file.

Define a Preprocessor Macro Name

-Dname[=string]

The **-D** option defines the value of a preprocessor macro. If you do not specify a string, the preprocessor macro has the value 1 (equivalent to putting **#define name 1** at the top of the source file).

You can define multiple names by preceding each name with **-D**. For example:

-Dabc -DABC

is equivalent to using the preprocessor directives **#define abc 1** and **#define ABC 1**.

Note



The **-U** (undefine macro) option is processed before the **-D** (define macro) option, regardless of the order in which they appear on the command line.

Read Options From File

-doption_file

The **-d** option directs the driver to read command line options from the specified file.

Display Preprocessor Output

-E[s]

The **-E** option tells the driver to execute the preprocessor only. Preprocessor output is sent to standard output. The **#line** and **#pragma** directives are not removed. The **#include** directives are converted to appropriate **#line** directives, so that the original structure of the source files can be determined. (See also the **-C** and **-P** options.)

If **-Es** is specified, all the **-E** actions are performed, but the echoing of the backslash (\) and newline is disabled.

This option conflicts with the **-c**, **-l**, **-P**, and **-S** options.

Pass Command File to Linker (Driver Option)

-ecommand_file

The **-e** option directs the driver to pass the specified command file to the linker.

Redirect Diagnostic Messages

-Fee (default)

-nFee

The **-Fee** option writes diagnostic messages to the standard error device (**stderr**). The diagnostic messages are written to the standard output device if **-nFee** is specified. **-Fee** and **-Feo** are mutually exclusive.

-Feo

-nFeo (default)

The **-Feo** option writes diagnostic messages to the standard output device (**stdout**). The diagnostic messages are written to the standard error device if **-nFeo** is specified. **-Fee** and **-Feo** are mutually exclusive.

Specify Format of Listing Files

-Fli

-nFli (default)

The **-Fli** option puts the contents of each **#include** file in the listing file. You must specify the **-l** option to generate a listing file.

-Flpnumber

The **-Flp** option sets the number of lines per page in the listing file. The default page length is 55 lines. A specification of less than 10 defaults to 55 lines. You must specify the **-l** option to generate a listing file.

Specify **-Flp0** to omit page breaks and the page header from the listing file.

-Fltstring

The **-Flt** option specifies a title string that appears at the top of each page of the listing file. You must specify the **-l** option to generate a listing file. Enclose *string* in double quotes (") if it contains any spaces or punctuation.

-Fsm

-nFsm

The **-Fsm** option includes high-level source code as comments in the assembler source file. You must specify the **-S** or **-H** option in order to generate an assembly file. The **-S** option enables **-Fsm**.

Control Floating Point Code Generation

-f (default)

-nf

The **-f** option instructs the backend to generate PowerPC floating point opcodes and the library to save and restore floating point registers.

With the **-nf** option, the backend generates calls to EABI-defined simulation functions and the library does not attempt saves or restores of floating point registers.

Code compiled with the **-nf** option should not be cross-linked with code compiled with the **-f** option. The results of calling a function between these types of code is undefined.

The **-f** option should not be specified for any e500v2 PowerPC processor variant. The compiler will automatically generate special floating-point opcodes that are appropriate for execution on those processors. Using the **-f** option will result in the use of standard floating-point opcodes that require simulation on the e500v2 processors. At present, these processors are e500v2, 8533, 8533e, 8543, 8543e, 8544, 8544e, 8545, 8545e, 8547e, 8548, 8548e, 8567, 8567e, 8568, 8568e, 8572, and 8572e.

Produce Debugging Information

-g

-ng (default)

The **-g** option generates line number, variable, and symbol information. The **_DEBUG** preprocessor symbol is predefined.

Information is in DWARF 2 format, which is readable by the EDGE PPC Debugger. With **-g**, use only the Microtec assembler and linker to pass complete debugging information to the EDGE Debugger.

Run-Time Checks

Run-time checks are provided by this C/C++ compiler and/or C run-time library. The checks and analysis that can be performed by a C/C++ compiler assist in day-to-day development, but they are not capable of solving more complex and hidden problems that can be introduced during the implementation and coding of your application. Compiler-generated run-time checks let you instrument your code to find and solve special problems. As the term “run-time checks” implies, those checks are performed when your program is executed, as opposed to static checks or analyses that are performed at compilation time. If errors are detected, a run-time error reporting function is called. See [Run-Time Error Reporting Function](#) for more details on this function.

To perform run-time checks, the C/C++ compiler automatically detects every statement and expression in the C source code where this kind of error checking is required and temporarily instruments your application code with a noticeable amount of additional code. This code is not visible at the source code level, but it increases the size of the binary code and the execution time of the application. Run-time checks should only be used during the debugging phase and are removed (by the compiler) when the application is built for release. The increased code size and execution time during the debugging phase helps to decrease the time needed for debugging the application and isolating problems.

-GP

-nGP (default)

NULL Pointer Dereferencing

The NULL pointer dereference check is done for pointer accesses with pointers where the base address is not known. The assumption is made that no object may be dynamically allocated at the address zero. Thus, if a pointer has a value of zero, a run-time error reporting function is called.

The application dereferences the pointer `ptrNumbers`, for which the base object is not known to the C/C++ compiler because the compiler cannot determine the valid memory range for such a pointer. It also cannot use the array bounds run-time check instrumentation; however, it can instrument the code with a NULL-pointer check.

```
int readPtr(int *ptrNumbers)
{
    return *ptrNumbers;
}
```

The instrumented code looks similar to (type is `rtcNullPointer`) the following:

```
int readPtr(int *ptrNumbers)
{
    // START: RTC-Instrumentation
    if (ptrNumbers == 0)
        rtc_checkfailed ( &info );
    // END: RTC-Instrumentation
    return *ptrNumbers;
}
```

-GP and **-Kknp** perform equivalent functions, but use different run-time library routines.

-GP is recommended as **-Kknp** may be removed in future releases.

-GS

-nGS (default)

Stack Overflow

Stack overflow checks whether the stack pointer is within valid bounds. An unconventional call is made to `rtc_stackoverflow` at the beginning of each function. (This is unconventional in the sense that the normal calling convention is not followed. This is done so that the incoming arguments to this function are not disturbed.) This routine checks whether the stack pointer is within the stack section. If an overflow is detected, a call is made to `rtc_checkfailed` with the type in the info block set to `rtcSupportStackOverflow`.

-GW

-nGW (default)

No Case Within a Switch Statement Is Hit

This inserts a call to a function within the run-time system for switch statements that do not have a default case. An error is generated when a switch statement in which no case fits the parameters is hit.

A switch statement with a missing default clause is instrumented as follows. The type passed in the info block is `rtcSwitch`.

```
void foo(int i)
{
    switch (i) {
        case 1: /* do something */ break;
        case 2: /* do something */ break;
        // START: RTC-Instrumentation
        default:
            rtc_checkfailed( &info );
        // END: RTC-Instrumentation
    }
}
```

-GZ

-nGZ (default)

Integer Division-By-Zero

The division-by-zero check works by inserting code to check that the divisor is not zero. A run-time error reporting function is called if the divisor is zero.

The application code performs integer division as shown here:

```
int div(int a, int b) {
    return a / b;
}
```

The C/C++ compiler instruments the application code as shown to check that the divisor is not equal to zero. The type in the info block is set to `rtcDivisionByZero`.

```
int div(int a, int b) {
    // START: RTC-Instrumentation
    if (b == 0)
        rtc_checkfailed( &info );
    // END: RTC-Instrumentation
    return a / b;
}
```

-GZ and **-Kkz** perform equivalent functions, but use different run-time library routines.

-GZ is recommended as **-Kkz** may be removed in future releases.

Run-Time Error Reporting Function

For most of the **-G** options, code for the specific test is generated, and if the test fails, a call is made to `rtc_checkfailed`. A single argument is passed containing the address of the following struct record:

```
typedef struct rtc_info_type
{
    short      version;
    short      type;
    const char* filename;
    int        line;
    int        column;
    const char* info;
} rtc_info;
```

'version' indicates this version of run-time checking. 'type' indicates the failure. Type failure codes are specified in the run-time library source file `'rtc.h'`. 'filename', 'line' and 'column' indicate where the failure occurred. 'info' may contain miscellaneous information.

Save Assembly File (Driver Option)

-H

-nH (default)

The **-H** option instructs the compiler not to remove the generated assembly file. The **-nH** option removes the generated assembly file.

The name of the assembly file is based on the source filename followed by a `.s` (UNIX) or `.src` (Windows) extension.

The **-o***filename* option does not affect the name of the assembly file.

Add Search Path for Nonstandard Include Files

-Idir

The **-I** option specifies a search path to be scanned to locate user-supplied **#include** files. If an include filename in your source file is enclosed in double quotes (""), the compiler searches for the include file in the following directories:

- The directory containing the source file that has the **#include** directive
- The directory containing the top-level source file
- The directories specified by the **-I** option
- The compiler searches multiple directories if you precede each directory name with **-I**. For example, if you enter **-I***dir1* **-I***dir2* **-I***dir3*, the compiler searches for **#include** files in the directory containing the source file, the *dir1* directory, the *dir2* directory, and the *dir3* directory, in that order.
- The order of directories as specified by the **-J** option.

See the **-J** option for information on include filenames enclosed in angle brackets (< >).

Add Search Path for Standard Include Files

-Jdir

The **-J** option specifies a search path to be scanned to locate standard **#include** files. When an include filename in your source file is enclosed in angle brackets (< >), the compiler searches for the file in the following directories:

- The directories specified by the **-J** option
- The compiler searches multiple directories if you precede each directory name with **-J**. For example, with **-Jdir1 -Jdir2 -Jdir3**, the compiler searches for standard **#include** files in the *dir1* directory, the *dir2* directory, and the *dir3* directory.
- The standard compiler include directories, which are parallel to the compiler binaries.
- For C, the directory is *installation_dir/./inc*, where *installation_dir* is the directory in which the compiler is installed.
- For C++, the directories are *installation_dir/./inc/stlport* and then *installation_dir/./inc*, where *installation_dir* is the directory in which the compiler is installed.

See the **-I** option for information on include filenames enclosed in quotes ("*name*").

Use Precompiled Headers

-jH

-njH (default)

The **-jH** option tells the compiler to automatically use and create a precompiled header file (if necessary). Subsequent occurrences of the **-jHc** and **-jHu** options override this option.

-jHcfile_name

The **-jHcfile_name** option tells the compiler to create a precompiled header file with the specified name. Subsequent occurrences of the **-jH** option override this option.

-jHddir (default: .)

The **-jHddir** option specifies the directory in which to search for and create precompiled header files.

-jHufile_name

The **-jH***file_name* option tells the compiler to use the specified precompiled header file as part of the current compilation. Subsequent occurrences of the **-jH** option overrides this option.

Refer to Chapter 9, “[Precompiled Header Files](#)” for more information about using precompiled header files.

Produce Minor Code Generation Variations

-KE (default)

-nKE

The **-KE** option generates code to be executed in a big-endian memory region. The processor addresses objects in big-endian memory from the big or most significant byte end. The **_BIG_ENDIAN** preprocessor symbol is set to 1 when this option is enabled.

The **-nKE** option specifies code to be executed in a little-endian memory region. The processor addresses objects in little-endian memory from the little or least significant byte end. The **_LITTLE_ENDIAN** preprocessor symbol is set to 1 when **-nKE** is used.

-Ken

-nKen (default)

The **-Ken** option tells the compiler to use the standard function call instruction (**bl**) for calls made from the vector offset code. With this option, the trap entry must be linked at the correct location and not dynamically moved.

With the **-nKen** option, function calls from the vector offset code load the function address into a register, then branch via the register. This technique allows the vector offset code to be created or moved dynamically.

-Keprefix

The **-Kep** option is used to generate names for **save** and **restore** routines. The default value is **_interrupt_**. The Run-Time Library (RTL) provides the routines **_interrupt_save** and **_interrupt_restore**.

-Kes

-nKes (default)

The **-Kes** option controls whether to save the link register and **R12** at the location pointed to by special register **SPRG0**. This is normally required when the interrupted process executes with data address translation enabled, as an interrupt causes address translations to be suppressed (that is, the stack pointer is no longer valid). Thus, special register **SPRG1** is modified. The first three instructions shown in the trap entry above are replaced with the following:

```

mtspr 273,11      # save R11 in SPRG1
mfspr 11,272      # move SPRG0 to R11
stw 12,4(11)      # save R12
mflr 12           # get link register
stw 12,0(11)      # .. and save
mfspr 11,273      # restore R11

```

Likewise, the registers are restored from the same locations when returning from the interrupt handler.

```

mfspr 12,272      # move SPRG0 to R12
lwz 12,0(12)      #
mtlr 12           # restore link register
mfspr 12,272      # move SPRG0 to R12
lwz 12,4(11)      # get R12

```

With the **-nKes** option, the link register is saved at (SP-8). **R12** is saved as (SP-4).

-Ket

-nKet (default)

The **-Ket** option specifies that the interrupt handler and body are placed in the **.text** section. You are responsible for copying the trap entry part to the correct physical address.

With the **-nKet** option, the interrupt handler is placed in the **SEC: *function_name*** section.

-Kf

-nKf (default)

The **-Kf** option controls function frame usage.

The **-Kf** option forces frames for all functions. In a framed function, variables and parameters are accessed on the stack with the FP-relative addressing mode. However, the stack pointer is not available for your use; user programs should never modify the stack pointer since this can cause unpredictable run-time errors.

With the **-nKf** option, the compiler will generate a frame for a function only if it is needed.

-KgM

-nKgM (default)

The **-KgM** option generates debugging information for preprocessor macros. Use this option in conjunction with the **-g** option.

-Khreg[,reg...]

The **-Khreg** option instructs the compiler to reserve the specified register(s). The compiler will not generate code that uses a reserved register. The value of *reg* must

be in the range of 14–30. Only the last instance of **-Khreg** on the command line will be used; all other instances of the option are ignored.

-nKia

-Kia (default)

The **-Kia** option permits inlining functions that contain **asm()** statements. This was not done prior to version 3.2. Inlining these functions can result in significant performance improvements, but it also has its drawbacks. If an **asm()** contains a label, and it is inlined multiple times, the assembler may issue a “Duplicate label” error. Also, some **asm()** statements may assume input arguments are in specific registers. This may not be true if the function is inlined. This option is only meaningful if the **-Oi** option (inlining optimization) is enabled.

-Kkargs

-nKkargs (default)

The **-Kkargs** option specifies that “hook” functions generated by the **-KKfe**, **-KKfx**, **-Kknp**, and **-Kkz** options take full arguments of the following types:

```
void (*) (char *function_name,
          char *file_name,
          long line_number);
```

Ordinarily, these hook functions take no arguments (they are of type **void (*) (void)**), and information used by the hook functions (such as the place from which the hook function was called) must be obtained by using assembly language routines generated by the **asm** pseudofunction and looking at the link register. Refer to the “[Inline Assembly](#)” section in Chapter 13, “[Embedded Environments](#)” for more information about the **asm** pseudofunction. The hooks generated by the **-Kkargs** option can result in a large amount of added code and data and greatly increase the memory requirements of the program.

The **-nKkargs** option indicates that generated hook functions take no arguments.

-Kkfunc (default 0)

The **-Kkfunc** option specifies that the function **func** is to be called when a procedure is entered. The function is called once the frame is established if a frame is necessary. No function is called if **func** is 0. **func** is a user-defined routine and is assumed not to return.

-Kkfxfunc (default 0)

The **-Kkfxfunc** option specifies that the function **func** is to be called whenever a procedure is entered. The function is called immediately before the procedure returns. No function is called if **func** is 0. **func** is a user-defined routine and is assumed not to return.

-Kknp

-nKknp

The **-Kknp** option generates code to check a pointer against zero when dereferencing a pointer at run-time. If the pointer is NULL, a call to the **__nullcheck_trap** routine (with no arguments unless the **-Kkargs** option is specified) is made. **__nullcheck_trap** is a user-defined routine and is assumed not to return. The linker will generate a “**__nullcheck_trap** symbol not found” error if an implementation of **__nullcheck_trap** is not supplied.

-GP and **-Kknp** perform equivalent functions, but they use different run-time library routines.

-GP is recommended as **-Kknp** may be removed in future releases.

-Kkz

-nKkz (default)

The **-Kkz** option generates code to check the divisor against 0 at run-time when a division operation (integer or floating point) is performed. If the divisor is ever found to be 0, the function **__mri_user_divcheck_hook** is called (with no arguments unless the **-Kkargs** option is specified).

-GZ and **-Kkz** perform equivalent functions, but use different run-time library routines.

-GZ is recommended as **-Kkz** may be removed in future releases.

-KLfn (default 0)

The **-KLf0** option causes function calls to translate to relative branches with link register setting. The **-KLf1** option makes relative branches for function calls within the same source file, while calls to external functions are translated as a four-word sequence that ends in an indirect branch. With the **-KLf2** option, all function calls are translated as indirect branches.

-KLin (default 0)

The **-Klin** option affects the generation of the **.init** section in memory. If **-Kli0** is specified, text static initializers are used, and the **.init** section contains code that startup routines branch into and that branches back to the label **._term_init**.

The **-Kli1** option specifies that data static initializers are to be used, and the **.init** section consists of an array of pointers to code with no special termination code.

-KP (default)

-nKP

The **-KP** option packs **structs** and **unions**. Enumerations are allocated as the smallest integral type that can hold all their values, rather than **int** by default.

-Kq

-nKq (default)

The **-Kq** option gives all **double** variables or constants the **float** type size (32 bits) and alignment (4 bytes). Since **-Kq** treats **long double** variables as double, all variables that are **float**, **double**, and **long double** have the **float** type size and alignment. A set of libraries that are built with the **-Kq** option is provided.

-KR

-nKR (default)

The **-KR** option directs the compiler to customize register notation generated by the compiler in assembly source files. Using **-KR** results in generating symbolic names for registers, like **r0**, instead of plain numbered notation, like **0**.

-Ksr

-nKsr (default)

The **-Ksr** option allows the compiler to call register save/restore routines in functions requiring numerous registers during execution. Although this allows the compiler to generate fewer instructions, the result takes longer to execute. The **-Os** option (optimize for size) implies **-Ksr**.

-Kst [default]

-nKst

Instructs the compiler to use a branch table when appropriate when handling a **switch** statement. A branch table is an array of code addresses representing code associated with each individual case of the switch. The compiled program selects a branch target from this array using the switch case value as an index. With **-nKst**, the compiler will not generate a branch table, but it will generate comparisons and conditional branches to determine the case to which to branch. Normally, when the compiler chooses to use the branch table, it is more optimal than the alternative.

-Ku (default)

-nKu

The **-Ku** option tells the compiler to consider char variables declared without the signed or unsigned keyword as unsigned. The preprocessor symbol **_CHAR_UNSIGNED** is predefined.

With the **-nKu** option, char variables without an explicit signed or unsigned keyword are treated as signed, and the preprocessor symbol **_CHAR_SIGNED** is predefined.

-Kv (default)

-nKv

The **-Kv** option tells the compiler to consider bit fields declared without the signed or unsigned keyword as unsigned.

With the **-nKv** option, bit fields without an explicit signed or unsigned keyword are treated as signed.

-Zan

The **-Zan** option sets the data bus width. The width of the data bus is determined by the processor selected by default. This setting affects the natural alignment of multi-byte data types; for example, **double** alignment is 4 for **-Za4** and 8 for **-Za8**. *n* may be any positive power of 2.

-Zcn

The **-Zcn** option specifies that the alignment of the body of each procedure is to be at least a multiple of *n*. The default value for *n* is determined by the processor selected; this value is 4 for all currently supported processors. *n* may be any positive power of 2.

-Zdn (default 4)

The **-Zdn** option specifies the alignment of data elements within a module. Data elements are allocated on an address that is at least a multiple of *n*. For example, specifying **-Zd4** forces each element (including strings) to start at a quad-aligned address. Since some data elements may have a naturally larger alignment than *n*, the alignment for an element is equal to **max(natural_alignment, n)**. *n* may be any positive power of 2.

-Zin (default -1)

The **-Zin** option indicates the maximum size of an **asm()** statement. This affects how local branches around **asm()** statements are generated. The default value (-1) indicates that **asm()** statements can have indefinite sizes, requiring long branches to span them.

-Zmn (default 1)

-Znn (default 1)

The **-Zmn** and **-Znn** options instruct the compiler how to size structures. They force the sizes of packed (**-Zm**) and unpacked (**-Zn**) structures to be a multiple of *n*. In order to accomplish this, trailing “padding” bytes are added to the end of the structure to adjust its size. *n* may be any positive value.

-Zssize (default 0)

The **-Zs** option tells the compiler to allocate all data items whose size in bytes is equal to or less than *size* to the small data area. Initialized data items are assigned to the **.sdata** section, and uninitialized data items are assigned to the **.sbss** section.

Up to 64 KB of data items with local or global scope can be placed in the small data area. The `_SDA_BASE_` symbol is defined by the linker to be an address relative to which all data in the `.sdata` and `.sbss` sections can be addressed with 16-bit signed offsets, or, if neither section exists, the value 0. The value of `_SDA_BASE_` in an executable is normally loaded into `r13` at process initialization time, and `r13` remains unchanged after that. If you set this value, the linker will attempt to place the `.sdata` and `.sbss` sections within the 64 KB region defined by the symbol. If you do not define the symbol, the linker attempts to place the `.sdata` and `.sbss` sections next to each other and then defines `_SDA_BASE_` as the midpoint of those two sections.

The compiler will generate “short-form” one-instruction references for all data items in the `.sdata` and `.sbss` sections. In executable files, those references are made relative to `r13`. Placing more data items in the small data areas usually results in smaller and faster program execution. However, the small data area is limited. If *size* is defined such that the small data area is too large, the linker will generate an error. This error can be corrected by compiling one or more portions of the code with a smaller value of *size*.

Generate Listing File

`-l[filename]`

The `-l` option writes a listing file containing high-level source code and any diagnostic messages to the standard output device or the specified file.

If the `-l` option is specified with `-P` or `-E`, the `-l` option is ignored.

`-m[filename]`

The `-m` option writes a link map to the standard output device or the specified file.

If the `-m` option is specified with `-P`, `-E`, `-c` or `-S`, the `-m` option is ignored.

Generate Position-Independent Code and Data

`-Mcn`

The `-Mcn` option directs the compiler to assume that the value held in register *n* (which must be in the range of 14 – 30) is the offset between the original link address for the code and the actual relocated code for the current module. Use of this option requires you to modify *entry.c* to ensure that register *n* is initialized with the correct value. The user-defined code that performs or recognizes the relocation should set the register containing the offset.

`-Mcp`

The `-Mcp` option directs the compiler to generate a code fragment in each function prologue that calculates the offset of the current location of the code from the

original link address. Use of this option does not require any additional setup on your part.

-Mcr (default)

The **-Mcr** option directs the compiler to use relative branches if the target is known (true for local branches and most **extern** calls) and absolute addresses loaded to the CNT register, for indirect calls.

Example 3-2. Position-Independent Code

Consider the following code fragment:

```
extern void f( void );

void (*p) (void);

void foo( void ) {
    p = &f;
}
```

If this code is compiled with the **-Mcr** option (the default), there is no code generated for the function prologue, and the code generated for the statement **p = &f** is as follows:

```
addis    3,0,ha(f)
addic    3,3,lo(f)
addis    4,0,ha(p)
stw      3,lo(p)(4)
```

If the same code is compiled instead with the **-Mcp** option, the function prologue contains code to calculate the code offset:

```
mflr     0
bl       L.PIC.1
L.PIC.1:
addis    12,0,ha(L.PIC.1)
addic    12,12,lo(L.PIC.1)
mflr     11
subfc    11,12,11
mtlr     0
ori      3,11,0x0
```

This code stores the current assumed location from the link register in register 0 using the **mflr** instruction, then uses the **bl** instruction to update the link register with the actual location. After that, the **mflr** instruction is used again to load the new location into register 11. The difference between the assumed and actual locations is calculated and stored in register 3, and the original value of the link register is restored from register 0 with the **mtlr** instruction.

The following assembly code is generated for the **p = &f** statement once this prologue is completed:

```
addis    4,0,ha(f)
```

```

addic    4,4,lo(f)
addc     3,3,4
addis    4,0,ha(p)
stw      3,lo(p)(4)

```

This code is almost identical in function to the code generated using the **-Mcr** option, but an additional line has been added. This new line, **addc 3,3,4**, adds the offset into the address in register 4 to account for any relocation of the code.

Compiling the same C source with the **-Mc19** option (in this case, reserving register 19 with **-Mc19**) again results in no prologue. The assembly code generated for the **p = &f** statement is similar to that generated with the **-Mcp** option:

```

addis    3,0,ha(f)
addic    3,3,lo(f)
addc     3,19,3
addis    4,0,ha(p)
stw      3,lo(p)(4)

```

Here, instead of the additional **addc** instruction using a value calculated in the function for the offset, it refers to a value assumed to exist already in register 19. You must ensure that this value is already set in the appropriate register before the function is called. Using this option can create potentially faster code than the **-Mcp** option, since the overhead for the prologue is substantially reduced.

-Mdn

The **-Mdn** option directs the compiler to use register-relative addressing for all data references. This results in data that is position-independent.

-Mda (default)

The **-Mda** option directs the compiler to use absolute addressing for all data references.

Example 3-3. Position-Independent Data

The following source code can be compiled with and without the **-Mdn** option:

```

main() {
    static int static_int;
    int local;
    int temp, temp1;

    temp = 0;
    temp1 = 0;

    local = temp;
    temp1 = local;

    static_int = temp;
    temp1 = static_int;
}

```

```
temp1 = ext_global;
ext_global = temp

global = temp1;
temp = global;
}
```

The assembly generated from this source code compiled without **-Mdn** is as shown:

```
main:
    stwu    1,-24(1)
    addi    3,0,0x0
    ori     4,3,0x0
    stw     4,0x8(1)
    addis   5,0,ha(LS.0.static_int)
    stw     4,lo(LS.0.static_int)(5)
    addis   5,0,ha(ext_global)
    lwz     6,lo(ext_global)(5)
    stw     6,0x10(1)
    stw     4,lo(ext_global)(5)
    lwz     4,0x10(1)
    addis   5,0,ha(global)
    stw     4,lo(global)(5)
    stw     4,0xc(1)
```

Source code compiled with **-Md20** results instead in the following output:

```
main:
    stwu    1,-24(1)
    addi    3,0,0x0
    ori     4,3,0x0
    stw     4,0x8(1)
    addis   5,0,ha(LS.0.static_int)
    addic   5,5,lo(LS.0.static_int)
    addc    5,20,5
    stw     4,0x0(5)
    addis   5,0,ha(ext_global)
    addic   5,5,lo(ext_global)
    addc    5,20,5
    lwz     6,0x0(5)
    stw     6,0x10(1)
    stw     4,0x0(5)
    lwz     4,0x10(1)
    addis   5,0,ha(global)
    addic   5,5,lo(global)
    addc    5,20,5
    stw     4,0x0(5)
    stw     4,0xc(1)
```

Rename Compiler Predefined Sections

These options are used to rename a compiler predefined section to any user-specified name, with the following limitations:

- The new section name must not match any other section name (either user- or predefined).
- The new section name must not match any symbol name (the assembler does not keep section and symbol name spaces separate).
- The new section name cannot begin with a period (.).
- No error checking is performed to check for name conflicts.

Using these options allows arbitrary grouping of sections from separate source modules together by name. These grouped sections can be loaded at prespecified load addresses in memory using the linker **ORDER** command.

-NCname

The **-NC** option sets the constant variables section name. (default: **.rodata**)

-NDname

The **-ND** option sets the data section name. (default: **.data**)

-NIname

The **-NI** option sets the initialization section name. (default: **.init**)

-NSname

The **-NS** option sets the strings section name. (default: **.rodata1**)

-NTname

The **-NT** option sets the code section name. (default: **.text**)

-NZname

The **-NZ** option sets the weak externals section name. (default: **.bss**)

Optimize Code

Safe Optimizations

-O

This option is equivalent to **-Ot**. Optimization defaults to **-ODglrU**.

-OD (default)

-nOD

The **-OD** option instructs the compiler to perform interprocedural call modification analysis. This analysis is performed only if it is possible to determine which routine is being called and if that routine is available. If two source files for the same

executable use the **-OD** option, they become dependent on each other for correct execution. All related files must be recompiled if one file is altered.

-Og (default)

-nOg

The **-Og** option instructs the compiler to perform the main optimizations globally.

-Oi

-nOi (default)

The **-Oi** option instructs the compiler to inline more aggressively than with the **-nOi** option selected. This option can be used in conjunction with the **-Os** and **-Ot** options. With **-Os**, it causes most inline functions and other small functions to be inlined. With **-Ot**, it causes the inlining the **-Ot** option already performs to become more aggressive.

-Oj

-nOj (default)

The **-Oj** option instructs the compiler to inline intrinsic routines when possible.

-Ol (default)

-nOl

The **-Ol** option instructs the compiler to perform the main optimizations locally.

-Or

-nOr (default)

The **-Or** option invokes the scheduler optimization. This reorders generated instructions to make optimal use of the instruction pipeline or instruction parallelism.

-Os

The **-Os** option instructs the compiler to optimize for space, suppressing optimizations that compromise space for speed, which include most inlining and loop unrolling optimizations.

-Ot

The **-Ot** option instructs the compiler to optimize for the execution time (speed) of the generated code on the target. It is equivalent to specifying **-ODgjlR** (that is, enabling all of the safe optimizations).

-OU (default)

-nOU

The **-OU** option enables loop unrolling.

-Ox

The **-Ox** option instructs the compiler to perform the maximum possible optimization, which may include some aggressive optimizations beyond those specified by **-Ot**.

When the **-Ox** option is used with the **-g** option, the resulting program cannot be easily debugged, because debug information is not provided for variables allocated to registers.

Potentially Unsafe Optimizations

-Of

-nOf (default)

The **-Of** option performs floating point expression rearrangement. While mathematically equivalent, these expressions may not be computationally equivalent due to accuracy issues.

-OJ

-nOJ (default)

The **-OJ** option disables intrinsic error checking. The ANSI C and C++ standards require intrinsic routines to do some error checking. For example, math routines are expected to set the variable **errno** on either a domain or range error and to provide certain specific values on range underflow or overflow. If intrinsic routines are inlined, they can be faster if it is known that these checks are not required. In C/C++, for example, the routine **sqrt** can only be inlined if the **-OJ** option is set. This speedup can have a significant impact for some programs.

Name the Output File (Driver Option)

-ofilename

The **-o** option names the output file with the specified name instead of its default name. The output file can be one of the following:

- Preprocessed file — If used with the **-E** or **-P** options
- Assembly source file — If used with the **-S** option
- Object file — If used with the **-c** option
- Library file — If *filename* is specified with a **.lib** extension
- Executable file — Assumed by default
- Preprocessed — By default, the output filename is the source filename stripped of any suffix or prefix.

Send Preprocessor Output to File (Driver Option)

-P[s]

The **-P** option tells the driver to execute the preprocessor only. Preprocessor output is sent to a file having the same name as the source file but with the suffix **.i**. This option is useful for debugging complicated macro definitions and deeply nested conditional directives. (See also the **-C** and **-E** options.)

This option conflicts with **-c**, **-E**, and **-y**.

If **-Ps** is specified, all **-P** actions are performed, but the echoing of the backslash and newline are disabled.

Compilation continues after preprocessing by default; the preprocessed source file is not saved.

Produce Code for Specified Processor

-pprocessor

The **-pprocessor** option produces code for the specified processor. The possible values for *processor* are: **401gf, 403ga, 403gb, 403gc, 403gcx, 405cr, 405ep, 405gp, 405gpr, 440ep, 440gp, 440gx, 505, 509, 5121e, 5123, 5200, 5200b, 533, 534, 535, 536, 555, 5553, 5554, 556, 560, 561, 562, 563, 564, 565, 566, 5xx, 603, 603e, 603e6, 603e7v, 603r, 604, 604e, 740, 7400, 7410, 7410t, 7441, 7445, 7445a, 7447, 7447a, 7448, 745, 7450, 7451, 7455, 7455a, 7457, 745b, 750, 750cx, 750cxe, 750cxr, 750fl, 750fx, 750gl, 750gx, 750gx_dd11_8, 750l, 755, 755b, 755c, 801, 821, 823, 823e, 8240, 8241, 8245, 8245a, 8247, 8248, 8250, 8250a, 8255, 8255a, 8260, 8260a, 8264, 8264a, 8265, 8265a, 8266, 8266a, 8270, 8270vr, 8271, 8272, 8275, 8275vr, 8280, 8313, 8313e, 8314, 8314e, 8315, 8315e, 8321, 8321e, 8323, 8323e, 8343, 8343e, 8347, 8347e, 8349, 8349e, 8358e, 8360e, 8377e, 8378e, 8379e,**

850, 850de, 850dsl, 850sr, 852t, 853t, 8533, 8533e, 8540, 8541e, 8543, 8543e, 8544, 8544e, 8545, 8545e, 8547e, 8548, 8548e, 8555e, 855t, 8560, 8567, 8567e, 8568, 8568e, 8572, 8572e, 857dsl, 857t, 859dsl, 859p, 859t, 860, 860de, 860dp, 860dt, 860en, 860p, 860sr, 860t, 8610, 862, 862p, 862t, 8640, 8640d, 8641, 8641d, 866, 866p, 866t, 870, 875, 880, 885, 8xx, com, e300, e300c2, e300c3, e500v1, e500v2, ec603e, ec603e6, ec603e7v, em603e, g2, g2_le, npe405h, npe405l, and ppc.

For processors that do not contain a floating point unit, specify the **-f** option to instruct the compiler to generate floating point instructions that will be emulated at run time through an exception handler. The assembler will allow these instructions without a diagnostic message.

The e500v2 PowerPC processor variants do not have a floating-point unit, but support floating-point operations through a different unit. As such, the **-f** option should not be specified for these variants. At present, these processors are e500v2, 8533, 8533e, 8543, 8543e, 8544, 8544e, 8545, 8545e, 8547e, 8548, 8548e, 8567, 8567e, 8568, 8568e, 8572, and 8572e.

The **-p505**, **-p509**, **-p533**, **-p534**, **-p535**, **-p536**, **-p555**, **-p556**, **-p560**, **-p561**, **-p562**, **-p563**, **-p564**, **-p565**, and **-p566** options provide a partial floating point unit. They fail to complete floating point operations under some circumstances (for example, handling floating point denormals) and raise the floating point assist exception. The floating point assist exception handler corrects the problem with software floating point emulation and then continues the execution.

For the 750GX processor, some instructions that operate on the condition register may fail to operate correctly at maximum frequency (see IBM PowerPC 750GX Microprocessor Errata Notice for DD1.1, item #8). The **-p750gx_dd11_8** option instructs the compiler to use other instructions to perform the equivalent operations.

Control Diagnostic Messages

Diagnostic messages have different severity levels: error, warning, or informational. You can change the severity level of messages that have **-D** appended to their message ID number.

-Qnumber

The **-Q** option stops compilation if more than *number* errors occur. (default: *number* is 20)

-QA

The **-QA** option suppresses all messages.

-Qe

The **-Qe** option suppresses error, warning, and informational messages.

-Qfn

The **-Qfn** option suppresses the display of the message ID number for each diagnostic. (default: message numbers are displayed)

-Qfs

The **-Qfs** option suppresses display of the source line for each diagnostic. (default: source lines are displayed)

-Qi (default)

The **-Qi** option suppresses informational messages.

-QmX

Selected warning and informational diagnostic messages can be upgraded, downgraded, or entirely suppressed by using the **-QmX** options. *X* can be one of the following:

- i — informational
- w — warning
- e — error
- s — suppress

A comma-separated list of message numbers can be specified on the command line or by using **#pragma options**. Refer to Appendix A, “[Compiler Messages](#)” for a full listing of message numbers. Specifying message numbers for diagnostics at a higher level than “warning” (that is, “error” or “fatal” diagnostics) has no effect.

For example, **-Qms177** completely suppresses the error message “variable ‘*varname*’ was declared but never referenced.” **-Qme177** upgrades this message to an error, stopping code generation.

-Qm{i|w|e}msgnum[,msgnum,...] upgrades or downgrades informational or warning messages to the specified level.

-Qmsgid[,msgid2,...] completely suppresses informational or warning messages.

-Qo

-nQo (default)

The **-Qo** option suppresses the listing (normally produced with **-l** option) of options that are currently active.

The **-nQo** option lists on the output listing options (produced by the **-l** option) that are active.

-Qs

The **-Qs** option suppresses the summary of diagnostic messages.

-Qw

The **-Qw** option suppresses warning and informational messages.

-Qwo

In some cases, the compiler's code generator issues warnings about situations it encounters that deserve your attention. These warnings are separate from the normal diagnostic reporting mechanism used by the compiler. These code generator warnings can be suppressed with the **-Qwo** option.

-nQ

The **-nQ** option permits the display of all messages.

Use Default Libraries for Linking

-q (default)

The **-q** option instructs the linker to use the default libraries for linking in addition to the libraries specified in the **-ecmdfile** option. The **-q** option instructs the linker to include the default libraries

-nq

The **-q** option instructs the linker to use the default libraries for linking in addition to the libraries specified in the **-ecmdfile** option.

The default libraries are not used for linking if **-nq** is specified.

Produce Assembler Source File (Driver Option)

-S

The **-S** option instructs the driver to produce an assembly language file (**.s** extension for UNIX and **.src** for Windows), which can be passed to the assembler to produce an object file (**.o** extension for UNIX and **.obj** for Windows).

This option conflicts with **-c**, **-E**, **-P**, and **-y**.

Control Template Instantiation

The **-tl**, **-tm**, and **-tu** options control the generation of template instances. They are effective only if your program uses templates. These options are mutually exclusive. The rightmost option specified overrides any previous template options.

-tl

The **-tl** option generates all instances of template functions as local functions; static data members of template classes are still generated as public and are shared among modules.

If you use the same template function in multiple files compiled with **-tl**, there will be duplicated instances in your final program.

-tm

The **-tm** option disables automatic generation of template instances; that is, template instances are not generated unless they are explicitly instantiated in the program. This option is used to avoid duplication of code at compilation time. When **-tm** is used, template definitions for the templates that are not generated need not be included.

Refer to the descriptions of **#pragma instantiate** and **#pragma do_not_instantiate** in Chapter 4, “C/C++ Extensions” to learn how to manually control individual template instances.

-tu (default)

The **-tu** option automatically generates all necessary template instances in a compilation unit, except those that are manually disabled, and places them into the object file. You must include all necessary template declarations and definitions in the source file. Duplicate template instances in object files are removed at link time.

Undefine a Preprocessor Macro Name

-Uname

The **-U** option undefines the defined preprocessor macro specified by *name* (this is equivalent to putting **#undef name** at the beginning of the source file). For example:

```
-UNAME1 -Uname2 -UNAME3
```

undefines the preprocessor macros **NAME1**, **name2**, and **NAME3**.

Note



The **-U** (undefine macro) option is processed before the **-D** (define macro) option, regardless of the order in which they appear on the command line.

Modify Naming Conventions

The compiler’s default naming convention puts an underscore at the beginning of symbol names. The C and C++ run-time libraries are built using the default naming convention. Therefore, if a module that contains a reference to a library item is not compiled with the default naming convention, that item is unresolved at link time.

-ui[*char*]

The **-ui** option changes the default insert character for **asm** pseudofunction calls from a back quote (`) to *char*. No insert character is defined if *char* is null.

Control Verbose Output Information (Driver Option)

-Vb

The **-Vb** option displays the copyright notice and version number and continues compilation.

-Vd

The **-Vd** option displays the commands that would be used to invoke each component of the toolkit without executing the commands. You can use the output to create a script for future compilations by redirecting your output to a file.

-Vi

The **-Vi** option displays the commands used to invoke each component of the toolkit as they are executed.

-Vt

The **-Vt** option displays a time stamp for the various stages of the compilation.

-V

The **-V** option displays the copyright notice and version number of each tool component and then exits.

Pass Options to Tools (Driver Option)

-Wa,*option1*[,*option2*,] . . . '

The **-Wa** option passes the specified options directly to the assembler. This option applies only if the driver is invoking the assembler; that is, if the **-E**, **-P**, or **-S** options are not specified.

Enclosing quotes are not necessary when a simple single command is passed with the **-Wa** option, as shown in the following example:

```
cccppc -Wa,-g main.s
```

Complex embedded commands passed by **-Wa** often require both single (') and double (") quotes as shown in the following examples:

```
ccppc -Vi y.cc -Wa,'-f "intfile, case" '  
ccppc "-Wa,-lmain.lis" main.cc
```


In the previous example, both single and double quotes were necessary because the **-f** command line option uses the assembler command line flags **infile** and **case**.

See the Mentor Graphics *Microtec Assembler/Linker/Librarian User's Guide* for your system for information about assembler command line options.

-Wl,option1[,option2,] . . .

The **-Wl** option passes the specified options directly to the linker. This option applies only if the driver is invoking the linker; that is, if **-c**, **-E**, **-P**, or **-S** options are not specified.

Enclosing quotes are not necessary when a simple single command is passed with the **-Wl** option, as shown in the following example:

```
cccppc -Wl,-mtest.map main.cc
```

See the Mentor Graphics *Microtec Assembler/Linker/Librarian User's Guide* for your system for information about linker command line options.

Initialize Uninitialized Global Data

-Xc (default)

The **-Xc** option treats uninitialized global variables as “weak externals” or “C common.” The compiler does not allocate any space for these variables. The linker resolves each such variable to its initialized declaration, if one exists.

For example:

```
int i;
```

is translated to:

```
.extern i  
.comm i,4,4
```

If this variable is not externally defined in another module, the linker allocates space for the largest type with which this variable is declared (but all modules should use the same type) in the **.bss** section.

-Xp

The **-Xp** option allocates space for global variables that have not been explicitly initialized but assigns them no value. The compiler places these variables in the **.bss** section, which is set to zero at program startup time.

For example:

```
int i;
```

is translated to:

```
.sect .bss
.align
.globl i
i:
.space 4
```

Enable Microtec Extensions

-x (default)

-nx

The **-x** option enables the following Microtec extensions to the C and C++ languages:

- Predefined Microtec preprocessor symbol **_MRI_EXTENSIONS**
- Acceptance of **\X** for hexadecimal sequences
- Additional bit field types **char**, **short**, and **long**
- Assembly in-lining with **asm**
- In-lining of selected library functions
- Preprocessor directives **#inform**, **#warning**, and **#error**
- Many C99 features, such as **__func__**, loop-scoped variables, variadic macros, and compound literals. Refer to Chapter 4, “[C/C++ Extensions](#)” for more details.
- Left type casting
- void pointer arithmetic
- **typeof** operator

The **strcpy** library function is expanded inline rather than called.

The **-nx** option disables the Microtec extensions.

Accept Out-of-range Floating Point Constants

-xf

-nxf(default)

The -xf option allows the compiler to accept out-of-range floating point constants without emitting a diagnostic message. When such a number is detected, the compiler will either use "infinity" or zero, as appropriate, for the value of the floating point constant.

```
double f1 = 1.8e308; // accepted with -xf, treated as "infinity"
double f2 = 4.9e-325; // accepted with -xf, treated as 0
```

Enable GNU Extensions

-xG[version]

-nxG

The **-xG** option enables supported GNU extensions to the C and C++ languages. Refer to [Supported GNU C/C++ Extensions](#) for more details. *version* specifies the GNU compiler version. The default value is 4.1.

Check Program Syntax (Driver Option)

-y

The **-y** option checks the program syntax without generating code.

The compiler checks the syntax and generates code by default.

Produce Minor Code Generation Variations (-Zx options)



Note

These options producing minor code generation variations (the **-Zx** options) appear with the **-Kx** options in the “[Produce Minor Code Generation Variations](#)” section, found earlier in this chapter.

Enable C++ Features

-ze

-nze (default)

In C++, the **-ze** option enables C++ exception handling in C++ code. See Chapter 10, “[Interlanguage Calling](#)” for further information on exception handling. When the

-ze option is specified, the macro **_EH** is predefined as 1; otherwise, it is predefined as 0.

The **-nze** option turns off exception handling.

-zr

-nzs (default)

The **-zr** option enables run-time type information. This option is required in order to use the C++ **dynamic_cast** and **typeid** operators.

Using Options

This section discusses how to use some of the compiler options to get the most out of the Microtec C and C++ compilers.

Customizing the Compilation Driver

The compiler features a powerful compile/assemble/link driver that automatically invokes the appropriate product. In addition to simplifying compiler invocation, the driver can be used to customize the compilation environment for your development needs.

Compiler options are read from left to right. If an option is specified multiple times on the command line, the last occurrence of that option is enforced. Options specified apply to all files being compiled (the position of filenames on the command line is not significant).

Example 3-4. Overriding Command Line Options

```
cccppc -g -l -Flp44 main.cc read.s scr.lib -ng -Flp100 -g -ng
```

This example compiles **main.cc** without debugging information (the rightmost option is **-ng**) and uses a listing page length of 100 lines (the rightmost option is **-Flp100**).

Example 3-5. Overriding Command Line Options in a Script

```
#!/bin/csh
set extraoptions="$argv[2-$#argv]"
ccppc $1 -c -g -O -Flp44 $extraoptions
```

This UNIX C shell script, named **compile**, assigns all the invocation arguments that follow the first argument (the filename) to **extraoptions**:

<code>compile test.cc</code>	Compiles test.cc with normal options.
<code>compile test.cc -g -l</code>	Compiles test.cc with normal options plus the -g and -l options.
<code>compile test.cc trick.cc</code>	Compiles test.cc and trick.cc with normal options.

You can use the driver option **-dfilename** to customize your compilation if you are not satisfied with the default setting of some of the options. This option also lets a system administrator establish installation defaults.

The driver reads the lines in *filename* as though they were specified on the command line. The file can contain multiple lines; the new-line character is interpreted as white space.

Example 3-6. Options File

```
-g
-Flp66
-e /usr/misc/myproject_linkfile
-DDEBUG_MODE -DHOST=SUN4
```

If the */usr/misc/myproject_defaults* file contains the lines shown in the preceding example, and if **-d/usr/misc/myproject_defaults** is specified on the command line, all of these options are applied to the compilation. Alternatively, a shell **alias** command can be used to force customization:

```
alias cccppc "ccppc -d/usr/misc/myproject_defaults"
```

Pragmas and Options

Options normally set on the command line can be set within the program file itself using the **#pragma options** construct. This directive lets you specify local compiler options inside the source file. These options override options on the invocation line because they are the last encountered and affect the compilation of the entire file, not just code subsequent to their occurrence.

Example 3-7. Multiple #pragma options Directives

```
#pragma options -g
funct1() /* always include debugging info */
...
#pragma options -ng
funct2() /* do not include debugging info */
...
```

If the lines shown in this example occur in the same file, the compiler applies the last option encountered (**-ng**) to the entire file. The compiler does not apply different options to **funct1** and **funct2**. In other words, the options cannot be turned on and off to affect only portions of a source file.

A typical use of **#pragma options** is with makefiles that compile a related group of sources. One of the files in this group might require particular options, such as optimizing for space or eliminating debugging information. The **#pragma options** directive lets you specify command line options that apply to that file only:

```
#pragma options -ng /* No debugging info for this file */
```

Driver options such as **-S**, **-e**, or **-I** are not accepted because the driver decides how to proceed before reading any input files, and all driver options have been processed prior to reading your file. The same happens with listing control options and preprocessor options. Generally, it is safe to give options that affect debugging, optimizations, and code generation.

Example 3-8. #pragma options Directives Across Modules

```
modulea.c:
    #pragma options -Og

moduleb.c:
    #pragma options -Or
```

This example shows two separate modules with the **#pragma options** using different levels of optimization within the different modules. Optimizations can be fine-tuned for the specific code in each module by specifying this information in the separate modules.

```
#pragma options push
#pragma options pop
```

The push/pop forms can be used to change options related to displaying diagnostics (that is, **-Q** options) in a specific interval.

Example 3-9. #pragma options push and pop

Assume you are annoyed by a particular compiler message such as:

```
(W) C0177-D; variable "a" was declared but never referenced
```

The “C” is the compiler signature. The message ID is (0)177. The **-D** after the message ID stands for discretionary, implying that the message can be suppressed, via the option **-Qms<num>**. Typically though, you may want to suppress a message only in a particular range. You can isolate the effect by surrounding the suppression with push/pop.

```
#pragma options push // remember the options in force
#pragma options -Qms177 // suppress annoying message
void X(...) {
    ...
}
```

```
#pragma options pop // restore the options in force before X
```

Pushes and pops can be nested.

Locating Header Files

Sometimes you need to know the exact origin of header files included during a compilation. The quickest way to access this information is to use the **-E** compiler option, which invokes the preprocessor and sends an annotated listing to the standard output device.

Example 3-10. Locating Header Files

```
#line 1 "input.c"
#line 1 "x.h"

    extern foo(struct tag *p);

#line 2 "input.c"
#line 1 "/usr/mri/include/cccppc/stdio.h"

    extern struct iobuf {
    ...
```

This example shows preprocessor output from the command line:

```
cccppc -E input.c
```

for a file *input.c* containing:

```
#include "x.h"
#include <stdio.h>
```

You can eliminate any uncertainty about the actual header files used with your program by viewing the full path names of the include files used. Refer to the **-J** compiler option for information on overriding the standard include file (<>) location.

Determining Option Defaults

The options you specify for a given compilation only represent a subset of all the options applied to your file(s). Many compiler options are turned on by default, and these defaults can affect compilation.

Compile an empty file to produce a listing file to determine the options in effect:

```
cccppc -l -y empty.c
```

All of the options in force are listed. The **-y** option limits the compiler to checking the syntax of the program so that errors produced by an empty file are kept to a minimum.

Producing Listing Files

The listing files produced by the compiler are intended for line printer output with a default page length of 55 lines. You can change this number with the **-Flpnumber** option. Specifying **-Flp0** tells the compiler to prepare the listing for a continuous printout with no page breaks.

The **#pragma list** directives can turn the listing option on and off. You can turn the listing option on and off several times within the same file since these directives apply to code following the directive.

Example 3-11. Turning a Listing File Off and On

```
foo() {  
#pragma list off  
    int invisible, visible;  
    invisible=1;  
#pragma list on  
    visible=99;  
#pragma list resume  
    invisible=1;  
#pragma list resume  
    visible=99;  
}
```

The following is the listing produced from this example:

```
foo() {  
#pragma list off  
    visible=99;  
#pragma list resume  
    visible=99;  
}
```

The **#pragma list resume** directive restores the status of the listing to the setting that was in force prior to its previous matching **#pragma list off** or **#pragma list on** directive. The listing option **-l** must be specified for the **#pragma list** directives to have any effect.

Command Line Examples

This section gives examples of the use of the compiler and driver.

See Chapter 5, “[Using Libraries](#)” for more information about the libraries described in these examples.

Example 3-12. Simple C++ Source Compilation

```
cccppc program.cc
```

This command compiles the C++ source file *program.cc* to produce the object file *program.o* (UNIX) or *program.obj* (Windows). The object file is then linked with the default linker command file and the default library to produce the executable file *program.x* (UNIX) or *program.abs* (Windows).

Example 3-13. C++ Compilation From Multiple Sources

```
cccppc module1.cxx module2.cc module3.cc -llisting
```

This command compiles the three C++ source files *module1.cxx*, *module2.cc*, and *module3.cc* to produce three object files *module1.o*, *module2.o*, and *module3.o* (UNIX) or *module1.obj*, *module2.obj*, and *module3.obj* (Windows). A compiler-generated listing of each source file is written to the file listing (**-l** option). The three object files are then linked with the default linker

command file and the default library to produce the executable file *module1.x* (UNIX) or *module1.abs* (Windows). The name of the executable file is derived from the name of the first source file that appears on the command line; the default extension is added to it. In this case, the object files are not deleted; the object file is deleted only if a single source file is successfully linked.

Example 3-14. Producing an Object File

```
cccppc -c module.s
```

This command assembles the assembly file *module.s* to produce the object file *module.o* (UNIX) or *module.obj* (Windows). The **-c** option prevents the driver from linking this module to produce an executable file.

Example 3-15. Compiling and Assembling Multiple Files With a Library

```
cccppc main.cc asmfile.s extra.o utils.lib
```

This command:

- Compiles *main.cc* to produce *main.o* (UNIX) or *main.obj* (Windows).
- Assembles *asmfile.s* to produce *asmfile.o* (UNIX) or *asmfile.obj* (Windows).
- Links the object files for *main.cc* and *asmfile.s* with the default linker command file, the default library, and *extra.o* to produce an executable file named *main.x* (UNIX) or *main.abs* (Windows). Modules from *utils.lib* are also linked if necessary.

Suggested Option Combinations

Here are some suggested command line option combinations that you may find useful:

To produce code that works best with the debugger:

```
mccppc -g -nO file.c ...
```

To produce code that is most compact:

```
mccppc -Osix -Zs8 file.c ...
```

To produce code that executes fastest:

```
mccppc -Otrix -Zs8 file.c ...
```


This chapter describes the extensions to the C and C++ languages.

Microtec C/C++ Extensions

Predefined Mirror Symbols

The Microtec C and C++ compilers provide predefined/mirror symbols that allow you to write code that is customized for the following:

- A target system
- A particular host compiler
- Various aspects of the application, such as data size and memory addressing

The following sections give detailed descriptions of various predefined macros available with the Microtec C/C++ compilers.

Standard Mirrors

Standard mirrors are defined by the C/C++ standard. [Table 4-1](#) describes these mirrors.

Table 4-1. Standard Mirrors

Mirror Symbol	Description
<code>__STDC__</code>	Contains a value of 1 if the -A option is specified.
<code>__FILE__</code>	A character string that consists of the ASCII representation of the current filename. Use the #line directive to change the value of this symbol.
<code>__LINE__</code>	A decimal constant that represents the current line number within the current file. Use the #line directive to change the value of this symbol.
<code>__DATE__</code>	A character string that consists of the ASCII representation of the current date in the following format: <i>Mmm dd yyyy</i> <i>Mmm</i> has the same format as the month given by the asctime run-time function. <i>dd</i> has a leading space if it is a one-digit number.

Table 4-1. Standard Mirrors (cont.)

Mirror Symbol	Description
<code>__FUNC__</code> (or <code>__func__</code>)	A character string that consists of the ASCII representation of the current function name. It is a predefined identifier and not only for preprocessing.

Compiler Mirrors

Compiler mirrors reflect the version of the compiler in use. [Table 4-2](#) describes these mirrors.

Table 4-2. Compiler Mirrors

Mirror Symbol	Description
<code>_MRI</code>	Defined as 1.
<code>_MICROTEC</code>	Defined as 1.
<code>_TARGET_PPC</code>	Defined as 1 if the compiler is being used for a PPC target; otherwise undefined.
<code>_VERSION</code>	The version of the compiler in literal string format.

Example 4-1. Using Compiler Mirror Symbols

```
#ifdef _MICROTEC
static char version[] =
"Module " __FILE__ ", compiled with Microtec "
"compiler version " _VERSION;
#endif
```

This example generates a string like the following:

```
Module prog.cc, compiled with Microtec C/C++ Compiler version 3.0
```

Language Mirrors

Language mirrors reflect the source language of the file being compiled. [Table 4-3](#) describes these mirrors.

Table 4-3. Language Mirrors

Mirror Symbol	Description
<code>_MCCPPC</code>	Defined as 1 if the invoking driver is for the C language; undefined otherwise.
<code>_CCCPPC</code>	Defined as 1 if the invoking driver is for the C++ language; undefined otherwise.

Table 4-3. Language Mirrors (cont.)

Mirror Symbol	Description
<code>_FILE_EXT</code>	A character string consisting of the ASCII representation of the extension of the current file. For example, for <i>test.c</i> , the value of <code>_FILE_EXT</code> is <code>c</code> .
<code>__cplusplus</code>	Defined as 1 if the source language of the file being compiled is C++; undefined otherwise.
<code>__c</code>	Defined as 1 if the source language of the file being compiled is C++; undefined otherwise.

Host Mirrors

Host mirrors reflect the compilation host. [Table 4-4](#) describes these mirrors.

Table 4-4. Host Mirrors

Mirror Symbol	Description
<code>_UNIX</code>	Defined as 1 if the compilation host is any UNIX variant.
<code>_SOLARIS</code>	Defined as 1 if the compilation host is Solaris.
<code>_LINUX</code>	Defined as 1 if the compilation host is Linux.
<code>_WINDOWS</code>	Defined as 1 if the compilation host is any version of Windows.

Type Mirrors

The size and alignment of basic types is available at compile time and, in general, not at the time of preprocessing. The `_SIZEOF_X` and `_ALIGNOF_X` symbols supply the size and alignment, respectively, of type *X* as shown in [Table 4-5](#). All values are given in reference to the PowerPC Family.

Table 4-5. Type Mirrors

Type	<code>_SIZEOF_X</code>	<code>_ALIGNOF_X</code>
CHAR	<code>_SIZEOF_CHAR</code>	<code>_ALIGNOF_CHAR</code>
WCHAR_T	<code>_SIZEOF_WCHAR_T</code>	<code>_ALIGNOF_WCHAR_T</code>
SHORT	<code>_SIZEOF_SHORT</code>	<code>_ALIGNOF_SHORT</code>
INT	<code>_SIZEOF_INT</code>	<code>_ALIGNOF_INT</code>
LONG	<code>_SIZEOF_LONG</code>	<code>_ALIGNOF_LONG</code>
LONG LONG	<code>_SIZEOF_LONG_LONG</code>	<code>_ALIGNOF_LONG_LONG</code>
FLOAT	<code>_SIZEOF_FLOAT</code>	<code>_ALIGNOF_FLOAT</code>
DOUBLE	<code>_SIZEOF_DOUBLE</code>	<code>_ALIGNOF_DOUBLE</code>

Table 4-5. Type Mirrors (cont.)

Type	<code>_SIZEOF_X</code>	<code>_ALIGNOF_X</code>
LONG DOUBLE	<code>_SIZEOF_LONG_DOUBLE</code>	<code>_ALIGNOF_LONG_DOUBLE</code>
CODE POINTER	<code>_SIZEOF_CODE_POINTER</code>	<code>_ALIGNOF_CODE_POINTER</code>
DATA POINTER	<code>_SIZEOF_DATA_POINTER</code>	<code>_ALIGNOF_DATA_POINTER</code>
POINTER	<code>_SIZEOF_POINTER</code>	<code>_ALIGNOF_POINTER</code>

The alignment and size of some types can be change with certain options. Refer to Chapter 3, “[Using Command Line Options](#)” for more details.

Note



POINTER is defined only if **CODE_POINTER** and **DATA_POINTER** have the same size; otherwise, it is undefined.

[Table 4-6](#) shows the underlying types used for `_WCHAR_T`, `_PTRDIFF_T`, and `_SIZE_T`.

Table 4-6. Underlying Types

Type	Underlying Type
<code>_WCHAR_T</code>	unsigned short
<code>_PTRDIFF_T</code>	The type of the result of subtracting two pointers (for example, signed int)
<code>_SIZE_T</code>	The type of the result of the sizeof operator (for example, unsigned int)

Target Mirrors

Target mirrors reflect properties of the compilation target. [Table 4-7](#) summarizes the functionality of target mirrors.

Table 4-7. Target Mirrors

Mirror Symbol	Description
<code>_VARIANT</code>	Defined for the variant given with -pVARIANT . <i>VARIANT</i> appears in all uppercase. Refer to the following example for more details.
<code>_BUS_WIDTH</code>	The size (in bits) of the target data bus.

Example 4-2. Using the `_VARIANT` Symbol

The variant specified by **-pVARIANT** has a corresponding macro, `_VARIANT`, which is set to 1. For example, if **-p405gp** is specified, then a macro `_405GP` is defined as 1.

If the compiler detects a change of target through the use of **#pragma options**, as in the following example:

```
#pragma options -p440ep
```

then the Microtec C/C++ compiler undefines the existing macro (such as **_405GP**) and defines the corresponding one (**_440EP**, in this example) as 1.

Example 4-3. Using the **_BUS_WIDTH** Symbol

```
#if _BUS_WIDTH == 16
printf("Size of Integer is 2 bytes\n");
#endif
```

Section Information Mirrors

The Microtec C/C++ compiler provides a method for accessing the starting address of any section using the **_STARTOF(*sec_name*)** macro, and the size of the section using the **_SIZEOF(*sec_name*)** macro.

Example 4-4. Using Section Information Mirrors

```
#include <mriext.h>
void main() {
    int start_addr, sec_size;
    . . .
    start_addr= _STARTOF(.text);
    sec_size = _SIZEOF(.text);
    . . .
}
```

Utilities

The Microtec C/C++ compilers introduce a number of macros to allow for string manipulation. All string manipulation macros have two versions, as described in the following sections and shown in [Table 4-8](#):

Parameter Non-Evaluating Macros

These macros begin with two leading underscores (**__**) and do not evaluate macros. The operands of these macros can only be strings. For example,

```
__TOUPPER("test.c")
```

returns **TEST.C**.

Parameter Evaluating Macros

These macros begin with a single leading underscore (`_`) and evaluate their parameters. The operands of these macros can be strings or other macros. For example,

```
_TOUPPER (__FILE__)
```

returns **TEST.C** if the current compilation unit is *test.c*. `_TOUPPER("test.c")` is also a valid construct and returns identical results.

Table 4-8. String Manipulation Mirrors

Parameter Evaluating Version	Parameter Non-Evaluating Version	Description
<code>_STR_CMP</code>	<code>__STR_CMP</code>	Compares its two argument strings and is replaced by the result. Both strings are compared in lexicographic order. The macro will be replaced by one of the following values: <ul style="list-style-type: none"> • -1 if <i>string1</i> < <i>string2</i> • 0 if <i>string1</i> = <i>string2</i> • 1 if <i>string1</i> > <i>string2</i>
<code>_STR_NOCASE_CMP</code>	<code>__STR_NOCASE_CMP</code>	This macro behaves exactly like <code>_STR_CMP</code> , but it employs a case-insensitive version of string comparison.
<code>_SUBSTR</code>	<code>__SUBSTR</code>	This macro takes two string arguments. <code>_SUBSTR</code> is replaced by 1 if one <i>string1</i> is a substring of <i>string2</i> , and 0 otherwise. The examples following this table give further details.
<code>_STRIP_QUOTES</code>	<code>__STRIP_QUOTES</code>	This macro takes a single argument that is a string literal and removes the quotes. See the examples following this table for further details.
<code>_TOLOWER</code>	<code>__TOLOWER</code>	This macro takes a single argument that is a string literal and returns a string literal with all letters in lowercase.
<code>_TOUPPER</code>	<code>__TOUPPER</code>	This macro takes a single argument that is a string literal and returns a string literal with all letters in uppercase.

__STR_CMP/__STR_CMP

Syntax

```
__STR_CMP(string1, string2)  
__STR_CMP(string1, string2)
```

Description

Compares its two argument strings and is replaced by the result. Both strings are compared in lexicographic order. The macro will be replaced by one of the following values:

- -1 if *string1* < *string2*
- 0 if *string1* = *string2*
- 1 if *string1* > *string2*

Example

```
__STR_CMP("hella", "hello")      is replaced by -1  
__STR_CMP("hello", "hello")     is replaced by 0  
__STR_CMP("hello", "hella")     is replaced by 1
```

If you pass another macro as an argument, this non-evaluating version cannot evaluate that macro. You must use the parameter evaluating version of this macro:

```
__STR_CMP( __FILE__ , "main.c")
```

is replaced by **0** if **__FILE__** returns a string literal **main.c**.

__STR_NOCASE_CMP/__STR_NOCASE_CMP

Syntax

`__STR_NOCASE_CMP(string1, string2)`

`__STR_NOCASE_CMP(string1, string2)`

Description

Compares its two argument strings without regard to case and is replaced by the result. Both strings are compared in lexicographic order. The macro will be replaced by one of the following values:

- -1 if *string1* < *string2*
- 0 if *string1* = *string2*
- 1 if *string1* > *string2*

Example

`__STR_NOCASE_CMP("hella", "heLLo")` is replaced by **-1**

`__STR_NOCASE_CMP("hello", "heLLo")` is replaced by **0**

`__STR_NOCASE_CMP("hello", "heLLa")` is replaced by **1**

If you pass another macro as an argument, this non-evaluating version cannot evaluate that macro. You must use the parameter evaluating version of this macro:

`__STR_NOCASE_CMP(__FILE__ , "mAin.c")`

is replaced by **0** if **__FILE__** returns either of the string literals **main.c** or **Main.c**.

__SUBSTR/___SUBSTR

Syntax

```
__SUBSTR(string_being_searched, string_to_search)  
___SUBSTR(string_being_searched, string_to_search)
```

Description

Returns 1 if *string_being_searched* is a substring of *string_to_search*, and 0 otherwise.

Example

```
___SUBSTR("hello", "helloworld")
```

is replaced by **0**, because the second string is not a substring of the first string.

If you pass another macro as an argument, this non-evaluating version cannot evaluate that macro. You must use the parameter evaluating version of this macro:

```
__SUBSTR("linkmain.c", __FILE__)
```

is replaced by **1** if **__FILE__** returns a string literal **main.c** and **0** if **__FILE__** does not return a string literal that is a substring of **linkmain.c**.

_STRIP_QUOTES/___STRIP_QUOTES

Syntax

`_STRIP_QUOTES(string)`

`___STRIP_QUOTES(string)`

Description

Removes any quotes from *string*.

Example

`___STRIP_QUOTES("hello")` returns **hello**.

Similarly, its parameter evaluation version:

`_STRIP_QUOTES(___FILE__)` returns **main.c** if the filename is **main.c**.

_TOLOWER/ __TOLOWER

Syntax

`_TOLOWER(string)`

`__TOLOWER(string)`

Description

Returns *string* with all letters in lowercase.

Example

`__TOLOWER("Hello.c")`

returns **hello.c**.

`__TOLOWER("HELLO.C")`

returns **hello.c**.

The parameter evaluating version:

`_TOLOWER(__FILE__)`

returns **main.c** if the filename is **Main.c**.

_TOUPPER/__TOUPPER

Syntax

_TOUPPER(*string*)
__TOUPPER(*string*)

Description

Returns *string* with all letters in uppercase.

Example

```
__TOUPPER("Hello.c")           returns HELLO.C.  
__TOUPPER("HELLO.C")          returns HELLO.C.
```

The parameter evaluating version:

```
_TOUPPER(__FILE__)             returns MAIN.C if the filename is Main.c.
```

Option Mirrors

Option mirrors reflect the current values of various options. They are logically updated after every **#pragma options** directive. The values of option mirrors include derived options, which are also updated at the end of every **#pragma options** directive.

The option mirrors are described in [Table 4-9](#). Refer to the “[Parameter Non-Evaluating Macros](#)” and “[Parameter Evaluating Macros](#)” sections in this chapter for the differences between parameter non-evaluating macros and parameter evaluating macros.

Table 4-9. Option Mirrors

Parameter Evaluating Version	Parameter Non- Evaluating Version	Description
_OPTION_AVAIL	__OPTION_AVAIL	Returns 0 or 1 depending on the availability of the specified option.
_OPTION_VALUE	__OPTION_VALUE	Returns the value of the option in a numeric form (using 0 or 1) for Boolean/numeric options and as a string literal for string options.

The **_OPTION_AVAIL/__OPTION_AVAIL** macros check to see if the specified command line option is available (that is, if it would be accepted if supplied in the command line). Its operand must be a string enclosed in double quotes (" ") and include a leading dash (-).

The following macro returns 1 if the **-Zd** option is currently implemented in the compiler, and 0 otherwise.

```
__OPTION_AVAIL ( "-Zd" )
```

The **__OPTION_VALUE/___OPTION_VALUE** macros are replaced according to the value of the option specified as an argument. If the option is a Boolean option, it is replaced by 1 if the option is **TRUE** and 0 if it is **FALSE**. If the option takes an integer value, it is replaced by the integer literal. If the option takes a literal string, the macro is replaced by the literal string in double quotes (" ").

Example 4-5. __OPTION_VALUE Macros

<code>__OPTION_VALUE ("-KE")</code>	returns 1 if the target is big-endian; otherwise, it returns 0.
<code>__OPTION_VALUE ("-Za")</code>	returns 4 if the -Za4 option was specified for this compilation unit.
<code>__OPTION_VALUE ("-p")</code>	returns 405GP if the -p405GP option was specified for this compilation unit.

Shorthand Forms

The Microtec C/C++ compiler provides a shorthand notation for important option mirrors. [Table 4-10](#) lists the shorthand forms of these options.

Table 4-10. Option Mirror Shorthand Forms

Shorthand Form	Option Mirror
<code>_MRI_EXTENSIONS</code>	<code>__OPTION_VALUE("-x")</code>
<code>_DEBUG</code>	<code>__OPTION_VALUE("-g")</code>
<code>_BIG_ENDIAN</code>	<code>__OPTION_VALUE("-KE")</code>
<code>_LITTLE_ENDIAN</code>	<code>__OPTION_VALUE("-nKE")</code>
<code>_CHAR_SIGNED</code>	<code>__OPTION_VALUE("-nKu")</code>
<code>_CHAR_UNSIGNED</code>	<code>__OPTION_VALUE("-Ku")</code>
<code>_PACKED_STRUCTS</code>	<code>__OPTION_VALUE("-KP")</code>
<code>_HW_SUPPORTS_FP</code>	<code>__OPTION_VALUE("-KF")</code>
<code>_FPU</code>	<code>__OPTION_VALUE("-f ")</code>
<code>_PID_REG</code>	<code>__OPTION_VALUE("-Md")</code>
<code>_PIC</code>	<code>__OPTION_VALUE("-Mcr")</code>
<code>_PID</code>	<code>__OPTION_VALUE("-Mda")</code>
<code>_PIC_REG</code>	<code>__OPTION_VALUE("-Mc")</code>

See the “[Preprocessor Directives](#)” section in this chapter for information about using the **#pragma macro** preprocessor directive to generate a list of predefined processor symbols and their values.

Preprocessor Directives

In addition to the standard directives provided by the ANSI C preprocessor, the Microtec C++ compiler provides several directives that extend beyond standard ANSI C. A complete listing of the ANSI C preprocessor directives can be found in ANSI C books available at bookstores. This section provides a syntax, description, and example of these preprocessor directives. [Table 4-11](#) lists Microtec preprocessor directives.

Table 4-11. Microtec Preprocessor Directives

Directive	Description
#error	Issues an error message
#informing	Issues an informational message
#pragma asm	Begins embedded assembly instructions
#pragma do_not_instantiate	Disables instantiation of specified template
#pragma eject	Controls page listing
#pragma endasm	Ends embedded assembly instructions
#pragma error	Issues an error message and halts compilation
#pragma info	Issues an informational message
#pragma instantiate	Instantiates specified template manually
#pragma list	Controls source line listings
#pragma macro	Lists predefined processor symbols
#pragma options	Specifies command line options
#pragma warn	Issues a warning message
#warning	Issues a warning message

#error — Issues an Error Message

Syntax

#error *message*

Parameters

- **message**

An error message. No quotes are required around *message*.

Description

The **#error** directive displays the error message that follows the **#error** directive. The message is displayed on the standard output device. Compilation stops after the message is displayed.

Example

```
#if _STR_CMP(_VERSION, "3.0") < 0
#error Not Valid Version Number
#endif
```

#informing — Issues an Informational Message

Syntax

#INForming *message*

Parameters

- **message**

An informational message. No quotes are required around *message*.

Description

The **#informing** directive displays an informational message on the standard output device. Compilation continues after the message is displayed. This message is displayed only if the **-nQA**, **-QmeC0627**, or **-QmeC0627** option is enabled on the command line.

Example

```
#if _STR_CMP(_VERSION, "3.0") < 0
#info Not Valid Version Number
#endif
```

#pragma asm — Begins Embedded Assembly Instructions

Syntax

`#pragma asm`

Description

The **#pragma asm** directive indicates that the source lines that follow are to be interpreted as assembly code. The **#pragma endasm** directive indicates the end of the assembly code sequence.

Use the **#pragma asm** and **#pragma endasm** directives only outside of a function body.

Notes

Use the **asm** pseudofunction to embed assembly code within a function body. Refer to the “[asm Support](#)” section in this chapter for more information.

Macros and preprocessor directives can be used within the assembly code.

Everything between the **#pragma asm** and **#pragma endasm** directives must be in assembly language format.

Example

```
#pragma asm
    (assembly source code)
#pragma endasm
```

#pragma do_not_instantiate — Disables Instantiation of Specified Template

Syntax

```
#pragma do_not_instantiate template
```

Parameters

- **template**
A template class or template function.

Description

The **#pragma do_not_instantiate** directive disables instantiation of the specified template.

Notes

See the detailed description of template instantiation under the “[Manual Template Instantiation](#)” section in Chapter 8, “[Template Instantiation](#)”.

Example

```
#pragma do_not_instantiate Stack<int>
// Do not instantiate any class or function for Stack<int>

#pragma do_not_instantiate void Stack<int>::size()
// Do not instantiate the size function from Stack<int>

#pragma do_not_instantiate void swap(int, int)
// Do not instantiate the swap(int, int) function defined in
// any template
// Only one template should define swap(int, int)
```

#pragma eject — Controls Page Listing

Syntax

`#pragma eject`

Description

The **#pragma eject** directive forces page ejects during listing output. When the **#pragma eject** command is used during listing output, the compiler terminates the output listing on the current page, ejects the page, then continues listing output at the beginning of the next page.

#pragma endasm — Ends Embedded Assembly Instructions

Syntax

```
#pragma endasm
```

Description

The **#pragma endasm** directive indicates the end of an assembly code sequence.

Use the **#pragma asm** and **#pragma endasm** directives only outside of a function body.

Notes

Use the **asm** pseudofunction to embed assembly code within a function body. Refer to the “[asm Support](#)” section in this chapter for more information.

Macros and preprocessor directives can be used within the assembly code.

Everything between the **#pragma asm** and **#pragma endasm** directives must be in assembly language format.

Example

```
#pragma asm  
    (assembly source code)  
#pragma endasm
```

#pragma error — Issues an Error Message and Halts Compilation

Syntax

`#pragma error message`

Parameters

- **message**

An error message. No quotes are required around *message*.

Description

The **#pragma error** directive displays the message that follows the **#pragma error** directive. The message is displayed on the standard output device. Compilation stops after the message is displayed.

Notes

Use this directive instead of the **#error** directive if you have disabled Microtec extensions. See the **-x** option in the “[Using Command Line Options](#)” chapter, for a description of the compiler command line options that enable and disable Microtec extensions.

Example

```
#pragma options -nx
#if _STR_CMP(_VERSION, "3.0") < 0
#pragma error Not Valid Version Number
#endif
```

#pragma info — Issues an Informational Message

Syntax

`#pragma info message`

Parameters

- **message**

An informational message. No quotes are required around *message*.

Description

The **#pragma info** directive displays the informational message that follows the **#pragma info** directive. The message is displayed on the standard output device. Compilation continues after the message is displayed. This message is displayed only if the **-nQA**, **-QmeC0625**, or **-QmeC0625** option is enabled on the command line.

Notes

Use this directive instead of the **#info** directive if you have disabled Microtec extensions. See the **-x** option in the “[Using Command Line Options](#)” chapter, for a description of the compiler command line options that enable and disable Microtec extensions.

Example

```
#pragma options -nx
#if _STR_CMP(_VERSION, "3.0") < 0
#pragma info Not Valid Version Number
#endif
```


#pragma instantiate — Instantiates Specified Template Manually

Syntax

```
#pragma instantiate template
```

Parameters

- **template**
A template class or template function.

Description

The **#pragma instantiate** directive explicitly creates an instance of the specified template. This **#pragma** has no effect on template functions when the **-tl** option is used.

Notes

See the detailed description of template instantiation under the “[Manual Template Instantiation](#)” section in the “[Template Instantiation](#)” chapter.

Example

```
#pragma instantiate Stack<int>
// Instantiate all classes and functions defined in Stack<int>

#pragma instantiate void Stack<int>::size()
// Instantiate the size function from Stack<int>

#pragma instantiate void swap(int, int)
// Instantiate the swap(int, int) function defined in any template
// Only one template should define swap(int, int)
```

#pragma list — Controls Source Line Listings

Syntax

`#pragma list {off | on | resume}`

Parameters

- **off**
Indicates that the source lines that follow the directive are not to be written to the listing file, regardless of the setting of the list file command line option.
- **on**
Indicates that the source lines that follow are to be written to the listing file. This directive has an effect only if the list file command line option is specified.
- **resume**
Restores the status of the listing to the setting that was in force prior to its previous matching **#pragma list off** or **#pragma list on** directive.

Description

This indicates how the following lines are to be treated in the listing file, according to the option supplied to **#pragma list**.

Example

Enclose the code as follows to disable the listing of a segment of source code:

```
#pragma list off
    (source_code)
#pragma list resume
```

The *source_code* in this example is not listed; the listing status of the surrounding code is unaffected.

#pragma macro — Lists Predefined Processor Symbols

Syntax

#pragma macro

Description

The **#pragma macro** directive causes the compiler to generate a list of predefined processor symbols and their values. This list is displayed on the standard output device.

Example

prog1.c:

```
#pragma macro

int main (int argc, char **argv) {
    . . .

> mccppc -x prog1.c

_603
_BIG_ENDIAN_OPTION_VALUE("-KE")
_BUS_WIDTH 64
_CHAR_SIGNED_OPTION_VALUE("-nKu")
_HW_SUPPORTS_FP_OPTION_VALUE("-KF")
_MCCPPC 1
_MICROTEC 1
_MSDOS 1
_SIZEOF_CHAR 1
. . .
```

#pragma options — Specifies Command Line Options

Syntax

#pragma options *option_list*

Parameters

- **option_list**

A list of the command line options specified.

Description

The **#pragma options** directive allows you to specify command line options in the source file. Generally, only command line options related to optimization or code generation can be used successfully with this **#pragma**. Options given with this **#pragma** have scope over the entire module.

Options specified using **#pragma options** take precedence over options specified on the command line.

Notes

Options for the driver or preprocessor are ignored and may produce a warning message.

Example

Use the following **#pragma** in the source code to ensure that a module is compiled with debugging information:

```
#pragma options -g
```

#pragma warn — Issues a Warning Message

Syntax

`#pragma warn message`

Parameters

- **message**

A warning message. No quotes are required around *message*.

Description

The **#pragma warn** directive displays the warning message that follows the **#pragma warn** directive. The message is displayed on the standard output device. Compilation continues after the message is displayed.

Notes

Use the **#pragma warn** directive instead of the **#warning** directive if you have disabled Microtec extensions. See the **-x** compiler option in the “[Using Command Line Options](#)” chapter for a description of the command line options that enable and disable Microtec extensions.

Example

```
#if ! _MRI_EXTENSIONS /* -nx active */  
#if _STR_CMP(_VERSION, "3.0") < 0  
#pragma warn Not Valid Version Number  
#endif  
#endif
```

#warning — Issues a Warning Message

Syntax

#WARNIng *message*

Parameters

- **message**

A warning message. No quotes are required around *message*.

Description

The **#warning** directive displays the warning message that follows the **#warning** directive. The message is displayed on the standard output device. Compilation continues after the message is displayed.

Example

```
#if _STR_CMP(_VERSION, "3.0") < 0
#warn Not Valid Version Number
#endif
```

Type Cast on Left

A type cast on the left offers efficient control of the architecture at a high language level. It allows using the C/C++ style cast operator on the left side of an assignment (or on an lvalue). This extension is available in both Microtec C and C++ compilers. The type being cast to and the operand of the cast operator should be scalars.

If there is a change in size of the value, type casting on the left works best for little-endian targets; for big-endian targets, a change in size causes the compiler to issue a diagnostic.

Penalties

The compiler issues a diagnostic message “cast used as an lvalue” based on the number of penalties the cast operation incurs. The penalties are computed as follows:

- 1 penalty if the **-nx** option is specified
- 1 penalty if the target is big-endian and there is a change in size

A diagnostic is issued according to the following rules once the penalties are calculated:

- 0 penalties — no diagnostic
- 1 penalty — warning message “cast used as an lvalue”
- 2 penalties — error message “cast used as an lvalue”

Example 4-6. Penalties in lvalue Type Casting

```
int *int_ptr;
char *char_ptr;

int int_var = 0;
int char_var = 'a';

(char)int_var = char_var; /* Change in size:
    one penalty for big-endian but no penalty
    for little-endian */
(char*)int_ptr = char_ptr; /* No change in size */
```

Void Pointer Arithmetic

In embedded programming, it is often necessary to explicitly type cast a **void pointer** to a **char pointer**. This extension implicitly type casts **void pointers** to **char pointers** and is useful in occasions where there is a need to perform an explicit cast. This implicit type cast is local to C/C++ expressions.

In addition, all of the following standard pointer arithmetic operations are also available to void pointers:

- Assignment of pointers of the same type
- Adding or subtracting a pointer and an integer
- Subtracting two pointers
- Comparing two pointers
- Assigning/comparing a pointer to zero

All other pointer arithmetic operations are illegal for void pointers.

Example 4-7. Void Pointer Arithmetic

```
void *void_ptr1, *void_ptr2;
int i=0;

i = void_ptr1 - void_ptr2;
void_ptr1++;
void_ptr1--;
```

Type Information

This section describes the Microtec extension **typeof** and related operators.

typeof operator

The **typeof** operator is a compile-time operator that returns the type of an expression, type, or typedef, including type qualifiers. If an expression is given as an argument, **typeof** returns the type of the expression but does not evaluate the expression. For example, **typeof(a++)** returns the type of the expression **a++** but does not actually increment **a**.

The syntax for the **typeof** operator is as follows:

```
typeof (operand)
```

where:

operand is an expression, **type**, or **typedef**. A type can be a primitive type (such as **char** or **int**) or a user-defined type.

Types returned by **typeof** can be integer types, pointer or reference types, floating point types, or aggregates. It also returns type qualifiers (such as **const** or **unsigned**). If a function is passed to **typeof**, it returns a pointer to that function.

Comparison Operators with typeof

The comparison operators **==** and **!=** can be used with the results of the **typeof** operator. They return true if the types are identical, for **==**, or not identical, for **!=**; otherwise, they return false.

Type qualifiers (both explicit and implied) are taken into account when comparing **typeof** results. For example, **const int** and **int** are not considered identical, nor are **signed int** and **unsigned int**. However, **int** and **signed int** are considered identical. For arrays, the size of the array is not taken into account, so, for example, **int array1[5]** and **int array2[10]** would be considered identical. Finally, a **typedef** is considered identical to its base type:

```
typedef size_t unsigned int;
```

size_t would be considered identical to **unsigned int** with this declaration.

Example 4-8. Using the typeof Operator

```
if (typeof(a) == typeof(int))
    printf("a is of type int\n");
else if (typeof(a) == typeof(unsigned int))
    printf("a is of type unsigned int\n");
else if (typeof(a) == typeof(struct A))
    printf("a is of type struct A\n");
else
    printf("a is not of a known type.\n");
```

nameoftype operator

The **nameoftype** operator is similar to the **typeof** operator; however, instead of returning an actual type, it returns a string containing the name of the type:

```
printf("The type of a_var is: %s\n", nameoftype(a_var));
```

Keywords

The Microtec C/C++ compiler supports the following keywords:

- **interrupt**
- **packed**
- **unpacked**

interrupt

The **interrupt** keyword instructs the compiler to generate a special code segment which can be put into the PowerPC Exception Table. This special code segment calls a register save routine, the actual interrupt function, and a register restore routine. Details about how to use the **interrupt** keyword is described in more detail in the “[Interrupt Handlers](#)” section of the “[Embedded Environments](#)” chapter.

Example 4-9. Using a C++ Interrupt Handler

```
// This example illustrates how to declare a C++ interrupt
// handler and how to use it with C++ classes and associated
```

```

// access rules.
typedef void (*FUNC) ();
const int UPAGECNT = 20;
struct Page;

// class SegU_Info contains some information necessary during
// signal interrupt processing or handling.
//
// class SegU_Info also permits a user interrupt handler
// (void handler()) to access its private members.

class SegU_Info {
    long uaddr[UPAGECNT];
    struct Page *diskaddr[UPAGECNT];
    static int user_handler_exist;
    FUNC catcher;
public:
    friend interrupt void handler (void);
};

// a handle to SetU_Info:
class SegU_Info segu_handle;

// This flag indicates whether or not a user handler exists
int SegU_Info::user_handler_exist = 1;

// This is a very simple sample of user interrupt handler.
// The key points to illustrate are:
// - The handler takes NO argument and returns NOTHING.
// - The handler, as a friend function to class SegU_Info,
//   can access members (including private members
//   such as SegU_Info::catcher) of class SegU_Info.
// The purpose of this example is not to illustrate general
// interrupt handler algorithms, and so forth
interrupt void handler (void) {

    /* some code */

    if (SegU_Info::user_handler_exist) {
        // Clear user signal catcher
        segu_handle.catcher = (FUNC) 0;
    }
    /* some code */
}

```

This example declares an interrupt. An interrupt handler cannot be a **member** function because an interrupt handler generally does not have any arguments. An interrupt handler can be a **friend** function without arguments or a **static member** function without arguments, because neither has a this pointer (or object handle), although either can be scoped within a class like the **static member** function.

packed and unpacked

The **packed** and **unpacked** keywords are Microtec extensions to the C and C++ languages. You can use these keywords with structure (**class**, **struct**, and **union**) and **enum** declarations to

control memory requirements. Refer to the “[Internal Data Representation](#)” chapter for more information on **packed** and **unpacked** data objects.

Padding bytes are not added between members when a structure is declared **packed**. Bit-field straddling is enabled for packed classes. By default, structures are considered unpacked unless the **-KP** option is used, in which case they are considered **packed**.

A **packed enum** uses the smallest integral type that can represent all of its enumerators, while an **unpacked enum** defaults to using **signed int**. An **enum** is always **unpacked** by default unless the **-KP** option is used.

You can declare a **class** or **enum** as either **packed** or **unpacked**, but not both:

```
unpacked struct packed_struct;
packed struct packed_struct {
    // Error: Inconsistent re-declaration
    // structure definition
};
```

Declarations without explicit **packed** or **unpacked** qualifiers do not conflict with other declarations:

```
packed struct no_explicit_qualifier;
struct no_explicit_qualifier; // Not an error
struct no_explicit_qualifier { // This structure is
    // packed structure definition
};
```

During **class** definition, if none of the previous declarations specifies a **packed** or **unpacked** qualifier, the default qualifier is assumed, in which case subsequent declarations cannot specify a **packed** or **unpacked** qualifier that is different from the default:

```
// Assume the -KP option was not specified.
// This structure is unpacked by default.
struct unpacked_struct {
    // structure definition
};
packed struct unpacked_struct ps1;
// Error, unpacked_struct is unpacked !
unpacked struct unpacked_struct ps1; // No error
```

Include File Optimization

The Microtec C and C++ compilers track which files are included in the compilation. When including a file for the first time, the compiler records if the file begins with one of the following directives, ignoring comments and whitespace, and ends with a matching **#endif**:

```
#ifdef A_SYMBOL
```

or

```
#ifndef A_SYMBOL
```

Compilation then proceeds, and if the same file is included again, the compiler checks to see if *A_SYMBOL* is undefined (in the first case) or defined (in the second). In either case, the contents of the included file are not processed any further. This mechanism can be used in header files that can be included multiple times during a single compilation. *A_SYMBOL* can be any symbol; it need not match the name of the include file. Make sure that *A_SYMBOL* is not defined in any other include file.

In the unlikely case that your header file has preprocessor directives following the **#endif**, as follows:

```
#ifndef A_SYMBOL
#define A_SYMBOL
... code to be included only once ...
#endif
#warn about something
```

this mechanism fails to consider the trailing preprocessor directives on the second and later inclusions. Change the first line to the following to work around this limitation:

```
#if !defined(A_SYMBOL)
```

asm Support

Embedded applications require close and efficient control of the target architecture. To accommodate this need, the Microtec C++ compiler supports assembler statement intermixing with C++ code.

You can include any number of assembler lines by using the pseudofunction **asm**, which can also be specified as **ASM**. See the “[Inline Assembly](#)” section in the “[Embedded Environments](#)” chapter for information on the use of the **asm** pseudofunction.

Predefined Identifier **__func__**

__func__ (or **__FUNC__**) gives the name of the function in which it is placed as a literal string. **__func__** is implicitly declared as if, immediately following the opening brace of each function definition, the following declaration were made:

```
static const char __func__[] = "function-name";
```

__func__ is different from a preprocessor macro. The preprocessor has no effect on it and treats it as a normal token.

Example 4-10. **__func__** Identifier

```
#include <stdio.h>
int main()
{
```

```
    printf("This is %s\n", __func__);
    return 1;
}
```

The output of this example would be as follows:

```
This is main
```

__func__ is allowed only inside a function. The following global declaration:

```
char * testing = __func__;
```

would generate the following error:

```
...error: the reserved identifier "__func__" may only be used inside a
function
char * testing = __func__;
                  ^
```

Variadic Macros

A macro can be declared to accept a variable number of arguments. This feature allows an ellipsis (...) in macro definitions. This kind of macro is called **variadic**. When a variadic macro is invoked, all the tokens in its argument list after the last named argument, including any commas, become a single argument (containing a variable number of arguments). This sequence of tokens replaces the identifier **__VA_ARGS__** in the macro body wherever it appears.

Example 4-11. Variadic Macros

```
#define show(where , ...) fprintf(where, __VA_ARGS__)
```

If this macro is invoked as follows:

```
show(stdout, "hello");
show(stderr, "hello %s", "world");
```

then the resulting expansions will be, respectively:

```
fprintf(stdout, "hello");
fprintf(stderr, "hello %s", "world");
```

The variable argument is completely macro-expanded before it is inserted into the macro expansion, just like an ordinary argument. The C99 standard mandates that the identifier **__VA_ARGS__** can only appear in the replacement list of a variadic macro. It may not be used as a macro name, macro argument name, or within a different type of macro. For example, following macro definition:

```
#define var_macro(__VA_ARGS__) __VA_ARGS__ + 1
```

produces this error message:

```
...error: the identifier __VA_ARGS__ can only appear in the replacement
lists of variadic macros
```

```
#define var_macro(__VA_ARGS__)  __VA_ARGS__ + 1
                                ^
```

If there is a named argument in a variadic macro, then the variable argument cannot be left empty during invocation; otherwise, it can be left out.

```
extern int fun();
#define empty_var_macro(...) fun(__VAR_ARGS__)

int main()
{
    empty_var_macro();
    return 0;
}
```

Loop-Scoped Variables

In C99, variables having loop scope are allowed. The syntax of the **for** loop is changed in the expression initialization part:

```
for ( declaration and/or initialization ; condition ; update )
```

Example 4-12. Loop-Scoped Variables

```
int j;
for(int i = 0 ; i < 10 ; i++){
{
    j = i;
    ...
}
```

The variable **i** is not available outside of the body of the loop. For example, after the close of the loop, the following statement would be invalid:

```
j = i;
```

The use of loop scoped variables in nested scopes is valid, as in the following example:

```
int array[10][10];
for(int i = 0 ; i < 10 ; i++){
    int j = i;
    for(int i = 0 ; i < 10 ; i++){
        array[j][i] = 0;
    }
}
```

Compound Literals

Compound literals (also called anonymous aggregates) can be created using the notation *(type-name) { initializer-list }*. A compound literal looks like a cast containing an initializer. Its value is an object of the type specified in the cast containing the elements specified in the initializer. For example, consider the following structure with a compound literal:

```
struct foo
{
    int a;
    char b;
}structure;

structure = ((struct foo) {1, 'a'});
```

This declaration is equivalent to the following code:

```
{
    struct foo temp = {1, 'a'};
    structure = temp;
}
```

Example 4-13. Compound Literals

```
char **foo = (char *[]) { "x", "y", "z" };
```

This example creates an array initialized with a compound literal.

Compound literals for **scalar** types and **union** types are also allowed, but in these cases the compound literal is equivalent to a cast. These cannot be used to initialize **static** objects.

```
static int i = ((int){1});
```

This statement generates the following error:

```
...error: expression must have a constant value
    static int i = ((int){1});
                   ^
```

A compound literal, unlike a cast, is an lvalue. For example, the address of a compound literal can be obtained in the following manner:

```
int * i = &((int){1});
```

C++ Style Comments

C++ style comments, which start with a double slash (//) and continue until the end of the line, are a C99 feature. This feature is available in the C language as well as in C++.

inline Keyword

The **inline** keyword, used with functions, indicates them as candidates for in-lining. Inlined functions are as fast as macros. Refer to the “[Function Inlining](#)” section in the “[Optimizations](#)” chapter for more details about function in-lining.

Dispersed Statements and Declarations

Dispersed statements and declarations allow the user to place a declaration between code fragments. Normally, all declarations must be made in a function before code is begun:

```
int main() {
    int j = 0;
    j++;
    int i = 0; /* This declaration follows
                * the line of code j++; */
    i++;
    return 0;
}
```

Variable Length Arrays

Variable length arrays provide a variable which specifies their length. This replaces the requirement for a constant that makes arrays fixed in length.

```
extern int n;
int foo() {
    int vla[n]; /* Variable length array */
    return 0;
}
```

Variable length arrays are disabled when exception handling is enabled.

Bool Keyword

The `_Bool` keyword was added to the C language as part of the C99 Standard. The standard `stdbool.h` include file is available for defining the 'bool' type, as well as for defining the 'true' and 'false' constants. Variables of type `_Bool` can have one of two values: 0 or 1.

Supported GNU C/C++ Extensions

GNU Extensions

The following is a list of acceptable GNU extensions used by the C/C++ compiler:

- Attributes, introduced by the keyword `__attribute__` can be used on declarations of variables, functions, types, and fields. The `aligned`, `const`, `constructor`, `deprecated`,

destructor, format, format_arg, init_priority, malloc, mode, nocommon, noreturn, packed, pure, section, sentinel, transparent_union, unused, used, and volatile attributes are supported. An example of `__attribute__` use is as follows:

```
int myvar __attribute__ ((section ("zerovariables"))) = 0;
```

- Compound literals are accepted. GNU C mode also allows such literals to be used to initialize variables in the file scope (if they only involve constant expressions) and to initialize elements of aggregate types in brace-enclosed aggregate initializers. In addition, variable initializers are accepted in GNU C mode.

```
struct foo {int a; char b[2];};  
struct bar { int a; struct foo b;};  
struct bar a = { 1, ((struct foo) {2, '0', 32})};  
int *p = (int []){1, 2, 3};
```

- The `__extension__` keyword can precede declarations and certain expressions. It has no effect on the meaning of a program.

```
__extension__ __inline__ int f(int a) {  
    return a > 0 ? a/2 : f(__extension__ 1-a);  
}
```

- Zero-length array types (specified by `[0]`) are supported. These are complete types of size zero.

```
int zero_length_array[0];
```

- Flexible array members are accepted. In addition, the last field of a class type contains a class type whose last field is a flexible array member. In GNU C++ mode, flexible array members are treated exactly like zero-length arrays and can appear anywhere in the class type.

```
struct foo { int a; int array[]; };  
struct bar { int a; struct foo b; };
```

- The C99 `_Pragma` operator is supported. It takes a string literal as an argument and is similar to “`#pragma`,” for example, the following statements have a similar effect:

```
_Pragma("options -Qms177")  
#pragma options -Qms177
```

- The `sizeof` operator is applicable to void and function types. It evaluates to the value of one.
- The keyword `inline` is ignored (with a warning) on variable declarations and on block-extern function declarations.
- Excess aggregate initializers are ignored with a warning.

```
struct S { int a, b; };  
struct S a1 = { 1, 2, 3 }; // "3" ignored with a warning; no error  
int a2[2] = { 7, 8, 9 }; // "9" ignored with a warning; no error
```

- The `__restrict__` keyword is accepted. It is identical to the `restrict` keyword.
- Out-of-range floating point values are accepted without a diagnostic. When an IEEE floating point is being used, the “infinity” value is included.

```
float f = 1.8e308L; //error in non GNU mode but accepted in GNU mode
```

- Extended variadic macros are supported. Extended variadic macros use a slightly different syntax and allow the name of the variadic parameter to be chosen (instead of `.../__VA_ARGS__`).

```
#define debug(str, args...) fprintf(stderr, str, args)
void f() { debug("file: %s, function %s", __FILE__, __FUNC__ ); }
/* Expands to: fprintf(stderr, "file: %s, function %s", __FILE__,
__FUNC__) */
```

- Hexadecimal floating point constants are recognized. In this format, the hex introducer ‘0x’ and the exponent specifier ‘p’ or ‘P’ are mandatory. The exponent is a decimal number and indicates a power of two by which the significant part is multiplied. Unlike for floating point numbers in the decimal notation, the exponent is always required in the hexadecimal notation. Otherwise, the compiler would not be able to resolve the ambiguity of 0x1.f, for example. This could mean 1.0f or 1.9375 since ‘f’ is also the extension for floating point constants of type float.

```
float f = 0xA.0p03F; // f will be assigned 10*2^3 = 80
```

- The `\e` escape sequence is recognized and stands for the ASCII “ESC” character.
- Multiline strings are supported (GNU version < 3.3).

```
char *p = "abc
def";
```

- ASCII “NULL” characters are accepted in source files.
- Case ranges are supported. In this extension, ranges can be provided in switch cases as shown here:

```
switch(ch) {
    case 'a' ... 'z':
        capital = 0;
        break;
    case 'A' ... 'Z':
        capital = 1;
        break;
};
```

- A predefined macro can be undefined.
- Some expressions are considered to be constant expressions even though they are not considered this in standard C and C++. Examples include the following:

```
((char *)&((struct S *)0)->c[0]) - (char *)0
(int)"Hello" & 0
```

- The macro `__GNUC__` is predefined to the major version number of the emulated GNU compiler. Similarly, the macros `__GNUC_MINOR__` and `__GNUC_PATCHLEVEL__` are predefined to the corresponding minor version number and patch level. Finally, `__VERSION__` is predefined to a string describing the compiler version.
- An extern inline function that is referenced, but not defined is permitted (with a warning).
- Nonstandard casts are allowed in null pointer constants, for example, `(int)(int *)0` is considered a null pointer constant despite the pointer cast in the middle.
- Non-evaluated parts of constant expressions can contain non-constant terms as shown here:

```
int i;
int a[ 1 || i ]; // Accepted in gcc/g++ mode
```

- Casts on an lvalue that do not fall under the usual “lvalue cast” interpretation (for example, they cast to a type with a different size) are ignored, and the operand remains an lvalue. A warning is issued. (GNU version < 3.4)

```
int i;
(short)i = 0; // Accepted, cast is ignored; entire int is set
```

- GNU C also allows VLA types for fields of local structures. The compiler treats such arrays as having length zero (with a warning).

```
void f(int n) {
    struct {
        int a[n]; /* Warning: n ignored and replaced by zero */
    };
}
```

GNU C Mode

- A non-constant may appear in the initializer list of an aggregate variable with an automatic storage class.
- structs and unions without fields are accepted. They have a size zero. Class types containing only fields of size zero (zero-length bit fields, zero-length arrays, and class types of size zero) also have this size.
- In conditional expressions, the middle operand can be left out. When this is the case, the result of the expression is the value of the first operand, if that first operand is “true”.

```
x = x ?: -1; /* If x is zero, evaluates to -1; otherwise, just x */
```

- Old-style definitions with unpromoted parameter types can follow prototype declarations with those same types.

```
void f(short); // Prototype
void f() short p; {} // Old-style definition OK
```

- An unprototyped declaration can follow a prototyped declaration. After such a redeclaration, the original prototype is ignored.

```
void f(char); // Original prototype
void f(); // Unprototyped redeclaration
void f(double x) {} // Accepted in GNU C mode
// (original prototype ignored)
```

- An unprototyped declaration (that is not a definition) can have a list of parameter names which may contain duplicates.

```
void f1(i, j); // Accepted (with a warning) in GNU C mode
void f2(i, i); // Also accepted (with the same warning)
```

- An old-style parameter list can be followed by an ellipsis to indicate that the function accepts variable-length argument lists.

```
void f(x) int x; ... {} // Accepted in GNU C mode
```

- Certain “generalized lvalues” are supported. The ternary conditional operator produces an lvalue if its second and third operand are lvalues. Similarly, the comma operator produces an lvalue if its second operand is an lvalue. Some casts also preserve the lvalueness of their operand. (GNU version < 4.0) The following is an example:

```
unsigned x, y, z;
(x ? y : z) = 4; // Updates y or z depending on x
(x += y, y) = 3; // Updates y
(int)x = -1; // Updates x
```

An “?” operator can also be treated as an lvalue if its operands have types that are different, but close enough that an lvalue cast can bridge the gap (for example, int and int *). A warning is issued.

- An expression can be cast to a union type that contains a field with that expression’s type, for example:

```
union X { int i; char c; };
union X f(int i) { return (union X)i; }
```

- Implicit conversions are allowed between integral and pointer types, and between incompatible pointer types (with a warning). Implicit conversions between pointer types can drop cv qualifiers (for example, const char * to char *). (GNU version < 4.0)
- Explicit casts are ignored if they are do nothing casts, that is, if the source type is the same as the destination type (cv-qualifier differences are allowed but ignored). The result is an lvalue, if the source is an lvalue. Casts to struct and union types (which are otherwise invalid) are allowed in such cases.
- Functions with a return type void can have return expressions. A warning is issued if the return expression does not have this.

- Bit field lengths that are too large for the associated type are ignored with a warning (that is, the field becomes an ordinary field whose address can be taken). (GNU version < 3.4) The following is an example:

```
struct S {
    char c: 9; // Oversized bitfield becomes regular field of
}; // ( type char, assuming a char can hold at most 8 bits)
```

- The `_Bool` keyword is accepted.
- “?” operators with mixed void/non-void operands in the second and third operands are accepted. The result has type void.
- Extraneous specifiers in typedef declarations that do not include a declarator are ignored with a warning. (GNU version < 4.0)

```
typedef long short; // "short" ignored.
```

```
typedef long long long; // Last "long" ignored.
```

- In function definitions, local declarations can hide parameters. (GNU version < 3.4)

```
void f(int x) {
    double x; // Accepted in GNU C mode (with a warning).
}
```

GNU C++ Mode

- A special keyword `__null` expands to the same constant as the literal “0”, but is expected to be used as a null pointer constant.
- When using a GNU version earlier than 3.4, names from dependent base classes are ignored only if another name would be found by the lookup.

```
const int n = 0;
template <class T> struct B {
    static const int m = 1; static const int n = 2;
};
template <class T> struct D : B<T> {
    int f() { return m + n; } // B::m + ::n in g++ mode
};
```

- A non-static data member from a dependent base class, which would usually be ignored as previously described, is found if the lookup would have otherwise found a non-static data member of an enclosing class. (GNU version is < 3.4)

```
template <class T> struct C {
    struct A { int i; };
    struct B: public A {
        void f() {
            i = 0; // g++ uses A::i not C::i
        }
    };
    int i;
```

```
};
```

- A new operation in a template is always treated as dependent. (GNU version is ≥ 3.4)

```
template <class T> struct A {  
    void f() {  
        void *p = 0;  
        new (&p) int(0); // calls operator new declared below  
    }  
};  
void* operator new(size_t, void* p);
```

- When doing a name lookup in a base class, the injected class name of a template class is ignored.

```
namespace N {  
    template <class T> struct A {};  
}  
struct A {  
    int i;  
};  
struct B : N::A<int> {  
    B() { A x; x.i = 1; } // g++ uses ::A, not N::A  
};
```

- The injected class name is found in certain contexts in which the constructor should be found instead. (GNU version < 3.4)

```
struct A {  
    A(int) {};  
};  
A::A a(1); // no error in GNU C++ mode
```

- In a constructor definition, what should be treated as a template argument list of the constructor is instead treated as the template argument list of the enclosing class.

```
template <int ul> struct A { };  
template <> struct A<1> {  
    template<class T> A(T i, int j);  
};  
template <> A<1>::A<1>(int i, int j) { } // accepted in GNU C++ mode
```

- The macro `__GNUG__` is defined identically to `__GNUC__` (that is, the major version number of the GNU compiler version that is being emulated).
- No connection is made between declarations of identical names in different scopes even when these names are declared extern "C," for example:

```
extern "C" { void f(int); }  
namespace N {  
    extern "C" {  
        void f() {} // Warning (not error) in g++ mode  
    }  
}  
int main() { f(1); }
```

- The definition of a class template member that appears outside of the class definition may declare a non-type template parameter with a type that is different than the one used in the definition of the class template. A warning is issued. (GNU version < 3.3)

```
template <int I> struct A { void f(); };
template <unsigned int I> void A<I>::f() {}
```

- A class template may be redeclared with a non-type template parameter that has a type different from the one used in the earlier declaration. A warning is issued.

```
template <int I> class A;
template <unsigned int I> class A {};
```

- A friend declaration may refer to a member typedef. (GNU version < 3.4)

```
class A {
    class B {};
    typedef B my_b;
    friend class my_b;
};
```

- When a friend class is declared with an unqualified name, the lookup of that name is not restricted to the nearest enclosing namespace scope.

```
struct S;
namespace N {
    class C {
        friend struct S; // ::S in GNU mode, N::S in default mode
    };
}
```

- In a catch clause, an entity may be declared with the same name as the handler parameter.

```
try { }
catch(int e) {
    char e;
}
```

- A template argument list may appear following a constructor name in constructor definition that appears outside of the class definition as shown here:

```
template <class T> struct A {
    A();
};
template <class T> A<T>::A<T>() {}
```

- When using a GNU version earlier than 3.4, an incomplete type can be used for a nonstatic data member of a class template.

```
class B;
template <class T> struct A {
    B b;
};
```

- A constructor is not required to provide an initializer for every non-static const data member (however, a warning is still issued if such an initializer is missing). (GNU version < 3.4)

```
struct S {  
    int const ic;  
    S() {} // Warning only in GNU C++ mode (error otherwise).  
};
```

- A friend declaration in a class template may refer to an undeclared template.

```
template <class T> struct A {  
    friend void f<>(A<T>);  
};
```

- A function template default argument may be redeclared. A warning is issued and the default from the initial declaration is used.

```
template<class T> void f(int i = 1);  
template<class T> void f(int i = 2){}  
int main() {  
    f<void>();  
}
```

- A definition of a class template member function that appears outside of the class may specify a default argument.

```
template <class T> struct A { void f(T); };  
template <class T> void A<T>::f(T value = T() ) { }
```

- Function declarations (that are not definitions) can have duplicate parameter names.

```
void f(int i, int i); // Accepted in GNU C++ mode
```

- A template may be redeclared outside of its class or namespace. (GNU version < 3.4)

```
namespace N {  
    template< typename T > struct S {};  
}  
template< typename T > struct N::S;
```

- The injected class name of a class template can be used as a template argument.

```
template <template <class> class T> struct A {};  
template <class T> struct B {  
    A<B> a;  
};
```

- A partial specialization may be declared after an instantiation has been completed. This instantiation uses the partial specialization if it is declared earlier. A warning is issued.

```
template <class T> class X {};  
X<int*> xi;  
template <class T> class X<T*> {};
```


- The “.” or “->” operator may be used in an integral constant expression if the result is an integral or enumeration constant. (GNU version < 3.4)

```
struct A { enum { e1 = 1 }; };
A a;
int x[a.e1]; // Accepted in GNU C++ mode
int main () {
    return 0;
}
```

- Strong using-directives are supported.

```
using namespace debug __attribute__((strong));
```

- Partial specializations that are unusable because of nondeducible template parameters are accepted and ignored.

```
template<class T> struct A {class C { };};
template<class T> struct B {enum {e = 1}; };
template <class T> struct B<typename A<T>::C> {enum {e = 2}; };
int main(int argc, char **argv) {
    printf("%d\n", B<int>::e);
    printf("%d\n", B<A<int>::C>::e);
}
```

- Template parameters that are not used in the signature of a function template are not ignored for partial ordering purposes when using a GNU version is earlier than 4.1.

```
template <class S, class T> void f(T t);
template <class T> void f(T t);
int main() {
    f<int>(3); // not ambiguous when GNU version is < 4.1
}
```

Unsupported GNU Features

- The gcc built-in `<stdarg.h>` and `<varargs.h>` facilities (`__builtin_va_list`, `__builtin_va_arg`, ...).

- Specification of the “assembler name” of variables and routines using the “asm” keyword:

```
int counter __asm__("counter_v1") = 0;
```

- Register variables mapped on specific registers using the asm keyword:

```
register int i asm("r1"); // Map "i" onto register r1
```

- GNU-extended syntax to indicate how assembly operands map to C/C++ variables:

```
asm("fsinx %1,%0" : "=f"(x) : "f"(a));
// Map the output operand on "x",
// and the input operand on "a"
```

- The address of a statement label using the prefix “&&” operator, for example, void *a = &&L and transfer to the address of a label using the “goto *” statement (for example, g., goto *a)
- Built-in functions of the form __builtin_xyz (for example, __builtin_alloca)
- The __thread specifier to indicate that a variable should be placed in thread-local storage.
- Statement expressions, for example:

```
#define maxint(a,b)\
    ({int _a = (a), _b = (b); _a > _b ? _a : _b; })
int max(int a,int b) {return maxint(a,b);}
```

- If an ‘extern inline’ function is defined and used in a file, its out-of-line copy is spilled, as opposed to GNU where no out-of-line copy is spilled.
- The forward declaration of function parameters (so they can participate in variable-length array parameters)
- alias attribute
- Complex types
- Nested functions

Chapter 5

Using Libraries

This chapter discusses the libraries distributed with the MCCPPC C Compiler and CCCPPC C++ Compiler and describes how they are used on a host computer running a UNIX or UNIX-like operating system or a DOS shell under Windows. The syntax is identical for these systems unless specifically noted. Examples are provided that demonstrate how to compile, assemble, and link a sample program.

C/C++ Compiler Libraries

Table 5-1 lists the libraries provided with MCCPPC and CCCPPC. These libraries are

Table 5-1. Libraries Distributed With MCCPPC and CCCPPC

Library	Big-Endian, FPU	Big-Endian, Software Floating Point	Little-Endian, FPU	Little-Endian, Software Floating Point
C run-time libraries and supplemental libraries ¹	<i>mppchb.lib</i> <i>mppc_b.lib</i>	<i>mppcsb.lib</i> <i>mppc_b.lib</i>	<i>mppchl.lib</i> <i>mppc_l.lib</i>	<i>mppcsl.lib</i> <i>mppc_l.lib</i>
C++ run-time libraries without exception handling	<i>cppcnhb.lib</i>	<i>cppcnsb.lib</i>	<i>cppcnhl.lib</i>	<i>cppcnsl.lib</i>
C++ run-time libraries with exception handling	<i>cppcehb.lib</i>	<i>cppcesb.lib</i>	<i>cppcehl.lib</i>	<i>cppcesl.lib</i>

1. The C run-time and supplemental libraries must be used together.

collectively referred to as the CCCPPC libraries. Library names can be summarized as follows:

```
mppc{h|s|_}{b|l}.lib  
cppc{n|e|_}{h|s}{b|l}.lib
```

where:

- mppc** Specifies C libraries (see the “[Intrinsic System Functions](#)” section in this chapter for a description of all intrinsic system functions provided)
- cppc** Specifies C++ libraries and includes support for multiple inheritance I/O stream classes/objects such as **cout**, **cin**, **cerr**, and **clog**, plus C++ run-time support routines such as **new** and **delete** (memory management routines)

h	Indicates a library with hardware floating point instructions.
s	Indicates a library with software floating point calculations.
n	Indicates a library without exception handling support.
e	Indicates a library with exception handling support.
b	Indicates a big-endian library
l	Indicates a little-endian library
_ (underscore)	Indicates a supplemental C library. Each supplemental C library must be used with its corresponding C run-time library. <i>mppc_b.lib</i> is used with <i>mppchb.lib</i> and <i>mppcsb.lib</i> , and <i>mppc_l</i> is used with <i>mppchl</i> and <i>mppcsl.lib</i> .

For most cases, libraries will be selected from the “*lib*” subdirectory, but besides those indicated in the previous table, some compiler options cause the resulting object file to be incompatible with other objects. Other subdirectories contain libraries which are selected as follows:

- **lib_dp_spe:** libraries used to support the E500 (v2) core double-precision floating point instruction set (for example, option `-p8548`). Libraries in this directory use 'f' to indicate the use of special floating-point instructions. The 'h' and 's' forms of the libraries are not present.
- **lib_kq:** libraries used when compiling with the `-Kq` option (treat “double” type as 32-bit single precision).
- **lib_kq_spe:** libraries used to support the E500 core single-precision floating point instruction set (for example, options `-Kq -p5554`). Libraries in this directory use 'f' to indicate the use of special floating-point instructions. The 'h' and 's' forms of the libraries are not present.
- **lib750gx_dd11_8:** libraries used to support 750GX restrictions (`-p750gx_dd11_8`).

Library Use

If unresolved symbol references exist after linking all user-supplied input files, the linker automatically selects the necessary libraries to link with based on information provided in the object files.

If the linker detects C++ input object modules, it will also search C++ run-time libraries to resolve outstanding references. The linker will search the C++ run-time libraries before the C run-time libraries. This ordering is required to correctly resolve all library references.

C Libraries

The C libraries provide the commonly used C language functions. See the “[Library Extensions](#)” section in this chapter for a description of extended library functions provided with the Microtec C++ compiler.

UNIX Level-1 Functions

UNIX level-1 functions are nonsystem-level I/O functions generally used by C. These routines read and write to streams and perform both formatted and unformatted I/O.

UNIX Level-2 Functions

Certain primitive functions, collectively referred to as the UNIX-style system functions, cannot be adequately implemented in the C library. These functions, which are mainly I/O related, must be provided by the operating system. [Table 5-2](#) lists the UNIX-style system functions used by the C compiler library.

Table 5-2. UNIX-Style System Functions

Function Name	Description
close	Closes a specified file
creat	Creates a specified file
_exit	Terminates a program
ftell	Gets the current location in a file
lseek	Sets the current location in a file
open	Opens a specified file
read	Reads bytes from a specified file
sbrk	Allocates memory space
unlink	Unlinks a filename
write	Writes bytes to a specified file

The Microtec C and C++ compilers are designed for developing programs that run in an embedded system. If your embedded system does not use a standard operating system, you might need to replace some or all of the UNIX-style system features to successfully run programs in an embedded environment. Refer to Chapter 13, “[Embedded Environments](#)” for more information about providing UNIX-style system support for the embedded environment.

Sample implementations of each UNIX-style function are provided with the distribution files. Refer to the *Release Notes* for the location of these files.

Startup Routine

A start-up module is also included in the library. This sample routine sets up an environment before the **main** function begins execution. The startup routine serves as the program's entry point and is responsible for a number of tasks, including opening standard files, building **argv** (the argument vector), and calling the main function.

If you are developing a program that will execute in an embedded system, you might need to write a special startup routine for your particular environment. Refer to Chapter 13, “[Embedded Environments](#)” for more information on using the startup module in the embedded environment. The compiler uses the startup module in the C library by default.

Termination Routine

A sample termination routine, **exit**, is included in the library. This routine calls all static destructors to deinitialize the C++ static objects (for example, values for class members are reset, object pointers are deleted, exit messages are printed, and so forth). C++ I/O level 1+ static objects are also deinitialized. Any data left in C++ level 1+ I/O buffers are flushed.

Non-Reentrant Functions

ANSI C library functions are generally reentrant. ANSI C defines some library functions in such a way that it is impossible to implement them in a reentrant fashion. If a library function is defined so that it uses static, nonautomatic data, the function is non-reentrant.

The largest class of functions that accesses static data are the routines in *stdio.h*. In general, if reentrancy is required, these routines should be avoided. Three notable exceptions are **sprintf**, **vsprintf**, and **sscanf**, which are completely reentrant.

Functions that modify **errno** are also non-reentrant. Many of the functions declared in *math.h* modify the global variable **errno**, along with the functions **strtod**, **strtol**, **strtoul**, and others. The list of non-reentrant functions is shown in [Table 5-3](#). All other functions are reentrant.

Table 5-3. Non-Reentrant Functions

Function Name	Static Variable(s) Used
acos	errno
acosf	errno
asctime	_ctbuf[]
asin	errno
asinf	errno
atan2	errno
atan2f	errno

Table 5-3. Non-Reentrant Functions (cont.)

Function Name	Static Variable(s) Used
atexit	_atexit_stack[], _atexit_top
calloc	_membase, _avail, _badlist
clearerr	_iob ¹
cos	errno
cosf	errno
cosh	errno
coshf	errno
ctime	_ctbuf[], _xtm
exp	errno
expf	errno
fclose	_iob ¹
fflush	_iob ¹
fgetc	_iob ¹
fgetpos	errno ¹
fgets	_iob ¹
fmod	errno
fmodf	errno
fopen	_iob ¹
fprintf	_iob ¹
fputc	_iob ¹
fputs	_iob ^a
fread	_iob ¹
free	_membase, _avail, _badlist
freopen	_iob ¹
frexp	errno
frexpf	errno
fscanf	_iob ¹
fseek	_iob ¹
fsetpos	_iob ¹ , errno
ftell	errno ¹

Table 5-3. Non-Reentrant Functions (cont.)

Function Name	Static Variable(s) Used
fwrite	_iob ¹
getc	_iob ¹
getchar	_iob ¹
gets	_iob
gmtime	_xtm
ldexp	errno
ldexpf	errno
localtime	_xtm
log	errno
logf	errno
log10	errno
log10f	errno
malloc	_membase, _avail, _badlist
modf	errno
modff	errno
perror	_iob ¹
pow	errno
powf	errno
printf	_iob ¹
putc	_iob ¹
putchar	_iob ¹
puts	_iob ¹
raise	errno
rand	_randx
realloc	_membase, _avail, _badlist
rewind	_iob ² , errno
scanf	_iob ¹
setbuf	_iob
setvbuf	_iob
signal	errno

Table 5-3. Non-Reentrant Functions (cont.)

Function Name	Static Variable(s) Used
sin	errno
sinf	errno
sinh	errno
sinhf	errno
sqrt	errno
sqrtf	errno
srand	_randx
sscanf	_iob ¹
strtod	errno
strtok	_lastp
strtol	errno
strtoul	errno
tan	errno
tanf	errno
tanh	errno
tanhf	errno
tmpfile	_iob ²
tmpnam	_iob ²
ungetc	_iob ¹
vfprintf	_iob ¹
vprintf	_iob ¹

1. This library function may have an I/O buffer attached through **_iob**. An I/O buffer is a buffer that is used for buffered I/O. Any I/O stream that is unbuffered will not access the buffer.

2. This library function is currently implemented as a sub that does not modify static data. However, an actual implementation of this function, which you might encounter in UNIX, would modify the described static data.

C++ Library

The Microtec PowerPC Toolkit is distributed with an open-source C++ library (STLport version 5.1.2). This library excludes language support features and intrinsic system functions, which are implemented by Mentor Graphics and the Edison Design Group. The following sections describe the different C++ library components and header files in more detail.

Note



The Microtec C++ Library does not support wide characters in the current version.

Language Support

The language support library component includes the features described in the following sections.

Dynamic memory management

This feature defines several functions for managing dynamic storage within a program. It also defines components for reporting memory allocation errors.

This feature is defined in the `<new>` or `<new.h>` header file.

C++ Exception Handling

C++ exception handling support involves handling **try**, **catch**, and **throw** constructs and managing run-time needs of exception handling.

This feature is defined in the `<exception>` or `<exception.h>` header file.

Run-Time Type Identification

Run-Time Type Identification (RTTI) support has classes for reporting type information at run-time and classes to report type identification errors.

This feature is defined in the `<typeinfo>` or `<typeinfo.h>` header file.

Diagnostics

This feature defines several classes for detecting and reporting exception conditions, such as overflow and underflow.

It is defined in the `<stdexcept>` header file.

General Utilities

These features include several helper functions and classes that are used by other library routines. They can also be used for general C++ programs (for allocator requirements, memory management utilities, and similar routines).

These features are defined in the `<utility>`, `<functional>`, and `<memory>` header files.

Strings

The set of functions and data structures in this portion of the library facilitate manipulating strings and characters.

These features are defined in the `<string>` header file.

Localization

These library functions address localization issues, such as character classification; string manipulation; and numeric, monetary, and date/time processing.

These features are defined in the `<locale>` header file

Numerics

This library section defines functions and classes to address number manipulation. It provides complex number operations, **valarrays** (for array operations), and several other helper functions for numbers.

These features are defined in the `<complex>`, `<valarray>`, and `<numerics>` header files.

Input/Output

This library section implements C++ stream I/O. C++ streams are implemented on top of standard C I/O library routines. The standard I/O stream objects available are **cin**, **cout**, **cerr**, and **clog**.

This functionality is defined in the following header files:

- `<iosfwd>`
- `<ios>` or `<ios.h>`
- `<fstream>` or `<fstream.h>`
- `<istream>` or `<istream.h>`
- `<ostream>` or `<ostream.h>`
- `<iostream>` or `<iostream.h>`
- `<stringstream>` or `<stringstream.h>`
- `<streambuf>` or `<streambuf.h>`
- `<iomanip>` or `<iomanip.h>`

Standard Template Library

The C++ Standard Template Library (STL) defines a set of containers, iterators, and algorithms. **Containers** are used to store data, **iterators** are used to implement independent transversal of containers, and algorithms manipulate containers through iterators in order to achieve a desired task. Such tasks include sorting a container's data or reversing the data.

Containers

Containers store data. The STL includes a wide variety of containers. Various containers have trade-offs when compared to other containers. For example, a simple vector (a container type) offers constant-time insertion and removal of elements from the end, but the time to manipulate elements in the middle or front is variable; this offers relatively quick operation in the case of a large set of data. In contrast, the same functionality can be obtained with a list (another container type), but it would be slower in comparison to a vector.

The following headers define containers:

- `<deque>`
- `<queue>`
- `<vector>`
- `<set>`
- `<list>`
- `<stack>`
- `<map>`
- `<bitset>`

Container Extensions

The following headers are extensions to the ISO/IEC 14882:1998 C++ standard:

- `<hash_set>`

set is a container that is defined in the standard; **hash_set** is an extension of this standard. It allows hash-based storage of data in a set, reducing the retrieval time. Any **hash** function can be passed as an argument to be used for hashing. This header defines the **hash_multiset** container as well, which is a similar expansion of the **multiset** container declared in `<set>`. **set** and **multiset** differ in that a multiset allows duplicate values to be stored, while a set does not allow this behavior.
- `<hash_map>`

This header defines the container **hash_map**, which is an extension of the standard **map** container. It allows hash-based storage of data in a map container, reducing the retrieval time. Any **hash** function can be passed as an argument to be used for hashing. A map differs from a set in that it stores values as (**key**, **value**) pairs, while a set stores only keys.

In addition, this header defines the **hash_multimap** container as an extension of the standard **multimap** container declared in `<map>`. The difference between a multimap and a map is that a multimap allows different values to be stored against a key value.

- **<slist>**

This header defines the **slist** container, a forward-only singly-linked list version of the standard **list** container. This container offers better speed than a list and can be used in scenarios where forward-only traversal is required.

- **<rope>**

This header defines the **rope** container, an advanced version of the **string** container that offers better performance for larger strings.

Iterators

Iterators are a generalization of pointers; they are objects that point to other objects. Iterators are often used to iterate over a range of objects. If an iterator points to one element in a range, it is possible to increment it so that it points to the next element.

Iterators are central to generic programming because they are an interface between containers and algorithms. Algorithms use iterators to access elements of a container in order to perform their tasks. This makes it possible to write a generic algorithm that can operate on many different kinds of containers.

Iterators are defined in the `<iterator>` header file.

Algorithms

The algorithms section of the STL defines several algorithms that can be used to manipulate containers. The tasks that these algorithms can perform include sorting, searching, and counting.

Algorithms are defined in the `<algorithm>` header file.

Intrinsic System Functions

These functions define the virtual function scheme and construction/destruction of **static** objects. [Table 5-4](#) summarizes these routines.

Table 5-4. Intrinsic System Functions

Routine / Source File	Description
_main ..\etc\cxx\src\cxxconde.c	This routine is called before the main function is executed. It constructs static objects by calling their constructors. If C++ exception handling is enabled, this routine also initializes exception handling data structures.
_cxxfini ..\etc\cxx\src\cxxconde.c	This routine calls static destructors for static objects in order to clean up memory before exiting a program.
__pure_virtual_called ..\etc\cxx\src\pure_virt.cpp	This routine terminates a program if a pure virtual function is called.

Library Extensions

The Microtec C and C++ compilers provide a run-time library that contains many functions familiar to C programmers. Some low-level system functions are also included with Microtec C and C++ for use on target systems; however, these functions may not be applicable to embedded systems.

The Microtec C and C++ standard include file directory is described in the “[Using Command Line Options](#)” chapter. A complete listing of the standard ANSI C library functions and include files can be found in any ANSI C reference book.

C and C++ extensions listed alphabetically in this section include:

- Reentrant functions
- Non-reentrant functions
- I/O system functions

C Function Groups and Include Files

In addition to the standard ANSI include files, the Microtec C and C++ compilers also provide extensions beyond the standard ANSI C function set.

mriext.h

The include file *mriext.h* includes Microtec extensions. [Table 5-5](#) lists the functions in *mriext.h*.

Table 5-5. mriext.h Functions

Function	Definition
<code>eprintf</code>	Prints to standard error

Table 5-5. mriext.h Functions (cont.)

Function	Definition
fileno	Gets file number
ftoa	Converts floating point numbers to ASCII
getl	Reads long int
getw	Reads short int
isascii	Checks if argument is ASCII
itoa	Converts integer to ASCII
itostr	Converts int to ASCII, variable base
ltoa	Converts long to ASCII
ltostr	Converts long to ASCII, variable base
max	Returns maximum value
memccpy	Copies memory looking for a character
memclr	Clears memory bytes
min	Returns minimum value
putl	Writes long int
putw	Writes short int
swab	Swaps bytes in memory
toascii	Guarantees argument is ASCII
_tolower (<i>char_string</i>)	Is the same as tolower , except does not perform argument checking
_toupper (<i>char_string</i>)	Is the same as toupper , except does not perform argument checking
zalloc	Allocates data space dynamically and initializes to zero

The *mriext.h* file also defines the macros listed in [Table 5-6](#).

Table 5-6. mriext.h Macros

Macro	Definition
BLKSIZE	Size of I/O buffer
FALSE	Value of 0
NULLPTR	Value of NULL pointer
stdaux	Pointer to a standard auxiliary output device
stdprn	Pointer to a standard printer device

Table 5-6. mriext.h Macros (cont.)

Macro	Definition
TRUE	Value of 1

Non-Reentrant Extensions

Microtec library extensions are mostly reentrant. However, there are library extensions that are non-reentrant due to the use of static, nonautomatic data.

Refer to ANSI C books available at computer bookstores for more information on ANSI C non-reentrant functions.

[Table 5-7](#) shows the Microtec non-reentrant extensions.

Table 5-7. Non-Reentrant Extensions

Function	Static Variables Used
<code>fprintf</code>	<code>_iob</code> ¹
<code>getl</code>	<code>_iob</code> ¹
<code>getw</code>	<code>_iob</code> ¹
<code>putl</code>	<code>_iob</code> ¹
<code>putw</code>	<code>_iob</code> ¹
<code>zalloc</code>	<code>_membase</code> , <code>_avail</code> , <code>_badlist</code>

1. This extension may have an I/O buffer attached through `_iob`. An I/O buffer is a buffer used for buffered I/O. Any I/O stream that is unbuffered does not access the buffer.

I/O System Functions

[Table 5-8](#) lists the system functions that perform input/output operations. See the “[Embedded Environments](#)” chapter for more information on system functions.

Table 5-8. System Functions

Function	Definition
<code>chdir</code>	Changes working directory.
<code>close</code>	Closes a specified file.
<code>connect</code>	Connects a socket to a server.
<code>creat</code>	Creates a new file.
<code>_exit</code>	Terminates a program.
<code>getcwd</code>	Gets working directory.

Table 5-8. System Functions (cont.)

Function	Definition
lseek	Sets the current location in a file.
open	Opens a specified file.
pclose	Closes a pipe.
popen	Opens a pipe.
read	Reads from a specified file.
rename	Renames a file.
sbrk	Allocates memory space.
socket	Opens a socket.
stat	Gets information on a file.
sys_in	Gets input from original stdin.
sys_out	Puts output to original stdout.
sys_stat	Checks keyboard status.
system	Executes host system command.
time	Gets system time.
tmpnam	Generates temporary filename.
unlink	Unlinks a filename.
write	Writes to a specified file.

Each function listed in [Table 5-8](#), except as noted, is implemented as a stub that does not modify static data. An actual implementation of these functions would make the function non-reentrant and modify static variables based on your target system or kernel.

Single Precision Math Functions

In addition to the normal double precision math functions, the library also contains single-precision versions of the standard math libraries. The names are derived from the double-precision names with an ‘f’ appended. All type ‘double’ input arguments and return types are changed to ‘float’. The prototypes are in `<math.h>`. These new routines are as follows:

Table 5-9. Single-Precision Versions of the Standard Math Libraries

acosf	asinf	atan2f	atanf	cosf	coshf
expf	fabsf	fmodf	frexpf	ldexpf	log10f
logf	modff	ceilf	floorf	powf	sinf

Table 5-9. Single-Precision Versions of the Standard Math Libraries (cont.)

sinhf	sqrtf	tanf	tanhf		
-------	-------	------	-------	--	--

Library Function Definitions

The remainder of this chapter describes library functions in alphabetical order. A fixed format that includes the following subsections is provided for easy reference:

Function Name

Each section begins with the name of the library function, followed by a short description. Some functions are also defined as macros.

Class

Each function is flagged as one of three classes: a UNIX-style system function, an intrinsic system function, or a Microtec C/C++ extension.

Syntax

The syntax shows the return type of the function and the types of its formal arguments. Declarations for some of these functions are in the include file indicated. Alternatively, functions that are not implemented as macros can be declared as external functions with the appropriate return type.

Description

The description gives the meaning of the function and its return values.

Notes

This section contains information on abnormal behavior, embedded environment considerations, and other issues that can affect function or macro behavior.

_cxxfini — Calls Static Destructors to Destroy Static Objects

Class

Intrinsic System Function

Syntax

```
extern "C" {  
    void _cxxfini();  
}
```

Description

The `_cxxfini` function calls the static destructors of static objects before the program exits. A static destructor is a compiler-generated destructor used to destroy static objects for each module. All static destructors must be executed before the program exits. More specifically, static destructors must be executed before the file cleanup operations, such as flushing file buffers and closing file channels.

The `exit` function calls the `_cxxfini` function.

Notes

The `_cxxfini` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

[_main — Calls Static Constructors to Create Static Objects](#)

eprintf — Provides Formatted Print to Standard Error

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    int fprintf (const char *format, ...);
}
```

Parameters

- **format**
The format of the output.
- ...
The arguments to be formatted.

Description

The `eprintf` function provides formatted output to the C standard error file. The format and results are the same as for **printf**. When `eprintf` is successful, it returns the number of characters transmitted. Further information on the `printf` function can be found in ANSI C books available at bookstores.

eprintf returns End-Of-File (**EOF**) if an output error is encountered.

Notes

The function `eprintf` modifies the static array **_iob**; no other static variables are modified.

fileno — Gets File Descriptor

Class

Microtec C/C++ Extension (implemented as a macro)

Syntax

```
#include <mriext.h>
extern "C" {
    char fileno (FILE * stream);
}
```

Parameters

- **stream**

The stream for the requested file descriptor.

Description

The **fileno** macro returns the file descriptor associated with *stream*.

Notes

The macro yields undefined results if *stream* does not point to an open file.

ftoa — Converts Floating Point Number to ASCII String

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    int ftoa (double value, char *str, int format, int prec);
}
```

Parameters

- **value**
The floating point number to be converted.
- **str**
The string where the output will be stored.
- **format**
Must be one of **f**, **e**, **E**, **g**, or **G**. These characters are defined in the same way as the format string for the **printf** function.
- **prec**
Identical to the *digits* parameter in **printf**.

Description

The **ftoa** function converts the floating point number *value* to a **NULL**-terminated ASCII string with a specified format and precision. The output is placed in the area pointed to by *str*. No leading blanks or zeros are produced.

The conversion format character *format* must be **f**, **e**, **E**, **g**, or **G**. The conversion format character and the precision *prec* are the same as those defined for the **printf** function. (The parameter *prec* is the same as the *digits* parameter in **printf**.) The **printf** function calls **ftoa** to convert floating point numbers.

The **ftoa** function returns the number of characters placed in the output string, excluding the **NULL** terminator.

Notes

The behavior is unpredictable if an invalid floating point number is passed to the **ftoa** function.

getl — Reads a Long Integer From a Stream

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    long getl (FILE * stream);
}
```

Parameters

- **stream**

The stream from which to read.

Description

The function **getl** reads a **long** integer from *stream* and returns it. **getl** returns **EOF** at the end of *stream*. Any **long** integer read from *stream* must have been written with a **putl** function.

Notes

getl yields unpredictable results if *stream* does not point to an open file.

This function can modify the static array **_iob**; no other static data is modified.

getw — Reads a Short Integer From a Stream

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    int getw (FILE *stream);
}
```

Parameters

- **stream**

The stream from which to read.

Description

The function **getw** reads a **short int** from *stream* and returns it as an **int**. **getw** returns **EOF** at the end of *stream*.

Any short integer read from *stream* must have been written with a **putw** function.

Notes

The function **getw** yields unpredictable results if *stream* does not point to an open file.

The function **getw** can modify the static array **_iob**; no other static data is modified.

isascii — Tests for an ASCII Character

Class

Microtec C/C++ Extension (implemented as a function and as a macro)

Syntax

```
#include <mriext.h>
extern "C" {
    int isascii (int c);
}
```

Parameters

- **c**

The character to be tested.

Description

The `isascii` function or macro returns a nonzero number if the character in `c` is an ASCII character. It returns 0 otherwise. ASCII characters range in value from 0 to 127.

The macro is invoked by default. You must do one of the following if you want to call the function:

- Enclose the name in parentheses as follows:

```
(isascii)(c);
```

- Undefine the **isascii** macro with **#undef** before using it, as follows:

```
#undef isascii
```

itoa — Converts Integer to ASCII String

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    int itoa (int i, char *cp);
}
```

Parameters

- **i**
The integer to be converted.
- **cp**
The string where the output will be stored.

Description

The itoa function converts the signed integer *i* to a **NULL**-terminated ASCII string and places it in the area pointed to by *cp*. The pointer *cp* must point to a string.

The itoa function returns the number of characters placed in the output string, excluding the **NULL** terminator.

itostr — Converts Unsigned Integer to ASCII String

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    int itostr (unsigned u, char *cp, int base);
}
```

Parameters

- **i**
The integer to be converted.
- **cp**
The string where the output will be stored.
- **base**
The numerical base for the output.

Description

The `itostr` function converts the **unsigned** integer *u* to a **NULL**-terminated ASCII string. The base number is specified by *base*, and the output is placed in the area pointed to by *cp*. The pointer *cp* must point to a string, and the specified base must be between 2 and 36 (otherwise, base 10 is assumed).

The `itostr` function returns the number of characters placed in the output string, excluding the **NULL** terminator.

ltoa — Converts Long Integer to ASCII String

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    int ltoa (long value, char *cp);
}
```

Parameters

- **value**
The integer to be converted.
- **cp**
The string where the output will be stored.

Description

The ltoa function converts the **long signed** integer *value* to a **NULL**-terminated ASCII string and places it in the area pointed to by *cp*. The pointer *cp* must point to a string.

The ltoa function returns the number of characters placed in the output string, excluding the **NULL** terminator.

ltostr — Converts Unsigned Long Integer to ASCII String

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    int ltostr (unsigned long ul, char *cp, int base);
}
```

Parameters

- **ul**
The integer to be converted.
- **cp**
The string where the output will be stored.
- **base**
The numerical base for the output.

Description

The ltostr function converts the **unsigned long** integer *ul* to a **NULL**-terminated ASCII string. The base number is specified by *base*, and the output is placed in the area pointed to by *cp*. The pointer *cp* must point to a string, and the specified base must be between 2 and 36 (otherwise, base 10 is assumed).

The ltostr function returns the number of characters placed in the output string, excluding the **NULL** terminator.

_main — Calls Static Constructors to Create Static Objects

Class

Intrinsic System Function

Syntax

```
extern "C" {  
    void _main();  
}
```

Description

The `_main` function constructs static objects before the first statement of the main function is executed. A static constructor is a compiler-generated constructor to initialize static objects on a per-module basis.

The main function calls the `_main` function.

Notes

The `_main` function follows the C linkage name rules, which implies that no function prototype information is encoded in its linkage name.

See Also

[_cxxfini — Calls Static Destructors to Destroy Static Objects](#)

max — Returns the Greater of Two Values

Class

Microtec C/C++ Extension (implemented as a macro)

Syntax

```
#include <mriext.h>
max (a, b);
```

Parameters

- **a, b**

The integers, floating point numbers, or pointers to be compared.

Description

The macro **max** returns the larger of *a* and *b*. *a* and *b* can be any valid C expression yielding an integral, floating point, or pointer type. **max** performs any type promotion necessary to make *a* and *b* the same type and returns that type. If either argument is a pointer type, both arguments must be pointers to the same type.

Notes

Both *a* and *b* are evaluated more than once.

memccpy — Copies Characters in Memory

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    char *memccpy (char *s1, const char *s2, int c, size_t i);
}
```

Parameters

- **s1**
The destination for the copied data.
- **s2**
The source for the copied data.
- **c**
A termination character for the memory copy.
- **i**
The maximum number of characters to be copied.

Description

The memccpy function copies characters from memory area *s2* into *s1*, stopping after the first occurrence of character *c* has been copied or after *i* characters have been copied, whichever comes first. This function returns a pointer to the character after the copy of *c* in *s1*.

The memccpy function returns a null pointer if *c* was not found in the first *i* characters of *s2*.

memclr — Clears Memory Bytes

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    char *memclr (char *cp, size_t i);
}
```

Parameters

- **cp**
The memory area to be cleared.
- **i**
The number of bytes to be cleared.

Description

The memclr function sets the first *i* bytes in the memory area *cp* to zero. The memclr function returns *cp*.

min — Returns the Lesser of Two Values

Class

Microtec C/C++ Extension (implemented as a macro)

Syntax

```
#include <mriext.h>
min (a, b);
```

Parameters

- **a, b**

The integers, floating point numbers, or pointers to be compared.

Description

The macro **min** returns the lesser of *a* and *b*. *a* and *b* can be any valid C expression. *a* and *b* can be any integral, floating point, or pointer type. **min** performs any type promotion necessary to make *a* and *b* the same type and returns that type. If either argument is a pointer type, both arguments must be pointers to the same type.

Notes

Both *a* and *b* are evaluated more than once.

open — Opens a Specified File

Class

UNIX-Style System Function

Syntax

```
extern "C" {  
    int open (char *filename, int mode);  
}
```

Parameters

- **filename**
The file to be opened.
- **mode**
The mode for the file to be opened. Refer to [Table 5-10](#) for details about the specific modes that can be used.

Description

The **open** function opens an existing file for reading, writing, or updating. The **NULL**-terminated string *filename* must correspond to a valid filename on the target operating system.

The function opens the file named by *filename* in the mode indicated by *mode*. The values for *mode* and the corresponding symbols are shown in [Table 5-10](#).

Table 5-10. Mode Values for the open Function

Mode Value	Symbol	Description
0x0000	O_RDONLY	Opens for read only
0x0001	O_WRONLY	Opens for write only
0x0002	O_RDWR	Opens for read and write
0x0008	O_APPEND	Performs writes at EOF
0x0200	O_CREAT	Creates a file
0x0400	O_TRUNC	Truncates the file
0x4000	O_FORM	Identifies a text file
0x8000	O_BINARY	Identifies a binary file

Notes

The function **open** is implemented as a stub that returns 0. Using this function can generate a link-time error message indicating that the **_WARNING_open_stub_used** symbol is unresolved. If your program calls the **open** stub, it writes the following message:

WARNING - open (stub routine) called

to **stderr**. You must provide your own **open** function for your environment. See the “[Embedded Environments](#)” chapter in this manual for more information.

The function **open** can modify the static array **_iob**; no other static data is modified.

The **open** function should return -1 if the file does not exist, if the file cannot be read or written, or if too many files are open.

__pure_virtual_function_called — Handles Abnormal Conditions With Pure Virtual Functions

Class

Intrinsic System Function

Syntax

```
extern "C" {
void abort();
void __pure_virtual_function_called();
}
```

Description

The `__pure_virtual_function_called` function calls the `abort` function to terminate execution. It is invoked by the C++ run-time support of virtual functions if the program calls a pure virtual function in an abstract class.

Notes

A pure virtual function can be called in a constructor.

Example

```
class Abstract {
public:
    virtual void pure_vf() = 0;
    void f1(int n) {
        if (n>1)
            pure_vf();
    }
    Abstract(int n) {
        f1(n);
    }
};

class B: public Abstract {
public:
    virtual void f2() { }
    B(int n): Abstract(n) { }
    // B(n) is okay as long as n<=1
    // B(2) calls A::pure_vf() indirectly
};
```

If the function `Abstract::pure_vf` is called, the run-time C++ execution environment calls `__pure_virtual_function_called` to abort the execution. If the program is not aborted when a pure virtual function is called, a custom function `__pure_virtual_function_called` or `abort` function can be supplied. For example, a custom function might print a warning string and the program location indicating that a pure virtual function was called by the run-time C++ execution environment.

putl — Writes a Long Integer to a Stream

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    long putl (long l, FILE *stream);
}
```

Parameters

- ***l***
The integer to be written.
- ***stream***
The stream to which the integer is to be written.

Description

The function `putl` writes a **long** integer to *stream*. `putl` returns *l*.

Notes

`putl` yields unpredictable results if *stream* does not point to an open file.

This function can modify the static array `_iob`; no other static data is modified.

putw — Writes a Short Integer to a Stream

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    int putw (int w, FILE *stream);
}
```

Parameters

- **w**
The integer to be written.
- **stream**
The stream to which the integer is to be written.

Description

The function `putw` writes a short integer to *stream*. `putw` returns *w*.

Notes

`putw` yields unpredictable results if *stream* does not point to an open file.

This function can modify the static array `_iob`; no other static data is modified.

sbrk — Allocates Memory Space

Class

UNIX-Style System Function

Syntax

```
extern "C" {  
    void *sbrk (int size);  
}
```

Parameters

- **size**
The amount of space to be allocated.

Description

The sbrk function allocates additional memory from the system. A new block of memory, at least *size* bytes, is allocated, and a pointer to the start of this memory block is returned.

The sbrk function returns a pointer to the starting address of the memory block or -1 if the requested amount of memory cannot be allocated.

Notes

See the “[Embedded Environments](#)” chapter for more information on system functions.

swab — Swaps Odd and Even Bytes in Memory

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    void swab (char *source, char *dest, int count);
}
```

Parameters

- **source**
The source for the copied data.
- **dest**
The destination for the copied data.
- **count**
The number of bytes to be moved.

Description

The `swab` function moves data from *source* to *dest*, swapping odd and even bytes in the process. The parameter *count* should be even. If *count* is odd, the last byte of *source* is not moved.

Notes

The behavior is undefined if *source* and *dest* memory spaces overlap.

toascii — Converts a Byte to ASCII Format

Class

Microtec C/C++ Extension (implemented as a function and as a macro)

Syntax

```
#include <mriext.h>
extern "C" {
    int toascii (int c);
}
```

Parameters

- **c**

The byte to be converted.

Description

The `toascii` function converts the value of its argument *c* to an ASCII character by truncating all but the lower order 7 bits. The argument *c* is not modified. `toascii` returns a value in the range 0 to 127.

The macro is invoked by default. To call the function, do one of the following:

- Enclose the name in parentheses as follows:

```
(toascii)(c);
```

- Undefine the **toascii** macro with **#undef** before using it, as follows:

```
#undef toascii
```

_tolower — Converts Characters to Lowercase Without Argument Checking

Class

Microtec C/C++ Extension (implemented as a function and as a macro)

Syntax

```
#include <mriext.h>
extern "C" {
    int _tolower (int c);
}
```

Parameters

- **c**

The character to be converted.

Description

The `_tolower` function converts the value of its argument `c` to a lowercase letter. The conversion is performed whether or not `c` is an upper-case letter. The argument `c` is not modified.

The macro is invoked by default. You must do one of the following if you want to call the function:

- Enclose the name in parentheses as follows:

```
(_tolower) (c);
```

- Undefine the **_tolower** macro with **#undef** before using it, as follows:

```
#undef _tolower
```

Notes

The result is undefined if `c` is not an uppercase letter.

_toupper — Converts Characters to Uppercase Without Argument Checking

Class

Microtec C/C++ Extension (implemented as a function and as a macro)

Syntax

```
#include <mriext.h>
extern "C" {
    int _toupper (int c);
}
```

Parameters

- **c**

The character to be converted.

Description

The `_toupper` function converts the value of its argument *c* to an uppercase letter. The conversion is performed whether or not *c* is a lowercase letter. The argument *c* is not modified.

The macro is invoked by default. Do one of the following to call the function:

- Enclose the name in parentheses as follows:

```
(_toupper) (c);
```

- Undefine the **_toupper** macro with **#undef** before using it, as follows:

```
#undef _toupper
```

Notes

The result is undefined if *c* is not a lowercase letter.

unlink — Unlinks a Filename

Class

UNIX-Style System Function

Syntax

```
extern "C" {  
    int unlink (char *filename);  
}
```

Parameters

- **filename**

The file to be unlinked.

Description

The **unlink** function removes the ability to access the file named by *filename*. Generally, this is accomplished by removing the directory entry for *filename*. The file is deleted if no other links or directory entries for the file exist.

Notes

The function unlink is implemented simply as a stub that returns 0. Using this function can generate a link-time error message indicating that the **_WARNING_unlink_stub_used** symbol is unresolved. If your program calls the **unlink** stub, it writes the message:

```
WARNING - unlink (stub routine) called
```

to **stderr**. You must provide your own unlink function for your environment. See the “[Embedded Environments](#)” chapter for more information.

The unlink function should return -1 if the operation fails.

write — Writes Bytes to a Specified File

Class

UNIX-Style System Function

Syntax

```
extern "C" {  
    int write (int fd, char *buffer, int nbyte);  
}
```

Parameters

- **fd**
The descriptor associated with the file to which to be written.
- **buffer**
The data to be written to the file.
- **nbyte**
The number of bytes written to the file.

Description

The write function writes *nbyte* bytes from a buffer to the file associated with the file descriptor *fd*. The starting buffer address is specified by *buffer*. The file descriptor *fd* is returned by **open** or **creat**. Up to 64K bytes may be written, starting at the current file position.

The **write** function returns the number of bytes actually written or a -1 if an error occurs.

Notes

See the “[Embedded Environments](#)” chapter for more information on system functions.

zalloc — Allocates Data Space Dynamically

Class

Microtec C/C++ Extension

Syntax

```
#include <mriext.h>
extern "C" {
    void *zalloc (size_t size);
}
```

Parameters

- **size**
The number of bytes to be allocated.

Description

The **zalloc** function allocates a new area of memory for program use and returns a pointer to that area. The size of the area allocated is equal to *size* bytes. The **zalloc** function sets the area to zeros. The space allocated is guaranteed to start on a boundary that is suitable for aligning any type.

If the requested amount of space cannot be allocated, **zalloc** returns a null pointer. Allocations are provided from a data structure called the heap, which is located in the **stack** section.

Notes

The function **zalloc** can modify the static structure **_membase** or the static variables **_avail** or **_badlist**; no other static data is modified.

Chapter 6

C Library Customizer

This chapter describes the process of customizing libraries using the Microtec C Library Customizer. A library is a set of objects, derived from library sources generated by a compilation set by the parameters of a configuration. The Microtec C compiler is distributed with a set of libraries, providing a number of general configurations for embedded systems applications.

A library can be compiled in many different ways and may contain parts that you do not want to include (for example, floating point components). The Microtec C Library Customizer lets you build your own custom libraries by altering an Microtec general distribution library to suit your application.

The C Library Customizer is actually the Microtec C compiler plus certain scripts (for this chapter, the term “script” refers to UNIX shell scripts or Windows batch files).

To customize a library, copy one of the configuration files, modify it to fit your needs, and then execute a simple script. You can validate such a library by running a test suite provided with the compiler.

When to Create a Custom Library

There are a variety of reasons to build a custom library. For example, a custom library may be required in order to:

- Eliminate the linking of the floating point package when **printf** is used (enable the definition of the **EXCLUDE_FORMAT_IO_FLOAT_FMT** preprocessor symbol)
- Debug the library by using the **-g** option
- Compile to take full advantage of a specific target
- Save space by excluding selected routines
- Rename sections to segregate library variables from program variables by using the **-Nx** options

Directories in the C Library Customizer

Under UNIX, the following directories containing the C Library Customizer will be installed along with the Microtec C compiler:

- **cmd** — Scripts
- **src** — Source Files
- **test** — Test Files
- **config** — Configuration Files
- **lib** — Created Library Files
- **log** — Log Files from Library Builds

These directories will be write protected. You will need to copy the directories into your own work area before you can proceed with the building of libraries. Please contact your system administrator to obtain the installation location of the C Library Customizer.

Under Windows, the C Library Customizer will be available in the directory *install_dir\etc\rtl*, where *install_dir* is the installation path of the compiler. However, it is recommended that you make a working copy of the C Library Customizer in the directory in which custom libraries will be built.

All scripts are designed to run with the *cmd* directory as the default directory and they assume the given directory structure. The *cmd* directory is the location where you will probably do most of your work. The *config* directory contains all the Microtec general distribution configuration files. The *config* directory is where you will place your customized configuration file.

Windows System Requirements

The C Library Customizer build and test scripts also make frequent use of environment variables. For that reason, it is recommended that you specify an environment size of at least 2048 bytes. The following is an example of how you can specify this environment size by placing a command in your *config.sys* file:

```
shell=c:\command.com /e:2048 /p
```

The actual path to your *command.com* file may vary for your system. Refer to your Windows documentation for more information about the **shell** command.

Using the C Library Customizer

Perform the following steps to build a customized library using the C Library Customizer:

1. Create a custom configuration file.
2. Build the library using the **bld_lib** script.
3. Test the custom library using the **test_lib** and **test_one** scripts.

Step 1: Creating a Custom Configuration File

The first step in generating a custom library is to generate a configuration file defining that library. A configuration file is a text file that you can edit using your personal text editor. The configuration file acts as a header file that is read by the compiler and contains compiler options in the form of **#pragma options** directives. The argument to the **#pragma options** directives are actually compiler command line options (see Chapter 4, “C/C++ Extensions” for more information on the **#pragma options** directive). The configuration file defines how your custom library will be built.

The simplest method to create a new configuration file is to identify which existing configuration file is closest to your requirements and to make a copy, assigning the copied configuration file a new name. This name will also be the label of the new library.

Note



Avoid giving the new library a name with more than eight characters when using Windows.

Perform the following steps to generate a configuration file:

1. In the *config* directory, select the general distribution configuration file that most closely matches the library you are going to build. Select *a_type.h* to start with a configuration file which only defines defaults.
2. Copy the existing configuration file to a new file. The base name of the configuration file must be identical to the base name of the library you are building. If the library is to be named *library.lib*, then the configuration file must be named *library.h*.
3. Edit your new configuration file to reflect your choice of compiler options.

There are several preprocessor symbols that may be defined to reconfigure the libraries. These symbols are also defined using the **#pragma options** directive. Currently, there are options in the configuration files that will reconfigure the formatted **printf** and **scanf** I/O routines using compiler preprocessor symbols. Enabling these symbols allows you to remove the features of the routines you are not currently using.

Table 6-1 shows the preprocessor symbols used to customize I/O routines. These preprocessor symbols will disable options, flags, and formats for the **printf** and **scanf** library functions.

Table 6-1. C Library Customizer Preprocessor Symbols

Preprocessor Symbol	Description
Options	
EXCLUDE_FORMAT_IO_h_OPT	Disables the h option (indicates I/O object is a signed or unsigned short int)

Table 6-1. C Library Customizer Preprocessor Symbols (cont.)

Preprocessor Symbol	Description
EXCLUDE_FORMAT_IO_I_OPT	Disables the I option (indicates I/O object is signed or unsigned long int)
EXCLUDE_FORMAT_IO_LL_OPT	Disables the ll option (indicates I/O object is signed or unsigned long long int)
EXCLUDE_FORMAT_IO_L_OPT	Disables the L option (indicates signed or unsigned long double)
EXCLUDE_FORMAT_IO_ASSGN_SUPP	Disables the * option (allow variable precision and field width specifications) in input (scanf) functions
EXCLUDE_FORMAT_IO_STAR_OPT	Disables the * option (allow variable precision and field width specifications) in output (printf) functions
Flags	
EXCLUDE_FORMAT_IO_MINUS_FLAG	Disables the - flag (left justify)
EXCLUDE_FORMAT_IO_PLUS_FLAG	Disables the + flag (right justify)
EXCLUDE_FORMAT_IO_SPACE_FLAG	Disables the white space flag (causes a space to be prefixed if no + or - is indicated)
EXCLUDE_FORMAT_IO_SHARP_FLAG	Disables the # flag (output is to be printed in an alternate form)
EXCLUDE_FORMAT_IO_ZERO_FLAG	Disables the 0 flag (pads the field width with zeroes)
Formats	
EXCLUDE_FORMAT_IO_BRAKT_FMT	Disables the [format (convert character sequences)
EXCLUDE_FORMAT_IO_CHAR_FMT	Disables the c format (convert individual characters)
EXCLUDE_FORMAT_IO_DEC_FMT	Disables the d format (convert signed integer)
EXCLUDE_FORMAT_IO_INT_FMT	Disables the i format (convert signed integer)
EXCLUDE_FORMAT_IO_FLOAT_FMT ¹	Disables the f format (convert floating point numbers)
EXCLUDE_FORMAT_IO_NUMB_FMT	Disables the n format (convert number of characters read or written)
EXCLUDE_FORMAT_IO_OCT_FMT	Disables the o format (convert octal number)
EXCLUDE_FORMAT_IO_PNT_FMT	Disables the p format (convert pointers)

Table 6-1. C Library Customizer Preprocessor Symbols (cont.)

Preprocessor Symbol	Description
EXCLUDE_FORMAT_IO_STR_FMT	Disables the s format (interpret argument as a string)
EXCLUDE_FORMAT_IO_UNF_FMT	Disables the u format (convert unsigned number)
EXCLUDE_FORMAT_IO_HEX_FMT	Disables the x format (convert hexadecimal number)

1. If you are not using floating point components in your program, you may enable the preprocessor symbol **EXCLUDE_FORMAT_IO_FLOAT_FMT**, which will automatically disable **%f** processing in both the **scanf** and **printf** functions. When you disable **%f** processing, the floating point library will not be linked in and a much smaller code size will result.

By setting the configuration file's preprocessor symbols efficiently, you can trim the formatted I/O routines down to minimal size while retaining necessary functionality.

Once you have created your configuration file, save it and move to the **cmd** directory.

Step 2: Building a Custom Library

To build a custom library, make the **cmd** directory the default directory. Execute the **bld_lib** script to create the library. The **bld_lib** script creates the library based on the contents of the configuration file.

The following command syntax invokes the library build script:

```
bld_lib configuration_file
```

where:

bld_lib

Specifies the name of the library build script.

configuration_file

Specifies the name of the configuration file. The configuration file must be stated without a filename extension.

The script **bld_lib** builds a custom library and places it in the *lib* directory. If the name of the configuration file was *name.h*, then the name of the library will be *name.lib*. A log file for the library build operation is created and placed in the *log* directory with the filename *name.log*.

Once the **bld_lib** operation is complete, the library is ready for testing.

Step 3: Testing a Custom Library

Once built, a library must be tested. Two scripts are provided to help with this process, **test_lib** and **test_one**. The **test_lib** script runs all available test routines. The **test_one** script, on the other hand, runs an individual test.

Note



Running the test scripts while the build script is running is not recommended, because it may cause compilation errors.

```
test_lib configuration_file
[ -cmdfile cmd_file ]
[ -output output_file ]
[ -testsrcfile src_file ... ]

test_one configuration_file test_file
[ -cmdfile cmd_file ]
[ -output output_file ]
[ -testsrcfile src_file ... ]
```

test_lib

Specifies the name of the test script which runs a complete set of tests.

test_one

Specifies the name of the test script which runs an individual test.

configuration_file

Specifies the filename of the configuration file. The configuration file must be declared without a filename extension.

test_file

Specifies the name of an individual test file. This file must be located in the *test* directory.

-cmdfile *cmd_file*

Assigns *cmd_file* as the linker command file.

-output *output_file*

Writes the results of the **test_lib** and **test_one** build operation to *output_file* (Windows only).

-testsrcfile *src_file*

Compiles and links *src_file* before the customized library. The file *src_file* will always be compiled with the full debugging option (**-g**) enabled. The file *src_file* must be located in the **src** directory.

Upon completion, executable files generated by the compiler are placed in the *test* directory along with the output files generated by the test routines. These output files are named *test_file.out*.

Assessing Test Results

Test script output has the following format:

- An individual test begins with the line:

```
TESTING: test_name LIBRARY: library_name
```

where *test_name* is the name of the test file being executed, and *library_name* is the name of the library being tested. Compiler warnings and errors will appear after this line.

- The results of the tests appear between two lines of asterisks (*). If the test was successful, the following message is issued:

```
This test has completed with no errors.
```

If any other message appears or if there is any other output along with this message, the test has failed.

Note



If there is no output between the lines of asterisks, the test has failed.

- Under UNIX, the **test_lib** script will print a summary of the test results. It will either indicate that the test suite was successful, or it will indicate that the test suite has failed and list those tests which failed.
- Under Windows, **test_lib** will not generate a summary; the user must examine the results of each individual test.

The custom library is ready for use once the **test_lib** script has completed with no errors.

Debugging Custom Libraries

If **test_lib** indicates that there is a problem with the custom libraries, you can debug these libraries. If the custom libraries have been built with debugging information enabled, you can bring up your debugger and step through the particular test that has failed. You can step easily through the library routines since the library was built with debugging information enabled.

On the other hand, if the custom library was built without debugging information enabled, you will not be able to step easily through the library routines. In this situation, consider performing one of the following actions:

- Rebuild the library with debugging information enabled. This method is perhaps the simplest, but it does take some time to rebuild the library.
- Use the **test_one** script to recompile the test routine and those library routines where you think there might be a problem. Specify those library routines as arguments to the **-testsrcfile** option. The routines will be recompiled with debugging information enabled, and they will be linked in ahead of your custom library. Once this is done, you can step easily through the library routines with the debugger.

Once the problem has been located and corrected, rerun the individual test using the **test_one** script. If the results of the individual test indicate that the problem has been fixed, rerun the entire test suite using the **test_lib** script. When the **test_lib** script indicates that the custom library is passing all tests, the library is ready for use in your application.

Using a Custom Library

The following sections describe the use of libraries after customization through the C Library Customizer.

Using Custom Libraries

To use a new custom library, link it as you would with any other objects (or library). However, custom libraries must precede any general distribution libraries during the link.

If a compilation driver is available, you can specify the custom library in the command line, as in the following example:

```
mcctar -o prog mod1.c mod2.c mod3.s mod4.o iBUILTIt.lib
```

In this example, *tar* is an abbreviation for your target processor. The custom library will be passed to the linker before the general distribution libraries supplied by the driver.

If the custom library supersedes the general distribution library, you may want to use the driver to manually include the custom library name in the linker command file. You can specify the custom library by using the **-elinker_commandfile** driver command line option, as in the following example:

```
mcctar mod1.c -eMyCmds
```

In this mode, the driver does not supply the Microtec general distribution libraries, which may result in a faster linking. You may want to examine the sample linker command file provided by Mentor Graphics.

Chapter 7

Optimizations

Optimizations are techniques employed by compilers to improve the object code produced. The improvements include reduced program code size and faster execution speed. The extent of improvement depends on the content of an individual program, its coding style, and the compiler's ability to recognize and optimize certain constructs.

The Microtec C and C++ compilers perform many commonly known optimizations, which are described in this chapter. Some of these optimizations are applied before, some during, and some after generating code for a C++ expression. Most optimizations are independent of the target processor, but some are specific to the microprocessor and the calling conventions.

General Optimizations

The Microtec C and C++ compilers perform several optimizations before generating code for a C or C++ program. These optimizations usually involve constant expressions or expressions with known results that can be evaluated at compile time. Programmers seldom write explicit expressions with only constants, but it is not uncommon for them to write an expression with defined constants.

Algebraic Simplification

These optimizations replace an expression with an equivalent but simplified expression.

Example 7-1. Algebraic Simplification

<code>a * 1</code>	<code>==></code>	<code>a</code>
<code>b - 0</code>	<code>==></code>	<code>b</code>
<code>c == c</code>	<code>==></code>	<code>1</code>
<code>x -= x</code>	<code>==></code>	<code>0</code>

Redundant Code Elimination

Code sequences that produce no real side effect are considered redundant and can be eliminated. They usually evolve as the result of other optimizations. This optimization is not performed if the variable is declared as **volatile**.

Example 7-2. Redundant Code Elimination

<code>a = a + 0</code>	<code>==></code>	No code generated
<code>b + c</code>	<code>==></code>	No code generated
<code>x &= x</code>	<code>==></code>	No code generated

Strength Reduction

These optimizations replace operations with equivalent but less expensive (reduced strength) operations.

Example 7-3. Strength Reduction

<code>a * 128</code>	<code>==></code>	<code>a << 7</code>
<code>(unsigned) b / 16</code>	<code>==></code>	<code>b >> 4</code>
<code>(unsigned) c % 32</code>	<code>==></code>	<code>c & 31</code>

Global Optimizations

The Microtec C and C++ compilers perform several optimizations that make programs smaller and faster by applying global data flow and control flow analysis on the entire function.

Dead Code Elimination

This optimization eliminates unreachable code (“dead code”). Unreachable code occurs through user error or conditional compilation techniques that exploit the preprocessor. Unreachable code sequences are usually detected after performing jump optimizations. Unlabeled instructions immediately following an unconditional jump can be removed. Conditional expressions that can be reduced to constants also produce unreachable code.

Example 7-4. Dead Code Elimination

```

#define TRUE 1
#define FALSE 0
#define DEBUG 0

while (a)
{
    test1();
    if (b)
        continue;
    else
        break;
    test2();
}                                     ==> No code generated

if (DEBUG)                           ==> No code generated
    printf("debug1 \n");              ==> No code generated

while (FALSE)
{
    test1();                         ==> No code generated
    test2();                         ==> No code generated
}
return (b);

    goto lab;
    i1 = 3;

    i2 = 4;                           ==> No code generated
    i3 = 5;

lab:
    i4 = 3;

a = ( 3 > 0 ) ? xx (6) : yy (7);      ==> a = xx(6);
if ( a != a ) b = 5;                 ==> No code generated

```

If a breakpoint is set on the line containing the statement **i2=4** in this example, the actual breakpoint is set on the line containing the statement **i4=3**. Then, whenever a jump is made to **lab**, the actual breakpoint stops the program and causes confusion.

A more typical example of dead code removal applies when the compiler determines that the condition of an **if** statement equates to a constant, which occurs if every operand is a constant, or a variable has been assigned to a constant value.

Example 7-5. Dead Code Elimination With an if Statement

```
#define control 0
int auxcontrol;
...
auxcontrol 0 ;

if (control | auxcontrol)           ==> No code generated
{
    i1 = 3;                         ==> No code generated
    i2 = 4;                         ==> No code generated
}
else                                ==> No code generated
    i3 = 5;
```

The preprocessor symbol **control** is replaced by the text **0** and is interpreted as the constant zero. The compiler determines that the integer variable **auxcontrol** still retains the value of zero at the beginning of the **if** statement and uses zero for **auxcontrol**. The compiler evaluates the **if** expression (**control | auxcontrol**) as (**0 | 0**) or zero and causes the first part of the **if** statement to become dead code and to be removed.

Factorization

Factorization optimization is particularly useful for storing a frequently used static address in a register. It references the static value or address through register indirect addressing and stores it at the beginning of the function. Factorization optimization is applied to the entire function.

Example 7-6. Factorization

```
fcn(){
    long test;

    if (test == 1) {...}
    if (test == 2) {...}
    if (test == 3) {...}
}

==>

fcn() {
    long test;
    long *t_test=&test;

    if (*t_test==1) {...}
    if (*t_test==2) {...}
    if (*t_test==3) {...}
}
```

Constant Propagation

When a constant is assigned to a variable, the constant can be carried forward to the subsequent uses of the value of that variable until a new value is assigned to that variable. This optimization is performed across basic blocks.

Example 7-7. Constant Propagation

```
int var1;

fcn() {
    int i;
    var1 = 2;
    if (var1)
        i = var1;
    f(var1,i);
}

==>

int var1;

fcn() {
    int i;
    var1 = 2;
    /* Test skipped, since var1 */
    /* is known to be true. */
    i = 2;
    /* Since var1 is known */
    /* to be 2. */
    f(2,i);
}
```

Copy Propagation

When a variable that is “cheap” to access (for example, a register variable) is assigned to a variable that is “expensive” to access (for example, a global variable), the “cheap” variable can be carried forward to the subsequent uses of the “expensive” variable until a new value is assigned to the “expensive” variable. This optimization is performed across basic blocks.

Example 7-8. Copy Propagation

```
int var1;

fcn(register int i) {
    var1 = i;
    if (var1)
        i = var1;
    f(var1+2);
}

==>

int var1;

fcn(register int i) {
    var1 = i;
    if (i)
        /* Since i and var1 */
        /* have the same value. */
        f(i+2);
}
```

Register Allocation

Optimizing register usage throughout an entire routine is also referred to as “register coloring”. This is particularly valuable for keeping most of the local variables in the registers at all times. Using data flow analysis, the compiler finds the lifetime of each variable. The register coloring algorithm can then increase the number of variables stored in registers using the same register for several variables in the same routine.

Unused Definition Elimination

An unused definition is an assignment in which the left-hand side variable is either never used or used after being reassigned. The compiler eliminates unused local definitions unless they are declared as **volatile**.

Example 7-9. Unused Definition Elimination

```
. . .  
. . .  
i = j + 1;      /* If the value of the variable i */  
. . .          /* is never used again, this */  
. . .          /* statement is eliminated. */  
  
i = j + 1; /* This assignment is eliminated since i */  
          /* is not used before it is reassigned. */  
i = k + 1;
```

Loop Optimizations

This section lists optimizations that can be applied to loop constructs.

Loop Invariant Code Optimization

Loop invariant analysis speeds up loops by eliminating computations of invariant expressions and addresses. These computations are moved out of the loop, and the value is stored in a register. This optimization is valuable for removing array subscripting from a loop when the subscript is a variable or expression that is not modified in the loop.

The loop control code generated by an **if** or other similar construct is removed at compile time if the compiler determines that the controlling loop variables cause a loop to be executed only once. The entire loop is removed in the same way as is dead code, if the compiler determines that the loop control variables keep the loop from ever being executed.

Example 7-10. Loop Invariant Code Optimization

```
for (i1 = 1; i1 < 1; i1++) {  
    ival = ival*2+1;  
}  
  
==>      ival = ival*2+1
```

The loop invariant code optimization searches loops for expressions that yield the same result regardless of the number of times the loop is executed. The optimization then places the expression before the loop.

```
while (i<10) {  
    test(j+k+1)  
    i++;  
}  
  
==>      tmp=j+k+1;  
while (i<10) {  
    test(tmp);  
    i++;  
}
```

Loop Rotation

In both the **for** and **while** loops, the controlling expression is located syntactically at the top of each loop. When translating these loops, the controlling expression is rotated to the bottom. This rotation reduces the number of instructions within the body of the loop.

The **while** loop on the left is translated as if it were written as the sequence of statements on the right. The **goto** statement is suppressed if the compiler determines that **expression** is initially false.

Example 7-11. Loop Rotation

<code>while (expression)</code>	<code>==></code>	<code>goto test;</code>
<code>{</code>	<code>==></code>	<code>loop:</code>
<code> statements</code>	<code>==></code>	<code>{</code>
<code>}</code>	<code>==></code>	<code> statements</code>
		<code>}</code>
		<code>test: if (expression)</code>
		<code>goto loop;</code>

The **for** loop on the left is translated as if it were written as the sequence of statements on the right. The **goto** statement is suppressed if the compiler determines that **expr2** is initially false.

<code>for (expr1; expr2; expr3)</code>	<code>==></code>	<code>expr1;</code>
<code>{</code>	<code>==></code>	<code>goto test;</code>
<code> statements</code>	<code>==></code>	<code>loop:</code>
<code>}</code>	<code>==></code>	<code>{</code>
		<code> statements</code>
		<code>}</code>
		<code>expr3;</code>
		<code>test: if (expr2)</code>
		<code>goto loop;</code>

Strength Reduction and Index Simplification

Strength reduction and index simplification is also referred to as induction variable elimination. The values of induction variables increment or decrement by a constant value for each iteration of the loop. These variables are often used to count or index an array. Also, multiples of these variables can be computed. It is often possible to replace all but one through the process of induction variable elimination.

Example 7-12. Strength Reduction and Index Simplification

<pre>for (i=1; i<10; i++) ary[i] = i;</pre>	<pre>==></pre>	<pre>int *pary = ary; for (i=1; i<10; i++) { *pary = i; pary++; }</pre>
--	-------------------	---

Local Optimizations

Local optimizations are generally applied to small sections of the generated code. Some of the jump optimizations described in the “[Jump Optimizations](#)” section in this chapter can also be considered local optimizations.

Common Subexpression Elimination

Common subexpression optimizations remove expressions that have been previously evaluated.

Example 7-13. Common Subexpression Elimination

<pre>if (a+b > 3) test(a+b)</pre>	<pre>==></pre>	<pre>tmp = a+b; if (tmp>3) test(tmp)</pre>
--	-------------------	---

Constant Folding

Arithmetic and logical operations involving only constants are evaluated at compile time.

Example 7-14. Constant Folding

<pre>i = 1 + 2 * 3 + 4</pre>	<pre>==></pre>	<pre>i = 11</pre>
<pre>(a >> 3) >> 1</pre>	<pre>==></pre>	<pre>a >> 4</pre>
<pre>i + 5 + j - 6 + k - 7</pre>	<pre>==></pre>	<pre>i + j + k - 8</pre>
<pre>(4 & 1) (5 > 7)</pre>	<pre>==></pre>	<pre>0</pre>

Redundant Load and Store Elimination

Loading the same variable after storing it is redundant and can be eliminated. This optimization can also be applied to pushing the same variable onto the stack.

Example 7-15. Redundant Load and Store Elimination

```
i = a * 5;  
j = i + 5; /* i is not loaded again, since it is still */
```



```
/* in a register. */
```

Jump Optimizations

The Microtec C++ compiler performs jump optimizations after generating code for each function. The compiler examines every jump instruction and its target location to recognize specific patterns for optimization. Since most jump optimizations create opportunities for further improvements, the optimizations are applied iteratively until no further changes result.

Branch Tail Merging

The compiler combines common instructions preceding two jump instructions that jump to the same location. This optimization is enabled with the **-Og** option.

In the next example, the compiler does not generate code for two **il=7** statements. Instead, it generates code for the **il=7** statement on line 7 and only a jump statement for the **il=7** statement on line 3.

Example 7-16. Branch Tail Merging

<pre>if (i) { i2 = 3; i1 = 7; } else { i2 = 21; i1 = 7; }</pre>	==>	<pre>if (i) i2 = 3; else i2 = 21; i1 = 7;</pre>
---	-----	---

Code Hoisting

In a similar manner to branch tail merging, code hoisting optimization moves common code up in the program to a common point. This optimization can be turned off with the appropriate compiler command line option. See Chapter 3, “[Using Command Line Options](#)” for more information.

Example 7-17. Code Hoisting

```
k = i+prime;  
while (k<10) {  
    flags[k] = 0;  
    k += prime;  
}
```

The instruction **k += prime** at the bottom of the loop is combined with the instruction **k = i+prime** before the loop entry point. When stepping through this loop in the debugger, the program counter steps out of the loop and highlights the **k = i+prime** statement. If you examine

the resulting assembly code, you can see that only part of this statement is being executed (**k=temporary_value**).

Multiple Jump Optimization

Generated code for complex control flow statements contain jumps to jumps, jumps to conditional jumps, or conditional jumps to jumps. These multiple jumps can be rerouted to jump directly to the target locations. In the next example, **goto** statements are used to show the control flow. The same optimization is performed on other control statements that generate jumps to jump instructions.

Example 7-18. Multiple Jump Optimization

goto L1;	==>	goto L2;
...		...
goto L1;		goto L2;
...		...
L1: goto L2;		L1: goto L2;

This optimization changes the perceived program flow.

Redundant Jump Elimination

A jump instruction that jumps to a location that immediately follows it can be eliminated. Redundant jumps usually appear after applying other jump optimizations.

Example 7-19. Redundant Jump Elimination

goto L3;	==>	No code generated
goto L3;	==>	L3: i = 5;

In the next example, the **if** statement is changed as shown. A breakpoint at line 4 is not encountered when **i3==3**.

1	if (i3==3) goto 11;	==>	if(i3==3) goto 12;
2	i1 = 3;		
3	i2 = 3;		
4	11:		
5	goto 12;		
6	k = 5;		
	...		

The next example shows a more typical example of redundant jumps. The test for the inner **while** loop generates a jump out of the loop to the next instruction following the closing brace. This is a jump to the test for the outer loop. The jump out of the inner loop is converted to a jump directly to the test for the outer loop.

```
while (x) {
```

```
    array(x) = 0;
    x = x+1;
    while (i) {
        istat(i) = 0;
        i = i-1;
    }
}
```

Function Inlining

Inline expansion of a user function saves the overhead usually associated with function calls, parameter passing, register saving, stack adjustment, and value return. Sometimes, further optimizations become possible because the actual parameters used in a call become visible, and the side effects of function calls also become exposed to the caller.

Example 7-20. Function In-Lining

```
func(char op, int a, int b) {
    if (op == 1)
        return(a+b);
    else
        return(a-b);
}

main()
{
    int x, y, z;
    ...

    z = func(1,x,y);           ==>      z = a + b;
}
```

inline Keyword

Any function can be qualified as inline to tell the compiler which functions are the best candidates for inlining when invoked within the same module. A function is not expanded inline under the following circumstances:

- The in-lining optimization (**-Oi** option) has not been selected.
- The function declares static local variables.
- The function is deemed too complex or too large by the compiler.
- The invocation point is not favorable. For example, not enough registers are available to make in-lining convenient at a given point in the program.
- The function is an **extern** function. Specifying **inline** for an **extern** function has no effect since inlining is not performed across modules.

Example 7-21. The inline Keyword

```
inline int func (char, int, int); /* prototype */
inline int /* definition */
func(char op, int a, int b) {
    if (op == 1)
        return(a+b);
    else
        return(a-b);
}
```

Instruction Scheduling

Improved performance can be achieved by rearranging the instruction sequence to avoid stalling the instruction unit pipeline. This includes overlapping CPU instructions with FPU instructions and inserting useful instructions between those instructions that could otherwise cause pipeline stall because of data dependence.

Chapter 8

Template Instantiation

This chapter describes the fundamentals of the Microtec C++ template implementation. Templates are a feature in C++ that replace most uses of macros. They also have some unique properties that cannot be simulated with macros.

Templates Versus Macros

Templates can be considered “smart macros”. Both templates and macros can be used to define generic functions or classes. The abstraction of these generic definitions is important in the design of reusable and maintainable software.

The following three examples show the ways in which macros and templates are commonly used to implement a generic “swap” function, which can be used with any type **T**.

Inline Macro

In the following example, the inline macro generates code where it is invoked to swap the values.

```
#define SWAP(T,X,Y) \
do { T tmp_var; tmp_var=X; X=Y; Y=tmp_var; } \
while (0) /* expand macro when used */

SWAP(int, int_var_X, int_var_Y);
SWAP(String, String_var_X, String_var_Y);
SWAP(int, int_var_Z, int_var_Y);
SWAP(String, String_var_Z, String_var_Y);
```

Macro Function

The following example uses a macro called **DEF_SWAP_FUN** to declare a function that performs the swap. This function is actually called by invoking the **SWAP_FUN** macro with two sets of parentheses. The type is specified within the first set, and the values to be swapped are specified within the second set.

```
#define SWAP_FUN(T) swap_function_##T
#define DEF_SWAP_FUN(T) \
void SWAP_FUN(T) (T &X, T &Y) \
{ \
    T tmp;\
    tmp=X; X=Y; Y=tmp;\
}
/* create instances */
```

```
DEF_SWAP_FUN(int)
DEF_SWAP_FUN(String)
/* use instances */
SWAP_FUN(int)(int_var_X, int_var_Y); SWAP_FUN(String)(String_var_X,
String_var_Y); SWAP_FUN(int)(int_var_Z, int_var_Y);
SWAP_FUN(String)(String_var_Z, String_var_Y);
```

Function Template

The following example uses a function template:

```
template <class T> void swap(T &X, T &Y)
{
    T tmp; tmp=X; X=Y; Y=tmp;
}
/* use automatically generated instances */
swap(int_var_X, int_var_Y);
swap(String_var_X, String_var_Y);
swap(int_var_Z, int_var_Y);
swap(String_var_Z, String_var_Y);
```

Templates are preferred over macros for the following reasons:

- Template names are recognized by the C++ compiler; thus, better semantic checking can be performed. Macro names are lost after the C++ preprocessor is finished running.
- A template unit has its own scope and can avoid naming problems in the definition of macros. For example, in the **SWAP** inline macro example, the name **tmp_var** cannot be used as the argument for **X** or **Y**.
- The creation of template instances is automatic. While macros can be used to define generic functions or classes, their instances must be manually created before they can be used.
- Arguments to templates are typed and checked for each instance. This catches errors in the arguments of template instances and also requires users to give each template a well-defined functionality.

Macros do not have this type checking feature. For example, **SWAP(int, a_int_var, a_char_var)** is usually accepted by a C/C++ compiler with automatic conversion between **int** and **char**, which might be incorrect for the user.

Compile-Time Template Instantiation

The Microtec C++ compiler implements a compile time template instantiation strategy. This strategy gives users early error messages, full control over the scope of template instances, easy creation of class libraries, and, in most cases, the shortest edit-compile-run cycle. The following is a typical organization of source files.

Both the *stack.h* file and *stack.def* file have inclusion guards to avoid multiple inclusion. Source files such as *user1.cc* include *stack.h*, which includes *stack.def*, to obtain the declaration and definition of the stack template.

Note



The nesting level of template class instantiation has a limit of 17.

Declare the Class Template Stack in stack.h

```
/* to avoid repeated inclusion */
#ifndef STACK_H
#define STACK_H

/* include required header files for Stack declaration */
#include <stdlib.h>

/* the declaration of Stack template */
template <class X> class Stack {
    public: static int some_data;
    void push(X);
    ...
};
#ifndef STACK_DEF
#include "stack.def"
#endif
#endif /* STACK_H */
```

Define the Class Template in stack.def

```
#ifndef STACK_DEF
#define STACK_DEF
#ifndef STACK_H
#include "stack.h"
#endif

template <class X> void Stack<X>::push(X data) {
    /* generic code for push */
}

void Stack<double>::push(double data) {
    /* special code for Stack<double> */
}

template <class X> int Stack<X>::some_data = 1234;
/* default initial value for an instance */

int Stack<double>::some_data = 2468;
/* special initial value for instance Stack<double> */

#endif /*STACK_DEF */
```

Use Template Class in Source File user1.cc

```
#include "stack.h"
...
Stack<int> myStack;
int n;
myStack.push(n);
```

Automatic Instantiation

The Microtec C++ compiler automatically generates all template instances used in each module of an object file by default. Duplicated instances are skipped at link time, so there is no duplication in the final program.

To avoid duplication in object files, the **-tm** option can be used to disable the creation of template instances in a compilation. Template instances that are used but not generated should be generated in some other modules and linked into the final program.

Template instances are checked only when they are created. Some syntax errors in a template definition are caught only if the template is instantiated. This is a requirement by the C++ language standard. Thus, using the **-tm** option also disables the syntax and semantic checks of template instances.

Scope of Template Instances

A template declaration or definition usually contains names undefined in its local scope. The binding of names in a template depends upon the scope of the template instance.

The Microtec C++ compiler implements the binding rules in *The C++ Annotated Reference Manual*. All names used in a template definition are bound according to the following two rules:

- If a name does not depend upon the template arguments, it is bound in the context of the template definition.
- If a name might depend upon the template arguments, it is bound by the file scope immediately before the first place where the template instance is used.

You are responsible for including a template definition at the right place in a file. To avoid inconsistent scoping of template instances, include template definitions in the header file that declares the template. All modules that use a specific template should include the header file for that template to obtain the correct declaration of the template, as well as the correct scope of the template definition.

Manual Template Instantiation

Manual control of template instantiation is necessary to have the shortest compile-and-link cycle or to save disk space. However, use these manual controls only as an optimization after a working version is built.

To avoid duplication, **-tm** can be used on some compilation modules to turn off the creation of template instances. You can still use all template instances when compiling with **-tm**, but you need to create those instances in some other module and link them to the final program.

Two instantiation **#pragmas** are provided to further control the creation of template instances:

```
#pragma instantiate a_template
#pragma do_not_instantiate a_template
```

For example, to design a C++ library that uses class `Stack<int>`, put only one instance of `Stack<int>` in your library and turn off its instantiation everywhere else. You can put the following statement in your library header file:

```
#ifndef BUILDING_MY_LIBRARY
#pragma do_not_instantiate Stack<int>
#else
#pragma instantiate Stack<int>
#endif
```

Hence, users of your library do not need to spend time creating the `Stack<int>` instance, because the instance is explicitly created in the library.

a_template can be one of the following in the instantiation pragmas:

Template class name: `Stack<int>`
 Member function name: `Stack<int>::push`
 Static data member name: `Stack<int>::some_data`
 Member function declaration: `void A<int>::f(int,char)`
 Template function declaration: `char* foo(int,float)`

When a template class name is used in an instantiation **#pragma**, all member functions and static data members of that class are affected.

A member function name cannot be used when it is overloaded. A full declaration of a member function should be used if there are overloaded member functions.

Specialization

Special definitions of template instances must be seen before they are used. Otherwise, the compiler generates instances from the generic template definition.

The Microtec C++ compiler does not consider multiple special instances an error at link time to make it easier for users to define special instances. For example, in *stack.h*:

```
template <class T> class stack {
public: static int some_data;
...
};

template <class T> int Stack<T>::some_data = 1;
// initial value for the generic definition of some_data

int stack<int>::some_data = 2;
// special definition of Stack<int>::some_data
```

in **a.cc**:

```
#include "stack.h"
...
Stack<char>::some_data;    // get value 1
Stack<int>::some_data;    // get value 2
```

in **b.cc**:

```
#include "stack.h"
...
Stack<char>::some_data;    // get value 1
Stack<int>::some_data;    // get value 2
```

When you compile and link **a.cc** and **b.cc**, instances of **Stack<char>::some_data** and **Stack<int>::some_data** are created in both **a.o** and **b.o** files. The Microtec linker removes the duplicated instances without any error or warning message.

On the other hand, if the special definition of **Stack<int>::some_data** is moved from *stack.h* to **a.cc**, **a.o** gets this special definition, but **b.o** gets the generic definition. Depending on your link options, you might not get any warning of this inconsistency. In the final program, whether **Stack<int>::some_data** has an initial value of 1 or 2 is unknown.

Chapter 9

Precompiled Header Files

This chapter describes how to create and use precompiled header files, which can be used to shorten compilation time for source files that include several of the same headers. Precompiled header files are produced by saving a “snapshot” of the current state of compiling source code to a file, so that when you compile the same source again or compile different source with the same set of headers, the compiler can recognize the “snapshot point” and read in the previously-compiled state.

Precompiled Header File Processing

When the **-jH** option is specified on the command line, automatic precompiled header processing is enabled. This tells the compiler to look automatically for a usable precompiled header file to read in and create one for use on subsequent compilations if necessary.

A precompiled header file contains a “snapshot” of all code preceding the header stop point. The header stop point generally occurs at the first token in the primary source file that does not belong to a preprocessing directive. It can also be forced by using **#pragma hdrstop**.

Example 9-1. Precompiled Header Processing

```
#include "xxx.h"
#include "yyy.h"
int i;
```

In the preceding code, the header stop point occurs at the **int** declaration; the precompiled header file for this source would contain a snapshot showing the inclusion of the files *xxx.h* and *yyy.h*.

```
#include "xxx.h"
#ifdef YY_H
#define YY_H 1
#include "yyy.h"
#endif
if TEST
    int i;
#endif
```

In this example, the first code not belonging to a preprocessing directive is again the **int** declaration; however, the **int** declaration occurs as part of a **#if** block. In this case, the header stop point occurs just before the **#if** beginning the block in which the **int** declaration is a part.

Precompiled header files are produced only if the header stop point and the code preceding it meet the following requirements:

- The header stop point must occur at file scope. This means that it cannot be with an unclosed scope established by a header file. For example, no precompiled header file would be produced for the following files:

```
// xxx.h
class A{

// xxx.C
#include "xxx.h"
int i;
};
```

- The header stop point cannot be inside a declaration started within a header file, nor can it be part of a declaration list of a linkage specification. For example, no precompiled header file would be generated for the following files, since the **int** declaration is a continuation of the declaration begun in *yyy.h*:

```
// yyy.h
static

// yyy.C
#include "yyy.h"
int i;
```

- The header stop point cannot be inside a **#if** block or a **#define** statement started in a header file.
- The processing preceding the header stop point cannot have produced any errors.
- No references to the predefined macros **__DATE__** or **__TIME__** can have occurred.
- The **#line** preprocessor directive cannot have been used.
- The code cannot contain **#pragma no_pch**.
- The code preceding the header stop point must have introduced enough declarations to justify the overhead associated with precompiled headers. This number can be configured by defining the macro **PCH_DECL_SEQ_THRESHOLD** as the minimal number of declarations required for generation of a precompiled header file.

Precompiled header files contain information that can be checked to determine when they can be reused. This information includes the following:

- Compiler version, including the date and time the compiler was built
- Current working directory
- Command line options
- Initial sequence of preprocessing directives from primary source file
- Date and time of header files specified in **#include** directives

This information is called the precompiled header file prefix. If the prefix for a precompiled header file matches the data for a file being compiled with the **-jH** option, the remainder of the precompiled header file is read in.

If a source file can use more than one precompiled header file, the one representing the most preprocessing directives from the primary source file is used. For example, if a primary source file begins with the following:

```
#include "xxx.h"  
#include "yyy.h"  
#include "zzz.h"
```

and there are precompiled header files for "xxx.h" and for "xxx.h" and "yyy.h", the latter is used if both files match the current compilation. Further, after the precompiled header for the first two header files has been read in and the third one compiled, a new precompiled header file encompassing all three headers is created.

Using Options With Precompiled Header Files

Several options interact directly with precompiled headers. These include the **-jH**, **-jHc**, **-jHd**, and **-jHu** options. These options are described in the “[Use Precompiled Headers](#)” section in Chapter 3, “[Using Command Line Options](#)”.

Some options cannot be used with precompiled header options. If options specifying precompiled headers are used alone with these options, the precompiled header options are ignored and the compiler generates a warning message. Options that cannot be used with precompiled headers include the following: **-C**, **-E**, **-Es**, **-Fli**, **-Flp**, **-Flt**, **-I**, **-P**, and **-Ps**.

Another set of options are independent of precompiled header file operations. Their settings are not saved in precompiled header files. These options include the following: **-jH**, **-jHc**, **-jHu**, **-m**, **-ND**, **-NI**, **-NS**, **-NT**, **-NZ**, **-o**, **-QA**, **-Qe**, **-Qfn**, **-Qfs**, **-Qi**, **-Qme**, **-Qmw**, **-Qo**, **-Qs**, **-Qw**, **-Q**, **-q**, **-S**, **-Vb**, **-V**, and **-y**.

Any option not represented in one of the preceding lists has its state saved in precompiled header files. If used from the command line (rather than with the **#pragma options** directive), they must match exactly in order to reuse a precompiled header file.

Using the Preprocessor With Precompiled Headers

There are several preprocessor macros and pragmas that interact with precompiled header processing. Some of these define properties to be used by precompiled header files.

The following pragmas can affect precompiled header processing:

- **#pragma asm**

When **#pragma asm** or **#pragma endasm** appears in the main source file, it behaves as **#pragma hdrstop** for precompiled header processing. This prevents repeated execution of assembly code denoted by these directives.

- **#pragma hdrstop**

This pragma forces a header stop point.

- **#pragma no_pch**

This pragma disables the use and generation of precompiled header files.

- **#pragma options**

If a precompiled header file uses **#pragma options**, the same options will be set upon reuse of the precompiled header file. Command line options must always match for precompiled header files to be used, regardless of the state of options set using **#pragma options**.

In addition, pragmas that have side effects that cannot be reproduced (such as **#pragma info**, **#pragma error**, **#pragma macro**, and **#pragma warn**) will not be executed.

The following macros can affect precompiled header processing:

- **PCH_FILE_SUFFIX** (default: **.pch**)
- **USE_MMAP_FOR_MEMORY_REGIONS** (default: **FALSE** (Windows), **TRUE** (UNIX))
- **USE_FIXED_ADDRESS_FOR_MMAP** (default: **FALSE**)
- **DEFAULT_PREALLOCATED_PCH_MEM_SIZE** (default: **1MB** (Windows), **4MB** (UNIX))
- **PCH_DECL_SEQ_THRESHOLD** (default: **1**)

Precompiled Header File Usage Guide

There are two ways to use precompiled header file options. One is to use the **-jH** option, and the other is to use the **-jHc** and **-jHu** options.

Using the **-jH** option is the simplest approach. The compiler will search for all precompiled header files in the current directory or the directory specified by the **-jHd** option, attempting to locate a best-match precompiled header file to use for the current compilation. If there is no suitable precompiled header file found, or if the match is not perfect, a new precompiled header file will be created with the same base name as the source file and the extension *.pch*. This file will be created in the directory specified by the **-jHd** option or in the current directory if none is specified.

By creating multiple precompiled header files, the **-jH** option provides the greatest possibility of reusing precompiled header files. However, it might create more precompiled header files than required if most of your source files use different sets of header files. This situation could consume disk space quickly, since each precompiled header file can take up half a megabyte or more.

The **-jHc** and **-jHu** options are designed to minimize the space used by precompiled header files. Suppose you have many source files that include a common set of header files, but each one also includes other header files. One way to avoid the creation of excess precompiled header files and still reuse precompiled header files as much as possible is to pick one key source file to generate a single, precompiled header file to be used by all the other files. For example, your key source file might contain the following lines:

```
#include <iostream.h>
#include "mycommon.h"
#pragma hdrstop
#include "extra.h"
```

When you compile this file with the **-jHc mypch.pch** option, the *mypch.pch* file is created with the compiled information of *iostream.h* and *mycommon.h*. Then you can compile the remaining files using the **-jHu mypch.pch** option. The precompiled header file will be reused for all files that have an inclusion sequence starting with *iostream.h* and *mycommon.h*.

A precompiled header file will be reused only when all included header files have unchanged time stamps. Many command line options also need to be the same to reuse a precompiled header file. This is usually not a problem for a project that compiles multiple files with a single set of options for all the files. When you compile the same file the second time with the same option, the precompiled header file created in the previous compilation will be reused unless some header files have been changed.

Precompiled header files should be considered intermediate files when performing parallel compilations. There is no concurrency control regarding the access of precompiled header files. When the same precompiled header file is accessed by two processes at the same time, concurrent write could result in corrupted precompiled header files, and concurrent write and read could cause the failure of the read operation. When a precompiled header file is found corrupted, it is deleted. When a precompiled header file read operation fails, normal compilation without precompiled header files is executed to generate the correct code. Concurrent compilation with the **-jH** or **-jHc** option is not recommended, in order to avoid concurrent write to the same precompiled header file. The **-jHu** option, however, is generally safe to use with concurrent compilation.

Chapter 10

Interlanguage Calling

This chapter describes C, C++, and assembly language calling conventions for the Microtec C/C++ Compiler. This chapter provides enough information for you to write assembly language routines to interface with C or C++ language functions.

Note



The Microtec C/C++ Compiler makes use of several intrinsic functions. These intrinsic functions are tuned for performance and do not follow any calling convention.

Parameter Passing

Assembly language routines must follow the C/C++ parameter passing conventions to pass values to, or to receive values from, C/C++ functions. C/C++ functions pass parameters by value in specific registers and on the stack. Parameters are always evaluated from right to left, with the first parameter being evaluated last. The next section describes the conventions on procedure calls.

Setting Parameters

Parameters are passed in registers and on the stack. Parameters are always at least word-aligned, and must always be an integral number of words long.

Integer Parameters

Each parameter that has a type of **int** or **pointer** is passed in a single register. The first parameter is passed in register **R3**. Parameters that follow are passed from left to right sequentially into registers **R4** to **R10**. Additional parameters are placed on the stack.

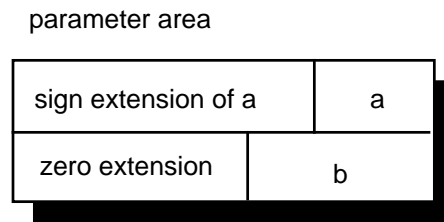
A single register never contains more than one parameter:

```
func(short s, short t)
```

A **short** integer (byte or word) is extended to a **long** word according to its sign before it is set in the parameter area as shown in [Figure 10-1](#).

Figure 10-1. Parameter Area and Short Integers

f(a, b), where a is a signed byte and b is an unsigned word



Floating Point Parameters

If floating point hardware is enabled, then each parameter that has a type of **float** or **double** is passed in a floating point register. The first parameter is passed in register **F1**. Parameters that follow are passed from left to right sequentially into registers **F2** to **F8**. Additional parameters are placed on the stack. If software floating point emulation is enabled (**-nf** option), then floating point parameters are passed in general-purpose registers as described above.

Parameters of type **double** require two general-purpose registers, and the first one must be odd-numbered. If the next available parameter register for a double argument is even, it is skipped.

Structure Parameters

Each parameter that has a type of **struct** or **union** is passed on the stack. All **int** and **float** type members allocate 4-bytes of storage. All **double** type members allocate 8-bytes of storage and require alignment on 8-byte boundaries. Padding is inserted as necessary to maintain the correct alignment.

Implicit Parameter for Structure/Union Return Value

When the return value type is a **structure** or **union** that is greater than 8-bytes, the calling function allocates space on the stack to place the return value. A pointer to this area is passed as a hidden parameter on the stack. The called function uses this information to store the return value.

Function Return Values

This section describes the type of return values that are possible.

An **integer** or **pointer** type return value is set in **R3**. An **integer** return value with a size of less than 4-bytes is not extended to a **long** word before returning. With the **-f** option, a **floating point** return value is set in **F1**. With **-nf**, a **float** type is returned in **R3**. A **double** type is returned in **R3** and **R4**.

A **structure** or **union** return value that is 8-bytes or less is returned in registers **R3/R4**. A **structure** or **union** return value greater than 8-bytes is returned through the implicit first parameter which points to the return value area.

When a **structure** return value is assigned to another structure or pushed onto the stack as a parameter, the implicit parameter that indicates the return value area can be selected as follows:

```
struct s {
    int r, i j;
} a, f();

    <implicit parameter is the address of a>
# a=f ();
addic    3,1,0x8
crxor    6,6,6
bl       f
lwz      3,0x8(1)
addis    4,0,ha(a)
stw      3,lo(a)(4)
lwz      3,0xc(1)
addis    4,0,ha((a)+4)
stw      3,lo((a)+4)(4)
lwz      3,0x10(1)
addis    4,0,ha((a)+8)
stw      3,lo((a)+8)(4)

    <implicit parameter is the address of
    parameter area>
# g(f(),1);
addic    3,1,0x14
crxor    6,6,6
bl       f
lwz      3,0x14(1)
stw      3,0x20(1)
lwz      3,0x18(1)
stw      3,0x24(1)
lwz      3,0x1c(1)
stw      3,0x28(1)
addic    3,1,0x20
addi     4,0,0x1
bl       g
```

Register and Stack Usage With Functions

Table 10-1 illustrates how the Microtec C/C++ Compiler uses registers:

Table 10-1. Register Usage

R0	Scratch Register	F0	Scratch Register
R1	Stack frame pointer	F1	Parameters and return values
R2	Secondary small data pointer	F2-F8	Parameter Passing
R3-R4	Parameters and return values	F9-F13	Scratch Registers

Table 10-1. Register Usage (cont.)

R5-R10	Parameter Passing	F14-F31	Local variables
R11-R12	Scratch Registers		
R13	Small data pointer		
R14-R31	Local variables		

Register **R31** is reserved for stack frame threading when the **-Kf** option is in effect.

Stack Frames

Local frames for functions are automatically generated by the Microtec C compiler if a function calls another function.

If a function does not call another function and has no stack variables, local frames are not generated, unless the **-Kf** option is in effect.

The format for a stack frame is defined in the *PowerPC Application Binary Interface*. This layout is illustrated in [Table 10-2](#):

Table 10-2. Stack Frame Layout

Floating point register save area	High Address
General register save area	
CR save area	
Local variable space	
Additional parameter list area	
LR save word	
Previous function's frame pointer	Low Address

Only the link register entry is required. The size of the space allocated is dependent upon the number and type of registers saved on the stack frame. However, the size must be a multiple of 16.

Note



Modifying the stack pointer can cause unpredictable run-time errors.

Prologue

The prologue is the sequence of code at the beginning of a function. The prologue sets up a local stack frame and a frame pointer so that function parameters and automatic variables can be

accessed through the pointer. The prologue can also include code that saves certain registers on the stack.

You can write your own assembly routines to interface to C/C++ programs. If the routines change the contents of any of the registers **R14** through **R31**, or **F14** through **F31**, you must save the old values.

If you write your own applications and the assembly language routine does not call another function and has no stack variables, it is not necessary to set up a local stack frame.

Epilogue

The epilogue is the sequence of code at the end of a function. The epilogue restores the saved registers and resets the stack frame to the condition at entry. It then passes control back to the calling routine with a **BCLR** instruction.

Registers that are saved in the prologue should be restored first. The local stack frame can be reset by restoring the Link Register and reloading **R1** with the previous frame pointer.

The following is the recommended instruction sequence for exiting a routine:

```
lwz      0,0x34(1)
mtlrr    0
addi     1,1,0x30
bclr     0x14,0x0
```

Assembly Language Interfacing

When interfacing to an assembly language routine, it is recommended that you create a minimal local stack frame. This provides an easy way of keeping local data on the stack. It will also allow the EDGE Debugger to display assembly language routine names in the procedure traceback window.

Perform the following steps to write an assembly language routine that can be called from C/C++:

1. Create a local stack frame.
2. Save registers.

Save any of the registers **R14** through **R31**, or **F14** through **F31**, if they were used in the assembly language routine. All registers should be saved in numerical order immediately after the stack frame is built.

3. Return values.

Return values should be correctly placed before returning. See the “[Function Return Values](#)” section in this chapter for more information.

4. Restore registers.

At the end of the assembly language routine, the saved registers and the old stack frame must be restored. This must be done immediately before the return instruction.

Assembly Language Routine Example

A prototype of an assembly language routine (named `_foo`) that can be called from a C/C++ function is shown in the following example:

```
_foo:mflr    0; Move the link register to r0
stwu       1,-32(1); Creates stack frame.
           ; Allocates 32 bytes of space
           ; for local data.
stw        0,0x24(1); Saves the link register on the stack.
.
           ; Body of procedure.
.
.
lwz        0,0x24(1); restores the link register
mtlr       0
addi       1,1,0x20; Undoes the stack frame.
           ; Restores r1 to original value.
bclr       0x14,0x0; Returns to caller.
```

Variable References from Assembly Routines

The compiler does not prepend any leading symbol to variable names. This allows them to be accessed using the same symbol name in assembly language or C/C++. However, class members in C++ will have mangled names, making them much more difficult to access from assembly language.

Defining a Function

C++ is a strongly typed language that does extensive cross-checking, which helps uncover programming errors. Every C++ function definition specifies the types of its arguments, and the compiler checks that every call to that function contains the appropriate number and type of arguments. In order to implement this feature, each C++ function is uniquely identified by the compiler. This unique identification (associating the function name with a specified number of arguments of a specified type) enables the C++ programmer to use duplicate function names, which is not permitted in the C language. Using the same function name for two different functions is referred to as “overloading” the function name. The following example demonstrates function overloading (this is legal in C++ since the unique identifier assigned to each function enables the compiler to differentiate between the two functions):

```
func_1 (char);/* unique identifier: func_1__Fc */
func_1 (int);/* unique identifier: func_1__Fi */
```

The C++ function declaration is called a prototype declaration. A function prototype declaration includes the function name, return type of the function, the argument types in parentheses, and a closing semicolon. A function prototype declaration must be specified before calling the function. The C++ compiler uses this prototype information (the function name and the argument types) to generate a unique identifier for the function, known as a C++ function signature or a C++ mangled function name.

Calling C Functions From C++

To prevent the C++ compiler from trying to create the unique C++ function signature, inform the C++ compiler when calling a C function from your C++ program. The compiler must prevent the mangling of C function names, since C code does not recognize mangled names. The meaning of the C keyword **extern** has been extended for this purpose:

```
extern "C" {  
    func_1 (char);  
    func_2 (char *, int);  
}
```

As shown in this example, declare all the C functions called from C++ programs as **extern "C"** (note that a capital C is required). Functions declared in this manner are treated as C functions and are subject to standard C rules.

After you modify a file to use the **extern "C"** declaration, that file can no longer be compiled by a C compiler, as the declaration is illegal in the C language. To compile the same header file with C and C++, use **#ifdef __cplusplus**, as follows:

```
#ifdef __cplusplus  
    extern "C" {  
        func_1(char);  
        func_2(char *, int);  
    }  
#else  
#ifdef __STDC__  
    extern func_1(char);  
    extern func_2(char *, int);  
#else  
    extern func_1(), func_2();  
#endif
```

Passing Arguments

In most cases, traditional C function declarations work without complications. Most ANSI C compilers, however, “promote” arguments of type **float** (4-bytes) to type **double** (8-bytes) if a nonprototyped C function declaration is used. This behavior can confuse programmers who do not explicitly define argument types in their **extern "C"** declarations.

Example 10-1. Passing Arguments Between C and C++

In a C source file, define two functions, **f** and **g**:

```
f (old_fd)/* traditional C function definition */
float old_fd;/* old_fd treated as double */
{
    ...
}

g (float ansi_fd)/* ANSI C function prototype definition */
{
    /* ansi_fd treated as float */
    ...
}
```

Add the following code to your C++ source file to call the two C functions from a C++ source file:

```
extern "C" {/* function definitions in C++ file */
    f(double);/* variable treated as double */
    g(float);/* variable treated as float */
}
```

Function **f()** is defined according to traditional C, with the argument name in parentheses. Traditional C compilers promote the variable **old_fd** to **double**, even though it is explicitly defined as **float**. All subsequent uses of this variable would be treated as **double**, which might affect the results of a program. No such problem exists in ANSI C, since ANSI C compilers enforce the variable type **float** specifically defined in the function prototype definition.

The C++ compiler uses types explicitly defined in function prototype definitions for ANSI C and C++ code. For traditional C code, the C++ compiler correctly interprets the function declaration of **f()** in the **extern "C"** declaration in a C++ file as **f(double)** unless specified as **f(float)**. Although a declaration of **f(float)** seems logical, it would be incorrect (since the C compiler always treats it as a **double**) and could even have unfortunate side effects.

To avoid confusion, explicitly define **float** arguments for traditional C functions as **double** in **extern "C"** declarations, as shown in the example, and add a comment. This explicit declaration instructs the C++ compiler to treat the argument as it was originally interpreted by the traditional C compiler. It also clearly indicates the argument type to the casual observer. Always add an explanatory comment when declaring a traditional C **float** argument as **f()** or **f(double)**. An inexperienced C++ programmer might change the traditional C function definition to **f(float)** to be consistent, thus introducing the possibility of side effects in code that was working properly.

A similar approach is to redefine traditional C function definitions to change **float** variables to **double**. This method eliminates the need for promotion and avoids misinterpretation by inexperienced C++ programmers. Unfortunately, this method requires altering the original function definitions on C source code, which might not be desirable.

Calling C++ Functions From C

This section outlines factors to be considered when calling C++ functions from C.

Overloaded Functions

You cannot call C++ overloaded functions from C without explicitly invoking the proper function using the corresponding C++ function signatures or mangled names.

Member Functions

Avoid calling member functions directly. It is more systematic and less error-prone to call a C++ nonmember function or **friend** function layer, which then invokes the proper member functions and performs the class object operations/manipulations. This C++ non-member function or **friend** function layer should be declared in C++ as the **extern "C"** linkage so that it is compatible with the C linkage name rules.

The following example shows the use of a non-member function layer in C++ to create the interlanguage relationship needed between the C and C++ files. [Figure 10-2](#) illustrates the relationship:

```
#include <stdio.h>
class Matrix {
    int row;
    int col;
public:
    Matrix (int row1=0, int col1=0) {
        row = row1;
        col=col1;
    }
    int get_row() { return row; }
    int get_col() { return col; }
    void self() {
        printf("Your location is: row (%d) col (%d)\n",
            get_row(), get_col() );
    }
};

extern "C" {
    void print_matrix(Matrix *);
}

void print_matrix(Matrix *mobj) {
    mobj->self();
}

extern "C" {
    int C_Matrix_get_row (Matrix *mobj);
}

int C_Matrix_get_row (Matrix *mobj) {
    return mobj->get_row();
}
```

```
    }                               /* for Matrix::get_row() */

extern "C" {
    int C_Matrix_get_col (Matrix *mobj);
}

int C_Matrix_get_col (Matrix *mobj) {
    return mobj->get_col();
}                               /* for Matrix::get_col() */
```

Figure 10-2. C/C++ Interlanguage Calling

C Code

```
extern struct Matrix;
typedef struct Matrix
    Matrix;
extern void
    print_matrix( struct
        Matrix *);
extern int
    C_Matrix_get_col( struct
        Matrix *);
extern int
    C_Matrix_get_row( struct
        Matrix *);
void f() {
    struct Matrix *p;
    int x;
    . . .
    print_matrix(p);
    x=C_Matrix_get_col(p);
    x=C_Matrix_get_row(p);
    . . .
}
```

C++ Code

```
class Matrix {
    . . .
public:
    void self() { . . . }
};
extern "C" {
    void print_matrix(Matrix *);
}
void print_matrix(Matrix *mobj) {
    mobj->self();
}
extern "C" {
    int C_Matrix_get_col (Matrix *, mobj);
    int C_Matrix_get_row (Matrix *, mobj);
}
int C_Matrix_get_col (Matrix *, mobj) {
    return (mobj->get_col());
}
int C_Matrix_get_row (Matrix *, mobj) {
    return (mobj->get_row());
}
. . .
```

To call the C++ **self()** function from your C++ code, refer to it as **Matrix::self()**. However, to call the C++ **self()** function from your C code, use another way to reference this function, because a term such as **Matrix::self()** is not recognized in the C language.

You can solve the problem by creating a C++ non-member function layer, also known as a C-callable C++ wrapper, in your C++ code. This wrapper is created by defining a new function **print_matrix** in your C++ code. This new function is passed as a pointer to the class (**Matrix**) containing the function that the C program wants to call.

Matrix is defined as **struct** in the C code and **class** in the C++ code if these structures are equivalent across the languages. Ensure that the structures are equivalent in terms of structure size, structure layout, and the offsets of the data members. Use an external declaration in your C code as follows to ensure that the **Matrix** types are usable across C and C++:

```
extern struct Matrix;
```

Your C++ code declares **Matrix** as follows:

```
class Matrix {...};
```

The **print_matrix** function is declared in your C code as follows:

```
extern void print_matrix (Matrix *);
```

The **print_matrix** function is declared in your C++ code as follows:

```
extern "C" {  
    void print_matrix (Matrix *);  
}
```

Declare the function **print_matrix** as **extern "C"** in your C++ code to prevent the function name from being mangled with a function signature (the C code would not be able to recognize the function name if it were encoded). If you declare **p** an external variable, its implementation names are the same in Microtec C and Microtec C++, which facilitates the passing of global objects between C and C++.

Using the code shown in [Figure 10-2](#), your C code call safely invokes the **print_matrix** function, which points to the C++ **self()** function contained in class **Matrix**. (The **print_matrix** function is a legal C function because of the **extern "C"** declaration.) This approach allows you to call C++ functions from C code.

You could also develop a **friend** function for each visible (public) member function of the class that you would call with C functions. The following example shows how to define a C-callable C++ wrapper through **friend** functions:

```
#include <stdio.h>  
extern "C" {  
    int C_Matrix_get_row (Matrix *);  
    int C_Matrix_get_col (Matrix *);  
    void print_matrix (Matrix *);  
}  
class Matrix {  
    int row;  
    int col;  
public:  
    Matrix (int row1=0, int col1=0) {  
        ...  
    }  
    int get_row () {  
        ...;  
    }  
}
```

```
int get_col () {  
    ...;  
}  
void self () {  
    ...;  
}  
friend int C_Matrix_get_row (Matrix *);  
friend int C_Matrix_get_col (Matrix *);  
friend void print_matrix (Matrix *);  
};
```

In this example, each C-callable C++ non-member function of class **Matrix** is declared as a **friend** function to class **Matrix**.

Chapter 11

Internal Data Representation

This chapter describes internal data formats of the C and C++ languages and the run-time organization of C and C++ programs for supported microprocessor families. Additional information is available in the *System V Application Binary Interface PowerPC Supplement* and the *PowerPC Embedded Application Binary Interface*.

Storage Layout

Memory is accessed according to the type of processor. There are two basic types of processors:

- Big-endian: addresses objects in memory from the “big” or most significant byte end. The preprocessor symbol **_BIG_ENDIAN** is defined for these processors.
- Little-endian: addresses objects in memory from the “little” or least significant byte end. The preprocessor symbol **_LITTLE_ENDIAN** is defined for these processors.

the instruction to load address N is handled in two different ways when addressing memory such as that shown in [Figure 11-1](#). A big-endian processor considers N the more significant byte, with $N+1$ as the less significant byte. A little-endian processor considers $N+1$ the more significant byte, with N as the less significant byte (see [Figure 11-2](#)).

Figure 11-1. Memory Layout

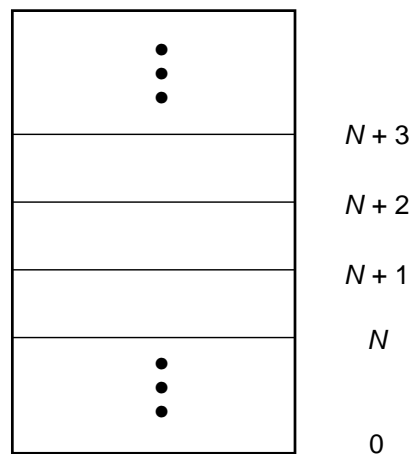
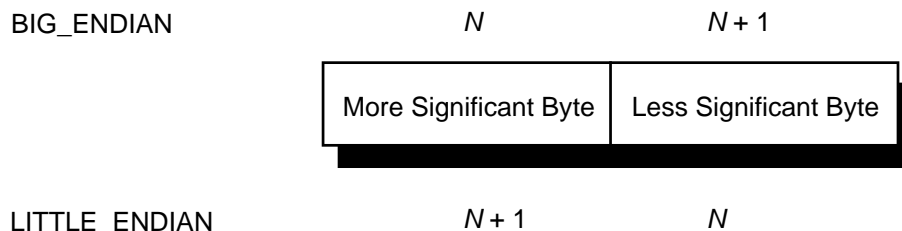
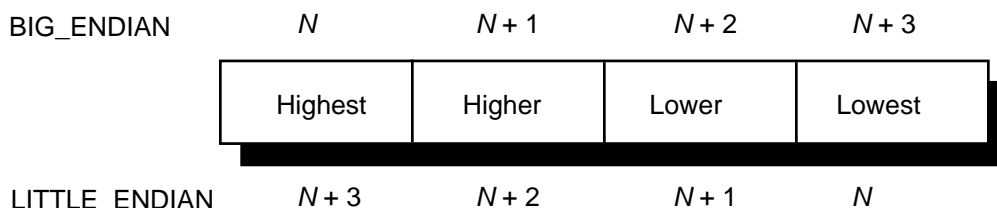


Figure 11-2. Loading Dependent on Processor Type



For a 4-byte quantity with address N in a big-endian processor, N represents the address of the most significant byte of the high-order word; the low-order word is located at address $N+2$, leaving the least significant byte at address $N+3$. The little-endian processor treats $N+3$ as the most significant byte, with N as the least significant byte. [Figure 11-3](#) shows a 4-byte quantity.

Figure 11-3. Byte Ordering in Words



Note



The term “word” refers to two bytes; “long word” refers to four bytes.

MCCPPC and CCCPPC support both little-endian and big-endian processor types via the **-KE** option.

Data Type Summary

Data types for the supported microprocessor families include both scalar and complex.

[Table 11-1](#) shows the size and value range of the scalar data types. The range of values are decimal representations, and the alignment is in bytes.

Table 11-1. Scalar Data Types

Data Type	Size	Range	Byte Alignment
signed char	8-bits = 1-byte	-128 to 127	1
unsigned char	8-bits = 1-byte	0 to 255	1

Table 11-1. Scalar Data Types (cont.)

Data Type	Size	Range	Byte Alignment
short int	16-bits = 2-bytes	-32768 to 32767	2
unsigned short int	16-bits = 2-bytes	0 to 65535	2
enum	32-bits = 4-bytes		4
int	32-bits = 4-bytes	-2147483648 to 2147483647	4
unsigned int	32-bits = 4-bytes	0 to 4294967295	4
pointer	32-bits = 4-bytes		4
long int	32-bits = 4-bytes	-2147483648 to 2147483647	4
unsigned long int	32-bits = 4-bytes	0 to 4294967295	4
long long int	64-bits = 8-bytes	-9223372036854775808 to 9223372036854775807	8
unsigned long long int	64-bits = 8-bytes	0 to 18446744073709551615	8
float	32-bits = 4-bytes	$1.18 * 10^{-38}$ to $3.4 * 10^{38}$	4
double	64-bits = 8-bytes	$1.18 * 10^{-308}$ to $3.4 * 10^{308}$	8
long double	64-bits = 8-bytes	$1.18 * 10^{-308}$ to $3.4 * 10^{308}$	8

Note

If the keyword **signed** or **unsigned** does not appear, the **char** data type is considered to be unsigned unless the **-nKu** option is used, in which case it is considered to be signed.

Structure Operations

The following sections discuss Microtec extensions to structure operations.

Structure Alignment

Structure members are aligned according to their type (see [Table 11-1](#) for alignment of scalar types). The compiler might add padding between structure members and at the end of a structure. Structures are always padded so that the size is a multiple of the maximum alignment of the members of the structure. For example, a structure of the type shown in the following example is allocated $4 + 1 + (3\text{-byte padding}) + 4 = 12\text{-bytes}$.

Example 11-1. Structure Alignment

```
struct date /* structure tag */
{
    int day;
    unsigned char month;
    int year;
} holiday; /* structure variable name */

main()
{
    holiday.day = 25;
    holiday.month = 12;
    holiday.year = 1988;
}
```

In this example, the structure variable is **holiday** and the members of the structure are **day**, **month**, and **year**. The **date** is an identifier called a structure tag. This structure tag can be used for later definitions and declarations without repeating the structure members. For example:

```
struct date workday;
```

More information on the syntax for structures can be found in ANSI C books available at bookstores.

Bit Fields

A member of a structure can be defined as a bit field. A bit field defines the number of bits of storage that the member requires. A bit field is specified by:

```
basetype member_name:number_of_bits;
```

In addition to the base types **int**, **signed int**, or **unsigned int**, the Microtec C/C++ Compilers provide support for bit fields of any integral type, including **char**, **signed char**, **unsigned char**, **short**, **signed short**, **unsigned short**, **long**, **signed long**, **unsigned long**, **packed** enumerated types, or **unpacked** enumerated types. Storage is allocated according to the base type. The minimum size of any bit field or group of bit fields of the same base type is the *number_of_bits* in the *basetype*. The maximum size of a single bit field is also the number of bits in its base type.

Example 11-2. Declaring Bit Fields

```
struct status {
    unsigned control:3;
    int data:13;
} pvar;
```

Bit fields in unpacked structs are stored in a base type, starting at higher order bits in big-endian mode and at lower order bits in little-endian mode. If a successive field element does not fit into the remaining space of the current base type, a new byte, half word, or word is allocated (according to the base type). The field member is placed into the new base type, starting at higher order bits in big-endian mode and at lower order bits in little-endian mode.

Bit fields in packed structs are stored in a base type in the same way, but if there is not enough room for a bit field in the current base type, it is partially allocated to the current base type and partially to a new base type. However, if a bit field allocated in this way overlaps four or more byte boundaries, padding is added to fill out the current base type and the bit field is allocated to a new base type. In both packed and unpacked structs, field members of zero length are not allocated any space. To fulfill the requirements of the base type of the zero-length bit field, the field members cause the current allocation unit to be completed and the next byte to be allocated so that they will be properly aligned. When the size of the base type of the next bit field differs from the size of the current base type, padding is added so that the next bit field is aligned to its base type boundary. Padding is also added if the next element is not a bit field, regardless of the type of element. Changing the type sign in a sequence of bit field declarations does not cause padding to be added.

Both **signed** and **unsigned** bit fields are implemented. **Signed** bit fields are handled the same way as **signed** data. Changing the sign in a sequence of bit field declarations does not introduce a new word.

Note

A **signed** bit field that is one-bit wide can only have the values 0 and -1, since the high-order bit is considered to be the sign bit.

Example 11-3. Bit Field Allocation

```
struct {  
    char bf1:5;  
    short bf2:3;    /* size changed, pad to next half word */  
    int bf3:2;      /* size changed, pad to next word */  
    signed int bf4:7;  
} s1;
```

Figure 11-4 shows the little-endian bit field allocation for the preceding code.

Figure 11-4. Sample Little-Endian Bit Field Allocation (Over 8-Bytes)

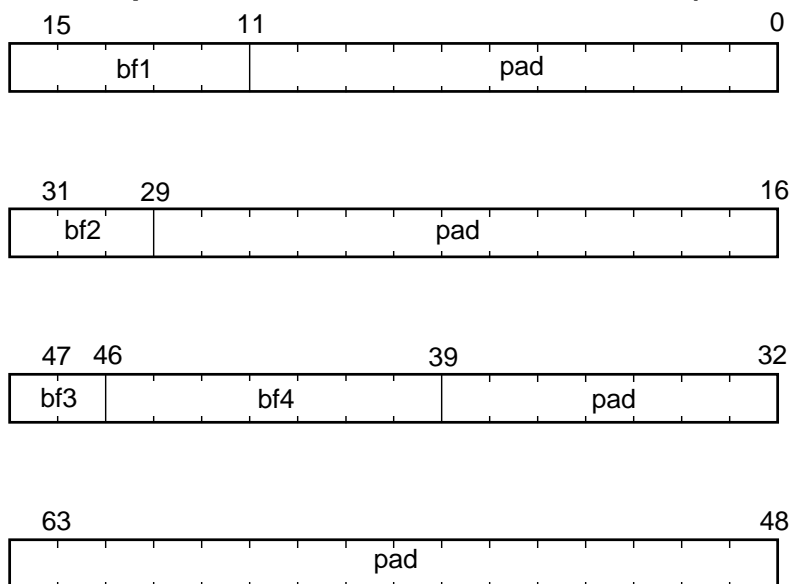
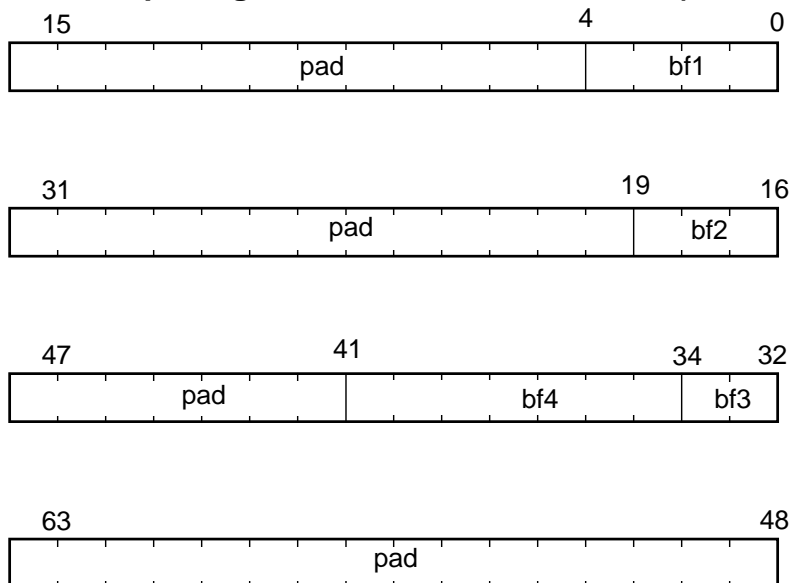


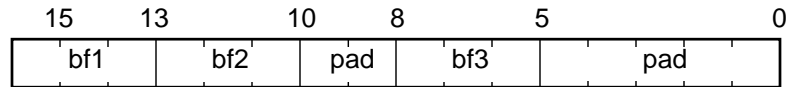
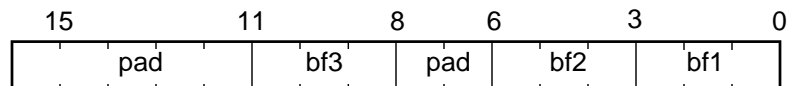
Figure 11-5 shows the big-endian bit field allocation for the preceding code.

Figure 11-5. Sample Big-Endian Bit Field Allocation (Over 8-Bytes)



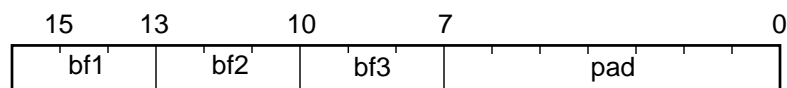
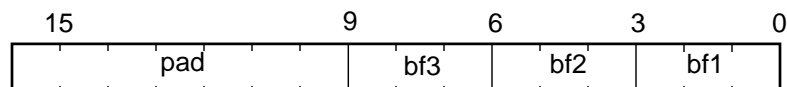
```
struct {
    char bf1 : 3;
    unsigned char bf2 : 3; /* changed sign */
    char bf3 : 3; /* does not fit in current
                  * byte */
} s2;
```

Figure 11-6 shows the bit field allocation for this code in both big- and little-endian modes.

Figure 11-6. Sample Bit Field Allocation (Over 2-Bytes)**Big-Endian:****Little-Endian:**

```
packed struct {  
    char bf1:3;  
    unsigned char bf2:3;  
    char bf3:3;  
} s3;
```

Figure 11-7 shows the bit field allocation for this code in both big- and little-endian modes.

Figure 11-7. Sample Packed Bit Field Allocation**Big-Endian:****Little-Endian:**

Allocation of Variables

Local and static variables are discussed in the following sections.

Register

The register storage class is used for local variables only. For each function, C++ allocates as many variables as possible to the hardware registers.

Local Variables

The compiler puts local (automatic) variables into registers even if a **register** declaration is not used. Eight, sixteen, and thirty-two bit variables can be allocated into registers. The lifetime of each local variable is examined by the compiler; several variables can share the same register in one routine, if possible. A variable allocated to a register always resides in that register. However, since other variables can share the register, the value of the register might not always contain the value of the variable in question.

Variables that are declared in **register** declarations are allocated to registers before nonregister variables. This allocation allows variables to be specified in their order of importance.

Static Variables

All static variables are initialized to zero automatically unless they are explicitly initialized to a value other than zero. Memory might not be allocated to static variables that are never used.

Uninitialized static variables are allocated in the **.bss** section by default. This section is cleared at program startup time. See Chapter 3, “[Using Command Line Options](#)” for options that can alter this behavior.

Memory Allocation

Memory allocation varies, depending on the type.

Big-Endian

For big-endian setup, memory is byte-addressed, with the least significant byte at the highest address (high to low ordered). Bits are numbered with bit zero as the least significant bit.

Little-Endian

For little-endian setup, memory is byte-addressed, with the least significant byte at the lowest address (low to high ordered). Bits are numbered with bit zero as the least significant bit.

Chapter 12

Run-Time Organization

This chapter describes the memory organization of C++ programs for PowerPC family microprocessor-based systems.

Code Organization

The CCCPPC C++ Compiler places each variable and function into a specific section. [Table 12-1](#) shows the sections that are generated by the compiler.

Table 12-1. Sections Generated by the Compiler

Section Name	Contents
.text	Program code
.rodata1	String literals
.rodata	Compiler-generated data and explicitly initialized const variables
ioports	Simulated I/O ports used with the EDGE Debugger
.data	Explicitly initialized non- const variables
.bss	Uninitialized variables
stack	Stack pointer initialized to point to the end of the section
heap	Heap pointer initialized to point to the start of the section

If a variable is not initialized in any module, it is treated as a “C common.” The linker allocates “C common” variables in the **.bss** section; it allocates uninitialized static variables in the **.bss** section.

Compiler-Generated Sections

This section provides a short example to illustrate how the compiler generates sections. The example uses two modules: *module1.c* and *module2.c*. These modules are compiled and assembled to produce *module1.o* and *module2.o*. Finally, they are linked together to produce an executable file.

Table 12-2 shows the contents of *module1.c* and *module2.c*.

Table 12-2. Example Modules

module1.c	module2.c
<pre> int a = 0; int b; static c; const ten = 10; main () { int d; func ("hello"); /* ... */ }</pre>	<pre> int a; int b; static c; func (char * string) { /* ... */ }</pre>

Table 12-3 shows the contents of each section in *module1.o* and *module2.o*.

Table 12-3. module1.o and module2.o Sections

Section Name	module1.o	module2.o
.text	main	func
.rodata1	"hello"	
.rodata	ten	
.data	a	
.bss	c	c

In *module1.o*, **b** is an unresolved reference that has not been allocated to any section. The section for **b** cannot be determined until linking; it can eventually reside in the **.rodata**, **.data**, or **.bss** sections, or in a different user-defined section. Similarly, in *module2.o*, **a** and **b** are unresolved references that have not been allocated to any section. The variables **d** and **string** are not allocated to any section. They are accessed through the stack at run-time. The two modules are linked to create an executable file. The linker combines all sections of the same name together.

Table 12-4 shows the sections contained in the resulting executable file.

Table 12-4. Executable File Sections

Section Name	Contents
.text	main, func
.rodata1	"hello"
.rodata	ten
.data	a

Table 12-4. Executable File Sections (cont.)

Section Name	Contents
.bss	b , c , c

The two static variables named **c** remain separate and are not linked together. Since **b** was not present in any object file section, the linker placed it into the **.bss** section.

The following sections describe the compiler-generated sections in more detail.

.text Section

The **.text** section contains all program code. All code is assumed to be read-only and, therefore, can be safely placed in ROM.

.rodata1 Section

The **.rodata1** section contains the contents of all string literals. The **.rodata1** section can be safely placed in ROM if your application does not attempt to modify string literals.

.rodata Section

The **.rodata** section is used for external or static variables that are explicitly initialized. These variables are declared using the **const** keyword. It also contains any compiler-generated literal data.

In some cases, scalar **static const** variables might not be placed in this section. In this case, the compiler would not allocate any storage for them and would substitute a constant value each time they are used. This generally happens for very small values.

You can safely place the **.rodata** section in ROM if your application does not attempt to modify **const** variables, then . Direct assignment to **const** variables is prohibited by the compiler but can be achieved by using type casting or by using **const** inconsistently across different modules.

.data Section

The **.data** section is used for all static and external variables that are explicitly initialized. These variables are declared without the **const** keyword. Typically, this data is placed in RAM.

ioports Section

When a program uses the **read** or **write** functions, the **_simulated_input** and **_simulated_output** variables used to facilitate I/O simulation with the EDGE Debugger are placed in the **ioports** section.

.bss Section

The **.bss** section is used for all uninitialized external or static variables. These variables are initialized to zero when the program is loaded into memory or when execution starts.

The entire variable is considered to be initialized and is placed in the **.data** section if a portion of a variable is explicitly initialized. For example, in the following declaration:

```
char buffer [1024] = {0};
```

the first element of **buffer** is explicitly initialized, so this variable cannot be placed in the **.bss** section.

Variables in the **.bss** section are initialized to zero when the program is loaded into memory or when execution starts. It does not occupy any physical space in the executable file because the **.bss** section contains only uninitialized variables. The allocation and initialization of **.bss** variables depends on the presence of a program loader, as follows:

- The loader must allocate memory space for the **.bss** data if the operating system has a program loader. Some program loaders initialize this memory to 0. Your startup routine should initialize this memory to 0 if your loader does not provide this service.
- Your program code must be permanently resident in ROM, and the **.bss** data must be allocated to an address space within RAM if the operating system does not have a program loader. Your startup routine should initialize the **.bss** memory to 0 each time the program is restarted.

Chapter 13

Embedded Environments

This chapter provides tips on using the Microtec C and C++ compilers to write embedded applications. It includes information on the following:

- Generating reentrant code
- Modifying the system functions and initialization routines included in the distribution
- Initializing data
- Using the linker to initialize program variables in RAM at start-up time
- Creating a linker command file

Embedded Applications

An embedded system consists of a microprocessor and its associated peripheral hardware and software. The software provides all operating system functions such as input/output, memory allocation, scheduling, and interrupt handling.

Typically, an embedded system does not have a standard operating system, run-time loader, or disk. Instead, its program code and constants usually reside permanently in ROM. The program starts execution at a memory location determined by the hardware whenever a RESET occurs (at power-up).

Considerations for Embedded Systems

Consider the following when working with an embedded system:

- The hardware starting address must coincide with the main entry point (or code that ultimately jumps to the main entry point). For this reason, the sample initialization routine in the *entry.c* file might need to be modified. If any modifications are made, *entry.c* must be recompiled, and the resulting object module must be linked with your program. (The new version must be loaded ahead of the Microtec C library in the linker command file; alternatively, the old version should be replaced by the new one in the library.) See the “[User-Modified Routines](#)” section in this chapter for information on the initialization routine and how to modify it.
- Since an embedded environment typically has its own unique memory organization and I/O system, it may be necessary to customize several run-time library routines in order to accommodate the environment. Routines that might need modification include I/O

routines (**read**, **write**), memory allocation routines (**malloc**, **calloc**), and system functions (**open**, **close**).

- Initialized variables can be placed either in ROM or RAM or initialized by the application at run time. Variables placed in ROM cannot be altered by the program. The “[Data Initialization](#)” section in this chapter discusses this topic in detail.
- You must modify several routines to adapt them to your embedded environment to use the C or C++ I/O and memory allocation library. See the “[User-Modified Routines](#)” section in this chapter for more information.
- Interrupt handlers written in assembly language must save and restore all registers they use. These routines can call C or C++ functions as described in Chapter 10, “[Interlanguage Calling](#)”.

Inline Assembly

Embedded applications require close and efficient control of the target architecture. The Microtec C and C++ compilers support assembler statement intermixing with C or C++ code to accommodate this need.

You can include any number of assembler lines by using the pseudofunction **asm** (which can also be specified as **ASM**).

Syntax

```
asm([type[,]] [string [,string] ...]);
```

Description

type Tells the compiler what is returned by the **asm** invocation and, implicitly, where it is returned. The value returned by the **asm** pseudofunction is of type **int** by default. The **asm** pseudofunction cannot return **structures** or **unions**.

string Represents assembler instructions. Generally, each *string* separated by a comma corresponds to a new assembler line.

Insert as many assembler instructions as required using a single **asm** statement, using the following rule:

- Since each assembler statement must be on its own line, use either a separate string for each assembler statement or **\n** to generate a newline character.

The **asm** pseudofunction behaves from the outside exactly like a function (it takes parameters and returns a type). However, **asm** does not generate a procedure call. Instead, it inserts your assembler code "inline".

Like any C function, **asm** can be invoked with or without taking into consideration its returned value. The global optimizer in the compiler disables certain optimizations based on the returned value.

The value of the **asm** pseudofunction is the result placed in the return value register. The simplest way to identify the return register is to write a small function that returns a value of the desired type. Compile this function using the **-S -Fsm** options and take note of the register or registers used to return the value.

The compiler turns off the recognition of register symbols in order to avoid collisions with the C/C++ symbol space. Either registers must be accessed by numeric values (for example, use **1** to refer to **r1**), or the list of assembly statements must start with a **.lflags r** statement and end with a **.lflags nr** statement to temporarily turn on register symbol recognition. It is also possible to avoid warnings from the assembler for the use of system-level instructions by surrounding any assembly statements with **.supervisoron** and **.supervisoroff** statements. These statements should only be used if you are certain that the code will be executed only while in system mode. These directives do not prevent exceptions from occurring if such system-level assembly instructions are executed in user mode. The use of labels with statements in the **asm** pseudofunction can result in multiple definitions of that label, due to function inlining. Because of this, labels should be avoided in **asm** pseudofunctions.

Follow these guidelines when you add **asm** pseudofunctions to your code:

1. Write your C program.
2. Compile the program.
3. Examine those places in the assembler output where you are considering adding **asm** pseudofunctions.
4. Add assembly code (also known as “inserts”) to **asm** pseudofunctions within procedures.
5. Use **#pragma asm** and **#pragma endasm** to add assembly code outside procedures.

The following examples demonstrate how to use **asm** to put assembler lines into a C program.

```
main()
{
    asm(" addi 1,0,0x1000 # initialize stack pointer");
    ...
}
```

In this example, the compiler inserts the quoted string immediately after the prologue code that it generates for **main()**. A newline character is added after the string to avoid concatenating it with the next line.

The return type is **int** because no type was specified in the above **asm** statement, . The return type is ignored since the return value is not assigned to any variable.

```
asm(char*, " addi 3,0,0 " ) [100]='*';
```

This example shows how to impose a char array view on physical memory.

```
asm( " ori 0,0,0", " sync");  
asm( " ori 0,0,0\n sync" );  
asm( " ori 0,0,0 ", " s" "ync " );  
/* no ', 'after s- same as " sync " */  
asm( " ori 0,0,0 " ); asm( " sync " );  
asm(int, " ori 0,0,0", " sync");
```

These statements all generate the following code:

```
ori 0,0,0  
sync
```

The following compiler supplies an End-of-Line character at the end of every string argument. Thus, in the statement:

```
asm( " ori 0,0,0\n", " sync");
```

the `\n` is unnecessary.

Because the compiler passes the string arguments of **asm** directly to the assembler, it does not perform syntax checking on your assembler statements. Be careful when using the **asm** pseudofunction.

Assembler Inlining Features

This section describes **asm** features and provides examples.

Assigning asm to a Variable

You can assign the return value of **asm** to a variable, as you can with any other C function. The following example determines the value of the stack pointer.

```
sp_reg=asm(" ori 3,1,0 # get value of sp register");
```

The preceding **asm** statement returns an integer (the default type). According to the calling conventions, integers (like other scalars) are returned in register r3. Since **asm** behaves like a function, the compiler generates code to move the returned value to the target.

The compiler produces the following sequence in the assembly file:

```
ori 3,1,0# get value of sp register  
addis 4,0 ha(sp_reg)  
stw 3,lo(sp_reg)(4)
```

Returning a Typed Value

Use the type argument to **asm** to tell the compiler what is returned by the **asm** invocation and, implicitly, where it is returned.

```
int *sp=asm(int *, " ori 3,1,0");
```

In this example, since the compiler knows a pointer is returned in r3, the compiler will, after the pseudo-invocation of **asm**, generate code to move r3 to the variable SP.

```
double d=asm(double, " fmr 1,9");
```

No real call to **asm** takes place. The passing of parameters and the call itself are replaced by the string(s) you supply. Store in r3 (or whatever the calling conventions dictate) the value that interests you.

Using #define for Readability

You can make **asm** statements more readable with the **#define** directive.

```
#define R11 asm(int, " ori 3,11,0")
#define R12 asm(int, " ori 3,12,0")
if (R11>R12)
...
```

This example shows that it is possible to access every machine device at the C level.

Variable Names Inside asm

You can insert local variables by name inside an **asm** string. However, representation of local variables at the assembler level is more complex. They are represented by frame offsets or registers that can change from one release of the compiler to the next. Their addresses can change between two compiles if new code or variables have been added to a procedure.

The Microtec C compiler lets you insert variable names in any **asm** string by marking them with back quotes (`). Since it is unlikely that this character will appear in any assembler statement, it has been chosen as the default. Specify the **-uichar** option to change the insert character to char to use a character other than the back quote. You can specify this option within a **#pragma options** statement to ensure that the insert character does not change between compilation.

Example 13-1. Using Variables Within asm()

```
/* Read/write of time base in supervisor mode */
void get_timebase (unsigned int *hi, unsigned int *lo) {
    unsigned int up, low;
    asm("l: ",
        " mftbu 3    #load from TBU",
        " mftb  4    #load from TBL",
        " mftbu 5    #load from TBU",
        " cmpw  5,3  #see if old-new",
```

```
        " bne 1b      #loop if carry occurred",
        " MOVE_LOCAL 5,`up`  # move TBU to up",
        " MOVE_LOCAL 3,`low` # move TBL to low");
*hi = up;
*lo = low;
}

void set_timebase (unsigned int hi, unsigned int lo) {
    asm(" MOVE_LOCAL `hi`,3 #load 64-bit value for",
        " MOVE_LOCAL `lo`,4 #TB into r3 and r4",
        " li 5, 0",
        ".supervisoron",
        " mttbl 5 #force TBL to 0",
        " mttbu 3 #set TBU",
        " mttbl 4 #set TBL",
        ".supervisoroff");
}
```

The assembly output generated for the preceding C example is as follows:

```
# Microtec C/C++ PowerPC Compiler
# "Mentor Graphics ANSI_C Front End 3.5 ; Back End 3.5 "
.file"asm.c"
# Options: asm.c
.cputype"603/F"
.aloff
.endianbig
.lflagsnr
.equ._PPC_EMB_initflag,1
.sect.text
.align4
.globlget_timebase
get_timebase:
# Allocation of Local Variables
# _____
# hi = r3 ;
# lo = r4 ;
/* Read/write of time base in supervisor mode */
#void get_timebase (unsigned int *hi, unsigned int *lo) {
    stwul,-24(1)
    stw28,0x8(1)
    stw29,0xc(1)
    stw30,0x10(1)
    stw31,0x14(1)
    ori31,4,0x0
    ori30,3,0x0
    addi28,0,0x0
    ori29,28,0x0
#     unsigned int up, low;
#     asm("1: ",
1:
    mftbu 3    #load from TBU
    mftb 4     #load from TBL
    mftbu 5    #load from TBU
    cmpw 5,3   #see if old==new
    bne 1b     #loop if carry occurred
    MOVE_LOCAL 5,28 # move TBU to up
    MOVE_LOCAL 3,29 # move TBL to low
```

```
#      " mftbu 3    #load from TBU",
#      " mftb  4    #load from TBL",
#      " mftbu 5    #load from TBU",
#      " cmpw 5,3   #see if old-new",
#      " bne 1b     #loop if carry occurred",
#      " MOVE_LOCAL 5,`up`  # move TBU to up",
#      " MOVE_LOCAL 3,`low` # move TBL to low");
#      *hi = up;
#      stw28,0x0(30)
#      *lo = low;
#      stw29,0x0(31)
#}
CLRET.get_timebase:
    lwz28,0x8(1)
    lwz29,0xc(1)
    lwz30,0x10(1)
    lwz31,0x14(1)
    addi1,1,0x18
    bclr0x14,0x0
# Code=68+asm() bytes Stack=24 bytes
    .sect.text
    .align4
    .globlset_timebase
set_timebase:
# Allocation of Local Variables
# _____
# hi  = r3 ;
# lo  = r4 ;
#
#void set_timebase (unsigned int hi, unsigned int lo) {
    stwu1,-16(1)
    stw30,0x8(1)
    stw31,0xc(1)
    ori31,4,0x0
    ori30,3,0x0
#      asm(" MOVE_LOCAL `hi`,3 #load 64-bit value for",
MOVE_LOCAL 30,3 #load 64-bit value for
MOVE_LOCAL 31,4 #TB into r3 and r4
    li 5, 0
    .supervisoron
    mttbl 5 #force TBL to 0
    mttbu 3 #set TBU
    mttbl 4 #set TBL
    .supervisoroff
#      " MOVE_LOCAL `lo`,4 #TB into r3 and r4",
#      " li 5, 0",
#      ".supervisoron",
#      " mttbl 5 #force TBL to 0",
#      " mttbu 3 #set TBU",
#      " mttbl 4 #set TBL",
#      ".supervisoroff");
#}
CLRET.set_timebase:
    lwz30,0x8(1)
    lwz31,0xc(1)
    addi1,1,0x10
    bclr0x14,0x0
# Code=36+asm() bytes Stack=16 bytes
```

```
.sect.data
.sect.data
.align8
CLDATA.:
.sect.data
.align8
CLRDATA.:
.sect.rodata1
.align8
STRINGDATA.:
.sect.rodata
.align8
CONSTDATA.:
.sect.data
.sect.libinfo
.byte"bhp#603#"
```

Variables in asm and Optimization Levels

The level of optimization as set by the **-On** option controls the way in which variables in the **asm** pseudofunction are handled. The compiler may handle such a variable in different ways depending on whether it is a global variable, external variable, a local variable, a local register variable, a function parameter, and so forth.

For example, in the following statement:

```
asm("addi `id`, 0, 0x0);
```

the variable ID can be any of the previously named types. [Table 13-1](#) shows the code that the compiler would generate for the preceding **asm** statement based on the type of variable ID is and the optimization level.

Table 13-1. Variables in asm and Optimization level

	Optimization Level	
	-nOg	-Og
Local Variable	addi 0x8(i),0,0x0	addi 3,0,0x0
Local Register Variable	addi 0x8(1),0,0x0	addi 3,0,0x0
Function Parameter	addi 0x8(1),0,0x0	addi 3,0,0x0
Global/External Variable	addi id,0,0x0	addi id,0,0x0

The predefined ASMPPC macro "MOVE_LOCAL" is provided to permit correct accessing of local variables in assembly regardless of the optimization options used. In this example, consider `asm(" MOVE_LOCAL `hi`,3 #load 64-bit value for")`. If 'hi' is allocated on the stack, MOVE_LOCAL generates a load (lwz) instruction. If 'hi' is in a register, a register move (ori) instruction is generated. Several variations of MOVE_LOCAL are also available for moving small integers and floating point values. Refer to the *Assembler/Linker/Librarian User's Guide*

and Reference for the PowerPC Family "chapter 8, ("Predefined Macros" section)" for more information.

Another common problem with using `asm()` statements has to do with duplicate labels. If an `asm()` occurs within a compiler macro (for example, by using `#define`), the `asm()` may be duplicated when the macro is used. Also, if the `-Oi` inlining option is used, functions that contain `asm()` statements may get "inlined" multiple times within the same file.

Note



The `-nKia` option can be used to prevent functions with `asm()` statements from being inlined.

In these cases, if the `asm()` contains a label, it can cause the assembler to issue a duplicate label error. This problem can be avoided by using a local label, where a single decimal digit instead is used instead of an identifier. A reference is made by appending the digit with a 'b' for a backward reference, or 'f' for a forward reference. The same local label can be defined multiple times within the same file. Only the closest local label definition in either direction (forwards or backwards) can be referenced. In this example, the assembler encounters the label reference "1b" and associates it with the previous "1:" label defined several instructions earlier. This `asm()` statement can be used several times in the same file without causing problems with duplicate labels. Refer to the *Assembler/Linker/Librarian User's Guide and Reference for the PowerPC Family* "chapter 3, ("Assembler Syntax/Labels" section)" for more information.

#pragma asm or asm

Since **asm** is a pseudofunction, it can only be used where a normal function can be invoked, namely inside a procedure. Outside a procedure, any number of unquoted assembler instructions can be inserted between the pair **#pragma asm** and **#pragma endasm**. You can use these directives to create your own assembly procedure.

Note



Everything between the **#pragma asm** and **#pragma endasm** directives must be in assembly language format.

Example 13-2. #pragma asm

```
#pragma options -QmsC0096
#pragma asm
.sect .text
.align 4
.globl __mri_start

__mri_start:

    addis 1,0,9
    addi 4,0,0x0
    addis 5,0,ha(_end)
```

```
    addic 5,5,lo(_end)
    addis 3,0,ha(_edata)
    addic 3,3,lo(_edata)
    subfc 5,3,5
    bl    .memset

    ori   3,0,0x0
    mfmsr 3

#ifdef _BIG_ENDIAN == 1
    ori   3,3,0x2000
#else
    ori   3,3,0x2001
#endif
    mtmsr 3
    bl    ._start
    addi  1,1,0
    addi  1,1,0
    addi  1,1,0
    .long 0xdead

    .extern _edata
    .extern _end
    .extern .memset
    .extern ._start
#pragma endasm
```

#pragma asm is more readable for long sequences of assembly language. You can use user-defined macros as well as predefined macros inside **#pragma asm/#pragma endasm** pairs, since full preprocessor functionality is available. Assembler inserts (**asm** function) cannot be used.

The compiler turns off the recognition of register symbols in order to avoid collisions with the C/C++ symbol space. Either registers must be accessed by numeric values (for example, use 1 to refer to r1), or the **#pragma asm/#pragma endasm** sequence must start with a **.lflags r** statement and end with a **.lflags nr** statement to temporarily turn on register symbol recognition. It is also possible to avoid warnings from the assembler for the use of system-level instructions by surrounding any assembly statements with **.supervisoron** and **.supervisoroff** statements. These statements should only be used, however, if you are certain that the code will be executed only while in system mode. These directives do not prevent exceptions from occurring if such system-level assembly instructions are executed in user mode.

Considerations for Assembler Inlining

The content of any string is passed to the assembler as is. No control is placed on the string, which generates direct consequences:

- You must be ready to read errors issued by the assembler.
- The compiler makes conservative assumptions on the size of the inserted instructions. Every jump crossing the instructions might be long. Consider the following example:

```
asm(" .space 400")
```

The **-Zin** option can be used to control whether long branches are needed.

- **asm** is not portable. It is likely, however, that a program that uses direct machine instructions was never intended to be portable.

Check the optimization effects when you use **asm**. Even though some optimizations are disabled, the Microtec C/C++ compiler takes freedom in rearranging, changing, and deleting code. This behavior could affect your program.

Addressing

A variety of methods exist for accessing addresses.

Direct Memory and I/O Port Addressing

You can examine and modify absolute memory locations by defining a macro that functions exactly like a normal C++ variable, except that it is located at a particular memory address given by a number. For example, a macro called **memloc** is defined in the following example. This macro represents the contents of the byte at location 1A (hexadecimal). The following is the macro definition:

```
#define memloc (*(char *)0x1A)
```

When **memloc** appears in the program, it is replaced with the macro definition text `*(char *)0x1A`, which means “1A hexadecimal, treated as a pointer to a character and dereferenced”. If **memloc** appears in an expression or on the right-hand side of an assignment, it represents the contents of byte location 0x1A. If **memloc** appears on the left-hand side of an assignment, it represents the byte address 0x1A.

The macro **memloc** can be used as an ordinary byte variable, as follows:

```
chr1 = memloc + 1;  
memloc = 'a';
```

The macro **memloc** can also correspond to a memory-mapped I/O location, in which case **memloc** must be **volatile**, since each use would have the side effect of reading or writing:

```
#define memloc (*(volatile char *)0x1A)
```

Calling a Function at an Absolute Address

Use the **asm** pseudofunction or assembly language to call a function at an absolute address (such as a system routine). However, if you are concerned about porting your program to a different processor, use the technique described here.

This section describes how a C++ program can call an independently-compiled and linked function that is placed at an absolute address without having to use assembly language. Use this technique when an application program is required to call a system routine at a fixed ROM address.

Define a macro that declares a function to call a function at an absolute location:

```
#define ABS_FUNC (*(int (*)())0x124)
```

The macro defines ABS_FUNC to be an integer-valued function at location 0x124. You can then call ABS_FUNC just like an ordinary function:

```
i = ABS_FUNC(); /* ABS_FUNC can be in any expression */
```

The routine you are calling must have a compatible calling sequence if parameters are being passed. See Chapter 10, “[Interlanguage Calling](#)” for more information on function parameter passing and return values.

Interrupt Handlers

Interrupt handlers perform the necessary functions when an interrupt occurs. They preserve the values in various registers and storage locations and transfer control to routines to service the interrupt.

The Microtec C/C++ Compiler assumes all registers will be used in an interrupt service routine by default. The **interrupt** function calls intrinsic functions to perform **save** and **restore** operations. These **save** and **restore** routines can be tailored to the specific needs of individual interrupt functions.

A function can be declared as an interrupt handler by using the **interrupt** keyword. Interrupt functions should have no parameters and should return the **void** type:

```
interrupt void handler (void)
{
    ...
}
```

The compiler will generate the following two distinct pieces of code for interrupt functions:

- The function body

This code would be produced even if the **interrupt** keyword were not present.

- The trap table entry

This is less than sixty-four words and can be located directly into the trap table. It makes calls to routines to save and restore volatile registers and calls the **interrupt** function body.

Example 13-3. Interrupt Assembly Code

```

.sect sectname[ax]
ENT.funcname:
    addil,1,-8    # make room for regs
    stw12,4(1)    # save r12
    mflr12        # get link register
    stw12,0(1)    # .. and save
    addis12,0,ha(prefixsave) # save registers
    addi12,12,lo(prefixsave)
    mtlr12
    blrl
    addis12,0,ha(funcname) # call function body
    addi12,12,lo(funcname)
    mtlr
    blrl
    addis12,0,ha(prefixrestore) # restore registers
    addi12,12,lo(prefixrestore)
    mtlr
    blrl
    lwz12,0(1)    # restore r12 and link
    mtlr12
    lwz12,4(1)
    addil,1,8     # restore stack pointer
END.funcname:
    rfi

```

Where *sectname* is the section name. This defaults to **SEC.funcname** but is affected by the **-Ket** option. *prefix* is used to name the register **save/restore** routines. This defaults to `_interrupt_save` and `_interrupt_restore` (provided with the C libraries) but can be modified with the **-Kep** option.

r12 is used because it does not conflict with the argument registers (**r0** does not work with **addi**).

END.funcname is provided so that you will have a label that can be used to compute the table entry size. This is placed immediately before the last instruction.

If *funcname* is global, the entry code, **ENT.funcname**, is also global. *funcname* can be called and will behave like a normal function, but **ENT.funcname** will be bypassed.

A version of the **save/restore** routines you can modify is provided with the library source. Alternatively, you can modify the naming of the **save/restore** routines so they can provide your own.

A big problem with interrupts is determining which registers should be saved and restored. A conservative approach is to save and restore all volatile GP, FP, and special-purpose registers whenever there is an **asm** or function call present. This would likely include at least 90% of the interrupt functions. With this approach, you can tailor the **save/restore** code to meet specific needs (for example, if interrupt handlers do not use floating point, there is no need to save/restore FP registers). The **save** routine can also be used to generate parameters to pass to the **interrupt** function. The **-Ken**, **-Kep**, **-Kes** and **-Ket** options affect code generation for

interrupt functions. These are described in detail in “[Command Line Options - Expanded Descriptions](#)” in Chapter 3, “[Using Command Line Options](#)”.

Built-in Exception Processing

The Microtec C/C++ compiler uses exception handlers in the following situations:

- Misaligned data accesses resulting from the use of packed data structures
- Floating point emulation for the MPC405, MPC821, and other processors without an FPU

The libraries contain the necessary code to support these exceptions. These routines are loaded into the vector table through commands in the linker command file. The default linker command file assumes that the exception table is located at 0xffff0000. Changes will be required to place these at 0 or to implement a run-time copy mechanism to move the vector table from ROM into RAM. The source to all exception handlers is provided.

For users of IBM405, MPC821, and similar processors, the default linker command file must be modified to include and locate the floating point exception processing routine. Failure to do so will result in a link time error message indicating the absence of floating point support.

Reentrant Code

A routine is reentrant if it can be interrupted during its execution and reinvoked any number of times by subsequent calls from, for example, an interrupt service routine. When each subsequent call has completed its invocation, the prior invocation resumes processing where it left off, without loss of data.

The code generated by the compiler is reentrant, provided you follow these rules:

- Do not write to global variables. Also on page 13-20, in the same section, it refers to the “data” section, with no dot in front of the name (.data) should this be .data?
- Do not write to local static variables.
- Do not perform any noninterruptible tasks such as certain I/O operations.
- Use the non-reentrant library routines only in the manner described below.

For the purposes of this discussion, a thread of computation, or “thread,” is a given sequence of instructions. A thread can be interrupted by an external interrupt, and control can be given to another thread at almost any time. Multiple threads can access the same executable code at any given time. An interrupt service routine, while it is active, is also a thread of computation.

Interrupt service routines, when they are interrupted and reentered by another interrupt, constitute multiple threads of computation. In a multiprocessing or multitasking environment,

each process or task is a thread of computation. A reentrant routine is one that functions correctly, even though multiple threads can be executing the routine simultaneously.

In general, the I/O routines declared in *stdio.h* are non-reentrant, because they modify elements of the array `_iob`, which is declared in *stdio.h*. However, the three string I/O functions **sprintf**, **vsprintf**, and **sscanf** are reentrant. To make file I/O reentrant, each task should allocate its own file structure buffer on its stack instead of calling **fopen**. The following code can be used for writing formatted output:

```
i = sprintf(buff, format_string, arguments);
write(1, buff, i);
```

where:

sprintf	Returns the number of characters written.
<i>i</i>	Contains the value of the number of characters written.
<i>buff</i>	Is an area of memory where the formatted output is temporarily written.
<i>format_string</i>	Is a printf format string.
<i>arguments</i>	Is a list of arguments used by <i>format_string</i> .
write	Outputs <i>i</i> bytes from <i>buff</i> to the output port.
1	Causes write to write to stdout .

There is another approach to using I/O routines when reentrancy is required. If each thread performs I/O through a unique FILE * or “stream,” the I/O routines can be considered reentrant, as each thread is modifying its own unique element of the `_iob` array. However, because `fopen` must scan the `_iob` array to find an unopened element, each stream must be opened in the application’s startup code before individual threads are activated.

Functions that modify **errno** are also non-reentrant. Many of the functions declared in *math.h* modify this variable, along with the **strtod**, **strtol**, **strtoul** functions and others. If a multi-threaded environment is required, the easiest way of using these routines is simply not to use or modify **asm** in any user-written code.

Most C++ library functions are non-reentrant. As a general rule, any function that uses the I/O stream class or new and delete is non-reentrant.

Non-reentrant functions can be used in a multi-threaded environment; however, only one thread can call any group of functions accessing the same static variable. For example, only one thread can call the **malloc**, **calloc**, **zalloc**, **realloc**, and **free** functions. These routines can be called by multiple threads only by using one of the techniques listed in the following sections.

Position-Independent Code and Data

This section describes position-independent code and data and how they relate to the Microtec C and C++ compilers, the C and C++ programming languages, and the PowerPC family of microprocessors.

The compiler generates references to absolute code (branches are relative) and data by default. However, the **-Mcx** command line options can be used to instruct the compiler to generate references to position-independent code. These options are described in [Table 13-2](#).

Table 13-2. Position-independent Code Options

Option	Meaning
-Mcr	The compiler uses relative branches if the target is known (true for local branches and most extern calls) and absolute addresses loaded to the CNT register for indirect calls. (default)
-Mcp	The compiler generates a code fragment in each function prologue that calculates the offset of the current location of the code from the original link address. Use of this option does not require any additional setup on your part, but it does necessitate that the entire code segment be relocated by the same amount.
-Mcn	The compiler assumes that the value held in register <i>n</i> (which must be in the range of 14 – 30) is the offset between the original link address for the code and the actual relocated code for the current module. Use of this option requires you to modify <i>entry.c</i> to ensure that register <i>n</i> is initialized with the correct value. The user-defined code that performs or recognizes the relocation should set the register containing the offset.

[Table 13-3](#) shows the options that affect position-independent data.

Table 13-3. Position-independent Data Options

Option	Meaning
-Mda	Data accesses use absolute addresses. (default)
-Mdn	Data accesses are relative to register <i>n</i> (which must be in the range 14 – 30).
-arc	ROM data is addressed according to the -Mdx option. This applies to both .rodata and .rodata1 options. (default)

Position-Independent Versus Position-Dependent

Position-independent code and data references are used for multitasking systems and other programs that are loaded at different addresses for different executions. A program that is

position-independent can be loaded and run by the operating system at any available memory location.

This position-independent flexibility may not be necessary for a stand-alone program targeted for a specific CPU board and memory configuration, since such a program would always be loaded at the same memory location.

The effective address is generated by adding a displacement (or offset) to the base address. This procedure for generating the effective address works because the relative positions of instructions and data in a program remain the same even though their absolute addresses can change each time the program is loaded.

Programming Considerations

Keep the following rules in mind when generating position-independent code:

- Do not call or jump to absolute addresses within the program (for example, by using the `asm()` pseudofunction).
- Do not use pointers initialized to absolute code addresses within the program.
- Do not position a data reference and its target far apart. The target of a data reference must be in the range of -32768 to 32767 bytes. This limitation is a result of the fact that the address operand of typical **LOAD** and **STORE** instructions is allocated a total of 16 signed bits.

Keep the following rules in mind when generating position-independent data:

- Do not make data references to absolute references within the program.

The following code cannot use register-relative data addressing because it uses an absolute data address:

```
read_from(0x02); /* read from data address 0x02 */
```

- Do not use pointers initialized to absolute data addresses within the program.

The following code fragment cannot use register-relative data addressing because it initializes pointers to absolute data addresses:

```
/* pointers initialized to absolute addresses of strings */
char *table[3] = {"table", "of", "strings"};
/* pointer initialized to absolute data address */
char c;
char *p_c = &c;
```

- Do not position a data reference and its target too far apart. The target of a data reference must be in the range of -32768 to 32767.

For example, an instruction at address 0, whose displacement is at address 2, can only reference an address up to 32769, but an instruction at address 5000, whose displacement is at address 5002, can reference an address up to 37769.

Assembler and Linker Considerations

The assembler and linker automatically handle position-independent code, so no special command-line options need to be specified. When code is made position-independent, the run-time linker adds the address assigned to the first instruction to all of the load addresses in the executable, allowing the program to execute at those new locations.

Absolute Versus Register-Relative Addressing

Refer to “[Generate Position-Independent Code and Data](#)” in Chapter 3, “[Using Command Line Options](#)” for an example showing the difference between the various types of code addressing.

User-Modified Routines

This section describes the routines you must modify when using the Microtec C/C++ compilers in certain environments. Sample versions of these routines are provided on the distribution, but it is assumed that these sample routines will either be modified extensively or used only as examples for writing your own routines.

The first part of this section covers the routines required for users running on either an embedded system or a non-UNIX system.

User-Modified Routines for Embedded Systems

When running on a non-UNIX or embedded system, the routines **read**, **write**, **entry**, **_START**, and the functions **sbrk** and **_exit** must be modified to work with your embedded system environment. These routines are described in the following sections. Sample versions of these routines and functions are provided in distribution files, will work with the EDGE Debugger, and are sufficient for initial debugging in that environment.

Start Routine

The start routine (**_START**) does the following:

- Initializes **_randx**
- Completes initialization of the heap
- Initializes the I/O system
- Opens the standard files **stdin**, **stdout**, **stderr**, **stdaux**, and **stdprn**

- Initializes global variables
- Places **_simulated_input** and **_simulated_output** in the **ioports** section (this section should be located in RAM)
- Optionally calls the **initcopy** routine for copying initialized data in the **.data** section from ROM to RAM (see the “[Using the Linker](#)” section in this chapter for a description of the LNKPPC **INITDATA** command)
- Calls your **main** function with the three arguments **argc**, **argv**, and **_environ**
- Calls **exit** when your **main** function returns (the **exit** function flushes and closes all opened files and then calls **_exit** to terminate the program)

Exit Routine

A simple version of the **_exit** routine is included in the *entry.c* file. The exit routine is implicitly called at the end of each **main** function to return control to the operating system. In this implementation, exit closes all opened files and then calls **_exit**.

Initialization Routine

The initialization routine initializes the embedded system environment. It should meet the following requirements:

- Be written in assembly language
- Set up values for the stack pointer
- Provide a static variable for the heap pointer
- Call the **_START** routine

The sample initialization routine (*entry*) is the main entry point. It does the following:

- Initializes the CPU for little-endian execution (when the **-nKE** option is used)
- Initializes **r1** to point to the end of the **stack** section
- Initializes **HEAP** as the starting address of the **heap** section, which is located after all other sections
- Initializes the frame pointer
- Clears the **.bss**, **.sbss**, and **.sbss2** sections
- Initializes the trap table entries for misaligned data access
- Initializes the trap table entries for FPU emulation (for processors without an FPU)
- Calls the C initialization routine **START**

Heap Management Routine

The **sbrk** function keeps track of the heap's growth and allocates space from the heap. It increments the heap pointer (HEAP) by size bytes and returns the previous heap pointer value in register r3. **sbrk** returns -1 if there is not enough space on the heap.

Removing Unneeded I/O Support

You can reduce your code size by not linking in the full set of UNIX system functions included in the distribution, although you might still have a requirement for formatted I/O. In some embedded applications, the formatted I/O requirements are not extensive enough to involve a UNIX-style system. In other applications, the formatted I/O is necessary only during project development and is removed before the code is finished.

In these cases, have the library routine **printf** call your character output routine directly. Library routines such as **printf** implicitly write to **stdout** by passing characters from the library routines **putc** and **flsbuf** to the user-modifiable routine **write**. In a typical application, you would modify the source code provided for the **write** routine to work with the particular hardware in your system. **write** is unavailable if you do not link in the UNIX system functions.

Use one of the following two methods to allow formatted output to be sent directly to a low-level output routine:

- Use **sprintf** rather than **printf** for formatted output. The **sprintf** function does the same formatting as **printf** but places the output in a user-specified string followed by the **NULL** character. The string can then be copied a character at a time to your output routine in a simple function:

```
str_out (char *str)
{
    while (*str)
        _simulated_output = *str++;
}
```

- Rewrite the **putc** routine so that all of the output characters are available directly from **printf**. In the C library, the **printf** routine calls **putc** to output individual characters. Your **putc** routine also intercepts the characters output by other library routines that use **putc**, such as **fprintf**, **fprintf**, **vfprintf**, and **vprintf**.

```
#include <stdio.h>
#undef putc /* shut off putc macro */
volatile extern char _simulated_output;
int putc(int c, FILE *stream)
{
    _simulated_output = c;
    return(c);
}
```

Removing Unneeded Floating Point Support

If **printf**, **fprintf**, or **sprintf** is used and no floating point values are read, avoid linking in full floating point support by adding the following stub for **_ftoa** and compiling and linking it before you link in the library:

```
int _ftoa() { return 0; }
```

Alternatively, you could add the following stub for **_sfef** and compile and link it before you link in the library:

```
void _sfef() { return; }
```

This technique reduces the size of the executable code in both cases.

Data Initialization

Variable values can be initialized at run-time or compile time:

Run-Time Initialization

```
int i  
i = 15;
```

Compile-Time Initialization

```
int i = 15;
```

Code is always ROMable when variables are initialized at run time. In this example, the constant value 15 is stored in the **code** section.

The compiler allocates the variable and its value in the **vars** section when a variable is initialized at compile time. Often the **vars** section is located in RAM. This presents a problem for most ROM-based embedded systems. RAM is typically not initialized with the variable values contained in the **data** section when your program starts executing. The variable must be in ROM.

The run-time libraries provided with the CCCPPC Compiler contain initialized variables. As a result, the problem exists for both run-time library initialized variables and user-initialized variables.

Saving Initialized Variables in ROM

Many embedded programmers use the convention that only constants can be initialized on the variable declaration because RAM is typically not initialized with the expected variable values when your program starts executing. The value of the variables cannot be changed at run-time. In this case, the **.data** section and the **.text** section can be placed in ROM.

Use the INITDATA facility available in most ROM/RAM systems if you do not want variables initialized at compile time on the data declaration to be constants. This facility initializes the values of these variables when the program restarts.

Using the Linker

This section provides linker command examples, including an example using a ROM-based system. See Chapter 12, “[Run-Time Organization](#)” for a description of the default section names used by the linker.

Default Sections

See Chapter 12, “[Run-Time Organization](#)” for information about default linker sections. See Chapter 3, “[Using Command Line Options](#)” for information on locating and addressing the various **code** and data sections.

Libraries

See Chapter 5, “[Using Libraries](#)” for information about which libraries to use.

INITDATA Command

The LNKPPC Linker command INITDATA provides a method for initializing program variables in RAM at program startup. The linker creates the section **initdat** if the INITDATA command is used. This section contains the initialization values for the section(s) specified with the INITDATA command. At program startup time, the **initcopy** function copies the initialization values from the **initdata** section to the section(s) specified with the INITDATA command.

The Microtec run-time libraries contain the **initcopy** function. The **initcopy** function is called from the **START** function in *csys.c* by default. The *csys.c* file on the distribution includes a call to **initcopy**.

Perform the following steps to disable the INITDATA command:

1. Edit your linker command file and remove this command:

```
initdata .data
```

2. Either:

- Define the preprocessor macro **EXCLUDE_INITDATA**, either on the compiler command line with the **-D** option, or in your program with the following directive:

```
#define EXCLUDE_INITDATA 1
```

- Edit the *csys.c* file and remove the **#if EXCLUDE_INITDATA** preprocessor directives and their associated **#endif** directives; then recompile the file and link it before the library.

See the Mentor Graphics Assembler/Linker/Librarian Reference Manual for your system for more information on the INITDATA command.

Linker Command Example

The following linker command file creates the memory configuration shown in [Figure 13-1](#).

```

RESNUM0, 0x10000; Start linking above 0x10000

INITDATA.data
ORDER .text
ORDER .init
ORDER initdat
ORDER .data
ORDER .bss
ORDER ioports
ORDER heap
ORDER stack

; ENT._unaligned_trap may be needed if the program contains
; packed structures.References to elements of packed
; structures are often at misaligned addresses. The PowerPC
; architecture can handle integral misaligned accesses when
; in big-endian mode, but other misaligned accesses cause an
; Alignment Exception to occur.

EXTERNENT._unaligned_trap
ORDER SEC._unaligned_trap=0xFFFF00600

; Uncomment the EXTERN below if the floating point simulation
; routines should be linked into the application. Uncomment
; the appropriate ORDER command to locate the routine in
; memory.The 603 and MPC860 processors have different memory
; locations where the code should be located.

;EXTERNENT._FPE_main
;ORDERSEC._FPE_main=0xFFFF00800; 603 vector location
;ORDERSEC._FPE_main=0xFFFF01000; MPC860 vector location

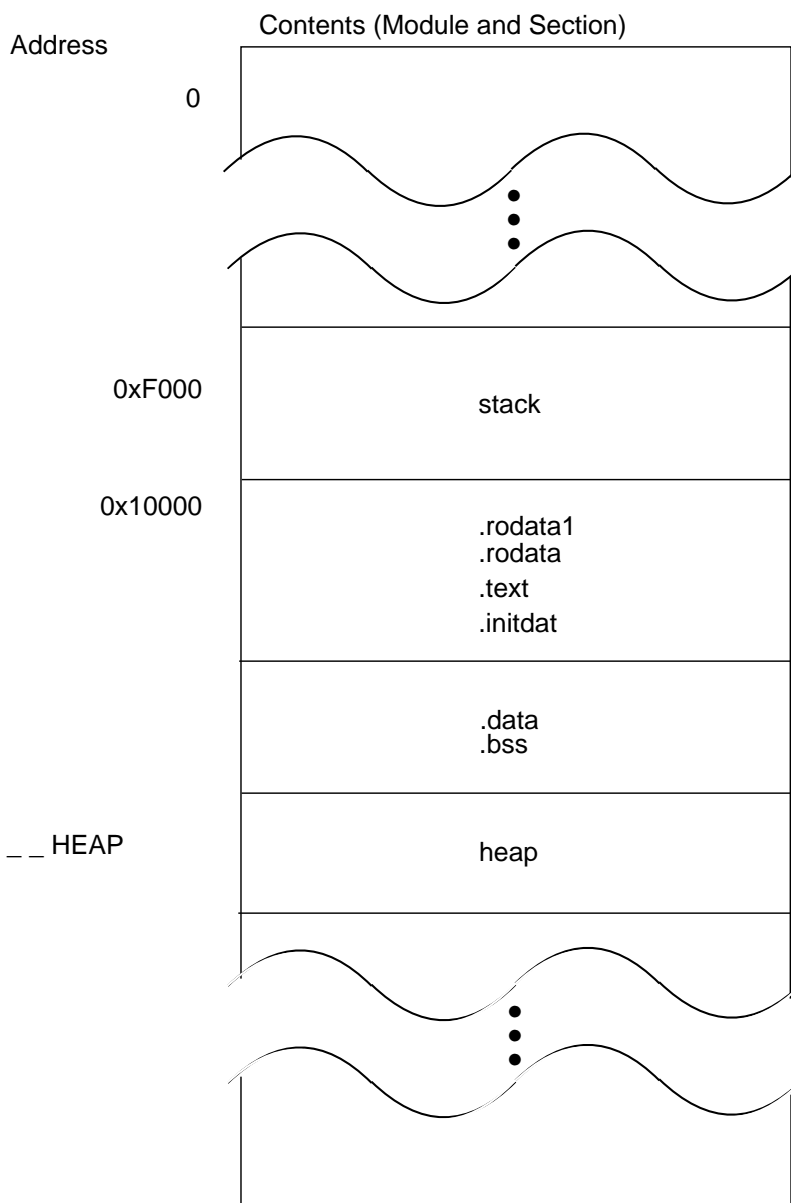
; Specify starting address
EXTERN __mri_start
START __mri_start

```

The compiler places code and data in the sections described in the “[Default Sections](#)” section in this chapter. The example assembly module ENTRY uses section **heap** for the heap.

The linker begins assigning memory at address 0 by default, which is unsuitable for many real applications.

Figure 13-1. Memory Configuration



Linker Command Example (ROM-Based System)

In some applications, such as ROM-based dedicated systems, code, data, and stack sections are to be assigned to preallocated areas of the processor's address space. In the following linker command file, it is assumed that the literals section is to be placed in the ROM area starting at

F000, the zerovars section is to start at location 2000, the vars section is to start at location A000, and the stack is to reside in the 0 to 2000 region.

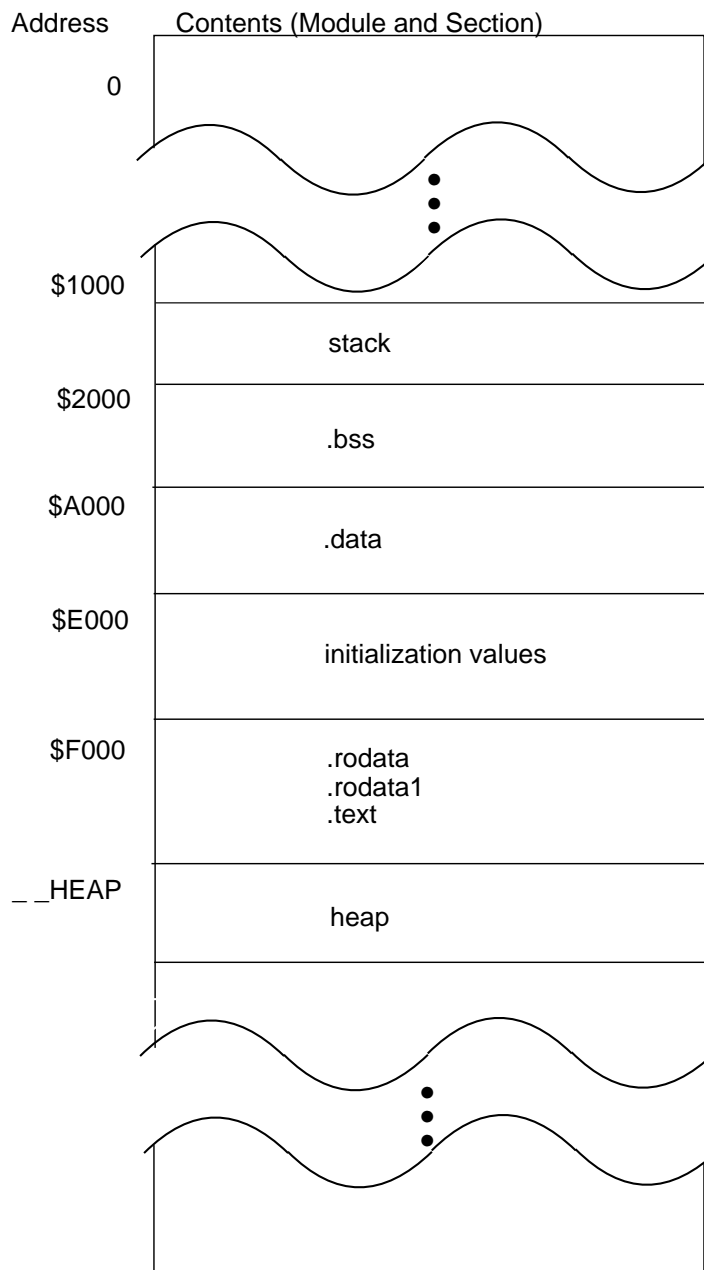
```
extern ENTRY    ; Force load of
                ; initialization routine
;
order stack=0x1000 ; Stack section
order .bss=0x2000,.data=0xA000,.initdat=0xE000
order .rodata=0xF000,.rodata1,.text
initdata .data ; Put init values in .initdat
load entry    ; Main entry point
load sieve    ; User-application routine
load mppchb.lib ; C run-time library
end           ; End of command file
```

Figure 13-2 shows this memory configuration.

The initialized variables are allocated in section .data, and their initialization values are allocated in the .initdat section, which is placed at E000. Call **initcopy** from the entry function to copy the initialization values from .initdat (ROM) to .data (RAM).

Further adjustments to the starting addresses might be necessary to ensure sufficient space for each program section if the allocation map generated by the linker is used.

Figure 13-2. Memory Configuration (ROM-Based)



Static Initialization and Termination

C++ static objects must be initialized during program startup or before the first reference to the static object. C++ static objects also must be destroyed upon program termination in the reverse order of their construction or initialization. This section deals only with C++ code because constructors and destructors do not apply to C.

Static Constructor

The C++ compiler generates the following code to ensure proper construction (initialization) of the static objects:

- A static constructor routine that calls constructors or performs necessary initialization code to construct each static object in a given file
- A function pointer to the static constructor in the predefined section **init** for the C++ run-time library to call at program startup time

All static constructors are typically prefixed with `__sti__`. One static constructor is generated on a per-file basis. A static constructor is not generated if the given file has no static objects.

Static Destructor

The C++ compiler emits the following code and data to ensure proper destruction of the static objects:

- A static destructor routine for each static object in the file.
- A data record for each object. The record contains a pointer to the data record of the next object to be destroyed, the address of the object, and the address of the destructor.
- A function call to `__record_needed_destruction` passing the address of the data record for each object requiring destruction.

The `__record_needed_destruction` routine maintains a linked list of records for each object that requires destruction. Object destruction is performed in the reverse order of construction.

init Section

The `.init` section in an object file contains addresses of each `__sti__` function; these functions must be called to initialize the static data of a given file. The linker combines the `.init` sections from all files in a program into a single `.init` section. Consult the linker command file for your compiler to see the exact placement of the `.init` section (this section is usually placed with other constant data).

If you do not want to use the default linker command file or you want to load sections in a different order, specify this information in your linker command file. Specify the `.init` section to ensure enough space is allocated in ROM if this section is produced.

Startup and Exit Routines

The startup routine `__mri_start` is invoked to initialize library globals and open standard files at the beginning of run-time and before the main routine of your program is called. `main` calls

the `_main` routine before any application code is executed. At this time, all of your static constructors are initialized, and the C++ I/O layer level 1 (referred to as layer 1+) is initialized.

The standard exit routine is invoked from the compiler library when you exit your program. This routine calls the C++ routine `__process_needed_destructions`, which calls all static destructors to deinitialize all C++ static objects in your program.

I/O Static Initialization and Termination

The C++ static I/O objects **`cout`**, **`cin`**, **`cerr`**, and **`clog`** are defined in the C++ I/O stream class libraries. The I/O channel for **`cout`** is tied with **`stdout`**, **`cin`** with **`stdin`**, and **`cerr`** with **`stderr`**.

I/O channel connections between `stdout/stdin/stderr` and `cout/cin/cerr` are part of the static initialization of `cout/cin/cerr` objects. If, in your program, you use static I/O stream objects such as `cout` in the startup code of your application, static constructors for the I/O stream objects are invoked to initialize the I/O channels, buffers, states, and so forth for the I/O static objects. As your application terminates, all corresponding static destructors are invoked to flush the I/O buffers before the I/O channels are closed.

Chapter 14

Compiler Output Files

Assembly Source File

The Microtec C/C++ Compiler produces an assembly source file that is in Microtec-compatible and Motorola-compatible format. There are several assembler directives that can optionally pass variable name and data type information from the compiler directly to the EDGE Debugger.

Advantages of Producing an Assembly File

You can examine and optimize the generated assembly code file by hand. Be careful when modifying the assembly code; an incorrect change can produce unpredictable results.

A compiler command line option is available to intermix high-level source statements as comments with the generated assembly code in the output file. See Chapter 3, “[Using Command Line Options](#)” for more information.

C variable and line numbers are mapped into symbol names and placed in the output object module as described in the following sections.

Variable Names

The Microtec C compiler does not alter the names of external, public, and static variables.

Code and Data Sections

See Chapter 12, “[Run-Time Organization](#)”, for information on section names, placement, and method of addressing.

Compiler Output Listing

The compiler can generate an output listing file when you use the appropriate command line option (see the “[Generate Listing File](#)” section in Chapter 3, “[Using Command Line Options](#)”, for more information). When a listing is generated, the compiler assigns a line number, beginning with number 1, to each source line in your C module. When files are included in the C source module, the compiler will also assign a number, beginning with number 1, to the include file lines. When the last line of the include file has been numbered, the compiler resumes numbering the source file lines from the line prior to the included file.

The compiler output lists error messages if any occurred during compilation; an arrow points to the error in the listing. The error message is printed immediately below the line where the error was encountered.

The output listing also includes a code summary for each routine and the entire module. These code sizes are estimates since it is not known at compile time whether certain address references will require 16- or 32-bits.

Generating an Executable Program

The CCCPPC Compiler allows you to take a source file and produce an object module. To build a program, assemble and then link the object module to produce an executable file. Do this for the sample program *sieve.cc* as follows:

1. Compile the sample program by entering:

```
cccppc -c -lsieve.lis -g -o sieve.o sieve.cc
```

This command compiles *sieve.cc* and produces the relocatable object module *sieve.o*. The **-l** option produces the listing file *sieve.lis*, and the **-g** option tells the compiler to include debugging information in the file. The **-c** option (used here for the purpose of illustration) prevents linking.

Note



This example uses the **-c** compiler option to prevent linking so that all compilation steps can be shown. Normally, the command line:

cccppc -lsieve.lis -g sieve.cc

would be used to create an executable file containing debugging information and the listing file *sieve.lis*. The assembler and linker are automatically invoked by the compilation driver.

You can use the **-Vd** or **-Vi** compiler option to examine the commands that the driver uses to invoke the assembler or linker. These commands can then be used as a guide for constructing your own invocation commands.


2. The object module automatically created by the assembler must now be linked with the library files and converted into absolute format. Mentor Graphics provides a default linker command file called **m603.cmd** (or a similar name for other processors).

Link the object files by entering:

```
cccppc -o sieve.abs sieve.o -Wl,-msieve.map
```

The compilation driver invokes the linker, links the appropriate libraries, and creates the absolute object module *sieve.abs* or *sieve.x*. The **-m** linker command generates a link map and places it in the file *sieve.map*.

The absolute file *sieve.abs* or *sieve.x* is ready to be debugged using the EDGE Debugger or downloaded to an in-circuit emulator or a target system.

 **Note** You can create linker command files for your applications based on the default command file provided with your distribution files.

Invoking the Linker

When the linker is invoked, the linker automatically selects the correct libraries based on information found in object files. You can choose an alternate linker command file with the **-e** option (see the “[Pass Command File to Linker \(Driver Option\)](#)” section in Chapter 3, “[Using Command Line Options](#)”). Use the **-q** option (see the “[Use Default Libraries for Linking](#)” section in Chapter 3, “[Using Command Line Options](#)”) to instruct the linker to include the default libraries.

See the Mentor Graphics *Assembler/Linker/Librarian User's Guide and Reference for the PowerPC Family* for more information on the LNKPPC Linker.

Appendix A

Compiler Messages

This appendix provides messages produced by Microtec C and C++ compilers. Error messages have the following format:

```
"filename", line line_num pos pos_num; (severity) msg_num msg
<line of source code with a circumflex pointing to the error position>
                        ^
```

where filename is the name of your file. The location of the error in that file is indicated by the line number (line_num) and column position (pos_num). Message severity level, described in the next section, is shown in parentheses, followed by the message number (msg_num) and the error message itself (msg). The offending line of source code is shown on the next line. A circumflex (^) points as closely as possible to the error position.

Note



This appendix contains explanations of compiler messages that can occur during the typical operation of this product. Messages that are outside the scope of this product (for example, operating system error messages) are not documented in this appendix. Refer to the *Release Notes* for additional information on error messages.

If you encounter an undocumented error message, contact Technical Support with the exact text of the message and the circumstances under which it occurred. The Technical Support representative will assist in determining the cause of the problem.

Error messages are generated with both numbers and offending source lines by default. Use the **-Qfn** option when compiling to suppress error message numbers. Use the **-Qfs** option to suppress offending source lines.

The examples in this appendix do not show the filename, line number, position number, or message number.

Message Severity Levels

Some errors that occur during compilation are fatal, in which case processing is abandoned. Other errors are nonfatal, and processing proceeds after the error is reported.

Messages are followed with a letter, in parentheses, indicating the severity type of the message. [Table A-1](#) lists the message severity levels.

Table A-1. Message Severity Levels

Type	Severity	Meaning
(F)	Fatal	Always fatal; processing aborts. Fatal errors arise from conditions that make further compilation impossible, such as missing source and include files or a premature End-Of-File (EOF). Most often, these errors can be corrected by using a more complete search path or by quickly examining the files and options used to invoke the compiler.
(E)	Error	Usually fatal, but processing continues for diagnostic purposes. In general, an error message indicates that the compiler has encountered a syntax or logical error. The compiler does not generate code after an error (E) occurs by default.
(W)	Warning	Not fatal. This message indicates code that is syntactically correct but can result in unexpected program behavior.
(I)	Informational	Informative, and often appears in conjunction with another error message. An informational message is similar to a warning message but is less likely to alter user program behavior. Informational messages are suppressed but can be displayed by the appropriate compiler option by default (see the “ Suppressing Error Messages ” section in this chapter).

Controlling Severity

Messages that display a “-D” after the message number are discretionary, and their severity level can be changed as follows:

-Qmmsg_nums

Makes messages with indicated numbers errors. For example, **-QmeC0177** makes message C0177 an error.

-Qmwmsg_nums

Makes messages with indicated numbers warnings.

-Qmimg_nums

Makes messages with indicated numbers informational.

Error Position Marker

The compiler puts a circumflex (^) either directly under the offending element or as close to the problem as possible if a line of C or C++ code contains an error:

```
1  main()
2  {
3      int i
4      printf("Hello, world\n");
5  }
```

produces the following output:

```
"example.cc", line 4 pos 2; (E) #C0065 expected a ";"
    printf("Hello, world\n");
    ^

"example.cc", line 3 pos 4; (W) #C0177-D variable "i" was declared but
never referenced.
    int i
    ^

1 Error 1 Warning
```

The source code is listed first, followed by the compiler output. This example shows how the omission of a semicolon on line 3 generates an error message with the error position marker pointing to the first statement after the error occurred.

The compiler displays the filename, line number, and position number by default. This extra information helps to identify the location of an error if it is not located immediately after the line with the error message. For example, template instantiation errors come at the end of a file.

Suppressing Error Messages

You can suppress error messages of a certain severity level by using the compiler options shown in [Table A-2](#). Fatal errors cannot be suppressed.

Table A-2. Using Compiler Options to Suppress Diagnostic Messages

Option	Suppresses
-Qi (default)	(I)
-Qw	(I), (W)
-Qe	(I), (W), (E)
-QA	All information
-Qs	Summary
-Qmsgnum	Message with the indicated message number

Only informational messages are suppressed by default. The default number of (E) messages that can be produced in every compilation is 20. This number can be changed with the **-Qnumber** compiler option. Setting number to zero (0) stops compilation if any (E) errors are detected:

```
1  array[100];
2  int *if;
```

produces the following output:

```
"example.cc", line 1 pos 1; (E) #C0077 this declaration has no storage
class or type specifier
array[100];
^
```

```
"example.cc", line 2 pos 6; (E) #C0040 expected an identifier
int *if
   ^
```

2 Errors

The following code:

```
1  foo()
2  {
3      undefined[*undefined](undefined);
4  }
```

produces the following output:

```
"example.cc", line 3 pos 2; (E) #C0020 identifier "undefined" is undefined
undefined[*undefined](undefined);
^
```

```
"example.cc", line 4 pos 1; (W) #C0117-D non-void function "foo" (declared
at line 1) should return a value
}
^
```

1 Error 1 Warning 1 Information

Although this example shows several uses of undefined (all errors), the compiler avoids multiple error messages and identifies the primary error rather than printing multiple error messages for the same error. The final error count is shown in a summary message at the end of the listing.

The next example shows the results when the summary message is not suppressed, warning and informational messages are suppressed, and the error message limit is set to 5. The compiler options are as follows:

```
-nQs -Qw -Q5
```

This code:

```
1  foo(){}
2  int vet[10;
```

produces the following output:

```
"example.cc", line 2 pos 11; (E) #C0017 expected a "]"
int vet[10;
```

```

      ^
1 Error 1 Warning      1 Information

```

The next example shows the results when the summary message is suppressed, warning and informational messages are suppressed, and the error message limit is set to 0. The compiler options are as follows:

```
-Qs -Qw -Q0
```

This code:

```

1  foo(){}
2  int vet[10;

```

produces the following output:

```

"example.cc", line 2 pos 11; (E) #C0017 expected a "]"
int vet[10;
      ^
(*) too many errors

```

The occurrence of any error terminates compilation.

Reporting Problems

Mentor Graphics performs extensive tests on its compilers. However, even with the most exhaustively tested software, you might still encounter a problem. If you find a problem, check the Release Notes first to see if there is any information on the problem. If there is no mention of the problem in the Release Notes, report the problem promptly to Technical Support. If possible, have a test case prepared.

Preparing a Test Case

A small test case that reproduces a problem can help Technical Support quickly determine a solution. Unfortunately, the complicated combinations of include files and intricate macros that comprise a program can make it difficult to isolate a problem and produce a clean test case.

Calling Technical Support

Perform the following steps to contact Technical Support:

1. Prepare a test case source file that, when compiled, exhibits the problem.
2. Add comments to your source file to indicate where the problem occurs.
3. Verify that the problem still exists.

4. Call Mentor Graphics Technical Support. The Support Center representative will make arrangements for you to deliver the test case source file.

C and C++ Compiler Messages

Table A-3 lists all possible compiler messages. Following the table are expanded descriptions and examples for messages that might not be self-explanatory.

Table A-3. Compiler Error Messages

Number	Message
C0000	unknown error
C0001	last line of file ends without a newline
C0002	last line of file ends with a backslash
C0003	#include file “filename” includes itself
C0004 ¹	out of memory
C0005	could not open source file “filename”
C0006	comment unclosed at end of file
C0007	unrecognized token
C0008	missing closing quote
C0009	nested comment is not allowed
C0010	“#” not expected here
C0011	unrecognized preprocessing directive
C0012	parsing restarts here after previous syntax error
C0013	expected a filename
C0014	extra text after expected end of preprocessing directive
C0016	“filename” is not a valid source filename
C0017	expected a “[”
C0018	expected a “)”
C0019 ¹	extra text after expected end of number
C0020	identifier “xxx” is undefined
C0021	type qualifiers are meaningless in this declaration
C0022	invalid hexadecimal number
C0023	integer constant is too large
C0024	invalid octal digit

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0025	quoted string should contain at least one character
C0026	too many characters in character constant
C0027	character value is out of range
C0028	expression must have a constant value
C0029	expected an expression
C0030	floating constant is out of range
C0031	expression must have integral type
C0032	expression must have arithmetic type
C0033	expected a line number
C0034	invalid line number
C0035	#error directive: <i>xxxx</i>
C0036	the #if for this directive is missing
C0037	the #endif for this directive is missing
C0038	directive is not allowed - an #else has already appeared
C0039	division by zero
C0040	expected an identifier
C0041	expression must have arithmetic or pointer type
C0042	operand types are incompatible (“ <i>type</i> ” and “ <i>type</i> ”)
C0044	expression must have pointer type
C0045	#undef cannot be used on this predefined name
C0046	this predefined name cannot be redefined
C0047	incompatible redefinition of macro “entity” (declared at line <i>xxxx</i>)
C0049	duplicate macro parameter name
C0050	“##” cannot be first in a macro definition
C0051	“##” cannot be last in a macro definition
C0052	expected a macro parameter name
C0053	expected a “.”
C0054	too few arguments in macro invocation
C0055	too many arguments in macro invocation
C0056	operand of sizeof cannot be a function

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0057	this operator is not allowed in a constant expression
C0058	this operator is not allowed in a preprocessing expression
C0059	function call is not allowed in a constant expression
C0060	this operator is not allowed in an integral constant expression
C0061	integer operation result is out of range
C0062	shift count is negative
C0063	shift count is too large
C0064	declaration does not declare anything
C0065	expected a “;”
C0066	enumeration value is out of “ int ” range
C0067	expected a “}”
C0068	integer conversion resulted in a change of sign
C0069	integer conversion resulted in truncation
C0070	incomplete type is not allowed
C0071	operand of sizeof cannot be a bit field
C0075	operand of “*” must be a pointer
C0076	argument to macro is empty
C0077	this declaration has no storage class or type specifier
C0078	a parameter declaration cannot have an initializer
C0079 ¹	expected a type specifier
C0080	a storage class cannot be specified here
C0081	more than one storage class cannot be specified
C0082 ¹	storage class is not first
C0083 ¹	type qualifier specified more than once
C0084 ¹	invalid combination of type specifiers
C0085 ¹	invalid storage class for a parameter
C0086 ¹	invalid storage class for a function
C0087	a type specifier cannot be used here
C0088	array of functions is not allowed
C0089	array of void is not allowed

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0090	function returning function is not allowed
C0091	function returning array is not allowed
C0092	identifier-list parameters may only be used in a function definition
C0093	function type cannot come from a typedef
C0094	the size of an array must be greater than zero
C0095 ¹	array is too large
C0096	a translation unit must contain at least one declaration
C0097 ¹	a function cannot return a value of this type
C0098 ¹	an array cannot have elements of this type
C0099	a declaration here must declare a parameter
C0100	duplicate parameter name
C0101	“xxx” has already been declared in the current scope
C0102	forward declaration of enum type is nonstandard
C0103 ¹	class is too large
C0104 ¹	struct or union is too large
C0105 ¹	invalid size for bit field
C0106 ¹	invalid type for a bit field
C0107	zero-length bit field must be unnamed
C0108	signed bit field of length 1
C0109	expression must have (pointer-to-) function type
C0110	expected either a definition or a tag name
C0111	statement is unreachable
C0112	expected “while”
C0114	entity-kind “entity” was referenced but not defined
C0115	a continue statement may only be used within a loop
C0116	a break statement may only be used within a loop or switch
C0117	non-void entity-kind “entity” should return a value
C0118	a void function cannot return a value
C0119	cast to type “type” is not allowed
C0120 ¹	return value type does not match the function type

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0121	a case label may only be used within a switch
C0122	a default label may only be used within a switch
C0123	case label value has already appeared in this switch
C0124	default label has already appeared in this switch
C0125	expected a “(“
C0126 ¹	expression must be an lvalue
C0127	expected a statement
C0128	loop is not reachable from preceding code
C0129	a block-scope function may only have extern storage class
C0130	expected a “{“
C0131	expression must have pointer-to-class type
C0132	expression must have pointer-to-struct-or-union type
C0133	expected a member name
C0134	expected a field name
C0135	entity-kind “entity” has no member “xxx”
C0136	entity-kind “entity” has no field “xxx”
C0137 ¹	expression must be a modifiable lvalue
C0138	taking the address of a register variable is not allowed
C0139	taking the address of a bit field is not allowed
C0140	too many arguments in function call
C0141	unnamed prototyped parameters not allowed when body is present
C0142	expression must have pointer-to-object type
C0143	program too large or complicated to compile
C0144	a value of type “ <i>type</i> ” cannot be used to initialize an entity of type “ <i>type</i> ”
C0145	entity-kind “entity” cannot be initialized
C0146	too many initializer values
C0147	declaration is incompatible with entity-kind “entity” (declared at line xxx)
C0148	entity-kind “entity” has already been initialized
C0149	a global-scope declaration cannot have this storage class
C0150	a type name cannot be redeclared as a parameter

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0151	a typedef name cannot be redeclared as a parameter
C0152	conversion of nonzero integer to pointer
C0153 ¹	expression must have class type
C0154 ¹	expression must have struct or union type
C0155 ¹	old-fashioned assignment operator
C0156 ¹	old-fashioned initializer
C0157	expression must be an integral constant expression
C0158 ¹	expression must be an lvalue or a function designator
C0159	declaration is incompatible with previous “entity” (declared at line <i>xxxx</i>)
C0160	name conflicts with previously used external name “ <i>xxxx</i> ”
C0161	unrecognized #pragma
C0163	could not open temporary file “filename”
C0164 ¹	name of directory for temporary files is too long (“ <i>xxxx</i> ”)
C0165	too few arguments in function call
C0166	invalid floating constant
C0167	argument of type “ <i>type</i> ” is incompatible with parameter of type “ <i>type</i> ”
C0168	a function type is not allowed here
C0169	expected a declaration
C0170 ¹	pointer points outside of underlying object
C0171	invalid type conversion
C0172 ¹	external/internal linkage conflict with previous declaration
C0173	floating point value does not fit in required integral type
C0174	expression has no effect
C0175	subscript out of range
C0177	entity-kind “entity” was declared but never referenced
C0178	“&” applied to an array has no effect
C0179	right operand of “%” is zero
C0180	argument is incompatible with formal parameter
C0181	argument is incompatible with corresponding format string conversion
C0182	could not open source file “ <i>filename</i> ” (no directories in search list)

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0183	type of cast must be integral
C0184	type of cast must be arithmetic or pointer
C0185	dynamic initialization in unreachable code
C0186	pointless comparison of unsigned integer with zero
C0187	use of “=” where “==” may have been intended
C0188	enumerated type mixed with another type
C0189	error while writing “filename” file
C0190	invalid intermediate language file
C0191	type qualifier is meaningless on cast type
C0192	unrecognized character escape sequence
C0193	zero used for undefined preprocessing identifier
C0194	expected an asm string
C0195	an asm function must be prototyped
C0196	an asm function cannot have an ellipsis
C0219	error while deleting file “filename”
C0220	integral value does not fit in required floating point type
C0221	floating point value does not fit in required floating point type
C0222	floating point operation result is out of range
C0223	function “xxx” declared implicitly
C0224 ¹	the format string requires additional arguments
C0225 ¹	the format string ends before this argument
C0226	invalid format string conversion
C0227	macro recursion
C0228	trailing comma is nonstandard
C0229	bit field cannot contain all values of the enumerated type
C0230	nonstandard type for a bit field
C0231	declaration is not visible outside of function
C0232	old-fashioned typedef of “ void ” ignored
C0233	left operand is not a struct or union containing this field
C0234	pointer does not point to struct or union containing this field

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0235	variable “xxxx” was declared with a never-completed type
C0236	controlling expression is constant
C0237	selector expression is constant
C0238	invalid specifier on a parameter
C0239	invalid specifier outside a class declaration
C0240	duplicate specifier in declaration
C0241	a union is not allowed to have a base class
C0242 ¹	multiple access control specifiers are not allowed
C0243	class or struct definition is missing
C0244	qualified name is not a member of class “ <i>type</i> ” or its base classes
C0245	a nonstatic member reference must be relative to a specific object
C0246	a nonstatic data member cannot be defined outside its class
C0247	entity-kind “entity” has already been defined
C0248	pointer to reference is not allowed
C0249	reference to reference is not allowed
C0250	reference to void is not allowed
C0251	array of reference is not allowed
C0252	reference entity-kind “entity” requires an initializer
C0253	expected a “,”
C0254	type name is not allowed
C0255	type definition is not allowed
C0256	invalid redeclaration of type name “entity” (declared at line xxxx)
C0257	const entity-kind “entity” requires an initializer
C0258	“this” may only be used inside a nonstatic member function
C0259	constant value is not known
C0260	explicit type is missing (“ int ” assumed)
C0261 ¹	access control not specified (“xxxx” by default)
C0262	not a class or struct name
C0263	duplicate base class name
C0264	invalid base class

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0265	entity-kind “entity” is inaccessible
C0266	“entity” is ambiguous
C0267 ¹	old-style parameter list (anachronism)
C0268	declaration can’t appear after executable statement in block
C0269	conversion to inaccessible base class “ <i>type</i> ” is not allowed
C0274	improperly terminated macro invocation
C0276	name followed by “::” must be a class or namespace name
C0277	invalid friend declaration
C0278	a constructor or destructor cannot return a value
C0279	invalid destructor declaration
C0280	declaration of a member with the same name as its class
C0281	global scope qualifier (leading “::”) is not allowed
C0282	the global scope has no “xxx”
C0283 ¹	qualified name is not allowed
C0284	NULL reference is not allowed
C0285	initialization with “{...}” is not allowed for object of type “ <i>type</i> ”
C0286	base class “ <i>type</i> ” is ambiguous
C0287	derived class “ <i>type</i> ” contains more than one instance of class “ <i>type</i> ”
C0288	cannot convert pointer to base class “ <i>type</i> ” to pointer to derived class “ <i>type</i> ” - base class is virtual
C0289	no instance of constructor “entity” matches the argument list
C0290	copy constructor for class “ <i>type</i> ” is ambiguous
C0291	no default constructor exists for class “ <i>type</i> ”
C0292	“xxx” is not a nonstatic data member or base class of class “ <i>type</i> ”
C0293	indirect nonvirtual base class is not allowed
C0294	invalid union member - class “ <i>type</i> ” has a disallowed member function
C0296 ¹	invalid use of non-lvalue array
C0297 ¹	expected an operator
C0298 ¹	inherited member is not allowed
C0299	cannot determine which instance of entity-kind “entity” is intended

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0300	a pointer to a bound function may only be used to call the function
C0301	typedef name has already been declared (with same type)
C0302	entity-kind “entity” has already been defined
C0304	no instance of entity-kind “entity” matches the argument list
C0305	type definition is not allowed in function return type declaration
C0306	default argument not at end of parameter list
C0307	redefinition of default argument
C0308 ¹	more than one instance of entity-kind “entity” matches the argument list:
C0309 ¹	more than one instance of constructor “entity” matches the argument list:
C0310	default argument of type “ <i>type</i> ” is incompatible with parameter of type “ <i>type</i> ”
C0311	cannot overload functions distinguished by return type alone
C0312	no suitable user-defined conversion from “ <i>type</i> ” to “ <i>type</i> ” exists
C0313	type qualifier is not allowed on this function
C0314	only nonstatic member functions may be virtual
C0315	the object has cv qualifiers that are not compatible with the member function
C0316	program too large to compile (too many virtual functions)
C0317	return type is not identical to nor covariant with return type “ <i>type</i> ” of overridden virtual function entity-kind “entity”
C0318	override of virtual entity-kind “entity” is ambiguous
C0319	pure specifier (“= 0”) allowed only on virtual functions
C0320	badly-formed pure specifier (only “= 0” is allowed)
C0321	data member initializer is not allowed
C0322	object of abstract class type “ <i>type</i> ” is not allowed:
C0323	function returning abstract class “ <i>type</i> ” is not allowed:
C0324	duplicate friend declaration
C0325	inline specifier allowed on function declarations only
C0326	“ inline ” is not allowed
C0327	invalid storage class for an inline function
C0328	invalid storage class for a class member
C0329	local class member entity-kind “entity” requires a definition

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0330	entity-kind “entity” is inaccessible
C0332	class “ <i>type</i> ” has no copy constructor to copy a const object
C0333	defining an implicitly declared member function is not allowed
C0334	class “ <i>type</i> ” has no suitable copy constructor
C0335 ¹	linkage specification is not allowed
C0336 ¹	unknown external linkage specification
C0337 ¹	linkage specification is incompatible with previous “entity” (declared at line <i>xxx</i>)
C0338 ¹	more than one instance of overloaded function “entity” has “C” linkage
C0339	class “ <i>type</i> ” has more than one default constructor
C0340	value copied to temporary, reference to temporary used
C0341	“ operator <i>xxx</i> ” must be a member function
C0342	operator cannot be a static member function
C0343	no arguments allowed on user-defined conversion
C0344	too many parameters for this operator function
C0345	too few parameters for this operator function
C0346	nonmember operator requires a parameter with class type
C0347	default argument is not allowed
C0348 ¹	more than one user-defined conversion from “ <i>type</i> ” to “ <i>type</i> ” applies:
C0349	no operator “ <i>xxx</i> ” matches these operands
C0350 ¹	more than one operator “ <i>xxx</i> ” matches these operands:
C0351	first parameter of allocation function must be of type “ <i>size_t</i> ”
C0352	allocation function requires “ void * ” return type
C0353	deallocation function requires “ void ” return type
C0354	first parameter of deallocation function must be of type “ void * ”
C0356	type must be an object type
C0357	base class “ <i>type</i> ” has already been initialized
C0358	base class name required - “ <i>type</i> ” assumed (anachronism)
C0359	entity-kind “entity” has already been initialized
C0360	name of member or base class is missing

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0361	assignment to “this” (anachronism)
C0362	“ overload ” keyword used (anachronism)
C0363	invalid anonymous union - nonpublic member is not allowed
C0364	invalid anonymous union - member function is not allowed
C0365	anonymous union at global or namespace scope must be declared static
C0366 ¹	entity-kind “entity” provides no initializer for:
C0367	implicitly generated constructor for class “ <i>type</i> ” cannot initialize:
C0368 ¹	entity-kind “entity” defines no constructor to initialize the following:
C0369	entity-kind “entity” has an uninitialized const or reference member
C0370	entity-kind “entity” has an uninitialized const field
C0371	class “ <i>type</i> ” has no assignment operator to copy a const object
C0372	class “ <i>type</i> ” has no suitable assignment operator
C0373	ambiguous assignment operator for class “ <i>type</i> ”
C0375	declaration requires a typedef name
C0377	“ virtual ” is not allowed
C0378	“ static ” is not allowed
C0379 ¹	cast of bound function to normal function pointer (anachronism)
C0380	expression must have pointer-to-member type
C0381	extra “;” ignored
C0382	nonstandard member constant declaration (standard form is a static const integral member)
C0384	no instance of overloaded “entity” matches the argument list
C0386	no instance of entity-kind “entity” matches the required type
C0387	delete array size expression used (anachronism)
C0389	a cast to abstract class “ <i>type</i> ” is not allowed:
C0390	function “main” cannot be called or have its address taken
C0391	a new-initializer cannot be specified for an array
C0392	member function “entity” cannot be redeclared outside its class
C0393	pointer to incomplete class type is not allowed
C0394	reference to local variable of enclosing function is not allowed

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0395 ¹	single-argument function used for postfix “ xxxx ” (anachronism)
C0397 ¹	implicitly generated assignment operator cannot copy:
C0398	cast to array type is nonstandard (treated as cast to “ <i>type</i> ”)
C0399	entity-kind “entity” has an operator new <i>xxxx</i> () but no default operator delete <i>xxxx</i> ()
C0400	entity-kind “entity” has a default operator delete <i>xxxx</i> () but no operator new <i>xxxx</i> ()
C0401	destructor for base class “ <i>type</i> ” is not virtual
C0403	entity-kind “entity” has already been declared
C0404	function “ main ” cannot be declared inline
C0405	member function with the same name as its class must be a constructor
C0406	using nested entity-kind “entity” (anachronism)
C0407	a destructor cannot have parameters
C0408	copy constructor for class “ <i>type</i> ” can’t have a parameter of type “ <i>type</i> ”
C0409	entity-kind “entity” returns incomplete type “ <i>type</i> ”
C0410	protected entity-kind “entity” is not accessible through a “ <i>type</i> ” pointer or object
C0411	a parameter is not allowed
C0412	an “ asm ” declaration is not allowed here
C0413	no suitable conversion function from “ <i>type</i> ” to “ <i>type</i> ” exists
C0414	delete of pointer to incomplete class
C0415	no suitable constructor exists to convert from “ <i>type</i> ” to “ <i>type</i> ”
C0416 ¹	more than one constructor applies to convert from “ <i>type</i> ” to “ <i>type</i> ”:
C0417 ¹	more than one conversion function from “ <i>type</i> ” to “ <i>type</i> ” applies:
C0418 ¹	more than one conversion function from “ <i>type</i> ” to a builtin type applies:
C0424	a constructor or destructor cannot have its address taken
C0425	dollar sign (“\$”) used in identifier
C0426	temporary used for initial value of reference to non- const (anachronism)
C0427	qualified name is not allowed in member declaration
C0428	enumerated type mixed with another type (anachronism)
C0429	the size of an array in “new” must be non-negative

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0430	returning reference to local temporary
C0432	“enum” declaration is not allowed
C0433	qualifiers dropped in binding reference of type <i>“type”</i> to initializer of type <i>“type”</i>
C0434	a reference of type <i>“type”</i> (not const -qualified) cannot be initialized with a value of type <i>“type”</i>
C0435	a pointer to function cannot be deleted
C0436	conversion function must be a nonstatic member function
C0437	template declaration is not allowed here
C0438	expected a “<”
C0439	expected a “>”
C0440	template parameter declaration is missing
C0441	argument list for entity-kind <i>“entity”</i> is missing
C0442	too few arguments for entity-kind <i>“entity”</i>
C0443	too many arguments for entity-kind <i>“entity”</i>
C0445	entity-kind <i>“entity”</i> is not used in declaring the parameter types of entity-kind <i>“entity”</i>
C0446	two nested types have the same name: <i>“entity”</i> and <i>“entity”</i> (declared at line <i>xxxx</i>) (cfront compatibility)
C0447	global <i>“entity”</i> was declared after nested <i>“entity”</i> (declared at line <i>xxxx</i>) (cfront compatibility)
C0449	more than one instance of entity-kind <i>“entity”</i> matches the required type
C0450	the type “long long” is nonstandard
C0451	omission of <i>“xxxx”</i> is nonstandard
C0452	return type cannot be specified on a conversion function
C0456 ¹	excessive recursion at instantiation of entity-kind <i>“entity”</i>
C0457	<i>“xxxx”</i> is not a function or static data member
C0458	argument of type <i>“type”</i> is incompatible with template parameter of type <i>“type”</i>
C0459	initialization requiring a temporary or conversion is not allowed
C0460	declaration of <i>“xxxx”</i> hides function parameter
C0461	initial value of reference to non- const must be an lvalue

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0463	“template” is not allowed
C0464	“ <i>type</i> ” is not a class template
C0466	“main” is not a valid name for a function template
C0467	invalid reference to entity-kind “entity” (union/nonunion mismatch)
C0468	a template argument cannot reference a local type
C0469 ¹	tag kind of <i>xxxx</i> is incompatible with declaration of entity-kind “entity” (declared at line <i>xxxx</i>)
C0470	the global scope has no tag named “ <i>xxxx</i> ”
C0471	entity-kind “entity” has no tag member named “ <i>xxxx</i> ”
C0472	member function typedef (allowed for cfront compatibility)
C0473	entity-kind “entity” may be used only in pointer-to-member declaration
C0475	a template argument cannot reference a non-external entity
C0476	name followed by “::~” must be a class name or a type name
C0477	destructor name does not match name of class “ <i>type</i> ”
C0478	type used as destructor name does not match type “ <i>type</i> ”
C0479	entity-kind “entity” redeclared “ inline ” after being called
C0481	invalid storage class for a template declaration
C0482	entity-kind “entity” is an inaccessible type (allowed for cfront compatibility)
C0484	invalid explicit instantiation declaration
C0485	entity-kind “entity” is not an entity that can be instantiated
C0486	compiler generated entity-kind “entity” cannot be explicitly instantiated
C0487	inline entity-kind “entity” cannot be explicitly instantiated
C0489	entity-kind “entity” cannot be instantiated - no template definition was supplied
C0490	entity-kind “entity” cannot be instantiated - it has been explicitly specialized
C0493	no instance of entity-kind “entity” matches the specified type
C0494	declaring a void parameter list with a typedef is nonstandard
C0495	global entity-kind “entity” used instead of entity-kind “entity” (cfront compatibility)
C0496	template parameter “ <i>xxxx</i> ” cannot be redeclared in this scope
C0497	declaration of “ <i>xxxx</i> ” hides template parameter

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0498	template argument list must match the parameter list
C0500 ¹	extra parameter of postfix “ operator xxxx” must be of type “ int ”
C0501	an operator name must be declared as a function
C0502	operator name is not allowed
C0503	entity-kind “entity” cannot be specialized in the current scope
C0504	nonstandard form for taking the address of a member function
C0505	too few template parameters - does not match previous declaration
C0506	too many template parameters - does not match previous declaration
C0507	function template for operator delete (void *) is not allowed
C0508	class template and template parameter cannot have the same name
C0510	a template argument cannot reference an unnamed type
C0511 ¹	enumerated type is not allowed
C0512	type qualifier on a reference type is not allowed
C0513	a value of type “ <i>type</i> ” cannot be assigned to an entity of type “ <i>type</i> ”
C0514	pointless comparison of unsigned integer with a negative constant
C0515	cannot convert to incomplete class “ <i>type</i> ”
C0516	const object requires an initializer
C0517	object has an uninitialized const or reference member
C0518	nonstandard preprocessing directive
C0519	entity-kind “entity” cannot have a template argument list
C0520 ¹	initialization with “{...}” expected for aggregate object
C0521 ¹	pointer-to-member selection class types are incompatible (“ <i>type</i> ” and “ <i>type</i> ”)
C0522 ¹	pointless friend declaration
C0523	“.” used in place of “::” to form a qualified name (cfront anachronism)
C0524	non- const function called for const object (anachronism)
C0525 ¹	a dependent statement cannot be a declaration
C0526	a parameter can’t have void type
C0529	this operator is not allowed in a template argument expression
C0530	try block requires at least one handler
C0531	handler requires an exception declaration

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0532	handler is masked by default handler
C0533	handler is potentially masked by previous handler for type “ <i>type</i> ”
C0534	use of a local type to specify an exception
C0535	redundant type in exception specification
C0536	exception specification is incompatible with that of previous entity-kind “entity” (declared at line <i>xxxx</i>):
C0540	support for exception handling is disabled; use -ze to enable
C0541	omission of exception specification is incompatible with previous entity-kind “entity” (declared at line <i>xxxx</i>):
C0542	could not create instantiation request file “ <i>filename</i> ”
C0543	non-arithmetic operation not allowed in nontype template argument
C0544	use of a local type to declare a nonlocal variable
C0545	use of a local type to declare a function
C0546 ¹	transfer of control bypasses initialization of:
C0548	transfer of control into an exception handler
C0549	entity-kind “entity” is used before its value is set
C0550	entity-kind “entity” was set but never used
C0551	entity-kind “entity” cannot be defined in the current scope
C0552	exception specification is not allowed
C0553 ¹	external/internal linkage conflict for entity-kind “entity” (declared at line <i>xxxx</i>)
C0554	entity-kind “entity” will not be called for implicit or explicit conversions
C0555	tag kind of <i>xxxx</i> is incompatible with template parameter of type “ <i>type</i> ”
C0556	function template for operator new(size_t) is not allowed
C0558	pointer to member of type “ <i>type</i> ” is not allowed
C0559	ellipsis is not allowed in operator function parameter list
C0560	“ entity ” is reserved for future use as a keyword
C0561 ¹	invalid macro definition:
C0562 ¹	invalid macro undefinition:
C0563	invalid preprocessor output file
C0564	cannot open preprocessor output file

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0569	cannot open C output file
C0570	error in debug option argument
C0571	invalid option:
C0574	invalid number:
C0576	invalid instantiation mode:
C0578	invalid error limit:
C0579	invalid raw-listing output file
C0580	cannot open raw-listing output file
C0582	cannot open cross-reference output file
C0583	invalid error output file
C0584	cannot open error output file
C0585	virtual function tables can only be suppressed when compiling C++
C0586	anachronism option can be used only when compiling C++
C0587	instantiation mode option can be used only when compiling C++
C0590	exception handling option can be used only when compiling C++
C0591	strict ANSI mode is incompatible with K&R mode
C0592	strict ANSI mode is incompatible with cfront mode
C0593	missing source filename
C0594	output files cannot be specified when compiling several input files
C0595	too many arguments on command line
C0596	an output file was specified, but none is needed
C0598	a template parameter cannot have void type
C0599	excessive recursive instantiation of entity-kind “entity” due to instantiate-all mode
C0600	strict ANSI mode is incompatible with allowing anachronisms
C0601	a throw expression cannot have void type
C0602	local instantiation mode is incompatible with automatic instantiation
C0603	parameter of abstract class type “ <i>type</i> ” is not allowed:
C0604	array of abstract class “ <i>type</i> ” is not allowed:
C0605	floating point template parameter is nonstandard

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0606	this pragma must immediately precede a declaration
C0607	this pragma must immediately precede a statement
C0608	this pragma must immediately precede a declaration or statement
C0609	this kind of pragma cannot be used here
C0611	overloaded virtual function “entity” is only partially overridden in entity-kind “entity”
C0612	specific definition of inline template function must precede its first use
C0613	invalid error tag in diagnostic control option:
C0614	invalid error number in diagnostic control option:
C0615	parameter type involves pointer to array of unknown bound
C0616	parameter type involves reference to array of unknown bound
C0617	pointer-to-member-function cast to pointer to function
C0618	struct or union declares no named members
C0619	nonstandard unnamed field
C0620	nonstandard unnamed member
C0622	invalid precompiled header output file
C0623	cannot open precompiled header output file
C0624	“xxx” is not a type name
C0625	cannot open precompiled header input file
C0626	precompiled header file “xxx” is either invalid or not generated by this version of the compiler
C0627	precompiled header file “xxx” was not generated in this directory
C0628	header files used to generate precompiled header file “xxx” have changed
C0629	the command line options do not match those used when precompiled header file “xxx” was created
C0630	the initial sequence of preprocessing directives is not compatible with those of precompiled header file “xxx”
C0631	unable to obtain mapped memory
C0632	“xxx”: using precompiled header file “xxx”
C0633	“xxx”: creating precompiled header file “xxx”
C0634	memory usage conflict with precompiled header file “xxx”

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0635	invalid PCH memory size
C0636	PCH options must appear first in the command line
C0637	insufficient memory for PCH memory allocation
C0638	precompiled header files cannot be used when compiling several input files
C0639	insufficient preallocated memory for generation of precompiled header file (xxxx bytes required)
C0640	very large entity in program prevents generation of precompiled header file
C0641	“xxxx” is not a valid directory
C0642	cannot build temporary file name
C0643	“restrict” is not allowed
C0644	a pointer or reference to function type cannot be qualified by “restrict”
C0645	“xxxx” is an unrecognized __declspec attribute
C0646	a calling convention modifier cannot be specified here
C0647	conflicting calling convention modifiers
C0648	strict ANSI mode is incompatible with Microsoft mode
C0649	cfront mode is incompatible with Microsoft mode
C0650	calling convention specified here is ignored
C0651	a calling convention cannot be followed by a nested declarator
C0652	calling convention is ignored for this type
C0654	declaration modifiers are incompatible with previous declaration
C0655	the modifier “xxxx” is not allowed on this declaration
C0656	transfer of control into a try block
C0657	inline specification is incompatible with previous “entity” (declared at line xxx)
C0658	closing brace of template definition not found
C0659	wchar_t keyword option can be used only when compiling C++
C0660	invalid packing alignment value
C0661	expected an integer constant
C0662	call of pure virtual function
C0663	invalid source file identifier string

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0664	a class template cannot be defined in a friend declaration
C0665	“asm” is not allowed
C0666	“asm” must be used with a function definition
C0667	“asm” function is nonstandard
C0668	ellipsis with no explicit parameters is nonstandard
C0669	“&...” is nonstandard
C0670	invalid use of “&...”
C0672	temporary used for initial value of reference to const volatile (anachronism)
C0673	a reference of type “ <i>type</i> ” cannot be initialized with a value of type “ <i>type</i> ”
C0674	initial value of reference to const volatile must be an lvalue
C0675	SVR4 C compatibility option can be used only when compiling ANSI C
C0676	using out-of-scope declaration of entity-kind “entity” (declared at line <i>xxxx</i>)
C0677	strict ANSI mode is incompatible with SVR4 C mode
C0678	call of entity-kind “entity” (declared at line <i>xxxx</i>) cannot be inlined
C0679	entity-kind “entity” cannot be inlined
C0680	invalid PCH directory:
C0681	expected __except or __finally
C0682	a __leave statement may only be used within a __try
C0688	“xxxx” not found on pack alignment stack
C0689	empty pack alignment stack
C0690	RTTI option can be used only when compiling C++
C0691	entity-kind “entity”, required for copy that was eliminated, is inaccessible
C0692	entity-kind “entity”, required for copy that was eliminated, is not callable because reference parameter cannot be bound to rvalue
C0693	<typeinfo> must be included before typeid is used
C0694	<i>xxxx</i> cannot cast away const or other type qualifiers
C0695	the type in a dynamic_cast must be a pointer or reference to a complete class type, or void *
C0696	the operand of a pointer dynamic_cast must be a pointer to a complete class type

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0697	the operand of a reference <code>dynamic_cast</code> must be an lvalue of a complete class type
C0698	the operand of a runtime <code>dynamic_cast</code> must have a polymorphic class type
C0699	<code>bool</code> option can be used only when compiling C++
C0701	an array type is not allowed here
C0702	expected an “=”
C0703	expected a declarator in condition declaration
C0704	“xxx”, declared in condition, cannot be redeclared in this scope
C0705	default template arguments are not allowed for function templates
C0706	expected a “,” or “>”
C0707	expected a template parameter list
C0708	incrementing a <code>bool</code> value is deprecated
C0709	<code>bool</code> type is not allowed
C0710	offset of base class “entity” within class “entity” is too large
C0711	expression must have <code>bool</code> type (or be convertible to <code>bool</code>)
C0712	array new and delete option can be used only when compiling C++
C0713	entity-kind “entity” is not a variable name
C0714	<code>__based</code> modifier is not allowed here
C0715	<code>__based</code> does not precede a pointer operator, <code>__based</code> ignored
C0716	variable in <code>__based</code> modifier must have pointer type
C0717	the type in a <code>const_cast</code> must be a pointer, reference, or pointer to member to an object type
C0718	a <code>const_cast</code> can only adjust type qualifiers; it cannot change the underlying type
C0719	<code>mutable</code> is not allowed
C0720	redeclaration of entity-kind “entity” is not allowed to alter its access
C0721	nonstandard format string conversion
C0722	use of alternative token “<:” appears to be unintended
C0723	use of alternative token “%:” appears to be unintended
C0724	namespace definition is not allowed
C0725	name must be a namespace name

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0726	namespace alias definition is not allowed
C0727	namespace-qualified name is required
C0728	a namespace name is not allowed
C0729	invalid combination of DLL attributes
C0730	entity-kind “entity” is not a class template
C0731	array with incomplete element type is nonstandard
C0732	allocation operator cannot be declared in a namespace
C0733	deallocation operator cannot be declared in a namespace
C0734	entity-kind “entity” conflicts with using-declaration of entity-kind “entity”
C0735	using-declaration of entity-kind “entity” conflicts with entity-kind “entity” (declared at line <i>xxxx</i>)
C0736	namespaces option can be used only when compiling C++
C0737	using-declaration ignored - it refers to the current namespace
C0738	a class-qualified name is required
C0742	entity-kind “entity” has no actual member “ <i>xxxx</i> ”
C0744	incompatible memory attributes specified
C0745	memory attribute ignored
C0746	memory attribute cannot be followed by a nested declarator
C0747	memory attribute specified more than once
C0748	calling convention specified more than once
C0749	a type qualifier is not allowed
C0750	entity-kind “entity” (declared at line <i>xxxx</i>) was used before its template was declared
C0751	static and nonstatic member functions with same parameter types cannot be overloaded
C0752	no prior declaration of entity-kind “entity”
C0753	a template-id is not allowed
C0754	a class-qualified name is not allowed
C0755	entity-kind “entity” cannot be redeclared in the current scope
C0756	qualified name is not allowed in namespace member declaration
C0757	entity-kind “entity” is not a type name

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0758	explicit instantiation is not allowed in the current scope
C0759	entity-kind “entity” cannot be explicitly instantiated in the current scope
C0760	entity-kind “entity” explicitly instantiated more than once
C0761	typename may only be used within a template
C0762	special_subscript_cost option can be used only when compiling C++
C0763	typename option can be used only when compiling C++
C0764	implicit typename option can be used only when compiling C++
C0765	nonstandard character at start of object-like macro definition
C0766	exception specification for virtual entity-kind “entity” is incompatible with that of overridden entity-kind “entity”
C0767	conversion from pointer to smaller integer
C0768	exception specification for implicitly declared virtual entity-kind “entity” is incompatible with that of overridden entity-kind “entity”
C0769	“entity”, implicitly called from entity-kind “entity”, is ambiguous
C0770	option “explicit” can be used only when compiling C++
C0771	“explicit” is not allowed
C0772	declaration conflicts with “xxxx” (reserved class name)
C0773	only “()” is allowed as initializer for array entity-kind “entity”
C0774	“virtual” is not allowed in a function template declaration
C0775	invalid anonymous union - class member template is not allowed
C0776	template nesting depth does not match the previous declaration of entity-kind “entity”
C0777	this declaration cannot have multiple “template <...>” clauses
C0778	option to control the for-init scope can be used only when compiling C++
C0779	“xxxx”, declared in for-loop initialization, cannot be redeclared in this scope
C0780	reference is to entity-kind “entity” (declared at line xxxx) - under old for-init scoping rules it would have been entity-kind “entity” (declared at line xxxx)
C0781	option to control warnings on for-init differences can be used only when compiling C++
C0782	definition of virtual entity-kind “entity” is required here
C0783	empty comment interpreted as token-pasting operator “##”

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0784	a storage class is not allowed in a friend declaration
C0785	template parameter list for “entity” is not allowed in this declaration
C0786	entity-kind “entity” is not a valid member class or function template
C0787	not a valid member class or function template declaration
C0788	a template declaration containing a template parameter list cannot be followed by an explicit specialization declaration
C0789	explicit specialization of entity-kind “entity” must precede the first use of entity-kind “entity”
C0790	explicit specialization is not allowed in the current scope
C0791	partial specialization of entity-kind “entity” is not allowed
C0792	entity-kind “entity” is not an entity that can be explicitly specialized
C0793	explicit specialization of entity-kind “entity” must precede its first use
C0794	template parameter “xxx” cannot be used in an elaborated type specifier
C0795	specializing entity-kind “entity” requires “template<>” syntax
C0798	option “old_specializations” can be used only when compiling C++
C0799	specializing entity-kind “entity” without “template<>” syntax is nonstandard
C0800	this declaration cannot have extern “C” linkage
C0801	“xxx” is not a class or function template name in the current scope
C0802	specifying a default argument when redeclaring an unreferenced function template is nonstandard
C0803	specifying a default argument when redeclaring an already referenced function template is not allowed
C0804	cannot convert pointer to member of base class “type” to pointer to member of derived class “type” - base class is virtual
C0805	exception specification is incompatible with that of entity-kind “entity” (declared at line xxx):
C0806	omission of exception specification is incompatible with entity-kind “entity” (declared at line xxx)
C0807	unexpected end of default argument expression
C0808	default-initialization of reference is not allowed
C0809	uninitialized entity-kind “entity” has a const member
C0810	uninitialized base class “type” has a const member

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0811	const entity-kind “entity” requires an initializer - class “ <i>type</i> ” has no explicitly declared default constructor
C0812	const object requires an initializer - class “ <i>type</i> ” has no explicitly declared default constructor
C0813	option “ implicit_extern_c_type_conversion ” can be used only when compiling C++
C0814	strict ANSI mode is incompatible with long preserving rules
C0815	type qualifier on return type is meaningless
C0816	in a function definition a type qualifier on a “ void ” return type is not allowed
C0817	static data member declaration is not allowed in this class
C0818	template instantiation resulted in an invalid function declaration
C0819	“...” is not allowed
C0820	option “ extern_inline ” can be used only when compiling C++
C0821	extern inline entity-kind “entity” was referenced but not defined
C0822	invalid destructor name for type “ <i>type</i> ”
C0824	destructor reference is ambiguous - both entity-kind “entity” and entity-kind “entity” could be used
C0825	virtual inline entity-kind “entity” was never defined
C0826	entity-kind “entity” was never referenced
C0827	only one member of a union may be specified in a constructor initializer list
C0828	support for “new[]” and “delete[]” is disabled
C0829	“ double ” used for “ long double ” in generated C code
C0830	entity-kind “entity” has no corresponding operator deletexxxx (to be called if an exception is thrown during initialization of an allocated object)
C0831	support for placement delete is disabled
C0832	no appropriate operator delete is visible
C0833	pointer or reference to incomplete type is not allowed
C0834	invalid partial specialization - entity-kind “entity” is already fully specialized
C0835	incompatible exception specifications
C0836	returning reference to local variable
C0837	omission of explicit type is nonstandard (“ int ” assumed)

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0838	more than one partial specialization matches the template argument list of entity-kind “entity”
C0840	a template argument list is not allowed in a declaration of a primary template
C0841	partial specializations cannot have default template arguments
C0842	entity-kind “entity” is not used in template argument list of entity-kind “entity”
C0843	the type of partial specialization template parameter entity-kind “entity” depends on another template parameter
C0844	the template argument list of the partial specialization includes a nontype argument whose type depends on a template parameter
C0845	this partial specialization would have been used to instantiate entity-kind “entity”
C0846	this partial specialization would have been made the instantiation of entity-kind “entity” ambiguous
C0847	expression must have integral or enum type
C0848	expression must have arithmetic or enum type
C0849	expression must have arithmetic , enum , or pointer type
C0850	type of cast must be integral or enum
C0851	type of cast must be arithmetic , enum , or pointer
C0852	expression must be a pointer to a complete object type
C0854	a partial specialization nontype argument must be the name of a nontype parameter or a constant
C0855	return type is not identical to return type “ <i>type</i> ” of overridden virtual function entity-kind “entity”
C0856	option “guiding_decls” can be used only when compiling C++
C0857	a partial specialization of a class template must be declared in the namespace of which it is a member
C0858	entity-kind “entity” is a pure virtual function
C0859	pure virtual entity-kind “entity” has no override
C0860	__declspec attributes ignored
C0861	invalid character in input line
C0862	function returns incomplete type “ <i>type</i> ”
C0863	effect of this “ #pragma pack ” directive is local to entity-kind “entity”

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0864	xxxx is not a template
C0865	a friend declaration cannot declare a partial specialization
C0866	exception specification ignored
C0867	declaration of “size_t” does not match the expected type “ <i>type</i> ”
C0868	space required between adjacent “>” delimiters of nested template argument lists (“>>” is the right shift operator)
C0869	could not set locale “xxxx” to allow processing of multibyte characters
C0870	invalid multibyte character sequence
C0871	template instantiation resulted in unexpected function type of “ <i>type</i> ” (the meaning of a name may have changed since the template declaration - the type of the template is “ <i>type</i> ”)
C0872	ambiguous guiding declaration - more than one function template “entity” matches type “ <i>type</i> ”
C0873	non-integral operation not allowed in nontype template argument
C0874	option “embedded_c++” can be used only when compiling C++
C0875	Embedded C++ does not support templates
C0876	Embedded C++ does not support exception handling
C0877	Embedded C++ does not support namespaces
C0878	Embedded C++ does not support run-time type information
C0879	Embedded C++ does not support the new cast syntax
C0880	Embedded C++ does not support using-declarations
C0881	Embedded C++ does not support “mutable”
C0882	Embedded C++ does not support multiple or virtual inheritance
C0883	invalid Microsoft version number:
C0884	pointer-to-member representation “xxxx” has already been set for entity-kind “entity”
C0885	“ <i>type</i> ” cannot be used to designate constructor for “ <i>type</i> ”
C0886	invalid suffix on integral constant
C0887	operand of __uuidof must have a class or enum type for which __declspec(uuid(“...”)) has been specified
C0888	invalid GUID string in __declspec(uuid(“...”))
C0889	option “vla” can be used only when compiling C

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0890	variable length array with unspecified bound is not allowed
C0891	an explicit template argument list is not allowed on this declaration
C0892	an entity with linkage cannot have a type involving a variable length array
C0893	a variable length array cannot have static storage duration
C0894	entity-kind “entity” is not a template
C0896	expected a template argument
C0898	nonmember operator requires a parameter with class or enum type
C0899	option “enum_overloading” can be used only when compiling C++
C0901	qualifier of destructor name “ <i>type</i> ” does not match type “ <i>type</i> ”
C0902	type qualifier ignored
C0903	option “nonstd_qualifier_deduction” can be used only when compiling C++
C0905	incorrect property specification; correct form is __declspec(property(get=name1,put=name2))
C0906	property has already been specified
C0907	__declspec(property) is not allowed on this declaration
C0908	member is declared with __declspec(property), but no “ get ” function was specified
C0909	the __declspec(property) “ get ” function “xxxx” is missing
C0910	member is declared with __declspec(property), but no “ put ” function was specified
C0911	the __declspec(property) “ put ” function “xxxx” is missing
C0912	ambiguous class member reference - entity-kind “entity” (declared at line xxxx) used in preference to entitykind “entity” (declared at line xxxx)
C0913	missing or invalid segment name in __declspec(allocate(“...”))
C0914	__declspec(allocate) is not allowed on this declaration
C0915	a segment name has already been specified
C0916	cannot convert pointer to member of derived class “ <i>type</i> ” to pointer to member of base class “ <i>type</i> ” - base class is virtual
C0917	invalid directory for instantiation files:
C0918	option “one_instantiation_per_object” can be used only when compiling C++
C0919	invalid output file: “xxxx”

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0920	cannot open output file: “xxx”
C0921	an instantiation information file name cannot be specified when compiling several input files
C0922	option “one_instantiation_per_object” cannot be used when compiling several input files
C0923	more than one command line option matches the abbreviation “--xxx”:
C0925	type qualifiers on function types are ignored
C0926	cannot open definition list file: “xxx”
C0927	late/early tiebreaker option can be used only when compiling C++
C0928	incorrect use of va_start
C0929	incorrect use of va_arg
C0930	incorrect use of va_end
C0931	pending instantiations option can be used only when compiling C++
C0932	invalid directory for #import files:
C0933	an import directory can be specified only in Microsoft mode
C0934	a member with reference type is not allowed in a union
C0935	“typedef” cannot be specified here
C0936	redeclaration of entity-kind “entity” alters its access
C0937	a class or namespace qualified name is required
C0938	return type “ int ” omitted in declaration of function “ main ”
C0939	pointer-to-member representation “xxx” is too restrictive for entity-kind “entity”
C0940	missing return statement at end of non-void entity-kind “entity”
C0941	duplicate using-declaration of “entity” ignored
C0942	enum bit-fields are always unsigned, but enum “type” includes negative enumerator
C0943	option “ class_name_injection ” can be used only when compiling C++
C0944	option “ arg_dep_lookup ” can be used only when compiling C++
C0945	option “ friend_injection ” can be used only when compiling C++
C0946	name following “template” must be a member template

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0948	nonstandard local-class friend declaration - no prior declaration in the enclosing scope
C0949	specifying a default argument on this declaration is nonstandard
C0950	option “ nonstd_using_decl ” can be used only when compiling C++
C0951	return type of function “ main ” must be “ int ”
C0952	a nontype template parameter cannot have class type
C0953	a default template argument cannot be specified on the declaration of a member of a class template outside of its class
C0954	a return statement is not allowed in a handler of a function try block of a constructor
C0955	ordinary and extended designators cannot be combined in an initializer designation
C0956	the second subscript must not be smaller than the first
C0957	option “designators” can be used only when compiling C
C0958	option “extended_designators” can be used only when compiling C
C0959	declared size for bit field is larger than the size of the bit field type; truncated to <i>xxxx</i> bits
C0960	type used as constructor name does not match type “ <i>type</i> ”
C0961	use of a type with no linkage to declare a variable with linkage
C0962	use of a type with no linkage to declare a function
C0963	return type cannot be specified on a constructor
C0964	return type cannot be specified on a destructor
C0965	incorrectly formed universal character name
C0966	universal character name specifies an invalid character
C0967	a universal character name cannot designate a character in the basic character set
C0968	this universal character is not allowed in an identifier
C0969	the identifier <code>__VA_ARGS__</code> can only appear in the replacement lists of variadic macros
C0970	the qualifier on this friend declaration is ignored
C0971	array range designators cannot be applied to dynamic initializers
C0972	property name cannot appear here

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0973	“inline” used as a function qualifier is ignored
C0974	option “compound_literals” can be used only when compiling C
C0975	a variable-length array type is not allowed
C0976	a compound literal is not allowed in an integral constant expression
C0977	a compound literal of type “type” is not allowed
C0978	a template friend declaration cannot be declared in a local class
C0979	ambiguous “?” operation: second operand of type “type” can be converted to third operand type “type” , and vice versa
C0980	call of an object of a class type without appropriate operator() or conversion functions to pointer-to-function type
C0982	there is more than one way an object of type “type” can be called for the argument list:
C0983	typedef name has already been declared (with similar type)
C0984	operator new and operator delete cannot be given internal linkage
C0985	storage class “mutable” is not allowed for anonymous unions
C0986	invalid precompiled header file
C0987	abstract class type “type” is not allowed as catch type:
C0988	a qualified function type cannot be used to declare a nonmember function or a static member function
C0989	a qualified function type cannot be used to declare a parameter
C0990	cannot create a pointer or reference to qualified function type
C0991	extra braces are nonstandard
C0992	invalid macro definition:
C0993	subtraction of pointer types “type” and “type” is nonstandard
C0994	an empty template parameter list is not allowed in a template template parameter declaration
C0995	expected “class”
C0996	the “class” keyword must be used when declaring a template template parameter
C0997	entity-kind “entity” is hidden by “entity” - virtual function override intended?
C0998	a qualified name is not allowed for a friend declaration that is a function definition

Table A-3. Compiler Error Messages (cont.)

Number	Message
C0999	entity-kind “entity” is not compatible with entity-kind “entity”
C1000	a storage class can’t be specified here
C1001	class member designated by a using-declaration must be visible in a direct base class
C1003	Sun mode is incompatible with cfront mode
C1004	strict ANSI mode is incompatible with Sun mode
C1005	Sun mode is only allowed when compiling C++
C1006	a template template parameter cannot have the same name as one of its template parameters
C1007	recursive instantiation of default argument
C1009	entity-kind “entity” is not an entity that can be defined
C1010	destructor name must be qualified
C1011	friend class name can’t be introduced with “typename”
C1012	a using-declaration can’t name a constructor or destructor
C1013	a qualified friend template declaration must refer to a specific previously declared template
C1014	invalid specifier in class template declaration”
C1015	argument is incompatible with formal parameter
C1016	option “dep_name” can be used only when compiling C++
C1017	loop in sequence of “operator->” functions starting at class “type”
C1018	entity-kind “entity” has no member class “xxx”
C1019	the global scope has no class named “xxx”
C1020	recursive instantiation of template default argument
C1021	access declarations and using-declarations cannot appear in unions
C1022	“entity” is not a class member
C1023	nonstandard member constant declaration is not allowed
C1024	option “ ignore_std ” can be used only when compiling C++
C1025	option “ parse_templates ” can be used only when compiling C++
C1026	option “ dep_name ” cannot be used with “no_parse_templates
C1027	language modes specified are incompatible
C1028	invalid redeclaration of nested class

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1029	type containing an unknown-size array is not allowed
C1030	a variable with static storage duration cannot be defined within an inline function
C1031	an entity with internal linkage cannot be referenced within an inline function with external linkage
C1032	argument type “ <i>type</i> ” does not match this type-generic function macro
C1034	friend declaration cannot add default arguments to previous declaration
C1035	entity-kind “entity” cannot be declared in this scope
C1036	the reserved identifier “xxxx” may only be used inside a function
C1037	this universal character cannot begin an identifier
C1038	expected a string literal
C1039	unrecognized STDC pragma
C1040	expected “ON”, “OFF”, or “DEFAULT”
C1041	a STDC pragma may only appear between declarations in the global scope or before any statements or declarations in a block scope
C1042	incorrect use of va_copy
C1043	xxxx can only be used with floating point types
C1044	complex type is not allowed
C1045	invalid designator kind
C1046	floating point value cannot be represented exactly
C1047	complex floating point operation result is out of range
C1048	conversion between real and imaginary yields zero
C1049	an initializer cannot be specified for a flexible array member
C1050	imaginary *= imaginary sets the left-hand operand to zero
C1051	standard requires that entity-kind “entity” be given a type by a subsequent declaration (“ int ” assumed)
C1052	a definition is required for inline entity-kind “entity”
C1053	conversion from integer to smaller pointer
C1054	a floating point type must be included in the type specifier for a _Complex or _Imaginary type
C1055	types cannot be declared in anonymous unions

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1056	returning pointer to local variable
C1057	returning pointer to local temporary
C1058	option “export” can be used only when compiling C++
C1059	option “export” cannot be used with “no_dep_name
C1060	option “export” cannot be used with “implicit_include
C1061	declaration of entity-kind “entity” is incompatible with a declaration in another translation unit
C1062	the other declaration is at line <i>xxxx</i>
C1065	a field declaration cannot have a type involving a variable length array
C1066	declaration of entity-kind “entity” had a different meaning during compilation of “ <i>xxxx</i> ”
C1067	expected “template”
C1068	“export” cannot be used on an explicit instantiation
C1069	“export” cannot be used on this declaration
C1070	a member of an unnamed namespace cannot be declared “export”
C1071	a template cannot be declared “export” after it has been defined
C1072	a declaration cannot have a label
C1073	support for exported templates is disabled
C1074	cannot open exported template file: “ <i>xxxx</i> ”
C1075	entity-kind “entity” already defined during compilation of “ <i>xxxx</i> ”
C1076	entity-kind “entity” already defined in another translation unit
C1077	a non-static local variable cannot be used in a <code>__based</code> specification
C1078	the option to list makefile dependencies cannot be specified when compiling more than one translation unit
C1080	the option to generate preprocessed output cannot be specified when compiling more than one translation unit
C1081	a field with the same name as its class cannot be declared in a class with a user-declared constructor
C1082	“implicit_include” cannot be used when compiling more than one translation unit
C1083	exported template file “ <i>xxxx</i> ” is corrupted

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1084	entity-kind “entity” cannot be instantiated - it has been explicitly specialized in the translation unit containing the exported definition
C1086	the object has cv-qualifiers that are not compatible with the member entity-kind “entity”
C1087	no instance of entity-kind “entity” matches the argument list and object (the object has cv-qualifiers that prevent a match)
C1088	an attribute specifies a mode incompatible with “ <i>type</i> ”
C1089	there is no type with the width specified
C1090	invalid alignment value specified by attribute
C1091	invalid attribute for “ <i>type</i> ”
C1092	invalid attribute for entity-kind “entity”
C1093	invalid attribute for parameter
C1094	attribute “ <i>xxxx</i> ” does not take arguments
C1096	expected an attribute name
C1097	attribute “ <i>xxxx</i> ” ignored
C1098	attributes cannot appear here
C1099	invalid argument to attribute “ <i>xxxx</i> ”
C1100	the “packed” attribute is ignored in a typedef
C1101	in “goto * <i>expr</i> ”, <i>expr</i> must have type “ void * ”
C1102	“goto * <i>expr</i> ” is nonstandard
C1103	taking the address of a label is nonstandard
C1104	file name specified more than once:
C1105	#warning directive: <i>xxxx</i>
C1106	#informing directive: <i>xxxx</i>
C1107	attribute “ <i>xxxx</i> ” is only allowed in a function definition
C1108	the “transparent_union” attribute only applies to unions, and “ <i>type</i> ” is not a union
C1109	the “ transparent_union ” attribute is ignored on incomplete types
C1110	“ <i>type</i> ” cannot be transparent because entity-kind “entity” does not have the same size as the union
C1111	“ <i>type</i> ” cannot be transparent because it has a field of type “ <i>type</i> ” which is not the same size as the union

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1112	only parameters can be transparent
C1113	the “xxxx” attribute does not apply to local variables
C1114	attributes are not permitted in a function definition
C1115	declarations of local labels should only appear at the start of statement expressions
C1116	the second constant in a case range must be larger than the first
C1117	an asm name is not permitted in a function definition
C1118	an asm name is ignored in a typedef
C1119	unknown register name “xxxx”
C1120	modifier letter “xxxx” ignored in asm operand
C1121	unknown asm constraint modifier “xxxx”
C1122	unknown asm constraint letter “xxxx”
C1123	asm operand has no constraint letter
C1124	an asm output operand must have one of the '=' or '+' modifiers
C1125	an asm input operand cannot have the '=' or '+' modifiers
C1126	too many operands to asm statement (maximum is 10)
C1127	too many colons in asm statement
C1128	register “xxxx” used more than once
C1129	register “xxxx” is both used and clobbered
C1130	register “xxxx” clobbered more than once
C1131	register “xxxx” has a fixed purpose and cannot be used in an asm statement
C1132	register “xxxx” has a fixed purpose and cannot be clobbered in an asm statement
C1133	an empty clobbers list must be omitted entirely
C1134	expected an asm operand
C1135	expected a register to clobber
C1136	“format” attribute applied to entity-kind “entity” which does not have variable arguments
C1137	first substitution argument is not the first variable argument
C1138	format argument index is greater than number of parameters
C1139	format argument does not have string type

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1140	the “ template ” keyword used for syntactic disambiguation may only be used within a template
C1143	attribute does not apply to non-function type “ <i>type</i> ”
C1144	arithmetic on pointer to void or function type
C1145	storage class must be auto or register
C1146	“ <i>type</i> ” would have been promoted to “ <i>type</i> ” when passed through the ellipsis parameter; use the latter type instead
C1147	“ <i>xxx</i> ” is not a base class member
C1148	__super cannot appear after “::”
C1149	__super may only be used in a class scope
C1150	__super must be followed by “::”
C1152	mangled name is too long
C1153	declaration aliased to unknown entity “ <i>xxx</i> ”
C1154	declaration does not match its alias entity-kind “entity”
C1155	entity declared as alias cannot have definition
C1156	variable-length array field type will be treated as zero-length array field type
C1157	nonstandard cast on lvalue ignored
C1158	unrecognized flag name
C1159	void return type cannot be qualified
C1160	the auto specifier is ignored here (invalid in standard C/C++)
C1161	a reduction in alignment without the “packed” attribute is ignored
C1162	a member template corresponding to “entity” is declared as a template of a different kind in another translation unit
C1163	excess initializers are ignored
C1164	va_start should only appear in a function with an ellipsis parameter
C1165	the “short_enums” option is only valid in GNU C and GNU C++ modes
C1166	invalid export information file “ <i>xxx</i> ” at line number <i>xxx</i>
C1167	statement expressions are only allowed in block scope
C1169	an asm name is ignored on a non-register automatic variable
C1170	inline function also declared as an alias; definition ignored
C1171	unrecognized UPC pragma

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1172	shared block size does not match one previously specified
C1173	bracketed expression is assumed to be a block size specification rather than an array dimension
C1174	the block size of a shared array must be greater than zero
C1175	multiple block sizes not allowed
C1176	strict or relaxed requires shared
C1177	THREADS not allowed in this context
C1178	block size specified exceeds the maximum value of <i>xxxx</i>
C1179	function returning shared is not allowed
C1180	only arrays of a shared type can be dimensioned to a multiple of THREADS
C1181	one dimension of an array of a shared type must be a multiple of THREADS when the number of threads is nonconstant
C1182	shared type inside a struct or union is not allowed
C1183	parameters can't have shared types
C1184	a dynamic THREADS dimension requires a definite block size
C1185	shared variables must be static or extern
C1186	argument of upc_blocksizeof is a pointer to a shared type (not shared type itself)
C1187	affinity expression ignored in nested upc_forall
C1188	branching into or out of a upc_forall loop is not allowed
C1189	affinity expression must have a shared type or point to a shared type
C1190	affinity has shared type (not pointer to shared)
C1191	shared void* types can only be compared for equality
C1192	UPC mode is incompatible with C++ and K&R modes
C1193	null (zero) character in input line ignored
C1194	null (zero) character in string or character constant
C1195	null (zero) character in header name
C1196	declaration in for-initializer hides a declaration in the surrounding scope
C1197	the hidden declaration is at line <i>xxxx</i>
C1198	the prototype declaration of entity-kind "entity" (declared at line <i>xxxx</i>) is ignored after this unprototyped redeclaration

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1200	entity-kind “entity” (declared at line <i>xxxx</i>) must have external C linkage
C1201	variable declaration hides declaration in for-initializer
C1202	typedef “ <i>xxxx</i> ” cannot be used in an elaborated type specifier
C1203	call of zero constant ignored
C1204	parameter “ <i>xxxx</i> ” can’t be redeclared in a catch clause of function try block
C1205	the initial explicit specialization of entity-kind “entity” must be declared in the namespace containing the template
C1206	“cc” clobber ignored
C1207	“template” must be followed by an identifier
C1208	MYTHREAD not allowed in this context
C1209	layout qualifier cannot qualify pointer to shared
C1210	layout qualifier cannot qualify an incomplete array
C1211	declaration of “ <i>xxxx</i> ” hides handler parameter
C1212	nonstandard cast to array type ignored
C1213	this pragma cannot be used in a _Pragma operator (a #pragma directive must be used)
C1214	field uses tail padding of a base class
C1215	GNU C++ compilers may use bit field padding
C1216	use of entity-kind “entity” (declared at line <i>xxxx</i>) is deprecated
C1217	an asm name is not allowed on a nonstatic member declaration
C1218	unrecognized format function type “ <i>xxxx</i> ” ignored
C1219	base class “entity” uses tail padding of base class “entity”
C1220	the “init_priority” attribute can only be used for namespace scope variables of class types
C1221	requested initialization priority is reserved for internal use
C1222	this anonymous union/struct field is hidden by entity-kind “entity” (declared at line <i>xxxx</i>)
C1223	invalid error number
C1224	invalid error tag
C1225	expected an error number or error tag
C1226	size of class is affected by tail padding

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1227	labels can be referenced only in function definitions
C1228	transfer of control into a statement expression is not allowed
C1229	transfer of control out of a statement expression is not allowed
C1230	this statement is not allowed inside of a statement expression
C1231	a non-POD class definition is not allowed inside of a statement expression
C1232	destructible entities are not allowed inside of a statement expression
C1233	a dynamically-initialized local static variable is not allowed inside of a statement expression
C1234	a variable-length array is not allowed inside of a statement expression
C1235	a statement expression is not allowed inside of a default argument
C1236	nonstandard conversion between pointer to function and pointer to data
C1237	interface types cannot have virtual base classes
C1238	interface types cannot specify “private” or “protected”
C1239	interface types can only derive from other interface types
C1240	“ <i>type</i> ” is an interface type
C1241	interface types cannot have typedef members
C1242	interface types cannot have user-declared constructors or destructors
C1243	interface types cannot have user-declared member operators
C1244	interface types cannot be declared in functions
C1245	cannot declare interface templates
C1246	interface types cannot have data members
C1247	interface types cannot contain friend declarations
C1248	interface types cannot have nested classes
C1249	interface types cannot be nested class types
C1250	interface types cannot have member templates
C1251	interface types cannot have static member functions
C1252	this pragma cannot be used in a __pragma operator (a #pragma directive must be used)
C1253	qualifier must be base class of “ <i>type</i> ”
C1254	declaration must correspond to a pure virtual member function in the indicated base class

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1255	integer overflow in internal computation due to size or complexity of “ <i>type</i> ”
C1256	integer overflow in internal computation
C1257	__w64 can only be specified on int , long , and pointer types
C1258	potentially narrowing conversion when compiled in an environment where int , long , or pointer types are 64-bits wide
C1259	current value of pragma pack is <i>xxxx</i>
C1260	arguments for pragma pack(show) are ignored
C1261	invalid alignment specifier value
C1262	expected an integer literal
C1263	earlier __declspec(align(...)) ignored
C1264	expected an argument value for the “ <i>xxxx</i> ” attribute parameter
C1265	invalid argument value for the “ <i>xxxx</i> ” attribute parameter
C1266	expected a boolean value for the “ <i>xxxx</i> ” attribute parameter
C1267	a positional argument cannot follow a named argument in an attribute
C1268	attribute “ <i>xxxx</i> ” has no parameter named “ <i>xxxx</i> ”
C1269	expected an argument list for the “ <i>xxxx</i> ” attribute
C1270	expected a “,” or “]”
C1271	attribute argument “ <i>xxxx</i> ” has already been given a value
C1272	a value cannot be assigned to the “ <i>xxxx</i> ” attribute
C1273	a throw expression cannot have pointer-to-incomplete type
C1274	alignment-of operator applied to incomplete type
C1275	“ <i>xxxx</i> ” may only be used as a standalone attribute
C1276	“ <i>xxxx</i> ” attribute cannot be used here
C1277	unrecognized attribute “ <i>xxxx</i> ”
C1278	attributes are not allowed here
C1279	invalid argument value for the “ <i>xxxx</i> ” attribute parameter
C1280	too many attribute arguments
C1281	conversion from inaccessible base class “ <i>type</i> ” is not allowed
C1282	option “ export ” requires distinct template signatures
C1284	<i>xxxx</i>

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1285	old-style parameter list of “xxx”
C1286	illegal -A option :
C1287	too few arguments in macro invocation of “xxx”
C1288	too many arguments in macro invocation of “xxx”
C1289	too many arguments in function call to “xxx”
C1290	too few arguments in function call to “xxx”
C1291	improperly terminated macro invocation of “xxx”
C1292	type of xxx argument to “xxx” is incompatible (“type” and “type”)
C1293	operand types of xxx are incompatible (“type” and “type”)
C1294	argument of type “type” is incompatible with parameter xxx of type “type”
C1295	basis class “type” for pointer to member has not been defined
C1296	__except not allowed inside a __finally handler
C1297	syntax error in ‘warning’ pragma; ignored
C1298	syntax error in ‘message’ pragma; ignored
C1299	conflicting memory designators
C1300	void pointer treated as char pointer
C1301	cast used as an lvalue
C1302	pragma asm only allowed at file scope; moved to file scope
C1303	assembler inlining function interpreted as user function due to “no mri extension” option
C1304	no matching insert character “xxx” found in asm string
C1305	non interrupt call of an interrupt function
C1306	“ interrupt ” keyword is not allowed
C13076	entity-kind “entity” cannot be redeclared “ interrupt ” after being called
C1308	interrupt specifier allowed on function declarations only
C1309	an interrupt function cannot have a non-void return type
C1310	an interrupt function cannot have any parameters
C1311	a template function cannot be declared interrupt
C1312	a non-static class member function cannot be declared interrupt
C1313	packed/unpacked mismatch

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1314	entity-kind “entity” assumed to be “xxxx” from a previous declaration
C1315	expected “ class ”, “ enum ”, “ struct ” or “ union ”
C1316	invalid pragma option
C1317	unknown option “xxxx” found during preprocessing
C1318	could not open raw list file “xxxx”
C1319	could not open list file “xxxx”
C1320	could not open diagnostic file “xxxx”
C1321	non int in bit-field declaration; accepted
C1322	incomplete type “ <i>type</i> ” is not allowed
C1323	pointer to incomplete class type “ <i>type</i> ” is not allowed
C1324	unknown option “xxxx”; “xxxx” ignored
C1325	invalid error number/tag in option “xxxx”
C1326	target “xxxx” not implemented
C1327	unimplemented target “xxxx”
C1328	target information unavailable
C1329	option “xxxx” takes only one value
C1330	closing parenthesis expected on option “xxxx”
C1331	boolean option option-name cannot have an argument “xxxx”; argument ignored
C1332	non-boolean option option-name must have an argument; “xxxx” ignored
C1333	-n not valid with non-boolean option option-name; negation ignored
C1334	invalid numeric argument in option “xxxx”; “xxxx” ignored
C1335	potentially unsafe option “xxxx”; assumes entire program in compilation
C1336	deprecated option “xxxx”
C1337	unknown optimization “xxxx”
C1338	could not open file “file-name” mentioned in option “xxxx”; “xxxx” ignored
C1339	invalid register “xxxx” specified for option -Kh <reg>; “xxxx” ignored
C1340	expected a string after “ -Kep ”; “ -Kep ” ignored
C1341	“xxxx” is not a power of 2, option “ -Za ” requires a power of 2
C1342	invalid pragma option “xxxx”; “xxxx” ignored

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1343	option -l conflicts with -E , -Es , -P and -Ps options; -l ignored
C1344	option -l conflicts with -jH , -jHc and -jHu options; -l overrides
C1345	invalid character in file name in option “xxx”; “xxx” ignored
C1346	file name cannot be null in option “xxx”; “xxx” ignored
C1348	invalid macro definition: xxx
C1349	unknown diagnostic control option “-Qxxx”; “-Qxxx” ignored
C1350	invalid PCH directory: xxx
C1351	invalid PCH memory size: xxx
C1352	option <i>option-name1</i> not valid with option <i>option-name2</i> ; “xxx” ignored
C1353	both operands for “==” or “!=” should be “typeof()”
C1354	type used as argument; replaced with its sizeof: xxx
C1355	Option xxx no longer necessary – ignored
C1356	“long long” type downgraded to “long”
C1357	option “-Kq” implicitly implies “-nf”
C1358	options “-Kkz” and “-GZ” implicitly imply “-nf”
C1359	#pragma xxx
C1360	cannot specify a source file in #pragma options
C1361	too many -d's in command line
C1362	due to option “-Kq”, double precision variable or constant has been reduced to 32-bit precision
C1363	options stack underflow or overflow (ignored)
C1364	string literals with different character kinds cannot be concatenated
C1365	GNU layout bug not emulated because it places virtual base xxx outside xxx object boundaries
C1366	virtual base xxx placed outside xxx object boundaries
C1367	nonstandard qualified name in namespace member declaration
C1368	reduction in alignment ignored
C1369	const qualifier ignored
C1371	invalid GNU asm qualifiers
C1372	non-POD class type passed through ellipsis

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1373	a non-POD class type cannot be fetched by va_arg
C1374	the 'u' or 'U' suffix must appear before the 'l' or 'L' suffix in a fixed-point literal
C1375	option "fixed_point" can be used only when compiling C
C1376	integer operand may cause fixed-point overflow
C1377	fixed-point constant is out of range
C1378	fixed-point value cannot be represented exactly
C1379	constant is too large for long long; given unsigned long long type (nonstandard)
C1380	layout qualifier cannot qualify pointer to shared void
C1381	duplicate THREADS in multidimensional array type
C1382	a strong using-directive may only appear in a namespace scope
C1383	xxxx declares a non-template function -- add <> to refer to a template instance
C1384	operation may cause fixed-point overflow
C1385	expression must have integral, enum, or fixed-point type
C1386	expression must have integral or fixed-point type
C1387	function declared with "noreturn" does return
C1388	asm name ignored because it conflicts with a previous declaration
C1389	class member typedef may not be redeclared
C1390	taking the address of a temporary
C1391	attributes are ignored on a class declaration that is not also a definition
C1392	fixed-point value implicitly converted to floating-point type
C1393	fixed-point types have no classification
C1394	a template parameter may not have fixed-point type
C1395	hexadecimal floating-point constants are not allowed
C1396	option "named_address_spaces" can be used only when compiling C
C1397	floating-point value does not fit in required fixed-point type
C1398	value cannot be converted to fixed-point value exactly
C1399	fixed-point conversion resulted in a change of sign
C1400	integer value does not fit in required fixed-point type
C1401	fixed-point operation result is out of range
C1402	multiple named address spaces

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1403	variable with automatic storage duration cannot be stored in a named address space
C1404	type cannot be qualified with named address space
C1405	function type cannot be qualified with named address space
C1406	field type cannot be qualified with named address space
C1407	fixed-point value does not fit in required floating-point type
C1408	fixed-point value does not fit in required integer type
C1409	value does not fit in required fixed-point type
C1410	option "named_registers" can be used only when compiling C
C1411	a named-register storage class is not allowed here
C1412	xxxx redeclared with incompatible named-register storage class
C1413	named-register storage class cannot be specified for aliased variable
C1414	named-register storage specifier is already in use
C1415	option "embedded_c" cannot be combined with options to control individual Embedded C features
C1416	invalid EDG_BASE directory:
C1417	cannot open predefined macro file: xxxx
C1418	invalid predefined macro entry at line xxxx: xxxx
C1419	invalid macro mode name xxxx
C1420	incompatible redefinition of predefined macro xxxx
C1421	redeclaration of xxxx is missing a named-register storage class
C1422	named register is too small for the type of the variable
C1423	arrays cannot be declared with named-register storage class
C1424	const_cast to enum type is nonstandard
C1425	option "embedded_c" can be used only when compiling C
C1426	a named address space qualifier is not allowed here
C1427	an empty initializer is invalid for an array with unspecified bound
C1428	function returns incomplete class type type
C1429	xxxx has already been initialized; the out-of-class initializer will be ignored
C1430	declaration hides xxxx

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1431	a parameter cannot be allocated in a named address space
C1432	invalid suffix on fixed-point or floating-point constant
C1433	a register variable cannot be allocated in a named address space
C1434	expected "SAT" or "DEFAULT"
C1435	xxxx has no corresponding member operator delete (to be called if an exception is thrown during initialization of an allocated object)
C1436	a thread-local variable cannot be declared with "dllimport" or "dllexport"
C1437	a function return type cannot be qualified with a named address space
C1438	an initializer cannot be specified for a flexible array member whose elements have a nontrivial destructor
C1439	an initializer cannot be specified for an indirect flexible array member
C1440	invalid GNU version number:
C1441	variable attributes appearing after a parenthesized initializer are ignored
C1442	the result of this cast cannot be used as an lvalue
C1443	negation of an unsigned fixed-point value
C1444	this operator is not allowed at this point; use parentheses
C1445	flexible array member initializer must be constant
C1446	register names can only be used for register variables
C1447	named-register variables cannot have void type
C1448	__declspec modifiers not valid for this declaration
C1449	parameters cannot have link scope specifiers
C1450	multiple link scope specifiers
C1451	link scope specifiers can only appear on functions and variables with external linkage
C1452	a redeclaration cannot weaken a link scope
C1453	link scope specifier not allowed on this declaration
C1454	nonstandard qualified name in global scope declaration
C1455	implicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)
C1456	explicit conversion of a 64-bit integral type to a smaller integral type (potential portability problem)

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1457	conversion from pointer to same-sized integral type (potential portability problem)
C1459	friend specifier is not allowed in a class definition; friend specifier is ignored
C1460	only static and extern variables can use thread-local storage
C1461	multiple thread-local storage specifiers
C1462	virtual xxxx was not defined (and cannot be defined elsewhere because it is a member of an unnamed namespace)
C1463	carriage return character in source line outside of comment or character/string literal
C1464	expression must have fixed-point type
C1465	invalid use of access specifier is ignored
C1466	pointer converted to bool
C1467	pointer-to-member converted to bool
C1468	storage specifier ignored
C1469	dllexport and dllimport are ignored on class templates
C1470	base class dllexport/dllimport specification differs from that of the derived class
C1471	redeclaration cannot add dllexport/dllimport to xxxx
C1472	dllexport/dllimport conflict with xxxx; dllexport assumed
C1473	cannot define dllimport entity
C1474	dllexport/dllimport requires external linkage
C1475	a member of a class declared with dllexport/dllimport cannot itself be declared with such a specifier
C1476	field of class type without a DLL interface used in a class with a DLL interface
C1477	parenthesized member declaration is nonstandard
C1478	white space between backslash and newline in line splice ignored
C1479	dllexport/dllimport conflict with xxxx; dllimport/dllexport dropped
C1480	invalid member for anonymous member class -- class type has a disallowed member function
C1481	nonstandard reinterpret_cast
C1482	positional format specifier cannot be zero
C1483	a local class cannot reference a variable-length array type from an enclosing function

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1484	member xxxx already has an explicit dllexport/dllimport specifier
C1485	a variable-length array is not allowed in a function return type
C1486	variable-length array type is not allowed in pointer to member of type type
C1487	the result of a statement expression cannot have a type involving a variable-length array
C1488	support for trigraphs is disabled
C1489	the xxxx attribute can only appear on functions and variables with external linkage
C1490	strict mode is incompatible with treating namespace std as an alias for the global namespace
C1491	in expansion of macro "xxxx" xxxx,
C1493	in expansion of macro "xxxx" xxxx
C1494	[xxxx macro expansions not shown]
C1495	in macro expansion at
C1496	invalid symbolic operand name xxxx
C1497	a symbolic match constraint must refer to one of the first ten operands
C1498	use of __if_exists is not supported in this context
C1499	__if_exists block not closed in the same scope in which it was opened
C1500	thread-local variable cannot be dynamically initialized
C1501	conversion drops "__unaligned" qualifier
C1502	some enumerator values cannot be represented by the integral type underlying the enum type
C1503	default argument is not allowed on a friend class template declaration
C1504	multicharacter character literal (potential portability problem)
C1505	expected a class, struct, or union type
C1506	second operand of offsetof must be a field
C1507	second operand of offsetof may not be a bit field
C1508	cannot apply offsetof to a member of a virtual base
C1509	offsetof applied to non-POD types is nonstandard
C1510	default arguments are not allowed on a friend declaration of a member function
C1511	default arguments are not allowed on friend declarations that are not definitions

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1512	redeclaration of xxxx previously declared as a friend with default arguments is not allowed
C1513	invalid qualifier for type (a derived class is not allowed here)
C1514	invalid qualifier for definition of class type
C1515	no prior push_macro for xxxx
C1516	wide string literal not allowed
C1518	xxxx is only allowed in C
C1519	__ptr32 and __ptr64 must follow a "*"
C1520	__ptr32 and __ptr64 cannot both apply
C1521	template argument list of xxxx must match the parameter list
C1522	an incomplete class type is not allowed
C1523	complex integral types are not supported
C1524	__real and __imag can only be applied to complex values
C1525	__real/__imag applied to real value
C1526	xxxx was declared "deprecated (xxxx)"
C1527	invalid redefinition of xxxx
C1528	dllimport/dllexport applied to a member of an unnamed namespace
C1529	__thiscall can only appear on nonstatic member function declarations
C1530	__thiscall not allowed on function with ellipsis parameter
C1531	explicit specialization of xxxx must precede its first use (xxxx)
C1532	a sealed class type cannot be used as a base class
C1533	duplicate class modifier
C1534	a member function cannot have both the "abstract" and "sealed" modifiers
C1535	a sealed member cannot be pure virtual
C1536	nonvirtual function cannot be declared with "abstract" or "sealed" modifier
C1537	member function declared with "override" modifier does not override a base class member
C1538	cannot override sealed xxxx
C1539	xxxx was declared with the class modifier "abstract"
C1612	missing cannot-redefine flag

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1613	missing mode after ','
C1614	missing macro name
C1615	invalid cannot-redefine value
C1616	duplicate function modifier
C1617	invalid character for char16_t literal
C1618	__LPREFIX cannot be applied to char16_t or char32_t literals
C1619	unrecognized calling convention xxxx, must be one of:
C1621	option "--uliterals" can be used only when compiling C
C1622	attribute xxxx not allowed on parameter declarations
C1623	underlying type of enum type must be an integral type other than bool
C1624	some enumerator constants cannot be represented by type
C1625	xxxx not allowed in current mode
C1626	option "--type_traits_helpers" can be used only when compiling C++
C1627	attribute "sentinel" requires an ellipsis parameter
C1628	argument must be a constant null pointer value
C1629	insufficient number of arguments for sentinel value
C1630	sentinel argument must correspond to an ellipsis parameter
C1631	__declspec(implementation_key(...)) can appear only between #pragma start_map_region and #pragma stop_map_region
C1632	#pragma start_map_region already active: pragma ignored
C1633	no #pragma start_map_region is currently active: pragma ignored
C1634	xxxx cannot be used to name a destructor (a type name is required)
C1635	nonstandard empty wide character literal treated as L'\0'
C1636	"typename" may not be specified here
C1637	a non-placement operator delete must be visible in a class with a virtual destructor
C1638	name linkage conflicts with previous declaration of xxxx
C1639	alias creates cycle of aliased entities
C1640	subscript must be constant
C1641	a variable with static storage duration allocated in a specific register cannot be declared with an initializer

Table A-3. Compiler Error Messages (cont.)

Number	Message
C1642	a variable allocated in a specific register must have POD type
C1643	predefined meaning of xxxx discarded
C1644	option xxxx requires run-time initialization of pointers
C1645	"__retain" is not allowed
C1646	__retain specifier allowed on function declarations only
C1647	exiting upon receipt of signal xxxx
C1648	options "-Kkz" and "-GZ" mutually exclusive; "-Kkz" ignored
C1649	options "-Kknp" and "-GP" mutually exclusive; "-Kknp" ignored
C1650	invalid GNU version in option "-xxxx"; "-xxxx" ignored, setting default GNU version 4.1
C1651	unsupported GNU version in option "-xxxx"; "-xxxx" ignored, setting default GNU version 4.1
C1652	"section" attribute ignored, can't be used with uninitialized variable
C1653	source file appears to be incorrectly formatted; line feed character is required to separate source lines

1. An expanded description is provided following this table.

Expanded Descriptions

This section provides further descriptions and examples of selected error messages from the list in [Table A-3](#).

C0004 out of memory

The compiler failed to get enough memory from the system. Refer to the Release Notes for specific information on memory requirements.

C0019 extra text after expected end number

A number was terminated by an invalid character:

```
i = 111aaa; // extra text after expected end of number
      ^
```

C0079 expected a type specifier

An expected type specifier was not found:

```
char * not_a_type;
int * ip = new not_a_type; // expected a type specifier
              ^
```

```
int * ip = new int;           // OK: 'int' is a valid type
```

C0082 storage class is not first

The storage class was not first in a declaration. This is an informational message in normal mode and a warning in strict ANSI mode.

```
int static i; // storage class is not first
  ^
```

C0083 type qualifier specified more than once

A type qualifier (**const** or **volatile**) was specified more than once in a declaration.

```
// type qualifier specified more than once
const const int i = 1;
  ^
```

C0084 invalid combination of type specifiers

An invalid combination of type specifiers (such as **short** and **float**) was used in the declaration.

```
// short cannot be used with float
short float f;
  ^
```

C0085 invalid storage class for a parameter

The storage class specified for a parameter is not valid. Only register (C/C++) and auto (C++) are valid storage classes for parameters.

```
// invalid storage class for a parameter
extern int f(static int i);
             ^
```

C0086 invalid storage class for a function

The storage class specified for the function is not valid. Only **extern** and **static** are valid storage class specifiers for a nonmember function.

```
// register cannot be specified as the storage class
// for a function
register int f();
          ^
```

C0095 array is too large

The size of the array, which is the product of the size of one element multiplied by the declared number of elements, is larger than the largest value of type *size_t*. Therefore, the array cannot fit in the address space of the target processor. For example, in PowerPC systems the maximum size is 4,294,967,295 bytes.

C0097 a function may not return a value of this type

The following are the legal return values are:

- **void**
- Object types other than arrays
- References to object types other than arrays

You cannot return functions or arrays, but you can return pointers to functions or pointers to the first element of an array.

C0098 an array may not have elements of this type

The element type cannot be incomplete; for example:

```
struct to_be_named;  
to_be_named team[10];
```

The element type cannot be a type that cannot be put into an array (**void**, a function, or a reference). In general, do not use anything that cannot be used in a **sizeof(type)** expression.

C0103 class is too large

The total size of the class declared is larger than the maximum value of type *size_t*. Therefore, the class or struct will not fit in the address space of the target processor. In PowerPC systems the maximum size is 4,294,967,295 bytes.

C0104 struct or union is too large

The total size of the **struct** or **union** declared is larger than the maximum value of type *size_t*. Therefore, the class or struct will not fit in the address space of the target processor. In PowerPC systems the maximum size is 4,294,967,295 bytes.

C0105 invalid size for bit field

Bit fields cannot have negative sizes, nor can they have sizes larger than the number of bits supported for their type (**char**, **short**, **int**, and so forth).

In PowerPC systems no bitfield can have more than 32 bits.

C0106 invalid type for a bit field

Bit fields must be declared with integral types (**char**, **short**, **int**, **long**, and their unsigned variants).

C0120 return value type does not match the function type

The value of the expression in return expression must be the same as, or convertible to, the value declared for the function.

C0126 expression must be an lvalue

See *The Annotated C++ Reference Manual* for a definition of lvalue. The context in which this diagnostic was issued required an object that was (or can be considered to be) allocated to memory, so that its address could be taken. A literal object will not suffice.

C0137 expression must be a modifiable lvalue

See *The Annotated C++ Reference Manual* for a definition of lvalue. The context in which this diagnostic was issued required an object that could be modified. Literal constants are not acceptable; neither are functions nor objects with the **const** attribute:

```
#include <string.h>
...
*strcpy = *strcat;
```

You cannot copy the function pointed to by `strcat` into the space allocated for the function **strcpy**.

```
const int x = 1;
x = 2;
```

x is **const**, and cannot be assigned a value.

C0153 expression must have class type

The expression that received the diagnostic contains a field selection operator (“.” or “.*”). If the second operand was the name of a member function, the first operand should be an object of a class that has a member of that name.

C0154 expression must have struct or union type

The expression that received the diagnostic contains a field selection operator (“.”) and the first argument is not an object of a **struct** or **union** type.

C0155 old-fashioned assignment operator

An obsolete K&R C style assignment operator (for example, ‘=’ instead of ‘-=’) was used.

C0156 old-fashioned initializer

An obsolete K&R C style initializer (without ‘=’ between the variable and the initial value) was used.

C0158 expression must be an lvalue or a function designator

See *The Annotated C++ Reference Manual* for a definition of lvalue. This diagnostic is issued when the argument of the unary operator (&) (that is, the `address_of` operator) is not something that has an address:

```
int *x = &7;
```

C0164 name of directory for temporary files is too long ("xxxx")

The length of a temporary file (composed of a directory name you specified using the `TMPDIR` environment variable, plus a filename generated by the C or C++ compiler) exceeds the internal buffer capacity of 150 bytes.

C0170 pointer points outside of underlying object

The result of a pointer arithmetic operation (pointer +/- constant) has produced a pointer whose validity cannot be guaranteed.

```
int x[10];
int *y;

y = x+2;    /* legal */
y = x-2;    /* illegal, outside x[] */
y = x+11;   /* illegal, outside x[] */
y = x+10;   /* legal, _just past_ x[] */
```

The second and third assignments are illegal, because the pointer constructed (y) does not clearly point to an **int** object. The last example is legal, because obtaining the address of the object just beyond the end of an array is legal. It is not legal to dereference that pointer, but it is legal to use it in comparisons and further arithmetic.

C0172 external/internal linkage conflict with previous declaration

The variable/function flagged by this diagnostic has had a prior definition (perhaps in a header file), and the current (re)definition is changing the linkage:

```
static int x;
extern x;

extern int f(int);

static int
f(int) {
...
}
```

C0224 the format string requires additional arguments

Too few arguments are supplied when using **printf** or **scanf**.

C0225 the format string ends before this argument

Too many arguments are supplied when using **printf** or **scanf**.

C0242 multiple access control specifiers are not allowed

```
class B: public
    private A {};
```

Only one access control specifier (public, private, or protected) can be specified.

C0261 access control not specified ("xxxx" by default)

```
class B:A {
...
};
// Access control of A is not specified.
// It is "private" by default
```

C0267 old-style parameter list (anachronism)

A K&R C nonprototype style parameter declaration was used:

```
void f(a)
int a;
{
    ...
}
```

C0271 access adjustment in a "private" section is not allowed

You cannot change the accessibility of an inherited member.

```
class A { public:
    int a;
};
class B: public A {
    private: A::a; // illegal to change from public to private
```

C0283 qualified name is not allowed

A qualified name is used where it is not permitted by the C++ syntax:

```
struct A {};
struct B: public A{
    typedef A::f;
}; // Qualified name A::f is not allowed in B.
```

C0296 invalid use of non-lvalue array

An array returned by a function is not an lvalue.

```
struct A {
    char name[5];
    A(){}
};
char *f() {
    return A().name;
}
```

C0297 expected an operator

After the **operator** keyword, an operator (such as +, -, ~, or >>) is required:

```
struct A {
    operator f();
}; // f is not a valid operator name.
```

C0298 inherited member is not allowed

A derived class cannot define a member function that is declared in one of its base classes:

```
struct A {
    void f();
};
struct B: public A {};
void B::f(){}
```

```
// error: cannot define B::f, which is inherited from A.
```

C0308 more than one instance of entity-kind "entity" matches the argument list:

An error occurs and all matching template instances are printed after the error message when more than one instance of a template matches the argument list.

C0309 more than one instance of constructor "entity" matches the argument list:

An error occurs and all matching constructors are printed after the error message when more than one instance of a constructor matches the argument list.

C0335 linkage specification is not allowed

A linkage specification (**extern** or **static**) is not permitted in the current context:

```
struct A {  
    extern "C" A();  
}; // extern "C" is not allowed; member functions  
    // always have C++ linkage.
```

C0336 unknown external linkage specification

Only "C" and "C++" linkages are supported:

```
extern "D" void f();  
// D is not recognized.
```

C0337 linkage specification is incompatible with previous "entity" (declared at line xxxx)

A declaration contains a linkage specification that is different from the previous declaration:

```
extern int a;  
static int a;  
// The second declaration is incompatible with the first one.
```

C0338 more than one instance of overloaded function "entity" has "C" linkage

Functions declared with "C" linkage cannot be overloaded:

```
extern "C" void f(int);  
extern "C" void f(char);
```

C0348 more than one user-defined conversion from "type" to "type" applies:

This diagnostic is followed by a list of all possible conversion functions. This happens during the resolution of overloaded functions.

C0350 more than one operator "xxxx" matches these operands:

This diagnostic is followed by a list of all possible operators. This happens during the resolution of overloaded functions.

C0366 entity-kind "entity" provides no initializer for:

This diagnostic is followed by a list of the related **const** and reference data members. A user-defined constructor of some class that has **const** or reference data members must have initializers for the data members listed.

C0368 entity-kind "entity" defines no constructor to initialize the following:

This diagnostic is followed by a list of the related **const** and reference data members. When a class contains **const** or reference data members, a user-defined constructor is required.

C0379 cast of bound function to normal function pointer (anachronism)

A cast of a pointer to a member function to a normal function pointer is illegal:

```
struct A {
    int f();
};
A *p = new A;
int (*pf)() = (int(*)())p->f;
// Incorrect. pf is a regular function pointer.
int (A::*pmf)() = p->f;
// pmf is a pointer to a member function in A.
```

C0395 single-argument function used for postfix "xxxx" (anachronism)

Older versions of C++ did not permit the prefix and postfix operators to be overridden independently. The definition of an overloaded operator that was used for both is now only the prefix operator. See *The Annotated C++ Reference Manual*.

```
struct A {
    operator ++();
    operator --();
};
void f() {
    A x;
    x++;    // warning 0395
    x--;    // warning 0395
}
```

C0397 implicitly generated assignment operator cannot copy:

This message is given when a class contains reference or **const** data members; an implicitly generated assignment operator cannot handle this case.

C0416 more than one constructor applies to convert from "type" to "type":

This diagnostic is followed by a list of all possible conversion functions.

C0417 more than one conversion function from "type" to "type" applies:

This diagnostic is followed by a list of all possible conversion functions.

C0418 more than one conversion function from "type" to a built-in type applies:

If class A and B have a conversion operator to **int** and class C is derived from A and B, the conversion from C to **int** is ambiguous.

This diagnostic is followed by a list of all possible conversion functions.

C0456 excessive recursion at instantiation of entity-kind "entity"

A loop in template instantiation occurred:

```
template <class X>
int foo(X n) {
    return foo (&n);
}
// Calling foo (an_int) creates infinite recursion on
// instantiation of foo(int), foo(int*), foo(int**), and so
// forth.
```

C0469 tag kind of xxxx is incompatible with declaration of entity-kind "entity" (declared at line xxxx)

You cannot use the same name tag for a **struct**, **class**, **enum**, **union**, or **typedef**:

```
class A {};
enum A {x,y};
// A is reused.
```

C0500 extra parameter of postfix "operatorxxxx" must be of type "int"

The extra parameter for post-increment and post-decrement operators must be an **int**. (See *The Annotated C++ Reference Manual* for more information.)

C0511 enumerated type is not allowed

The **enum** promotes to integer for arithmetic operations, and it cannot then be converted back to enum. An integer cannot be converted to an enum automatically. (See *The Annotated C++ Reference Manual* for more information.)

C0520 initialization with "{...}" expected for aggregate object

An initializer for an aggregate type must be surrounded by curly braces "{ }".

C0521 pointer-to-member selection class types are incompatible ("type" and "type")

Pointer-to-member types cannot be converted to other types (for example, to an **int**).

```
class A; class B;
int foo(int A::*p, B*x) {
    return x->*p; // incompatible with int
}
```

C0522 pointless friend declaration

There is no need to define a member function as a **friend**. Also, there is no need to define a class as its own **friend**.

C0525 a dependent statement may not be a declaration

The following statement is an error in Cfront-compatibility mode:

```
if (n) int x;
```

C0546 transfer of control bypasses initialization of:

This diagnostic is followed by a list of variables. These are not initialized, since control is transferred around their declaration.

C0553 external/internal linkage conflict for entity-kind "entity" (declared at line xxxx)

The function declaration says that the function is static, but the template declaration says the function is external:

```
static int foo(int);
template <class T>
int foo(T f);
```

C0561 invalid macro definition:

This diagnostic is followed by either command line or #pragma input:

```
cccfamily t.cc -D123=1
```

where *family* represents the processor family, or in the file *t.cc*:

```
#pragma options -D123=1
```

The preceding examples result in an invalid macro definition since the identifier “123” starts with a digit.

C0562 invalid macro undefinition:

This diagnostic is followed by either command line or #pragma input.

```
cccfamily t.cc -U123=1
```

where *family* represents the processor family, or in the file *t.cc*:

```
#pragma options -U123=1
```

The preceding examples result in an invalid macro undefinition since the identifier “123” starts with a digit.

C1653 source file appears to be incorrectly formatted; line feed character is required to separate source lines

This file lacks the line-feed character (ASCII 13), which suggests it may have come from an environment that does not use this character to separate source lines. This file should be converted to Unix or Windows format.

Appendix B

C and C++ Differences

Although C++ was originally based on the C language, it is not a simple extension of C. C++ has evolved into a language that has some of the same operations as the C language, modifies some of the original C operations and contains many new features not available in C. This chapter discusses C++ language operations that also exist in C but do not perform as defined by the C language.

C++ Program Structure

The following sections describe the differences between the C program structure and the C++ program structure.

void*

Both ANSI C and C++ permit assignment of any pointer type to a void pointer (**void***).

Example B-1. void Pointers

```
void legal () {
    int *ip;
    void *vp;
    vp = ip;    /* assigns int pointer to void pointer */
}
```

The code in this example is correct for both ANSI C and C++. However, the converse (conversion of a **void** pointer to any pointer type) is valid only in ANSI C.

```
void illegal() {
    char *cp;
    void *vp;
    cp = vp;    /* assigns void pointer to char pointer */
}              /* C++ error */
```

The code in this example is correct for ANSI C only. C++ generates an assignment type error. The last line can be modified as follows, so that it is correct in both C and C++:

```
cp = (char*) vp;
```

Global and Local Scope Issues

In ANSI C, a global variable can be redeclared without error. C++ allows only one declaration; any other declarations of that variable must be declared as **extern**.

Example B-2. Global and Local Declarations

```
int i=1;    /* first global declaration of i */
int my_function(int j);

int main() {
    int i;    /* local declaration of i */
    i = 9;    /* local i */
    my_function(i); /* local i */
    return(i); /* local i */
}

int i;    /* Second declaration of i */
/* ERROR - Redefinition in C++ */
/* Allowed in C++ if declared as extern. */
/* Redefinition in C */
int my_function(int j)
{
    j=10 + i; /* global i */
}
```

This example declares a local variable **i** within the function **main** and uses it within that function. It also defines a global variable **i** once at the top of the program and again before the **my_function** routine. ANSI C accepts the second declaration of the global variable **i**, but C++ generates an error message stating that there are two definitions of **i**. In order to modify the preceding example for C++, remove one of the global declarations of **i**, or change the second global declaration of **i** to read: **extern int i**. Similarly, if the two global declarations of **i** appear in separate files, one of the declarations must use the keyword **extern**.

Declarations

Variables must be declared before they are used in both C and C++. However, C++ permits the declaration to occur closer to the point of the first usage by permitting declarations to appear among executable statements rather than at the beginning of a block only.

Type Qualifiers

In C++, the **const** keyword instructs the compiler to avoid modifying the value of an object through a specific name:

```
int x = 0;
const int *ip = &x;  // x cannot be updated using ip
```

In C, however, the **const** keyword typically implies a compile-time constant that is stored in read-only memory:

```
struct A{
    int i;
};
const struct A aobj;
```

The C compiler stores **aobj** in the **.rodata** section (ROM). The C++ compiler, however, places **const** objects such as **aobj** into ROM only if the object has no constructor and no destructor. Since **aobj** has no constructor or destructor, it can be safely placed in ROM. Microtec C++ compilers perform special **const** storage optimizations so that read-only **const** objects such as **aobj** are placed in the **.rodata** section:

```
class Number {
    int i;
    int j;
public:
    Number() { // constructor
        i = 0;
        j = 0;
    }
    ~Number() { // destructor
        i = -1;
        j = -1;
    }
};
Number bobj;
```

In this example, **bobj** cannot be placed in ROM since it has both a constructor and a destructor. Additionally, the constructor **Number::Number()** changes the value of the data members of **bobj** at run time.

In ANSI C, a global **const** is automatically considered to be external and recognized across files. In C++, this declaration is only recognized within the scope of the current file. The **const** is discarded if its address is not taken:

```
const int debug = 0;
```

In ANSI C, this declaration is recognized as a global constant. In C++, the constant **debug** is recognized only within the file in which it was declared. To avoid confusion, you should define **const** as follows:

- **extern** to identify a constant that is recognized across files
- **static** to identify a constant that is only recognized within the scope of the current file

You must declare **const** explicitly as either **extern** or **static** to write an ANSI C program with code that is compatible with C++.

Example B-3. Declaring const for C++

```
extern const external_const;
static const static_const=200;
```

Initialization

C++ does not permit initialization to be skipped over by jumps such as **goto** statements:

```
char c;
void main() {
    goto label1;
    {
        int x = 1;
    label1:
        c = 20;
    }
}
```

The code shown in this example is illegal for C++ since the **goto label1** statement skips the initialization of **x** (**int x = 1**).

Type Definitions

class, **struct**, **union**, **enum**, and type definitions belong to the same name space in C++. In ANSI C, the **struct** names and type definition names belong to distinct name spaces:

```
typedef char A;
A cobj;
struct A { /* C++ error: A redefined */
    long foo;
    int i;
};
struct A aobj;
void main () {
    cobj = 'a';
    aobj.foo = 2;
}
```

The code in this example is legal for ANSI C, since **struct A** and **typedef A** belong to separate name spaces. However, the code is illegal for C++ because a **class** that includes a **struct**, **union**, or **enum** cannot have the same name as a **typedef** name. The code can be legal for C++ only if the **typedef** name and the **class** name refer to the same type.

```
class T {
    ...
};
typedef class T T;
```

In this example, **class T** and **typedef T** can coexist because they refer to the same basic type.

```
struct FIRST { . . . };
typedef int FIRST; /* C++ error: FIRST redefined */
```

In this example, the **typedef** declaration of **FIRST** generates an error message stating that **FIRST** was redefined. C++ does not allow a name to be declared as both a structure tag and a different **typedef** name in the same scope.

C++ allows the following explicit constructs:

```
typedef struct REDECL REDECL;
typedef struct TWICE {...} TWICE;
```


Keywords

In addition to the ANSI C keywords, C++ also includes the following keywords:

asm	friend	private	throw
bool	inline	protected	try
catch	new	public	virtual
class	operator	template	wchar_t
delete	overload ¹	this	

1. The **overload** keyword is an anachronism and might not be supported in future releases.

C++ Grammar Rules

Unlike ANSI C, C++ does not allow redundant parentheses in declarations. The additional restrictions are necessary because C++ supports overloading of the conversion operator ().

Example B-4. Conversion Operator

```
class Dummy {
    int c;
public:
    operator int () { return c; }
};
```

In this example, a conversion function from (**class Dummy ***) to (**int**) is defined in **class Dummy**'s class definition. You can convert objects of **class Dummy** to **int**-typed objects, as shown:

```
int cobj;
Dummy *d;
...
cobj = int (d);    // conversion operator function called.
```

You can achieve the same results by calling the conversion operator function:

```
int (d);
```

Consider the following code:

```
int (x);
void proc() {
    int (y);
}
```

In ANSI C, the code in this example declares a global **int x** and a local **int y**. In C++, it also declares a global **int x**, but the **int (y)** declaration is ambiguous. **int (y)** can be interpreted as a conversion of the variable **y** to **int** or a potential constructor call. In both cases, the statement **int (y)** is not considered to be a declaration, and the compiler considers the use of **y** to be an error since the variable **y** is undefined.

You can resolve the ambiguity by removing the redundant parentheses, thus declaring **y** with the declaration **int y**, or by adding an explicit declaration of **y** to the existing code, as shown:

```
int (x);
void proc() {
    int y;    /* declaration of y */
    int (y); /* call y's constructor */
}
```

Avoid using extra parentheses to write ANSI C code that is compatible with C++:

ANSI C

```
short (sa[2]);
char *(one_char);
```

C++-Compatible

```
short sa[2];
char *one_char;
```

Functions

All executable statements in a C program are contained in functions. In C++, however, a declaration is a statement and can be executed.

Function Declaration

The function declaration **f()**; is interpreted differently for pre-ANSI C, ANSI C, and C++, as shown in [Table B-1](#).

Table B-1. Function Declarations

C Language	f()	f(void)
Traditional C	Takes any number of arguments	Illegal
ANSI C	Takes any number of arguments	Takes no arguments
C++	Takes no arguments	Takes no arguments

To avoid confusion, declare a function that takes no arguments as **f(void)** for both ANSI C and C++.

Example B-5. Function Declarations

```
int f(void);
void fcall() {
    f();
}
```

Functions Used Without Being Declared

When using the Microtec ANSI C compiler, you can call a function before it has been declared. C++, however, requires a function prototype definition or declaration before that function can be called.

Example B-6. Prototype Declarations

```
void main() {
    int i, j = 2;
    i = my_function(j);
}

int my_function (int j)
{
    return (j+10);
}
```

The code in this example is legal in ANSI C; however, this example is illegal in C++. You must add the **my_function** function prototype declaration in order to make this example legal in C++:

```
int my_function (int j);
```

prior to calling **my_function**. Alternatively, move the **my_function** definition above the calling routine—**main()**, in this case. The next example shows how C++ enforces the declaration of a function before its use:

```
void main() {
    printf("hello\n");
}
```

This example is legal in ANSI C even though *stdio.h*, which contains the definition of the **printf()** function, is not included. This example is not legal for C++, however, because there is no default type information for the **printf()** function. As a result, an error message is generated stating that the undefined function **printf** was called. Include the *stdio.h* header file to fix this error.

Using Prototype Function Declarations

Microtec C compilers accept traditional C function declarations, such as nonprototyped declarations. Function prototype declarations allow stronger type checking, which is preferred in ANSI C and required in C++:

```
int func (s,i)
char *s;
int i;
{
    if (*s == 'z')
        i++;
    else
        return (-1);
    return (i);
}
```

In this example, the function **func** is declared in the traditional C language format. The prototype function declaration for the same function would be:

```
int func (char *s, int i)
```

The following code is interpreted in ANSI C to mean the arguments of **p()** are not known, which is not a problem. C++, however, rejects the code, since the **void p()** declaration is interpreted as declaring that the function **p** does not take any arguments. Therefore, the function call **p(1,2)** is considered an error:

```
void p();  
void main() {  
    p(1,2);  
}
```

The compiler must be informed if **p** is an external C function. In order to use an ANSI C linkage name from C++ code, you must explicitly state that the function is a C routine:

```
extern "C" {  
    void p();  
}
```

Refer to Chapter 10, “[Interlanguage Calling](#)” in this manual for more information on the **extern “C”** construct.

Comments

The ANSI C language uses one set of tokens (**/* ... */**) to identify comments. Comment text is placed between the tokens. C++, in addition to supporting the standard comment token set, also supports the double slash (**//**) comment token. This comment token is terminated by the **end-of-line** character. Any of the comment tokens can appear within the double slash (**//**) token:

```
// this is an example /* */ //
```

In the example shown, all tokens following the opening double slash (**//**) comment token are considered part of the comment until the **end-of-line** character.

/* ... */ comments cannot be nested, although the double slash (**//**) and slash-star (**/***) tokens are permitted within the **/* ... */** comments.

Internal Data Representation

This section describes the differences between the C program structure and the C++ program structure.

Enumeration Types

Assigning an invalid enumeration literal causes the C++ compiler (and some ANSI C compilers) to generate a warning.

Example B-7. Enumeration Types

```
enum E0 { red, green, blue };
enum E1 { orange, yellow };
void main() {
    enum E0 eobj;
    eobj = yellow;    // yellow does not belong to enum type E0
}
```

The C++ compiler generates an error message for the code shown, indicating that you have assigned an invalid enumerator (**yellow**) to the variable **eobj** of enumeration type **E0**.

In ANSI C, a function that takes an enumerator can be called with the name of an enumerator of that type or with its enumerator value. In C++, you must use the appropriate enumerator name:

```
enum E0 { red, green, blue };
void efunc (enum E0 eparm) {
    ...    /* some code */
}
void main () {
    efunc(red); /* call efunc() with proper enumerator */
    efunc(1); /* call efunc() with number instead */ /* of enumerator ->
warning message. */
}
```

In this example, both ANSI C and C++ accept the function call **efunc(red)** as legal. However, only ANSI C accepts the function call **efunc(1)** as legal (and invokes the **efunc()** function with **green**). C++ generates an error message telling you that you have assigned an **int** to an **enum**.

Arrays

The C compiler accepts strings longer than a character array to initialize the character array, since this is permitted by the C language (extra characters are truncated). The C++ compiler generates an error message reporting that there are too many values to initialize:

```
static char array[3] = "abc";
```

This example is considered legal by the C compiler but not by the C++ compiler. There is no space for the three characters **abc** plus the (implied) null termination byte because **array** can only hold three characters.

Nested Structures

In the C language, a **struct** declared within another **struct** (a nested structure) is promoted to the next higher scope. In C++, a **struct** within a **struct** is considered to have a scope of its own and is not recognized in a higher scope unless it is correctly referenced by the C++ scope resolution operator (**::**).

Example B-8. Nested Structures

```
struct S {
    struct Inner {
        short ss;
    };
    int i;
};
void main() {
    struct Inner *innerp;
}
```

In this example, the variable **innerp** is recognized by C but not by C++. Change the declaration of **innerp** as follows to change the code in this example for C++:

```
S::Inner *innerp;
```

The term **S::Inner** refers to the **struct Inner** defined in **struct S**. The pointer variable **innerp** is considered to be of type **S::Inner *** in C++, not type **struct Inner ***. The nested structure, **struct Inner**, is also known as the nested class of **struct S**.

Enumerators Within Structures

In the C language, enumerators declared within a structure are promoted to the next higher scope. In C++, enumerators declared within a structure are considered nested within the scope of their enclosing structure, and they are not exported to the global scope or promoted to the next higher scope.

The following example illustrates the subtle difference between the scope processing of enumerators and enum tags by ANSI C and C++. This difference is applied to the enumerators and the enum tags declared within a **struct**, **class**, or **union**.

Example B-9. Processing of Enumerators

```
#include <stdio.h>

struct Passenger {
    enum Flight {Domestic, International } code;
    /* whether the passenger is taking a
    ** domestic or international flight
    */
    char *name;
    /* flight name prefixed by 'D' for domestic
    ** flight and 'I' for international flight
    */
};

void AccessPassenger (struct Passenger *p) {
    enum Flight fcode;
    /* Strictly speaking, this declaration is
    ** not legal since the enumerator type tag
    ** Flight is really not exported outside of
```

```
    ** struct Passenger. However, you are
    ** permitted, though not encouraged, to use
    ** this construct for backward
    ** compatibility and migration from C to
    ** C++. One should declare 'fcode' as
    ** follows:
    **     Passenger::Flight fcode;
    */

    if (p->name && p->name[0] == 'D')
        fcode = Domestic;
    /* NOT OK since the enumerator Domestic is
    ** within the scope of struct Passenger and
    ** is not exported to the global scope.
    */
#ifdef __cplusplus
    else if (p->name && p->name[0] == 'I')
        fcode = Passenger::International;
    /* OK since we qualify the enumerator
    ** International with proper enclosing
    ** class name Passenger
    */
#endif

    printf ("Flight Name: %s; Flight Code: %s\n",
            p->name, p->code);
}

#ifdef __cplusplus
/*
** SetCode: set the given flight code to the
** given passenger object pointer.
*/
void SetCode (struct Passenger *p,
              Passenger::Flight f) {
    /* Note that the enum type Flight is nested
    ** inside struct Passenger in struct
    ** Passenger's definition. Per the AT&T C++ 2.1
    ** Product Reference Manual, section 7.2, to
    ** access the tag 'Flight', you must state
    ** 'Passenger::Flight' instead of just
    ** 'Flight'.
    */
    p->code = f;
}
#endif
```


Appendix C

Migration Guide

This appendix describes some of the changes to this version of the compiler from previous versions.

Optimization Option Changes

Table C-1 describes changes made to the **-O** (optimization) options controlling optimizations in the compiler.

Table C-1. Deprecated Optimization Options

Option	Description	Notes
-On	Turned on a predefined set of optimizations	-O option turns on all safe optimizations.
-OA	Allocated registers based on reducing overhead between procedures.	Now enabled with -Og .
-OI	Inlined function calls within and across modules.	Ineffective optimization. Removed.
-OLfn	Regulated forward code motion.	Now enabled with -Og .
-OLh	Used a register to hold array elements passed between iterations.	Ineffective optimization. Removed.
-OLiv	Removed loop control variables.	Ineffective optimization. Removed.
-OLue	Controlled number of iterations of an unrolled loop based on assumptions.	Unsafe optimization. Removed.
-OLuln	Determined number of times to unroll a loop.	Now enabled with -OU .
-ONfen	Determined treatment of optimizations based on possibility of non-numeric values.	Now enabled with -Of .
-ONffn	Controlled floating point constant folding.	Now enabled with -Of .
-ONfr	Permitted expression rearrangement across parentheses.	Now enabled with -Of .
-ONie	Ignored integer overflow when applying optimizations.	Unsafe optimization. Removed.
-OR	Allocated heavily used variables to registers.	Now enabled with -Og .
-OXan	Controlled alias analysis.	Now enabled with -Og .

Table C-1. Deprecated Optimization Options (cont.)

Option	Description	Notes
-OXcx	Controlled copy propagation.	Now enabled with -Ol .
-OXd	Performed main optimizations to entire routines.	Now enabled with -Oh .
-OXf	Performed control flow optimization.	Now enabled with -Og .
-OXiin	Controlled aggressiveness of in-lining on inline functions.	Now enabled with -Oi .
-OXiln	Controlled library function in-lining.	Now enabled with -Oj .
-OXinx	Excluded named functions from in-lining.	No longer available.
-OXiun	Controlled aggressiveness of in-lining on user routines.	Now enabled with -Oi .
-OXiyx	Specified functions to consider for in-lining.	No longer available.
-Oxkx	Controlled constant propagation.	Now enabled with -Ol .
-OxMn	Controlled optimization memory usage.	Deoptimization no longer available.
-OXmx	Controlled call modification analysis.	Now enabled with -Og .
-OXon	Controlled location of “main” optimizations.	Now enabled with -Og .
-Oxrn	Controlled allocation of register variables.	Now enabled with -Og .
-OXS	Used assumption that domain and range errors would not occur in intrinsic routines.	Now enabled with -OJ .
-OXsn	Controlled instruction scheduling.	Now enabled with -Or .
-OXvg	Assumed all global variables were volatile.	Deoptimization no longer available.
-OXvp	Assumed all pointers pointed to volatile objects.	Deoptimization no longer available.
-OXvs	Assumed all static variables were volatile.	Deoptimization no longer available.
-OXw	Assumed whole source program was present in current compilation.	Ineffective optimization. Removed.
-OXxn	Controlled “extra” optimizations.	Now enabled with -Og .
-OXz	Performed zone expansion.	Now enabled with -Og .

Option Specification Format Changes

All command line options now begin with a minus sign instead of beginning with a plus sign (+) or minus sign (-). Most options indicated by a plus sign have been removed; those that remain (-

tl, **-tm**, and **-tu**) have been changed to begin with a minus sign. Refer to the following section, “[Removed Options](#)”, for more information about removed options.

Removed Options

[Table C-2](#) lists options removed from version 3.0 of the compiler.

Table C-2. Options Removed in Version 3.0

Option	Former Behavior
+e	Generated uninitialized virtual tables. The compiler now always generates initialized virtual tables, and the linker removes redundant copies.
-Gf	Generated fully qualified pathnames. This behavior is now always done.
-I@	Affected search rules for source file directories. This behavior is no longer available.
-KF	Instructed the compiler to use floating point instructions. This behavior is controlled by the -f option.
-Knsadm	Used memcpy() for struct assignments. This behavior is no longer available.
+lx	Specified language dialects. The -A and -x options, and the file extensions, are now used to control the language dialect.
-X0	Allocated space for uninitialized global variables. Use the -Xp option for similar behavior.
-zc	Permitted C++-style double-slash (//) in C code. This is now always permitted.
+zsuffix	Permitted nonstandard suffixes for C++ files. This behavior is no longer available.

Added Behaviors

[Table C-3](#) describes options that control behaviors added to version 3.0 of the compiler.

Table C-3. New Options in Version 3.0

Option	Description
-ar{a d}	Controls accessing ROM with position-independent code or data.
-NIsect	Renames the .init section.
-uichar	Changes the insert character in asm statements.
-Z{a c d i m n}n	Controls alignment and padding within structures and classes.
-ze	Enables exception handling.

In addition, C99 support was added to the compiler. This added features including variadic macros, the `__func__` identifier, loop-scoped variables, compound literals, and dispersed statements and declarations. Refer to Chapter 4, “[C/C++ Extensions](#)” for more information about these features.

Appendix D

XRAY Commands and Functionality

The features and functionality described in this appendix are specific to the XRAY Debugger.

I/O System Functions

The functions described in the “[I/O System Functions](#)” section in Chapter 5, “[Using Libraries](#)”, are currently implemented to use the XRAY Debugger’s SysHost mechanism and are non-reentrant.

Functions Implemented for Use With SysHost

[Table D-1](#) lists the functions that are implemented to work with the XRAY Debugger’s SysHost feature. See Chapter 13, “[Embedded Environments](#)” for more information on system functions.

Table D-1. Functions Implemented for Use With SysHost

chdir	_popen	sys_out
connect	socket	sys_stat
getcwd	stat	
_pclose	sys_in	

User-Modified Routines for Embedded Systems

In Character Routine

All input is automatically provided through the **read** function if the SysHost feature is being used (XRAY version 4 and beyond). The XRAY Simulator **stdin** command can be used to specify the source of input data:

```
stdin f=<filename>; get input from file <filename>
```

or

```
stdin win ; fetch from stdio window
```

The SysHost feature can be disabled by compiling the libraries with the symbol **EXCLUDE_SYSHOST** defined.

The **inchrw()** routine reads characters from the **_simulated_input** variable if SysHost is not used. The XRAY Debugger INPORT command may be used to specify the source of input data for **_simulated_input**:

```
inport &_simulated_input [,input_source]
```

The XRAY Debugger reads from the standard input device if no INPORT command is issued,.

inchrw() returns an integer with a value between 0 and 255, or -1 if an error occurs.

Out Character Routine

All output is automatically provided through the **write** function if the SysHost feature is being used (XRAY version 4.0 and beyond). The following XRAY Simulator OUTPORT command can be used to specify the destination of standard output data:

```
stdout f=<filename> ; write output to file <filename>
```

Similarly, the following command can be used to specify the destination of standard error data:

```
stdout,err f=<filename> ; write errors to file <filename>
```

Output can be rerouted to the **stdio** window using the following command:

```
stdout[,err] win
```

The SysHost feature can be disabled by compiling the libraries with the symbol **EXCLUDE_SYSHOST** defined.

The **outchr(ch)** routine writes characters to the **_simulated_output** variable if SysHost is not used. The XRAY Debugger OUTPORT command may be used to specify the destination of the output data for **_simulated_output**:

```
outport &_simulated_output [,output_destination]
```

The XRAY Debugger writes to the standard output device if no OUTPORT command is issued. Its return value is ignored.

System Functions and the SysHost Feature

A part of the set of functions provided in the Microtec C/C++ Compiler run-time library includes low-level UNIX-style system functions, which are usually called indirectly through other library functions. How these functions are implemented relies on the nature of the execution environment. Most of the system functions provided are designed to work with the XRAY Debugger (Simulator or Monitor) SysHost feature. These routines (and the functions that call them) are intended to be used only during debugging.

The SysHost feature is provided to allow the application access to host resources (such as the file system) during the development phase. This feature works by filling a transfer buffer with data indicating the desired function to be performed on the host. The XRAY Debugger then stops the application and performs the requested function on the host system.

The SysHost feature is enabled in XRAY version 4.0 or later. If the SysHost feature is not enabled, then the system routines, except read and write, essentially act as stubs. A table of system functions is provided in Chapter 5, “[Using Libraries](#)”. The source code for the system functions is provided with the libraries. The SysHost-specific code can be removed by compiling the libraries with the symbol `EXCLUDE_SYSHOST` defined.

If you want to use these routines outside of the debugger, modify them to suit your environment or obtain them from a different library suitable to your environment. System routines coming from alternate sources can supersede the compiler’s libraries by specifying the filenames on the command line or in the linker command file, provided that they are in a compatible format. If the compiler library pathname is also specified, then the alternate source names must precede it. Other system routines that will require changes include the initialization routine and `_exit()` in *entry.c*; `_START()` in *csys.c*; and `_sbrk()` in *s_sbrk.c*. A description of these routines is provided in the “[User-Modified Routines](#)” section in Chapter 13, “[Embedded Environments](#)”.

If the SysHost feature is not used, then the **read** function, located in *s_read.c*, reads from the variable `_simulated_input`. `read()` returns the number of bytes read, 0 for EOF, or -1 to indicate an error. The **write** function, located in *s_write.c*, writes to the variable `_simulated_output` or returns -1 to indicate an error. Link in your implementation before the run-time library to replace any of these functions with one of your own.

The **creat** function uses the **open** function, which is implemented as a stub. You must provide your own **open** function to allow **creat** to work in your environment.

The **sbrk** function is provided as a minimal implementation. If you are developing code for an environment that requires a different **sbrk**, you can provide a replacement for the **sbrk** function by linking in your implementation of **sbrk** before the run-time library.

SysHost Functions

The following functions are implemented with XRAY SysHost functionality.

chdir — Changes Working Directory

Class

System (SysHost) Function

Syntax

```
#include <direct.h>  
  
int chdir(const char *dir);
```

Description

Changes current working directory pathname to name specified by *dir*.

Returns 0 when successful or -1 if an error is detected and sets **errno** to indicate the error.

close — Closes a Specified File

Class

System (SysHost) Function

Syntax

```
extern "C" {  
    int close (int fd);  
}
```

Parameters

- **fd**

A file descriptor previously returned by **open** or **creat**.

Description

The **close** function closes a file previously opened by either the **open** or **creat** function. A file descriptor must be specified (as opposed to a stream address).

The **close** function returns 0 when successful or -1 if the function detects an unknown file descriptor.

Notes

See Chapter 13, “[Embedded Environments](#)” for more information on system functions.

connect — Initiates a Socket Connection on the Host System

Class

System (SysHost) Function

Syntax

```
#include <socket.h>
```

```
int connect (int sock, struct sockaddr *name, int length)
```

Description

Passes the arguments to the local host's **connect** or **_connect** system call. *sock* is a socket descriptor returned from a socket call. *name* is an implementation-dependent socket name, and *length* indicates the length of this name.

Returns 0 when successful or -1 if an error is detected and sets **errno** to indicate the error.

Notes

This function has been implemented to work with the XRAY SysHost feature.

See Chapter 13, “[Embedded Environments](#)” for more information on system functions.

creat — Creates a Specified File

Class

System (SysHost) Function

Syntax

```
extern "C" {  
    int creat (char *filename, int mode);  
}
```

Parameters

- **filename**
The name of the file to be created.
- **mode**
The protection mode of the new file.

Description

The **creat** function creates and opens the file *filename* for writing. For devices that cannot be created (such as terminals), the **creat** function simply opens the device.

If **creat** is successful, it returns a file descriptor for *filename*. It returns -1 if the file cannot be written to, if the file is a directory, or if there are already too many files opened.

Notes

See Chapter 13, “[Embedded Environments](#)” for more information on system functions.

_exit — Terminates a Program

Class

System (SysHost) Function

Syntax

```
extern "C" {  
    void _exit (int status);  
}
```

Parameters

- **status**

The status code of the terminated program.

Description

The **_exit** function causes normal program termination to occur with a status code specified by *status*.

The **_exit** function is always called implicitly at the end of an **exit** function call.

Notes

A call to **_exit** never returns.

See Chapter 13, “[Embedded Environments](#)” for more information on system functions.

getcwd — Returns Current Working Directory

Class

System (SysHost) Function

Syntax

```
#include <direct.h>  
  
char *getcwd(char *buffer, int len)
```

Description

Copies up to *len* characters of current working directory pathname into returned buffer.

Returns buffer when successful or 0 if an error is detected and sets **errno** to indicate the error.

Iseek — Sets the Current Location in a File

Class

System (SysHost) Function

Syntax

```
extern "C" {  
    long lseek (int fd, long offset, int whence);  
}
```

Parameters

- **fd**
The file descriptor for the file to set location in.
- **offset**
The distance of the offset.
- **whence**
The location from which the offset is taken.

Description

The current position in the file is updated to the value given in *offset*. The *whence* argument indicates how to interpret the offset:

whence = 0 *offset* is relative to the beginning of the file.

whence = 1 *offset* is relative to the current position.

whence = 2 *offset* is relative to the end of the file.

The **lseek** function returns the resulting file position, measured in bytes from the beginning of the file. If a failure occurs, -1 is returned.

Notes

The function **lseek** is implemented simply as a stub that returns 0. Using this function can generate a link-time error message indicating that the **_WARNING_lseek_stub_used** symbol is unresolved. If your program calls the **lseek** stub, it writes the message:

WARNING - lseek (stub routine) called to **stderr**. You must provide your own **lseek** function for your environment. See Chapter 13, “[Embedded Environments](#)” for more information on system functions.

The function **lseek** can modify the static array **_iob**; no other static data can be modified.

_pclose — Closes a Pipe from a Process

Class

System (SysHost) Function

Syntax

```
#include <stdio.h>
```

```
int _pclose(FILE *pipe)
```

or

```
int pclose(FILE *pipe)
```

Description

If the *pipe* is write enabled, the buffer is flushed. The pipe is closed and the file structure is reset. If a system buffer was allocated from the heap, it is freed.

Returns 0 when successful or **EOF** if the pipe is not opened with **_popen()**.

Notes

Only **_pclose** can be used with the **-nx** option.

See Also

_popen

_popen — Opens a Pipe to a Process

Class

System (SysHost) Function

Syntax

```
#include <stdio.h>
```

```
FILE *_popen(const char *cmd, const char *mode)
```

or

```
FILE *popen(const char *cmd, const char *mode)
```

Description

Returns a pointer to the stream structure when successful, otherwise a **NULL** pointer is returned.

cmd is a pointer to a command to be executed on the host system. *mode* indicates whether the pipe is to be written to, or read from. Only **r** and **w** are allowed. The SysHost implementation of **popen** creates a pipe between the application and the host system. If the opentype is **w**, then standard input to the command is provided by writing to the returned stream, if the opentype is **r**, then standard output from the command is fetched by reading from the returned stream.

Notes

Only **_popen** can be used with the **-nx** option.

See Also

_pclose

read — Reads Bytes From a Specified File

Class

System (SysHost) Function

Syntax

```
extern "C" {  
    int read (int fd, char *buffer, int nbyte);  
}
```

Parameters

- **fd**
The descriptor associated with the file to be read from.
- **buffer**
The string where the output will be stored.
- **nbyte**
The number of bytes to be read.

Description

The **read** function reads *nbyte* bytes from the file associated with *fd* into the buffer pointed to by *buffer*. The file descriptor *fd* is returned by **open** or **creat**.

The **read** function returns the number of bytes actually read or 0 when **EOF** (End-Of-File) is reached. In the case of an error, the **read** function returns -1.

Notes

See Chapter 13, “[Embedded Environments](#)” for more information on system functions.

socket — Creates a Communication Endpoint on the Host System

Class

System (SysHost) Function

Syntax

```
#include <socket.h>
```

```
int socket(long domain, long type, long protocol)
```

Description

Pass the arguments to the host **socket()** call.

Returns non-negative descriptor when successful or -1 if an error is detected and sets **errno**.

Notes

This function has been implemented to work with the XRAY SysHost feature.

See Chapter 13, “[Embedded Environments](#)” for more information on system functions.

stat — Gets Information About File

Class

System (SysHost) Function

Syntax

```
#include <stat.h>
```

```
int stat(const char *path, struct stat *info);
```

Description

Returns information about the file specified by *path*. The information is fetched via the host's **stat()** or **_stat()** system call, and moved into the associated fields of *info*.

Returns 0 when successful, or -1 if an error is detected and sets **errno**.

sys_in — Reads Characters From Standard Input Window

Class

System (SysHost) Function

Syntax

```
#include <sysio.h>
```

```
int sys_in(char *buffer, int count, SYS_INMODE mode);
```

Description

Gets characters from the normal standard input and stores them into a buffer. *mode* indicates whether to wait for a line, multiple lines, or just return what has already been input. Depending on *mode*, *count* can indicate the maximum number of characters to read, or the maximum number of lines.

This function is not affected by other operations on the **stdin** file descriptor (for example, it is unaffected by **close(0)**).

Returns the number of characters read or -1 if an error is detected and sets **errno**.

sys_out — Sends Characters to Standard Output Window

Class

System (SysHost) Function

Syntax

```
#include <sysio.h>

int sys_out(const char *buffer, int maxbytes);
```

Description

Sends characters to standard output. This function is not affected by other operations on the **stdout** file descriptor (1) (for example, **close(1)**).

Returns non-zero if successful, or 0 if an error is detected and sets **errno**.

Notes

This function has been implemented to work with the XRAY SysHost feature.

See Chapter 13, “[Embedded Environments](#)” for more information on system and SysHost functions.

sys_stat — Checks for Input Characters Waiting

Class

System (SysHost) Function

Syntax

```
#include <sysio.h>
```

```
int sys_stat(void)
```

Description

Checks if characters are available from standard input. Returns non-zero if characters are waiting, or zero otherwise. This function is not affected by other operations on file descriptor 0 (for example, **close(0)**).

Abstract Class

A class containing a general structure that can be used only as a base class for producing specific implementations of the structure in derived classes.

Anachronism

A feature currently supported by C++ for backward compatibility with the ANSI C language or earlier C++ versions. This feature might not be supported in future releases.

Base Class

A group of data items declared in a class structure. Other classes can be derived from a base class (also known as the parent class).

Call Chain

A list of the functions linked in reverse order of their calling.

catch

A keyword in C++ that indicates a section of code for handling a particular error or exception condition. Catch statements can appear only at the end of a **try** block. When a **catch** statement is executed, it means that an exception was raised and the exception type matched the argument of this particular **catch** statement.

Catch-All Handler

An exception handler that contains statements designed to handle all exceptions. A catch-all handler catches any exception that reaches it:

```
catch (...) {  
    ...  
}
```

Class

A user-defined type. **struct** and **union** are also classes in C++.

Class Members

See [Data Members](#), [Member Functions](#).

Class Template

A template that defines a parameterized class. For example, **Stack** is a class template in the following declaration:

```
template <class T>  
class Stack { ..... };
```

const Member Function

A member function defined with the **const** keyword. Whenever a **const** member function is invoked, only **const** operations are performed; **const** operations do not change the value of the object. **const** member functions can be invoked for both **const** and non-**const** objects, whereas non-**const** member functions are not permitted on **const** objects. See also Member Functions.

const object

An object defined with the **const** keyword. The value of this object cannot be changed; only **const** operations (such as reading) can be performed on this object. **const** objects cannot call non-**const** functions.

Constructor

A constructor is a special member function used to initialize or construct a class object. It has the same name as the class.

Current Exception

An exception that has been thrown but has not been completely handled. The current exception is not considered handled until the program exits the handler.

Data Members

Data defined in a given **class**, **struct**, or **union** are called data members or class members.

Data Template

A static data member of a class template.

delete Operator

The **delete** operator destroys and deallocates the object created by the **new** operation.

Demangling

The act of decoding a C++ mangled name.

Derived Class

If a class has one or more parent classes, the class is said to be derived from those parent classes and is sometimes referred to as a derived class.

Destructor

A destructor is a special member function that destructs (deletes) class objects that must be destroyed.

Exception Handler

A section of code designated to handle a particular error or exception condition. When a particular exception occurs, the handler can put the run-time system into a stable state without having to terminate program execution.

Exception Object

An object created by a **throw** expression that is compared to the argument of **catch** statements in order to match an error or exception to a handler. See also *Thrown Object*.

Exception Specification

A list of the exceptions that a particular function can raise to the next **try** block.

friend Class

If class *X* is a **friend** of class *Y*, all member functions of **friend** class *X* are **friend** functions of class *Y*.

friend Function

A **friend** function of a class is a function that is not a member of the class but is allowed to access the private/protected members of the class.

Function Signature

Microtec C++ encodes the argument type information into the C++ function linkage name to enforce cross-module parameter type checking and overloaded functions.

Function Template

A template that defines a generic function. For example, **swap** is a function template in the following declaration:

```
template <class T>
void swap(T &x, T &y);
```

A member function of a class template is also called a function template.

Global Scope

To have the data variable, function, or class available to be called from any level in the program.

Handler

See [Exception Handler](#).

Inheritance

A derived class can inherit properties of its base class(es). The inheritable properties include data members and member functions from base class(es). A derived class can also override class members of the base class(es) or declare additional class members of the base class(es). If a class is derived from one base class, such an inheritance relation is called single inheritance. If a class is derived from more than one base class, such an inheritance relation is called multiple inheritance.

Mangling

Microtec Compiler C++ encodes type information in the linkage names it generates for function names, local variable names, and so forth. These encoded names are known as mangled names. Mangled function names are also known as function signatures.

Match

For exception handling, the pairing of an exception to the correct handler. The exception object type and the **catch** expression argument are compared for the match.

Member Functions

Operations defined on a given **class**, **struct**, or **union** are called member functions.

Name Demangling

See [Demangling](#).

Name Mangling

See [Mangling](#).

new Operator

The **new** operator creates and allocates space for an object of a given class.

Overloaded Function

Two or more functions that perform different operations can be defined with the same name. A function that uses the same function name as the other function but has different parameter types from the other function is called an overloaded function. An overloaded function can be a member or nonmember function. All overloaded functions should have the same return type.

Overloaded Operator

C++ allows most operators to be overloaded. You can define an overloaded operator function with arguments of user-defined types such as a complex arithmetic class type. All overloaded operators should have the same return type.

Pointer to Member

Pointer to a data member or a member function in a class.

Pure Virtual Function

A virtual function declared in a base class that does nothing. The details of the implementation can be left to subsequent derivations of the class.

Raise An Exception

See [Throw Expression](#).

Rethrow

When an exception has been raised, that exception can be thrown again from inside an exception handler or a custom unexpected exception function. A rethrow gives an opportunity for an exception to be handled by more than one handler.

Special Template Instance

A user-defined template instance. This usually has a different implementation from the generic template. For example, if **Stack<int>** is to be implemented differently from other generic instances, the following syntax could be used to define a special instance.

```
class Stack<int> {  
    /* special definition */  
};  
void Stack<int>::pop() {  
    /* special definition */  
}
```

When a special template instance exists, a C++ compiler does not generate the same instance from the generic template definition.

Static Data Members

A static data member's space is shared among all objects of the class. Static data members are scoped to the class. A static data member can be accessed using the scope operator.

Static Member Function

A static member function is scoped to the class. It is global to all objects of the class and, thus, does not have and does not need to be invoked with a **this** pointer.

Template Class

An instance of a class template. For example, class **Stack<int>**.

Template Data

A static data member of a template class. For example, **Stack<int>::data**.

Template Declaration

The declaration part of a class or function template.

Template Definition

The definition part of a class, function, member function, or static data member template.

Template Function

An instance of a function template or member function of a template class. For example, **swap<int>** and **Stack<int>::pop()**.

Template Instance

A template class, template function, or template data.

Template Instantiation

The creation of a template instance. Microtec C++ generates template instances at compile time.

this Pointer

The handle or the implicit pointer to an object of a given class. Each member function of the given class (except static member functions) has the **this** pointer as its first implicit argument.

throw

A keyword in C++ that indicates an error or other exception conditions exist. When a **throw** expression is executed, the exception handling system searches the appropriate **catch** block, which matches the type of the object thrown.

throw Expression

A C++ expression that raises an exception. The **throw** expression specifies the object type that is used to match the exception to the appropriate exception handler.

Thrown Object

An object created by a **throw** expression that is compared to the argument of **catch** statements in order to match an error or exception to a handler. The object can contain information that describes the exceptional condition. Also called exception object.

try Block

A block of statements that contain **throw** expressions or calls to functions that contain **throw** expressions. The **try** block indicates to the compiler that exceptions can occur in this section of code and that the handlers for these exceptions appear in **catch** blocks that follow the **try** block.

Type-Safe Linkage

For C++ functions, the mangled name encodes the name of the function and the types of its parameters. A function definition matches a use of the function only if the mangled names match. This ensures that the caller and the function definition agree on parameter types and members, thus avoiding common errors.

Unexpected Exception

An exception that was raised in a function that did not list the exception type in its exception specification list. The exception handling system reacts to an unexpected exception by calling the **unexpected** library function.

Unhandled Exception

An exception that was never matched to a handler. The exception handling system reacts to an unhandled exception by calling the **terminate** library function.

Virtual Function

A class member function whose implementation is dependent on its class type. The details of the implementation can be left to subsequent derivations of the class.

volatile Member Function

A member function defined with the **volatile** keyword. All volatile objects can invoke only volatile member functions, insuring that all operations of the volatile object are treated as **volatile**. volatile member functions will not be optimized. See also [Member Functions](#).

volatile object

An object defined using the **volatile** keyword. A volatile object can change independent of the program flow. Optimization is disabled for operations on these objects.

— Symbols —

__nullcheck_trap routine, 49
 __pure_virtual_function_called function, 157
 _exit function, 125, 348
 _main function, 150
 _simulated_input variable, 343
 _simulated_output variable, 343
 _tolower function, 163
 _toupper function, 164
 // comment token, 332
 /* and */ comment tokens, 332
 #pragma hdrstop directive, 198

— A —

-A option, 37
 Addressing ROM, absolute
 -ara, 37
 per -Md, 37
 Algebraic simplification, 177
 __ALIGNOF_CHAR mirror symbol, 77
 __ALIGNOF_CODE_POINTER mirror symbol, 78
 __ALIGNOF_DATA_POINTER mirror symbol, 78
 __ALIGNOF_DOUBLE mirror symbol, 77
 __ALIGNOF_FLOAT mirror symbol, 77
 __ALIGNOF_INT mirror symbol, 77
 __ALIGNOF_LONG mirror symbol, 77
 __ALIGNOF_LONG_DOUBLE mirror symbol, 78
 __ALIGNOF_LONG_LONG mirror symbol, 77
 __ALIGNOF_POINTER mirror symbol, 78
 __ALIGNOF_SHORT mirror symbol, 77
 __ALIGNOF_WCHAR_T mirror symbol, 77
 Allocating
 data space, 167
 data types, 214
 memory space, 160

ANSI C

 comparison to C++, 332 to 335
 function declarations, 330

ANSI-compliant mode, setting, -A option, 37

-ara option, 37

-ard option, 37

Argument

 passing, 207

 promotion, 207

Arrays, initialization, 333

ASCII character, testing for, 145

ASCII format, convert from byte, 162

ASCII string

 convert from integer, 146

 convert from long integer, 148

 convert from unsigned integer, 147

 convert from unsigned long integer, 149

ASCII string, convert from floating-point number, 142

asm pseudofunction

 definition, 226

 enabling, -x option, 66

asm support, 108

Assembler

 description, 20

 options, passing directly

 -Wa option, 36, 64

Assembler in-lining

 considerations, 234

Assembly code, optimizing by hand, 253

Assembly language

 example of a routine, 206

 interface, 201, 205

Assembly source file, 253

 advantages to producing, 253

 contents, 253

 generating, -S option, 62

 naming, -o option, 58

 saving, -H option, 44

 variable names, 253

— B —

Backspace

disable for preprocessor

-Es option, [39](#)

-Ps option, [59](#)

Banner, -Vb option, [64](#)

BIG_ENDIAN, [213](#)

Big-endian code, [46](#)

Bit fields, [51](#), [216](#)

bld_lib script, [173](#)

Branch tail merging, [185](#)

.bss section, [220](#), [224](#)

_BUS_WIDTH mirror symbol, [78](#)

Bytes

clearing memory bytes, [153](#)

convert to ASCII format, [162](#)

reading from a file, [353](#)

swapping odd and even bytes, [161](#)

writing to a file, [166](#)

— C —

C comments, saving in preprocessor output, -C option, [38](#)

C common, unutilized variables, [65](#)

__c mirror symbol, [77](#)

-C option, [38](#), [197](#)

-c option, [38](#)

C++ compiler

command line syntax, [23](#)

description, [19](#)

file locations, [25](#)

C++ language

comparison to ANSI C, [332](#) to [335](#)

function declarations, [330](#)

grammar rules, [329](#)

keywords, [329](#)

parentheses, [330](#)

wrapper, [210](#)

Call modification analysis, [56](#)

Calling functions

C++ from C, [209](#) to [212](#)

member functions, [209](#)

overloaded functions, [209](#)

member functions, [209](#)

calloc function, [239](#)

_CCPPC mirror symbol, [76](#)

CHAR mirror symbol, [77](#)

_CHAR_SIGNED preprocessor symbol, [50](#)

char type, [214](#)

_CHAR_UNSIGNED preprocessor symbol, [50](#)

Character

converting to lowercase, [163](#)

converting to uppercase, [164](#)

copying characters from memory, [152](#)

testing for ASCII character, [145](#)

chdir function, [344](#)

Checking syntax only, -y option, [67](#)

Class

declaring C++, [328](#)

nested, [334](#)

Clearing memory bytes, [153](#)

close a pipe, [351](#), [352](#)

close function, [125](#), [345](#), [346](#)

Closing a file, [345](#)

Code elimination, unreachable, [178](#)

Code hoisting, [185](#)

Code organization, compiler-generated sections, [221](#)

.code section (.text), naming, -NT option, [56](#)

CODE_POINTER mirror symbol, [78](#)

Command file, passing to linker, -e option, [39](#)

Command line options, specifying in file, -d, [38](#)

Command line, syntax, [23](#)

Comments

// token, [332](#)

/* and */ tokens, [332](#)

representing, [332](#)

saving in preprocessor output, -C option, [38](#)

Comparing values, [151](#), [154](#)

Compiler

description, [19](#)

driver, [23](#)

features, [19](#)

output

assembly source file, [253](#)

listing, [253](#)

Compiler sections, [221](#)

Concurrency control and precompiled header files, [199](#)

Configuration file, [171](#)

Conflicting options, specifying, [27](#)

const

avoiding confusing declarations, [327](#)

object, [326](#)

Constant folding optimization, [184](#)

Constant variables section (.rodata), naming, -NC option, [56](#)

Conversion operator (), [329](#)

Copying characters from memory, [152](#)

__cplusplus mirror symbol, [77](#)

creat function, [125](#), [347](#)

relationship to close, [345](#)

relationship to read, [353](#)

relationship to write, [166](#)

Creating a file, [347](#)

Cross-checking in C++, [206](#)

— D —

-D option, [38](#)

-d option, [38](#)

Data

formats, [213](#)

initialization, [245](#)

types, [214](#)

allocation, [214](#)

ranges, [214](#)

.data section, [223](#)

Data section (.data), naming, -ND option, [56](#)

Data space, allocating, [167](#)

DATA_POINTER mirror symbol, [78](#)

__DATE__ mirror symbol, [75](#)

Dead code elimination, [178](#)

_DEBUG preprocessor symbol, [41](#)

Debugging information, generating, -g option, [40](#)

Declarations, variables, [326](#)

Default initialization, [220](#)

DEFAULT_PREALLOCATED_PCH_MEM_SIZE macro, [198](#)

#define directive, [196](#), [229](#)

Defining macros on command line, -D option, [38](#)

Diagnostic messages

(see Messages, diagnostic)

Diagnostic messages, (see Messages, diagnostic)

Disable

backspace

-Es option, [39](#)

-Ps option, [59](#)

newline

-Es option, [39](#)

-Ps option, [59](#)

Display time stamp, -Vt option, [64](#)

DOUBLE mirror symbol, [77](#)

double type, [215](#)

Driver, compilation, [23](#)

Duplicate function names, [206](#)

— E —

-E option, [39](#), [197](#)

-e option, [39](#)

EDGE, [21](#)

EDGE Debugger, [21](#)

Embedded systems

code organization, [221](#)

user-modified routines, [242](#)

Enable GNU Extensions, [67](#)

end-of-line character, [332](#)

enum type, [215](#)

Enumeration

and scope, [334](#)

declaring within structures, [334](#)

types

invalid literal, [332](#)

reference by value, [333](#)

Epilogue function, [205](#)

eprintf function, [140](#)

#error directive

relation to #pragma error, [95](#)

definition, [89](#)

Error messages, [257](#)

enumeration types, [333](#)

ID numbers, -Qfn option, [61](#)

severity

-Qme option, [61](#)

-Qmw option, [61](#)

source line numbers, -Qfs option, [61](#)

suppression

- QA option, 60
- Qe option, 60
- Es option, 39, 197
- Examples
 - calling C functions from C++, 208
 - calling C++ member functions from C, 209
 - conversion operator, 329
 - declaring const for C++, 327
 - enumeration types, 333
 - function declarations, 330
 - global and local declarations, 326
 - interrupt keyword, 105 to 106
 - processing of enumerators, 334
 - prototype declarations, 331
 - void pointers, 325
- Exception handling, -ze, 67
- Exception processing, 238
- EXCLUDE_FORMAT_IO_ preprocessor
 - symbols, 171 to 173
- Executable file, suppressing, -c option, 38
- Executable program, generation, 254
- Executing only preprocessor
 - E option, 39
 - P option, 59
- _exit function, 125, 348
- exit function, relationship to _exit, 348
- Extensions
 - C language, -x option, 66
 - Microtec
 - (see Microtec extensions)
- extern "C" declaration, 207, 332
- extern declaration, 325
- External variable names, 253
- F —
- f option, 40
- Fee option, 39
- Feo option, 39
- File descriptor, getting, 141
- File locations, 25
- __FILE__ mirror symbol, 75
- _FILE_EXT mirror symbol, 77
- fileno macro, 141
- Files
 - #include, (see #include files)
 - closure, 345
 - creation, 347
 - current location, 350
 - linker command, 39
 - listing, (see Listing files)
 - making file inaccessible, 165
 - opening, 155
 - unlinking a filename, 165
- Fli option, 39, 197
- FLOAT mirror symbol, 77
- float type, 215
- Floating-point
 - convert to ASCII string, 142
 - removing unneeded support, 245
- Flp option, 39, 197
- Flp0 option, 40
- Flt option, 40, 197
- Formatted output to standard error, 140
- free function, 239
- friend, functions, calling from C, 209
- Fsm option, 40
- ftell function, 125
- ftoa function, 142
- __FUNC__ mirror symbol, 76
- __func__ mirror symbol, 76
- Function in-lining, 187
- Functions, 330
 - calling at an absolute address, 235
 - declaration, 330
 - duplicate names, 206
 - epilogue, 205
 - friend, 209
 - library, (see Library functions)
 - non-reentrant, 126, 136
 - prologue, 204
 - prototype declaration, 330
 - signature, 207
 - undeclared, 330
- G —
- g option, 40
- General optimizations, 177
- getcwd function, 349
- getl function, 143
- getw function, 144
- Global
 - constants, 327

- declarations, [325](#)
 - redefinition, [326](#)
- Global constant propagation optimization, [180](#)
- Global copy propagation, [181](#)
- Global optimizations, [178 to 183](#)
- Global variable, uninitialized, [65](#)
- H —
- H option, [44](#)
- I —
- I option, [44](#)
- I/O static initialization and termination, [252](#)
- #if preprocessor directive, [196](#)
- #include directive, use in precompiled headers, [196](#)
- #include files
 - mriext.h, [134, 135](#)
 - search path, specifying
 - I option, [44](#)
 - J option, [45](#)
 - stdio.h, [126](#)
- Index simplification optimization, [183](#)
- Induction variable elimination, [183](#)
- #info directive, relation to #pragma info, [96](#)
- Informational messages, suppression, -Qi
 - option, [61](#)
- #informing directive, [90](#)
- .init section, [251](#)
- INITDATA linker command, [246](#)
- Initialization, [245](#)
 - compile-time, [245](#)
 - data, [245](#)
 - default, [220](#)
 - run-time, [245](#)
 - static constructor, [250](#)
 - static destructor, [250](#)
 - static objects, [250](#)
 - static variables, [220](#)
- Initialization section (.init), naming, -NI
 - option, [56](#)
- In-lining
 - assembly code
 - enabling, -x option, [66](#)
 - assembly instructions, [108, 226](#)
 - intrinsic routines, [57](#)
- Input and output files, location, [25](#)
- int type, [215](#)
- Integer, convert to ASCII string, [146](#)
- Interrupt handlers, declaring, [236](#)
- interrupt keyword, [105, 236](#)
- INT mirror symbol, [77](#)
- Intrinsic routines, in-lining, [57](#)
- Invocation, compiler, [23](#)
- .ioports section, [223](#)
- isascii function, [145](#)
- itoa function, [146](#)
- itostr function, [147](#)
- J —
- J option, [45](#)
- jH option, [45, 195, 197, 198](#)
- jHc option, [45, 197, 198](#)
- jHd option, [45, 197, 198](#)
- jHu option, [45, 197, 198](#)
- Jump optimizations, [185 to 186](#)
- K —
- KE option, [46](#)
- Ken option, [46](#)
- Kep option, [46](#)
- Kes option, [46](#)
- Ket option, [47](#)
- Keywords, C++, [329](#)
- Kf option, [47, 48](#)
- Kh option, [47](#)
- Kk option, [48](#)
- Kkfe option, [48](#)
- Kkfx option, [48](#)
- Kknp option, [48](#)
- Kkz option, [49](#)
- KLf option, [49](#)
- KLi option, [49](#)
- KP option, [49](#)
- Kq option, [50](#)
- KR option, [50](#)
- Ksr, [50](#)
- KST, [50](#)
- Ku option, [50](#)
- Kv option, [50](#)

— L —

- l option, [52](#), [197](#)
- Librarian, description, [21](#)
- Libraries, [123](#), [246](#)
 - termination routine (exit), [126](#)
 - UNIX level-1 functions, [125](#)
 - UNIX level-2 functions, [125](#)
 - use, [124](#)
- Library customizer
 - building custom libraries, [173](#)
 - configuration file, [171](#)
 - directories, [169](#)
 - preprocessor symbols, [171](#)
 - testing custom libraries, [174](#)
- Library functions
 - (see under specific names)
 - non-reentrant, [126](#), [136](#)
 - read, [343](#)
 - write, [343](#)
- Library macros, [135](#)
- #line preprocessor directive, [196](#)
- __LINE__ mirror symbol, [75](#)
- Line numbers, information, -g option, [40](#)
- Linker
 - command example
 - IEEE, [247](#)
 - ROM-based system, IEEE, [248](#)
 - command file
 - passing, -e option, [39](#)
 - description, [20](#)
 - passing default libraries, [62](#)
 - passing options directly, -Wl option, [65](#)
 - suppressing call to, [38](#)
- Linker command files, using, [246](#)
- _LINUX mirror symbol, [77](#)
- Listing files
 - format, -Fli option, [39](#)
 - generating, -l option, [52](#)
 - generating, -m option, [52](#)
 - omitting page header, -Flp0 option, [40](#)
 - page length, specifying, -Flp option, [39](#)
 - source code comments, -Fsm option, [40](#)
 - title, -Flt option, [40](#)
- LITTLE_ENDIAN, [213](#)
- Little-endian code, [46](#)

- Local declarations
 - storage classes, [325](#)
- Local optimizations, [184](#)
- Local variables, [220](#)
- Location, current in a file, [350](#)
- Locations of input and output files, [25](#)
- long double type, [215](#)
- Long integer
 - convert to ASCII string, [148](#)
 - reading from a stream, [143](#)
 - writing to stream, [158](#)
- LONG mirror symbol, [77](#)
- long type, [215](#)
- LONG_DOUBLE mirror symbol, [78](#)
- LONG_LONG mirror symbol, [77](#)
- Loop optimizations, [182](#)
- Lowercase characters, converting to, [163](#)
- lseek function, [125](#), [350](#)
- ltoa function, [148](#)
- ltostr function, [149](#)

— M —

- m option, [52](#), [197](#)
- Macros
 - defining on command line, -D option, [38](#)
 - undefining, -U option, [63](#)
- _main function, [150](#)
- Main optimizations, [57](#)
- malloc function, [239](#)
- Mangling
 - extern "C" and, [211](#)
 - function names, [207](#)
 - overloaded functions and, [209](#)
 - preventing for C function names, [207](#)
- math.h include file, [239](#)
- max macro, [151](#)
- Mc option, [52](#)
- _MCCPPC mirror symbol, [76](#)
- Mcp option, [52](#)
- Mcr option, [53](#)
- Md option, [54](#)
- Mda option, [54](#)
- Member, function, calling from C, [209](#)
- memcpy function, [152](#)
- memclr function, [153](#)
- Memory layout, [220](#)

Memory space, allocating, [160](#)

Message severity levels, [257](#)

Messages, diagnostic

displaying, -nQ option, [62](#)

suppressing, -Q option, [60](#)

writing to stderr, -Fee option, [39](#)

writing to stdout, -Feo option, [39](#)

Messages, error, [257](#)

Microprocessor, (see Processor, specifying)

Microtec C/C++ Extensions, [75](#)

Microtec extensions

enabling, [66](#)

functions, [134](#)

eprintf, [140](#)

ftoa, [142](#)

getl, [143](#)

getw, [144](#)

isascii, [145](#)

itoa, [146](#)

itostr, [147](#)

ltoa, [148](#)

ltostr, [149](#)

memccpy, [152](#)

memclr, [153](#)

putl, [158](#)

putw, [159](#)

swab, [161](#)

toascii, [162](#)

_tolower, [163](#)

_toupper, [164](#)

zalloc, [167](#)

macros

BLKSIZE, [135](#)

FALSE, [135](#)

fileno, [141](#)

isascii, [145](#)

max, [151](#)

min, [154](#)

NULLPTR, [135](#)

stdaux, [135](#)

stdprn, [135](#)

toascii, [162](#)

_tolower, [163](#)

_toupper, [164](#)

TRUE, [136](#)

-x option, [66](#)

_MICROTEC mirror symbol, [76](#)

min macro, [154](#)

Mirror symbol

compiler

_MICROTEC, [76](#)

_MRI, [76](#)

_TARGET_PPC, [76](#)

_VERSION, [76](#)

host

__ALIGNOF_CHAR, [77](#)

__ALIGNOF_CODE_POINTER, [78](#)

__ALIGNOF_DATA_POINTER, [78](#)

__ALIGNOF_DOUBLE, [77](#)

__ALIGNOF_FLOAT, [77](#)

__ALIGNOF_INT, [77](#)

__ALIGNOF_LONG, [77](#)

__ALIGNOF_LONG_DOUBLE, [78](#)

__ALIGNOF_LONG_LONG, [77](#)

__ALIGNOF_POINTER, [78](#)

__ALIGNOF_SHORT, [77](#)

__ALIGNOF_WCHAR_T, [77](#)

__SIZEOF_CHAR, [77](#)

__SIZEOF_CODE_POINTER, [78](#)

__SIZEOF_DATA_POINTER, [78](#)

__SIZEOF_DOUBLE, [77](#)

__SIZEOF_FLOAT, [77](#)

__SIZEOF_INT, [77](#)

__SIZEOF_LONG, [77](#)

__SIZEOF_LONG_DOUBLE, [78](#)

__SIZEOF_LONG_LONG, [77](#)

__SIZEOF_POINTER, [78](#)

__SIZEOF_SHORT, [77](#)

__SIZEOF_WCHAR_T, [77](#)

_LINUX, [77](#)

_SOLARIS, [77](#)

_UNIX, [77](#)

_WINDOWS, [77](#)

language

__c, [77](#)

__cplusplus, [77](#)

__CCPPC, [76](#)

_FILE_EXT, [77](#)

_MCCPPC, [76](#)

standard

- `__DATE__`, 75
- `__FILE__`, 75
- `__FUNC__`, 76
- `__func__`, 76
- `__LINE__`, 75
- `__STDC__`, 75
- target
 - `_BUS_WIDTH`, 78
 - `_VARIANT`, 78
- type
 - `CHAR`, 77
 - `CODE_POINTER`, 78
 - `DATA_POINTER`, 78
 - `DOUBLE`, 77
 - `FLOAT`, 77
 - `INT`, 77
 - `LONG`, 77
 - `LONG_DOUBLE`, 78
 - `LONG_LONG`, 77
 - `POINTER`, 78
 - `SHORT`, 77
 - `WCHAR_T`, 77
- underlying type
 - `_PTRDIFF_T`, 78
 - `_SIZE_T`, 78
 - `_WCHAR_T`, 78
- Modes
 - ANSI-compliant, -A option, 37
 - processor, -p option, 59
 - verbose, -V option, 64
- `_MRI_EXTENSIONS` preprocessor symbol, 66
- `_MRI` mirror symbol, 76
- `mriext.h` include file, 134, 135
- Multiple jump optimization, 186
- N —
- nA option, 37
- Naming convention for symbols, 63
- NC option, 56
- nC option, 38
- ND option, 56, 197
- Negative option prefix, 27
- Nested
 - class, 334
 - structures, 333
- Newline
 - disable for preprocessor
 - Es option, 39
 - Ps option, 59
- nf option, 40
- nFee option, 39
- nFeo option, 39
- nFli option, 39
- nFsm option, 40
- ng option, 40
- nH option, 44
- NI option, 56, 197
- njH option, 45
- nKE option, 46
- nKen option, 46
- nKes option, 46
- nKet option, 47
- nKf option, 47
- nKk option, 48
- nKknp option, 49
- nKkz option, 49
- nKP option, 49
- nKq option, 50
- nKR option, 50
- nKsr, 50
- nKST, 50
- nKu option, 50
- nKv option, 51
- nOD option, 56
- nOfs option, 58
- nOg option, 57
- nOi option, 57
- nOJ option, 58
- nOj option, 57
- nOl option, 57
- Nonmember function, calling from C, 209
- Non-reentrant functions, (see also Reentrant functions)
 - nOr option, 57
 - nOU option, 57
 - nQ option, 62
 - nq option, 62
 - nQo option, 61
 - NS option, 56, 197
 - NT option, 56, 197

Nucleus Debugger

generating debugging information, -g
option, [40](#)

__nullcheck_trap routine, [49](#)

-nx option, [66](#)

-NZ option, [56](#), [197](#)

-nze option, [67](#)

-nzs option, [68](#)

— O —

-o option, [58](#), [197](#)

Object files

naming, -o option, [58](#)

producing only, -c option, [38](#)

-OD option, [56](#)

-Of option, [58](#)

-Og option, [57](#), [185](#)

-Oi option, [57](#)

-OJ option, [58](#)

-Oj option, [57](#)

-Ol option, [57](#)

open function, [125](#), [155](#)

relationship to close, [345](#)

relationship to read, [353](#)

relationship to write, [166](#)

Opening a file, [155](#)

Optimizations, [177](#)

algebraic simplification, [177](#)

branch tail merging, [185](#)

constant folding, [184](#)

general, [177](#)

global, [178](#) to [183](#)

global constant propagation, [180](#)

global copy propagation, [181](#)

instruction scheduling, [188](#)

jump, [185](#) to [186](#)

local, [184](#)

loop, [182](#)

multiple jump, [186](#)

redundant code elimination, [177](#)

redundant jump elimination, [186](#)

strength reduction, [178](#)

strength reduction and index simplification,
[183](#)

Option descriptions

UNIX, [28](#)

Windows, [28](#)

Options

(see also under individual entries)

active on output listing, suppression, -Qo
option, [61](#)

and precompiled header files, [197](#)

command line, [27](#)

conflicting, specifying, [27](#)

descriptions, [37](#)

file, specifying, -d option, [38](#)

negative, [27](#)

passing to assembler

-Wa option, [36](#)

passing to assembler, -Wa option, [36](#), [64](#)

passing to linker, -Wl option, [65](#)

summary, [28](#)

using pragmas to set, [69](#)

-Or option, [57](#)

-Os option, [57](#)

-Ot option, [57](#)

-OU option, [57](#)

Output

assembly source file, [253](#)

listing, [254](#)

Output files

naming, -o option, [58](#)

specifying format, -Fli option, [39](#)

Overloaded functions, [209](#)

Overloading

conversion operator, [329](#)

function names, [206](#)

-Ox option, [58](#)

— P —

-P option, [59](#), [197](#)

-p option, [59](#)

packed keyword, [106](#)

Padding, [215](#)

Page header, specifying for listing file, -Flp0
option, [40](#)

Parameters

passing, [201](#)

setting, [201](#)

Parentheses

redundant, [330](#)

use in C++, [330](#)

Passing options

to assembler, -Wa option, [36](#), [64](#)

to linker, -Wl option, [65](#)

.pch file extension, [198](#)

PCH_DECL_SEQ_THRESHOLD macro,
[196](#), [198](#)

PCH_FILE_SUFFIX macro, [198](#)

POINTER mirror symbol, [78](#)

Pointers

types, [215](#)

void*, [325](#)

#pragma asm directive, [91](#), [197](#), [233](#)

#pragma do_not_instantiate directive, [92](#)

#pragma eject directive, [93](#)

#pragma endasm directive, [94](#), [198](#), [233](#)

#pragma error directive, [95](#)

#pragma hdrstop directive, [195](#), [198](#)

#pragma info directive, [96](#)

#pragma instantiate directive, [97](#)

#pragma list directive, [98](#)

#pragma macro directive, [99](#)

#pragma no_pch directive, [196](#), [198](#)

#pragma option directive, [100](#), [198](#), [197](#)

#pragma warn directive, [101](#)

Pragmas and options, [69](#)

Pre-ANSI C function declarations, [330](#)

Precompiled header files, header stop point,
[195](#)

Precompiled headers, [45](#)

Preprocessor

executing only

-E option, [39](#)

-P option, [59](#)

macros

defining on command line, -D option,
[38](#)

undefining, -U option, [63](#)

output

saving comments, -C option, [38](#)

sending to standard output, -E option,
[39](#)

Preprocessor directives

#define, [196](#)

#error, [89](#)

#if, [196](#)

#informing, [90](#)

#line, [196](#)

#pragma asm, [91](#), [197](#)

#pragma do_not_instantiate, [92](#)

#pragma eject, [93](#)

#pragma endasm, [94](#), [198](#)

#pragma error, [95](#)

#pragma hdrstop, [195](#), [198](#)

#pragma info, [96](#)

#pragma instantiate, [97](#)

#pragma list, [98](#)

#pragma macro, [99](#)

#pragma no_pch, [196](#), [198](#)

#pragma option, [100](#), [197](#), [198](#)

#pragma warn, [101](#)

#WARNING, [102](#)

Preprocessor symbols, [171](#)

printf function

preprocessor symbols, [171](#)

removing unneeded I/O support, [244](#)

Processor

modes, -p option, [59](#)

specifying, -p option, [59](#)

Program termination, [348](#)

Prologue, function, [204](#)

Promotion, [207](#)

Prototyped functions, [330](#) to [331](#)

declaration, [330](#)

-Ps option, [59](#), [197](#)

_PTRDIFF_T mirror symbol, underlying type,
[78](#)

Public, variable names, [253](#)

__pure_virtual_function_called function, [157](#)

putl function

definition, [158](#)

relationship to getl, [143](#)

putw function

definition, [159](#)

relationship to getw, [144](#)

— Q —

-Q option, [60](#), [197](#), [259](#)

-q option, [62](#), [197](#)

-QA option, [60](#), [197](#), [259](#)

-Qe option, [60](#), [197](#), [259](#)

-Qfn option, [61](#), [197](#)

- Qfs option, [61](#), [197](#)
- Qi option, [61](#), [197](#), [259](#)
- Qm option, [61](#)
- Qme option, [61](#), [197](#)
- Qms option, [259](#)
- Qmw option, [61](#), [197](#)
- Qo option, [61](#), [197](#)
- Qs option, [61](#), [197](#), [259](#)
- Qw option, [62](#), [197](#), [259](#)
- Qwo, [62](#)

— R —

- read function, [125](#), [343](#), [353](#)
- Reading bytes from a file, [353](#)
- realloc function, [239](#)
- Redundant code elimination, [177](#)
- Redundant jump optimization, [186](#)
- Redundant store elimination, [184](#)
- Reentrant code, generating, [238](#)
- Reentrant functions, [126](#), [136](#), [238](#)
- register, keyword, [220](#)
- Register, storage class, [220](#)
- Registers, use of, [203](#)
- Returning a typed value, [229](#)
- .rodata section, [223](#), [327](#)
- .rodata1 section, [223](#)
- ROM addressing, absolute, -ara option, [37](#)
- ROM addressing, per -Md, -ard option, [37](#)
- ROM storage, const objects, [327](#)
- ROM-based systems, [248](#)
- Routines, user-modified, [242](#)
- Run-Time Checks, [41](#)

— S —

- S option, [62](#), [197](#)
- sbrk function, [125](#), [160](#)
- scanf function, preprocessor symbols, [171](#)
- Scope issues, [325](#)
- Search paths
 - (see also Environment variables)
 - nonstandard #include files, -I option, [44](#)
 - standard #include files, -J option, [45](#)
- Section
 - .bss, [220](#), [224](#)
 - .data, [223](#)
 - .ioports, [223](#)

- .rodata1, [223](#)
- .text, [223](#)
- rodata, [223](#)
- Setting sign of char variables, -Ku option, [50](#)
- Severity of errors, [257](#)
- Short integer
 - reading from a stream, [144](#)
 - writing to stream, [159](#)
- Short integers, [201](#)
- SHORT mirror symbol, [77](#)
- short type, [215](#)
- _simulated_input variable, [343](#)
- _simulated_output variable, [343](#)
- _SIZE_T mirror symbol, underlying type, [78](#)
- Size, data types, [214](#)
- __SIZEOF_CHAR mirror symbol, [77](#)
- __SIZEOF_CODE_POINTER mirror symbol, [78](#)
- __SIZEOF_DATA_POINTER mirror symbol, [78](#)
- __SIZEOF_DOUBLE mirror symbol, [77](#)
- __SIZEOF_FLOAT mirror symbol, [77](#)
- __SIZEOF_INT mirror symbol, [77](#)
- __SIZEOF_LONG mirror symbol, [77](#)
- __SIZEOF_LONG_DOUBLE mirror symbol, [78](#)
- __SIZEOF_LONG_LONG mirror symbol, [77](#)
- __SIZEOF_POINTER mirror symbol, [78](#)
- __SIZEOF_SHORT mirror symbol, [77](#)
- __SIZEOF_WCHAR_T mirror symbol, [77](#)
- socket function, [354](#)
- _SOLARIS mirror symbol, [77](#)
- Source code comments for listing file, -Fsm option, [40](#)
- sprintf function, [126](#), [239](#)
- sscanf function, [126](#), [239](#)
- Stack frames, [203](#)
- stat function, [355](#)
- Static variables, [220](#)
- __STDC__ mirror symbol, [75](#)
- __STDC__ preprocessor symbol, [37](#)
- stderr, directing diagnostic messages, -Fee option, [39](#)
- stdio.h include file, [126](#), [239](#)

stdout, directing diagnostic messages, -Feo
option, 39

Storage

allocation of data types, 214
layout, 213

strcpy function, 66

Stream

reading a long integer, 143
reading a short integer, 144
writing a long integer, 158
writing a short integer, 159

Strength reduction, optimization, 183

Strength reduction, optimizations, 178

strings section (.rodata1), naming, -NS option,
56

strtod function, 239

strtol function, 239

strtoul function, 239

struct scope, 333

Structures, 215

Summary message suppression, -Qs option, 61

Summary of command line options, 28

Supported GNU C/C++ Extensions, 112

Suppressing executable file, -c option, 38

swab function, 161

Swapping bytes in memory, 161

Symbol names convention, 63

Symbols, information, -g option, 40

Syntax checking only, -y option, 67

sys_in function, 356

sys_out function, 357

sys_stat function, 358

SysHost feature, 342

System functions, 136

_popen, 352

chdir, 344

close, 345

connect, 346

creat, 347

relationship to close, 345

relationship to read, 353

relationship to write, 166

_exit, 348

getcwd, 349

lseek, 350

memclr, 351

open, 155

relationship to close, 345

relationship to read, 353

relationship to write, 166

read, 343, 353

sbrk, 160

socket, 354

stat, 355

sys_in, 356

sys_out, 357

sys_stat, 358

unlink, 165

write, 166, 343

— T —

_TARGET_PPC mirror symbol, 76

Termination, normal program, 348

test_lib script, 174

test_one script, 174

.text section, 223

Threads

definition, 238

multiple, 239

Time stamp, compilation, -Vt option, 64

Title for listing file, -Flt option, 40

-tl option, 62

-tm option, 63

toascii function, 162

_tolower function, 163

tolower function, 135

Toolkit components, 19

_toupper function, 164

toupper function, 135

-tu option, 63

typedef declarations, 328

Types

(see also Data, types)

returned for functions, 229

— U —

-U option, 63

-ui option, 64

Undefining preprocessor macros, -U option, 63

Uninitialized global variables, 65

UNIX

- command line syntax, [23](#)
- compiler invocation, [23](#)
- compiler use, [23](#), [27](#), [123](#)
- compiling a program, [254](#)
- file locations, [25](#)
- level-1 functions, [125](#)
- level-2 functions, [125](#)
- option form, positive and negative, [27](#)
- option, command line, descriptions of, [27](#)
- system functions, [125](#)
- _UNIX mirror symbol, [77](#)
- unlink function, [125](#), [165](#)
- Unlinking a filename, [165](#)
- unpacked keyword, [106](#)
- Unreachable (dead) code elimination, [178](#)
- Unsigned char default, -Ku option, [50](#)
- unsigned int type, [215](#)
- Unsigned integer, convert to ASCII string, [147](#)
- Unsigned long integer, convert to ASCII string, [149](#)
- unsigned types, [214](#)
- Uppercase characters, converting to, [164](#)
- USE_FIXED_ADDRESS_FOR_MMAP
macro, [198](#)
- USE_MMAP_FOR_MEMORY_REGIONS
macro, [198](#)
- User-modified routines for embedded systems, [242](#)

— V —

- V option, [64](#), [197](#)
- Value comparison
 - returning the greater of two values, [151](#)
 - returning the lesser of two values, [154](#)
- Variables
 - declaring, [326](#)
 - global, uninitialized, [65](#)
 - information, -g option, [40](#)
 - initializations, [245](#)
 - initializing, [327](#)
 - local, [220](#)
 - names, [206](#)
 - external, [253](#)
 - inside asm, [229](#)
 - public, [253](#)
 - saving initialized variables in ROM, [245](#)

- static, [126](#), [136](#), [220](#)
- _VARIANT mirror symbol, [78](#)
- Vb option, [64](#), [197](#)
- Vd option, [64](#)
- Verbose mode, enabling and disabling, -V
option, [64](#)
- _VERSION mirror symbol, [76](#)
- Vi option, [64](#)
- void* pointer, [325](#)
- vsprintf function, [126](#), [239](#)
- Vt option, [64](#)

— W —

- Wa option, [36](#), [64](#)
- #WARNingdirective, [102](#)
- #warning directive
 - relation to #pragma warn, [101](#)
- Warning messages
 - compiler, [257](#)
 - suppression, -Qw option, [62](#)
- _WARNING_xxx_stub_used symbol
 - unresolved
 - lseek, [350](#)
 - open, [155](#)
 - unlink, [165](#)
- WARNING-xxx (stub routine) called, ftell, [156](#), [165](#), [350](#)
- _WCHAR_T mirror symbol, underlying type, [78](#)
- WCHAR_T mirror symbol, [77](#)
- Weak externals
 - no value, -Xp option, [65](#)
 - section naming (.bss), -NZ option, [56](#)
 - uninitialized globals, -Xc option, [65](#)
- Windows
 - command line syntax, [23](#)
 - compiler invocation, [23](#)
 - compiler use, [23](#), [27](#), [123](#)
 - compiling a program, [254](#)
 - file locations, [25](#)
 - option form, positive and negative, [27](#)
 - option, command line, descriptions of, [27](#)
- _WINDOWS mirror symbol, [77](#)
- Wl option, [65](#)
- Wrapper, C++, [210](#)
- write function, [125](#), [166](#), [343](#)

Writing a short integer to a stream, [159](#)

Writing bytes to a file, [166](#)

— X —

-x option, [66](#)

-Xc option, [65](#)

-Xp option, [65](#)

— Y —

-y option, [67](#), [197](#)

— Z —

-Z option, [56](#)

-Za option, [51](#)

zalloc function, [167](#), [239](#)

-Zc option, [51](#)

-Zd option, [51](#)

-ze option, [67](#)

-Zi option, [51](#)

-Zm option, [51](#)

-Zn option, [51](#)

-zr option, [68](#)

-Zs option, [51](#)

Third-Party Information

This section provides information on open source and third-party software that may be included in the Microtec C/C++ Compiler for the PowerPC Family product.

- This software application may include STLport third-party software

©1994 Hewlett-Packard Company

©1996-1998 Silicon Graphics Computer Systems, Inc.

©1997 Moscow Center for SPARC Technology

©1999 Boris Fomitchev

© 2000 Pavel Kuznetsov

© 2001 Meridian'93

©1997 Mark of the Unicorn, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Copyright holders make no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

©1999 Boris Fomitchev

This material is provided "as is", with absolutely no warranty expressed

or implied. Any use is at your own risk. Permission to use or copy this software for any purpose is hereby granted without fee, provided the above notices are retained on all copies. Permission to modify the code and to distribute modified code is granted, provided the above notices are retained, and a notice that the code was modified is included with the above copyright notice.

©1997 Mark of the Unicorn, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Mark of the Unicorn makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

This software application may include STLport version 5.1.2 third-party software. STLport version 5.1.2 is distributed under the terms of the STLport License Agreement and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view the complete license (stlport_2000.pdf) in the <Installdir>/legal directory accompanying this documentation.. STLport version 5.1.2 may be subject to the following copyrights:

© 1994 Hewlett-Packard Company

© 1996,97 Silicon Graphics Computer Systems, Inc.

© 1997 Moscow Center for SPARC Technology.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Moscow Center for SPARC Technology makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

This software application may include STLport version 5.1.2 third-party software. STLport version 5.1.2 is distributed under the terms of the STLport License Agreement and is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. See the license for the specific language governing rights and limitations under the license. You can view a copy of the license at: <InstallDir>/legal/stlport_2000.pdf. STLport version 5.1.2 may be subject to the following copyrights:

© 1994 Hewlett-Packard Company

© 1996-1999 Silicon Graphics Computer Systems, Inc.

© 1997 Moscow Center for SPARC Technology.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Moscow Center for SPARC Technology makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1998 Mark of the Unicorn, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Mark of the Unicorn, Inc. makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

This software application may include STLport version 5.1.2 third-party software, which is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied STLport version 5.1.2 may be subject to the following copyrights:

© 1996-1999 Silicon Graphics Computer Systems, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Silicon Graphics makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1994 Hewlett-Packard Company

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Hewlett-Packard Company makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

© 1998 Mark of the Unicorn, Inc.

Permission to use, copy, modify, distribute and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appear in all copies and that both that copyright notice and this permission notice appear in supporting documentation. Mark of the Unicorn, Inc. makes no representations about the suitability of this software for any purpose. It is provided "as is" without express or implied warranty.

This software application may include David Gay's dtoa version 20120423 third-party software, which is distributed on an "AS IS" basis, WITHOUT WARRANTY OF ANY KIND, either express or implied. David Gay's dtoa version 20120423 may be subject to the following copyrights:

© 1991, 2000, 2001 by Lucent Technologies.

Permission to use, copy, modify, and distribute this software for any purpose without fee is hereby granted, provided that this entire notice is included in all copies of any software which is or includes a copy or modification of this software and in all copies of the supporting documentation for such software.

THIS SOFTWARE IS BEING PROVIDED "AS IS", WITHOUT ANY EXPRESS OR IMPLIED WARRANTY. IN PARTICULAR, NEITHER THE AUTHOR NOR LUCENT MAKES ANY REPRESENTATION OR WARRANTY OF ANY KIND CONCERNING THE MERCHANTABILITY OF THIS SOFTWARE OR ITS FITNESS FOR ANY PARTICULAR PURPOSE.

Embedded Software and Hardware License Agreement

The latest version of the Embedded Software and Hardware License Agreement is available on-line at:
www.mentor.com/eshla

IMPORTANT INFORMATION

USE OF ALL PRODUCTS IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF PRODUCTS INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.

EMBEDDED SOFTWARE AND HARDWARE LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Products (as defined in Section 1) between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Products received electronically, certify destruction of Products and all accompanying items within five days after receipt of such Products and receive a full refund of any license fee paid.

1. **Definitions.** As used in this Agreement and any applicable quotation, supplement, attachment and/or addendum ("Addenda"), these terms shall have the following meanings:
 - 1.1. "Customer's Product" means Customer's end-user product identified by a unique SKU (including any Related SKUs) in an applicable Addenda that is developed, manufactured, branded and shipped solely by Customer or an authorized manufacturer or subcontractor on behalf of Customer to end-users or consumers;
 - 1.2. "Developer" means a unique user, as identified by a unique user identification number, with access to Embedded Software at an authorized Development Location. A unique user is an individual who works directly with the embedded software in source code form, or creates, modifies or compiles software that ultimately links to the Embedded Software in Object Code form and is embedded into Customer's Product at the point of manufacture;
 - 1.3. "Development Location" means the location where Products may be used as authorized in the applicable Addenda;
 - 1.4. "Development Tools" means the software that may be used by Customer for building, editing, compiling, debugging or prototyping Customer's Product;
 - 1.5. "Embedded Software" means Software that is embeddable;
 - 1.6. "End-User" means Customer's customer;
 - 1.7. "Executable Code" means a compiled program translated into a machine-readable format that can be loaded into memory and run by a certain processor;
 - 1.8. "Hardware" means a physically tangible electro-mechanical system or sub-system and associated documentation;
 - 1.9. "Linkable Object Code" or "Object Code" means linkable code resulting from the translation, processing, or compiling of Source Code by a computer into machine-readable format;
 - 1.10. "Mentor Embedded Linux" or "MEL" means Mentor Graphics' tools, source code, and recipes for building Linux systems;
 - 1.11. "Open Source Software" or "OSS" means software subject to an open source license which requires as a condition for redistribution of such software, including modifications thereto, that the: (i) redistribution be in source code form or be made available in source code form; (ii) redistributed software be licensed to allow the making of derivative works; or (iii) redistribution be at no charge;
 - 1.12. "Processor" means the specific microprocessor to be used with Software and implemented in Customer's Product;
 - 1.13. "Products" means Software, Term-Licensed Products and/or Hardware;
 - 1.14. "Proprietary Components" means the components of the Products that are owned and/or licensed by Mentor Graphics and are not subject to an Open Source Software license, as more fully set forth in the product documentation provided with the Products;

- 1.15. “Redistributable Components” means those components that are intended to be incorporated or linked into Customer’s Linkable Object Code developed with the Software, as more fully set forth in the documentation provided with the Products;
- 1.16. “Related SKU” means two or more Customer Products identified by logically-related SKUs, where there is no difference or change in the electrical hardware or software content between such Customer Products;
- 1.17. “Software” means software programs, Embedded Software and/or Development Tools, including any updates, modifications, revisions, copies, documentation and design data that are licensed under this Agreement;
- 1.18. “Source Code” means software in a form in which the program logic is readily understandable by a human being;
- 1.19. “Sourcery CodeBench Software” means Mentor Graphics’ Development Tool for C/C++ embedded application development;
- 1.20. “Sourcery VSIPL++” is Software providing C++ classes and functions for writing embedded signal processing applications designed to run on one or more processors;
- 1.21. “Stock Keeping Unit” or “SKU” is a unique number or code used to identify each distinct product, item or service available for purchase;
- 1.22. “Subsidiary” means any corporation more than 50% owned by Customer, excluding Mentor Graphics competitors. Customer agrees to fulfill the obligations of such Subsidiary in the event of default. To the extent Mentor Graphics authorizes any Subsidiary’s use of Products under this Agreement, Customer agrees to ensure such Subsidiary’s compliance with the terms of this Agreement and will be liable for any breach by a Subsidiary; and
- 1.23. “Term-Licensed Products” means Products licensed to Customer for a limited time period (“Term”).

2. Orders, Fees and Payment.

- 2.1. To the extent Customer (or if agreed by Mentor Graphics, Customer’s appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (“Order(s)”), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement and any applicable Addenda, whether or not these documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order will not be effective unless agreed in writing by an authorized representative of Customer and Mentor Graphics.
- 2.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. All invoices will be sent electronically to Customer on the date stated on the invoice unless otherwise specified in an Addendum. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice(s). Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer’s sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer’s behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 2.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics’ delivery of Software by electronic means is subject to Customer’s provision of both a primary and an alternate e-mail address.

3. Grant of License.

- 3.1. The Products installed, downloaded, or otherwise acquired by Customer under this Agreement constitute or contain copyrighted, trade secret, proprietary and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software as described in the applicable Addenda. The limited licenses granted under the applicable Addenda shall continue until the expiration date of Term-Licensed Products or termination in accordance with Section 12 below, whichever occurs first. **Mentor Graphics does NOT grant Customer any right to (a) sublicense or (b) use Software beyond the scope of this Section without first signing a separate agreement or Addenda with Mentor Graphics for such purpose.**
- 3.2. License Type. The license type shall be identified in the applicable Addenda.
 - 3.2.1. Development License: During the Term, if any, Customer may modify, compile, assemble and convert the applicable Embedded Software Source Code into Linkable Object Code and/or Executable Code form by the number of Developers specified, for the Processor(s), Customer’s Product(s) and at the Development Location(s) identified in the applicable Addenda.

- 3.2.2. End-User Product License: During the Term, if any, and unless otherwise specified in the applicable Addenda, Customer may incorporate or embed an Executable Code version of the Embedded Software into the specified number of copies of Customer's Product(s), using the Processor Unit(s), and at the Development Location(s) identified in the applicable Addenda. Customer may manufacture, brand and distribute such Customer's Product(s) worldwide to its End-Users.
- 3.2.3. Internal Tool License: During the Term, if any, Customer may use the Development Tools solely: (a) for internal business purposes and (b) on the specified number of computer work stations and sites. Development Tools are licensed on a per-seat or floating basis, as specified in the applicable Addenda, and shall not be distributed to others or delivered in Customer's Product(s) unless specifically authorized in an applicable Addenda.
- 3.2.4. Sourcery CodeBench Professional Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software (i) if the license is a node-locked license, by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, or (ii) if the license is a floating license, by the authorized number of concurrent users on one or more machines provided that only the authorized number of copies of the Software are in use at any one time, and (b) distribute the Redistributable Components of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.5. Sourcery CodeBench Standard Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on up to two machines provided that only one copy of the Software is in use at any one time, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer's Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.6. Sourcery CodeBench Personal Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.2.7. Sourcery CodeBench Academic Edition License: During the Term specified in the applicable Addenda, Customer may (a) install and use the Proprietary Components of the Software for non-commercial, academic purposes only by a single user who uses the Software on one machine, and (b) distribute the Redistributable Component(s) of the Software in Executable Code form only and only as part of Customer Object Code developed with the Software that provides substantially different functionality than the Redistributable Component(s) alone.
- 3.3. Mentor Graphics may from time to time, in its sole discretion, lend Products to Customer. For each loan, Mentor Graphics will identify in writing the quantity and description of Software loaned, the authorized location and the Term of the loan. Mentor Graphics will grant to Customer a temporary license to use the loaned Software solely for Customer's internal evaluation in a non-production environment. Customer shall return to Mentor Graphics or delete and destroy loaned Software on or before the expiration of the loan Term. Customer will sign a certification of such deletion or destruction if requested by Mentor Graphics.

4. Beta Code.

- 4.1. Portions or all of certain Products may contain code for experimental testing and evaluation ("Beta Code"), which may not be used without Mentor Graphics' explicit authorization. Upon Mentor Graphics' authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. This grant and Customer's use of the Beta Code shall not be construed as marketing or offering to sell a license to the Beta Code, which Mentor Graphics may choose not to release commercially in any form.
- 4.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer's use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer's evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 4.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer's feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 4.3 shall survive termination of this Agreement.

5. Restrictions on Use.

- 5.1. Customer may copy Software only as reasonably necessary to support the authorized use, including archival and backup purposes. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except where embedded in Executable Code form in Customer's Product, Customer shall maintain a record of the number and location of all copies of Software, including copies merged with other software and products, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer's employees, authorized manufacturers or authorized contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics immediate written notice of any unauthorized disclosure or use of the Products as soon as Customer learns or becomes aware of such unauthorized disclosure or use.
- 5.2. Customer acknowledges that the Products provided hereunder may contain Source Code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such Source Code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, reverse-compile, reverse-engineer or in any way derive any Source Code from Products that are not provided in Source Code form. Except as embedded in Executable Code in Customer's Product and distributed in the ordinary course of business, in no event shall Customer provide Products to Mentor Graphics competitors. Log files, data files, rule files and script files generated by or for the Software (collectively "Files") constitute and/or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Under no circumstances shall Customer use Products or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
- 5.3. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense or otherwise transfer the Products, whether by operation of law or otherwise ("Attempted Transfer"), without Mentor Graphics' prior written consent, which shall not be unreasonably withheld, and payment of Mentor Graphics' then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics' prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics' option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer's permitted successors in interest and assigns.
- 5.4. Notwithstanding any provision in an OSS license agreement applicable to a component of the Sourcery CodeBench Software that permits the redistribution of such component to a third party in Source Code or binary form, Customer may not use any Mentor Graphics trademark, whether registered or unregistered, in connection with such distribution, and may not recompile the Open Source Software components with the --with-pkgversion or --with-bugurl configuration options that embed Mentor Graphics' trademarks in the resulting binary.
- 5.5. The provisions of this Section 5 shall survive the termination of this Agreement.

6. Support Services.

- 6.1. Except as described in Sections 6.2, 6.3 and 6.4 below, and unless otherwise specified in any applicable Addenda to this Agreement, to the extent Customer purchases support services, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current End-User Software Support Terms located at <http://supportnet.mentor.com/about/legal/>.
- 6.2. To the extent Customer purchases support services for Sourcery CodeBench Software, support will be provided solely in accordance with the provisions of this Section 6.2. Mentor Graphics shall provide updates and technical support to Customer as described herein only on the condition that Customer uses the Executable Code form of the Sourcery CodeBench Software for internal use only and/or distributes the Redistributable Components in Executable Code form only (except as provided in a separate redistribution agreement with Mentor Graphics or as required by the applicable Open Source license). Any other distribution by Customer of the Sourcery CodeBench Software (or any component thereof) in any form, including distribution permitted by the applicable Open Source license, shall automatically terminate any remaining support term. Subject to the foregoing and the payment of support fees, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased in accordance with Mentor Graphics' then-current Sourcery CodeBench Software Support Terms located at <http://www.mentor.com/codebench-support-legal>.
- 6.3. To the extent Customer purchases support services for Sourcery VSIPL++, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Sourcery VSIPL++ Support Terms located at <http://www.mentor.com/vsipl-support-legal>.
- 6.4. To the extent Customer purchases support services for Mentor Embedded Linux, Mentor Graphics will provide Customer updates and technical support for the number of Developers at the Development Location(s) for which support is purchased solely in accordance with Mentor Graphics' then-current Mentor Embedded Linux Support Terms located at <http://www.mentor.com/mel-support-legal>.

7. **Third Party and Open Source Software.** Products may contain Open Source Software or code distributed under a proprietary third party license agreement. Please see applicable Products documentation, including but not limited to license notice files, header files or source code for further details. Please see the applicable Open Source Software license(s) for additional rights and obligations governing your use and distribution of Open Source Software. Customer agrees that it shall not subject any Product provided by Mentor Graphics under this Agreement to any Open Source Software license that does not otherwise apply to such Product. In the event of conflict between the terms of this Agreement, any Addenda and an applicable OSS or proprietary third party agreement, the OSS or proprietary third party agreement will control solely with respect to the OSS or proprietary third party software component. The provisions of this Section 7 shall survive the termination of this Agreement.
8. **Limited Warranty.**
- 8.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual and/or specification. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Products under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification or improper installation. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY, PROVIDED CUSTOMER HAS OTHERWISE COMPLIED WITH THIS AGREEMENT. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; OR (B) PRODUCTS PROVIDED AT NO CHARGE, WHICH ARE PROVIDED "AS IS" UNLESS OTHERWISE AGREED IN WRITING.
- 8.2. THE WARRANTIES SET FORTH IN THIS SECTION 8 ARE EXCLUSIVE TO CUSTOMER AND DO NOT APPLY TO ANY END-USER. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES, EXPRESS, IMPLIED, OR STATUTORY, WITH RESPECT TO PRODUCTS OR OTHER MATERIAL PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
9. **LIMITATION OF LIABILITY.** EXCEPT WHERE THIS EXCLUSION OR RESTRICTION OF LIABILITY WOULD BE VOID OR INEFFECTIVE UNDER APPLICABLE LAW, AND EXCEPT FOR EITHER PARTY'S BREACH OF ITS CONFIDENTIALITY OBLIGATIONS, CUSTOMER'S BREACH OF LICENSING TERMS OR CUSTOMER'S OBLIGATIONS UNDER SECTION 10, IN NO EVENT SHALL: (A) EITHER PARTY OR ITS RESPECTIVE LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF SUCH PARTY OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES; AND (B) EITHER PARTY OR ITS RESPECTIVE LICENSORS' LIABILITY UNDER THIS AGREEMENT, INCLUDING, FOR THE AVOIDANCE OF DOUBT, LIABILITY FOR ATTORNEYS' FEES OR COSTS, EXCEED THE GREATER OF THE FEES PAID OR OWING TO MENTOR GRAPHICS FOR THE PRODUCT OR SERVICE GIVING RISE TO THE CLAIM OR \$500,000 (FIVE HUNDRED THOUSAND U.S. DOLLARS). NOTWITHSTANDING THE FOREGOING, IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 9 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.
10. **Hazardous Applications.**
- 10.1. Customer agrees that Mentor Graphics has no control over Customer's testing or the specific applications and use that Customer will make of Products. Mentor Graphics Products are not specifically designed for use in the operation of nuclear facilities, aircraft navigation or communications systems, air traffic control, life support systems, medical devices or other applications in which the failure of Mentor Graphics Products could lead to death, personal injury, or severe physical or environmental damage ("Hazardous Applications").
- 10.2. CUSTOMER ACKNOWLEDGES IT IS SOLELY RESPONSIBLE FOR TESTING PRODUCTS USED IN HAZARDOUS APPLICATIONS AND SHALL BE SOLELY LIABLE FOR ANY DAMAGES RESULTING FROM SUCH USE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS SHALL BE LIABLE FOR ANY DAMAGES RESULTING FROM OR IN CONNECTION WITH THE USE OF PRODUCTS IN ANY HAZARDOUS APPLICATIONS.
- 10.3. CUSTOMER AGREES TO INDEMNIFY AND HOLD HARMLESS MENTOR GRAPHICS AND ITS LICENSORS FROM ANY CLAIMS, LOSS, COST, DAMAGE, EXPENSE OR LIABILITY, INCLUDING REASONABLE ATTORNEYS' FEES, ARISING OUT OF OR IN CONNECTION WITH THE USE OF PRODUCTS AS DESCRIBED IN SECTION 10.1.
- 10.4. THE PROVISIONS OF THIS SECTION 10 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.
11. **Infringement.**

- 11.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay any costs and damages finally awarded against Customer that are attributable to the action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
 - 11.2. If a claim is made under Subsection 11.1 Mentor Graphics may, at its option and expense, and in addition to its obligations under Section 11.1, either (a) replace or modify the Product so that it becomes noninfringing; or (b) procure for Customer the right to continue using the Product. If Mentor Graphics determines that neither of those alternatives is financially practical or otherwise reasonably available, Mentor Graphics may require the return of the Product and refund to Customer any purchase price or license fee(s) paid.
 - 11.3. Mentor Graphics has no liability to Customer if the claim is based upon: (a) the combination of the Product with any product not furnished by Mentor Graphics, where the Product itself is not infringing; (b) the modification of the Product other than by Mentor Graphics or as directed by Mentor Graphics, where the unmodified Product would not infringe; (c) the use of the infringing Product when Mentor Graphics has provided Customer with a current unaltered release of a non-infringing Product of substantially similar functionality in accordance with Subsection 11.2(a); (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells, where the Product itself is not infringing; (f) any Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) Open Source Software, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such Open Source Software; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorneys' fees and other costs related to the action.
 - 11.4. THIS SECTION 11 IS SUBJECT TO SECTION 9 ABOVE AND STATES: (A) THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS AND (B) CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.
12. **Termination and Effect of Termination.** If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized Term.
- 12.1. Termination for Breach. This Agreement shall remain in effect until terminated in accordance with its terms. Mentor Graphics may terminate this Agreement and/or any licenses granted under this Agreement, and Customer will immediately discontinue use and distribution of Products, if Customer (a) commits any material breach of any provision of this Agreement and fails to cure such breach upon 30-days prior written notice; or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination. For the avoidance of doubt, nothing in this Section 12 shall be construed to prevent Mentor Graphics from seeking immediate injunctive relief in the event of any threatened or actual breach of Customer's obligations hereunder.
 - 12.2. Effect of Termination. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination or expiration of the Term, Customer will discontinue use and/or distribution of Products, and shall return Hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form, except to the extent an Open Source Software license conflicts with this Section 12.2 and permits Customer's continued use of any Open Source Software portion or component of a Product. Upon termination for Customer's breach, an End-User may continue its use and/or distribution of Customer's Product so long as: (a) the End-User was licensed according to the terms of this Agreement, if applicable to such End-User, and (b) such End-User is not in breach of its agreement, if applicable, nor a party to Customer's breach.
13. **Export.** The Products provided hereunder are subject to regulation by local laws and United States government agencies, which prohibit export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export Products in any manner without first obtaining all necessary approval from appropriate local and United States government agencies. Customer acknowledges that the regulation of product export is in continuous modification by local governments and/or the United States Congress and administrative agencies. Customer agrees to complete all documents and to meet all requirements arising out of such modifications.
14. **U.S. Government License Rights.** Software was developed entirely at private expense. All Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to US FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in this Agreement, except for provisions which are contrary to applicable mandatory federal laws.
15. **Third Party Beneficiary.** For any Products licensed under this Agreement and provided by Customer to End-Users, Mentor Graphics or the applicable licensor is a third party beneficiary of the agreement between Customer and End-User. Mentor

Graphics Corporation, Mentor Graphics (Ireland) Limited, and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.

16. **Review of License Usage.** Customer will monitor the access to and use of Software. With prior written notice, during Customer's normal business hours, and no more frequently than once per calendar year, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system, records, accounts and sublicensing documents deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all Customer information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. Such license review shall be at Mentor Graphics' expense unless it reveals a material underpayment of fees of five percent or more, in which case Customer shall reimburse Mentor Graphics for the costs of such license review. Customer shall promptly pay any such fees. If the license review reveals that Customer has made an overpayment, Mentor Graphics has the option to either provide the Customer with a refund or credit the amount overpaid to Customer's next payment. The provisions of this Section 16 shall survive the termination of this Agreement.
17. **Controlling Law, Jurisdiction and Dispute Resolution.** This Agreement shall be governed by and construed under the laws of the State of California, USA, excluding choice of law rules. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the state and federal courts of California, USA. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer or its Subsidiary in the jurisdiction where Customer's or its Subsidiary's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
18. **Severability.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
19. **Miscellaneous.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements, including but not limited to any purchase order terms and conditions. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.