CodeWarrior Development Studio for Microcontrollers V10.x S12Z Architectures Build Tools Reference Manual

Document Number: CWMCUS12ZCMPREF
Rev 10.6, 02/2014

freescale™

Contents

Section number	Title	Page

Chapter 1
Overview

1.1 Accompanying Documentation....23

1.2 Additional Information Resources....24

1.3 Miscellaneous....24

Chapter 2
Introduction

2.1 Compiler Architecture....27

Chapter 3
Creating Project

3.1 Creating and Managing Project Using CodeWarrior IDE....31

3.1.1 Creating Project Using New Bareboard Project Wizard....31

3.1.2 Analysis of Groups in CodeWarrior Projects View....34

3.1.3 Analysis of Files in CodeWarrior Projects View....35

3.2 Highlights....37

3.3 CodeWarrior Integration of Build Tools....38

3.3.1 Combined or Separated Installations....38

3.3.2 S12Z Compiler Build Settings Panels....39

3.3.2.1 S12Z Compiler....39

3.3.2.2 S12Z Compiler > Input....40

3.3.2.3 S12Z Compiler > Access Paths....41

3.3.2.4 S12Z Compiler > Warnings....41

3.3.2.5 S12Z Compiler > Code Generation....43

3.3.2.6 S12Z Compiler > Optimization....44

3.3.2.7 S12Z Compiler > Language....45

3.3.2.8 S12Z Compiler > Messages....47

3.3.2.9 S12Z Compiler > General....47

3.3.3 CodeWarrior Tips and Tricks....48

Section number	Title	Page

Chapter 4
Using Build Tools on Command Line

4.1 Configuring Command-Line Tools....49

4.1.1 Setting CodeWarrior Environment Variables....49

4.1.2 Setting PATH Environment Variable....50

4.2 Invoking Command-Line Tools....50

4.3 Getting Help....51

4.3.1 Parameter Formats....52

4.3.2 Option Formats....52

4.3.3 Common Terms....53

4.4 File Name Extensions....53

Chapter 5
Command-Line Options for Standard C Conformance

5.1 -ansi....55

5.2 -stdkeywords....55

5.3 -strict....56

Chapter 6
Command-Line Options for Standard C++ Conformance

6.1 -ARM....57

6.2 -bool....57

6.3 -Cpp_exceptions....57

6.4 -dialect....58

6.5 -for_scoping....58

6.6 -instmgr....59

6.7 -iso_templates....59

6.8 -RTTI....60

6.9 -som....60

6.10 -som_env_check....60

6.11 -wchar_t....60

Section number	Title	Page

Chapter 7
Command-Line Options for Language Translation

7.1 -char....63

7.2 -encoding....63

7.3 -flag....64

7.4 -mapcr....65

7.5 -pragma....65

7.6 -requireprotos....66

7.7 -trigraphs....66

7.8 -nolonglong....66

Chapter 8
Command-Line Options for Diagnostic Messages

8.1 -help....67

8.2 -maxerrors....68

8.3 -maxwarnings....69

8.4 -msgstyle....69

8.5 -progress....70

8.6 -stderr....70

8.7 -verbose....70

8.8 -version....71

8.9 -timing....71

8.10 -warnings....71

8.11 -wraplines....75

Chapter 9
Command-Line Options for Preprocessing

9.1 -convertpaths....77

9.2 -cwd....78

9.3 -D+....78

9.4 -define....79

9.5 -E....79

9.6 -EP.....79

9.7 -gccincludes.....80

9.8 -gccdepends.....80

9.9 -I-.....80

9.10 -I+.....81

9.11 -include.....81

9.12 -ir.....82

9.13 -P.....82

9.14 -precompile.....82

9.15 -preprocess.....83

9.16 -ppopt.....83

9.17 -prefix.....84

9.18 -noprecompile.....84

9.19 -nosyspath.....84

9.20 -stdinc.....85

9.21 -U+.....85

9.22 -undefine.....85

9.23 -allow_macro_redefs.....86

Chapter 10
Command-Line Options for Object Code

10.1 -c.....87

10.2 -codegen.....87

10.3 -enum.....87

10.4 -min_enum_size.....88

10.5 -ext.....88

Chapter 11
Command-Line Options for Optimization

11.1 -inline.....91

11.2 -O.....92

Section number	Title	Page

11.3 -O+....................92

11.4 -opt....................93

Chapter 12
S12Z Command-Line Options

12.1 Diagnostic Command-Line Options....................95

12.1.1 -g....................95

12.1.2 -sym....................96

12.2 Code Generation Command-Line Options....................96

12.2.1 -model....................96

12.2.2 -peephole....................97

12.2.3 -coloring....................97

12.3 Bit Field Command-Line Options....................97

12.3.1 -bfield_gap_limit....................97

12.3.2 -[no]bfield_lsbit_first....................100

12.3.3 -[no]bfield_reduce_type....................101

12.4 Configuration Command-Line Options....................102

12.4.1 -schar_size....................103

12.4.2 -uchar_size....................103

12.4.3 -short_size....................104

12.4.4 -int_size....................105

12.4.5 -long_size....................106

12.4.6 -llong_size....................107

12.4.7 -float_size....................108

12.4.8 -double_size....................109

12.4.9 -ldouble_size....................110

12.4.10 -lldouble_size....................110

12.5 Alignment Options....................111

12.5.1 align_globals....................111

12.5.2 align_structs....................112

Section number	Title	Page

12.5.3 align_stack....113

Chapter 13
C Compiler

13.1 Extensions to Standard C....115

13.1.1 Controlling Standard C Conformance....115

13.1.2 C++-style Comments....116

13.1.3 Unnamed Arguments....116

13.1.4 Extensions to Preprocessor....116

13.1.5 Non-Standard Keywords....117

13.1.5.1 Global Variable Address Modifier (@address)....117

13.1.5.2 Variable Allocation using @ "SegmentName"....118

13.1.5.3 interrupt Keyword....118

13.1.5.4 Binary Constants (0b)....119

13.1.5.5 Hexadecimal Constants ($)....119

13.1.5.6 #warning Directive....119

13.2 C99 Extensions....119

13.2.1 Controlling C99 Extensions....120

13.2.2 Trailing Commas in Enumerations....120

13.2.3 Compound Literal Values....121

13.2.4 Designated Initializers....121

13.2.5 Predefined Symbol __func__....122

13.2.6 Implicit Return from main()....122

13.2.7 Non-constant Static Data Initialization....122

13.2.8 Variable Argument Macros....122

13.2.9 Extra C99 Keywords....123

13.2.10 C++-Style Comments....123

13.2.11 C++-Style Digraphs....123

13.2.12 Empty Arrays in Structures....124

13.2.13 Hexadecimal Floating-Point Constants....124

Section number	Title	Page

13.2.14 Variable-Length Arrays....125

13.2.15 Unsuffixed Decimal Literal Values....125

13.2.16 C99 Complex Data Types....126

13.3 GCC Extensions....126

13.3.1 Controlling GCC Extensions....127

13.3.2 Initializing Automatic Arrays and Structures....127

13.3.3 sizeof() Operator....127

13.3.4 Statements in Expressions....128

13.3.5 Redefining Macros....128

13.3.6 typeof() Operator....128

13.3.7 Void and Function Pointer Arithmetic....129

13.3.8 __builtin_constant_p() Operator....129

13.3.9 Forward Declarations of Static Arrays....129

13.3.10 Omitted Operands in Conditional Expressions....130

13.3.11 __builtin_expect() Operator....130

13.3.12 Void Return Statements....130

13.3.13 Minimum and Maximum Operators....131

13.3.14 Local Labels....131

Chapter 14 Precompiling

14.1 What can be Precompiled....133

14.2 Using Precompiled File....134

14.3 Creating Precompiled File....134

14.3.1 Precompiling File on Command Line....134

14.3.2 Updating Precompiled File Automatically....135

14.3.3 Preprocessor Scope in Precompiled Files....135

Section number	Title	Page

Chapter 15
C++ Compiler

15.1 C++ Compiler Performance....137

15.1.1 Precompiling C++ Source Code....137

15.1.2 Using Instance Manager....138

15.2 Extensions to Standard C++....138

15.2.1 __PRETTY_FUNCTION__ Identifier....138

15.2.2 Standard and Non-Standard Template Parsing....138

15.3 Implementation-Defined Behavior....141

15.4 GCC Extensions....143

Chapter 16
Inline Assembler

16.1 Syntax....145

16.2 Reserved Words....146

16.3 Pseudo-Opcodes....146

16.4 Accessing Variables....146

16.5 Constant Expressions....146

16.6 Addresses of Variables....147

16.7 Pure Inline Assembly Functions....148

16.8 Enforce Operators....148

16.8.1 Syntax....149

16.8.2 Keyworks....149

16.8.3 Examples....150

16.8.4 Special Cases....151

Chapter 17
Supported Intrinsic Functions

17.1 Functions....153

17.1.1 __abs8....153

17.1.2 __abs16....154

17.1.3 __abs32....154

Section number	Title	Page

17.1.4 __qmuls8....154

17.1.5 __qmuls16....155

17.1.6 __qmuls32....155

17.1.7 __qmuls32_16_16....155

17.1.8 __qmulu8....155

17.1.9 __qmulu16....156

17.1.10 __qmulu32....156

17.1.11 __qmulu32_16_16....156

17.1.12 __sat8....157

17.1.13 __sat16....157

17.1.14 __sat32....157

Chapter 18
Addressing

18.1 Memory Models....159

18.1.1 SMALL Memory Model....160

18.1.2 MEDIUM Memory Model....160

18.1.3 LARGE Memory Model....160

18.2 Segmentation....160

18.3 Data Types....161

18.3.1 Scalar Types....162

18.3.2 Floating Point Types....162

18.3.3 Pointer Types....163

18.3.4 Bitfields....163

18.4 Calling Convention....165

18.4.1 Argument Passing....165

18.4.2 Return Values....167

18.4.3 Entry and Exit Code....169

Section number	Title	Page

Chapter 19
Intermediate Analysis and Optimizations

19.1 Interprocedural Analysis....173

19.1.1 Invoking Interprocedural Analysis....173

19.1.2 Function-Level Optimization....174

19.1.3 File-Level Optimization....174

19.2 Intermediate Optimizations....174

19.2.1 Dead Code Elimination....175

19.2.2 Expression Simplification....176

19.2.3 Common Subexpression Elimination....177

19.2.4 Copy Propagation....178

19.2.5 Dead Store Elimination....179

19.2.6 Live Range Splitting....180

19.2.7 Loop-Invariant Code Motion....181

19.2.8 Strength Reduction....182

19.2.9 Loop Unrolling....184

19.3 Inlining....185

19.3.1 Choosing Which Functions to Inline 185

19.3.2 Inlining Techniques....187

Chapter 20
Alignment

20.1 Alignment of Global Data....189

20.1.1 Alignment Attributes....189

20.1.1.1 Alignment Attribute and Global Variable Declarations....190

20.1.1.2 Alignment Attribute and typedef Declarations....190

20.1.1.3 Alignment Attribute and Struct Members....191

20.1.2 Alignment of Struct Members....191

20.1.3 Stack Alignment....191

Section number	Title	Page

Chapter 21
Predefined Macros

21.1 __COUNTER__ 193

21.2 __cplusplus 194

21.3 __CWCC__ 194

21.4 __DATE__ 194

21.5 __embedded_cplusplus 195

21.6 __FILE__ 195

21.7 __func__ 195

21.8 __FUNCTION__ 196

21.9 __ide_target() 196

21.10 __LINE__ 197

21.11 __MWERKS__ 197

21.12 __PRETTY_FUNCTION__ 198

21.13 __profile__ 198

21.14 __STDC__ 198

21.15 __TIME__ 199

21.16 __optlevelx 199

Chapter 22
S12Z Predefined Symbols

22.1 __S12LISA__ 201

22.2 __S12Z__ 202

22.3 __HC12__ 202

22.4 __CHAR_IS_UNSIGNED__ 202

22.5 __CHAR_IS_SIGNED__ 203

22.6 __CHAR_IS_8BIT__ 203

22.7 __CHAR_IS_16BIT__ 203

22.8 __CHAR_IS_32BIT__ 204

22.9 __SHORT_IS_8BIT__ 204

Section number	Title	Page

22.10 __SHORT_IS_16BIT__ 204

22.11 __SHORT_IS_32BIT__ 204

22.12 __INT_IS_8BIT__ 205

22.13 __INT_IS_16BIT__ 205

22.14 __INT_IS_32BIT__ 205

22.15 __LONG_IS_8BIT__ 206

22.16 __LONG_IS_16BIT__ 206

22.17 __LONG_IS_32BIT__ 206

22.18 __LONG_LONG_IS_8BIT__ 206

22.19 __LONG_LONG_IS_16BIT__ 207

22.20 __LONG_LONG_IS_32BIT__ 207

22.21 __FLOAT_IS_IEEE32__ 207

22.22 __FLOAT_IS_IEEE64__ 208

22.23 __DOUBLE_IS_IEEE32__ 208

22.24 __DOUBLE_IS_IEEE64__ 208

22.25 __LONG_DOUBLE_IS_IEEE32__ 209

22.26 __LONG_DOUBLE_IS_IEEE64__ 209

22.27 __WCHAR_T_IS_UCHAR__ 209

22.28 __WCHAR_T_IS_USHORT__ 210

22.29 __WCHAR_T_IS_ULONG__ 210

22.30 __SIZE_T_IS_UCHAR__ 210

22.31 __SIZE_T_IS_USHORT__ 210

22.32 __SIZE_T_IS_UINT__ 211

22.33 __SIZE_T_IS_ULONG__ 211

22.34 __PTRDIFF_T_IS_CHAR__ 211

22.35 __PTRDIFF_T_IS_SHORT__ 212

22.36 __PTRDIFF_T_IS_INT__ 212

22.37 __PTRDIFF_T_IS_LONG__ 212

Section number	Title	Page

Chapter 23
Using Pragmas

23.1 Checking Pragma Settings....213

23.2 Saving and Restoring Pragma Settings....214

23.3 Determining Which Settings are Saved and Restored....215

23.4 Invalid Pragmas....216

23.5 Pragma Scope....216

Chapter 24
Pragmas for Standard C Conformance

24.1 ANSI_strict....217

24.2 c99....217

24.3 c9x....218

24.4 ignore_oldstyle....218

24.5 only_std_keywords....219

24.6 require_prototypes....219

Chapter 25
Pragmas for C++

25.1 access_errors....224

25.2 always_inline....224

25.3 arg_dep_lookup....225

25.4 ARM_conform....225

25.5 ARM_scoping....225

25.6 array_new_delete....226

25.7 auto_inline....226

25.8 bool....226

25.9 cplusplus....227

25.10 cpp1x....227

25.11 cpp_extensions....228

25.12 debuginline....229

25.13 def_inherited....229

Section number	Title	Page

25.14 defer_codegen....230
25.15 defer_defarg_parsing....230
25.16 direct_destruction....231
25.17 direct_to_som....231
25.18 dont_inline....231
25.19 ecplusplus....231
25.20 exceptions....232
25.21 extended_errorcheck....232
25.22 inline_bottom_up....233
25.23 inline_bottom_up_once....234
25.24 inline_depth....235
25.25 inline_max_auto_size....235
25.26 inline_max_size....236
25.27 inline_max_total_size....236
25.28 internal....237
25.29 iso_templates....237
25.30 new_mangler....238
25.31 no_conststringconv....238
25.32 no_static_dtors....239
25.33 nosyminline....239
25.34 old_friend_lookup....239
25.35 old_pods....240
25.36 old_vtable....240
25.37 opt_classresults....241
25.38 parse_func_templ....241
25.39 parse_mfunc_templ....242
25.40 RTTI....242
25.41 suppress_init_code....242
25.42 template_depth....243

Section number | Title | Page

25.43 thread_safe_init....243

25.44 warn_hidevirtual....244

25.45 warn_no_explicit_virtual....245

25.46 warn_no_typename....246

25.47 warn_notinlined....246

25.48 warn_structclass....246

25.49 wchar_type....247

Chapter 26
Pragmas for Language Translation

26.1 asmsemicolcomment....249

26.2 gcc_extensions....250

26.3 mpwc_newline....250

26.4 mpwc_relax....251

26.5 multibyteaware....252

26.6 multibyteaware_preserve_literals....252

26.7 text_encoding....252

26.8 trigraphs....253

26.9 unsigned_char....254

Chapter 27
Pragmas for Diagnostic Messages

27.1 extended_errorcheck....256

27.2 maxerrorcount....257

27.3 message....257

27.4 showmessagenumber....258

27.5 show_error_filestack....258

27.6 suppress_warnings....259

27.7 sym....259

27.8 unused....259

27.9 warning....261

Section number	Title	Page

27.10 warning_errors....261

27.11 warn_any_ptr_int_conv....262

27.12 warn_emptydecl....262

27.13 warn_extracomma....263

27.14 warn_filenamecaps....263

27.15 warn_filenamecaps_system....264

27.16 warn_hiddenlocals....265

27.17 warn_illpragma....265

27.18 warn_illtokenpasting....265

27.19 warn_illunionmembers....266

27.20 warn_impl_f2i_conv....266

27.21 warn_impl_i2f_conv....267

27.22 warn_impl_s2u_conv....267

27.23 warn_implicitconv....268

27.24 warn_largeargs....269

27.25 warn_missingreturn....269

27.26 warn_no_side_effect....269

27.27 warn_padding....270

27.28 warn_pch_portability....270

27.29 warn_possunwant....271

27.30 warn_ptr_int_conv....272

27.31 warn_resultnotused....272

27.32 warn_undefmacro....273

27.33 warn_uninitializedvar....273

27.34 warn_possiblyuninitializedvar....274

27.35 warn_unusedarg....274

27.36 warn_unusedvar....275

Section number	Title	Page

Chapter 28
Pragmas for Preprocessing

28.1 check_header_flags......277

28.2 faster_pch_gen......278

28.3 flat_include......278

28.4 fullpath_file......279

28.5 fullpath_prepdump......279

28.6 keepcomments......279

28.7 line_prepdump......280

28.8 macro_prepdump......280

28.9 msg_show_lineref......280

28.10 msg_show_realref......281

28.11 notonce......281

28.12 old_pragma_once......281

28.13 once......282

28.14 pop, push......282

28.15 pragma_prepdump......283

28.16 precompile_target......283

28.17 simple_prepdump......284

28.18 space_prepdump......284

28.19 srcrelincludes......285

28.20 syspath_once......285

Chapter 29
Pragmas for Code Generation

29.1 dont_reuse_strings......287

29.2 enumsalwaysint......288

29.3 explicit_zero_data......289

29.4 float_constants......289

29.5 longlong......290

Section number	Title	Page

29.6 longlong_enums....290

29.7 min_enum_size....290

29.8 readonly_strings....291

29.9 safe_index_expr....291

29.10 common_sub_expr_elim....291

29.11 const_propag....292

29.12 copy_propag....292

29.13 dead_store_elim....293

29.14 no_register_coloring....293

29.15 branch_tail_merge....293

Chapter 30 Pragmas for Optimization

30.1 global_optimizer....295

30.2 opt_common_subs....296

30.3 opt_dead_assignments....296

30.4 opt_dead_code....297

30.5 opt_lifetimes....297

30.6 opt_loop_invariants....297

30.7 opt_propagation....298

30.8 opt_strength_reduction....298

30.9 opt_strength_reduction_strict....298

30.10 opt_unroll_loops....299

30.11 optimization_level....299

30.12 optimize_for_size....300

30.13 strictheaderchecking....300

Chapter 31 S12Z Pragmas

31.1 CODE_SEG....303

31.2 DATA_SEG....305

Section number	Title	Page

31.3 CONST_SEG....307

31.4 NO_ENTRY....309

31.5 NO_EXIT....310

31.6 NO_RETURN....310

31.7 align_globals....311

31.8 safe_index_expr....311

31.9 common_sub_expr_elim....312

31.10 const_propag....313

31.11 copy_propag....313

31.12 dead_code_elim....313

31.13 dead_store_elim....314

31.14 branch_tail_merge....314

31.15 peephole....315

31.16 bfield_gap_limit....315

31.17 bfield_lsbit_first....316

31.18 bfield_reduce_type....316

Chapter 1
Overview

The S12Z Build Tools Reference Manual for Microcontrollers describes the compiler used for the Freescale 8-bit Microcontroller Unit (MCU) chip series.

The technical notes and application notes are placed at the following location:

`<CWInstallDir>\MCU\Help\PDF`

This section provides the information about the documentation related to the CodeWarrior Development Studio for Microcontrollers, Version 10.x and contains these major sections:

- Accompanying Documentation
- Additional Information Resources
- Miscellaneous

1.1 Accompanying Documentation

The **Documentation** page describes the documentation included in the *CodeWarrior Development Studio for Microcontrollers v10.x*. You can access the **Documentation** by:

- opening the `START_HERE.html` in `<CWInstallDir>\MCU\Help` folder,
- selecting **Help > Documentation** from the IDE's menu bar, or selecting the **Start > Programs > Freescale CodeWarrior > CW for MCU v10.x > Documentation** from the Windows taskbar.

NOTE

To view the online help for the CodeWarrior tools, first select **Help > Help Contents** from the IDE's menu bar. Next, select required manual from the **Contents** list. For general information about the CodeWarrior IDE and debugger, refer to the *CodeWarrior Common Features Guide* in this folder: `<CWInstallDir>\MCU\Help\PDF`

1.2 Additional Information Resources

- For Freescale documentation and resources, visit the Freescale web site:

  `http://www.freescale.com`

- For additional electronic-design and embedded-system resources, visit the EG3 Communications, Inc. web site:

  `http://www.eg3.com`

- For monthly and weekly forum information about programming embedded systems (including source-code examples), visit the Embedded Systems Programming magazine web site:

  `http://www.embedded.com`

- For late-breaking information about new features, bug fixes, known problems, and incompatibilities, read the release notes in the `CWInstallDir\MCU` folder, where `CWInstallDir` is the directory in which CodeWarrior is installed, and `MCU` is the CodeWarrior Microcontrollers folder.
- To view the online help for the CodeWarrior tools, select **Help > Help Contents** from the IDE's menu bar.

1.3 Miscellaneous

Refer to the documentation listed below for details about programming languages.

- *American National Standard for Programming Languages - C*, ANSI/ISO 9899-1990 (see ANSI X3.159-1989, X3J11)
- *The C Programming Language*, second edition, Prentice-Hall 1988
- *C: A Reference Manual*, second edition, Prentice-Hall 1987, Harbison and Steele
- *C Traps and Pitfalls*, Andrew Koenig, AT&T Bell Laboratories, Addison-Wesley Publishing Company, Nov. 1988, ISBN 0-201-17928-8
- *Data Structures and C Programs*, Van Wyk, Addison-Wesley 1988
- *How to Write Portable Programs in C*, Horton, Prentice-Hall 1989
- *The UNIX Programming Environment*, Kernighan and Pike, Prentice-Hall 1984
- *The C Puzzle Book*, Feuer, Prentice-Hall 1982
- *C Programming Guidelines*, Thomas Plum, Plum Hall Inc., Second Edition for Standard C, 1989, ISBN 0-911537-07-4

- *DWARF Debugging Information Format*, UNIX International, Programming Languages SIG, Revision 1.1.0 (October 6, 1992), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054
- *DWARF Debugging Information Format*, UNIX International, Programming Languages SIG, Revision 2.0.0 (July 27, 1993), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054
- *System V Application Binary Interface*, UNIX System V, 1992, 1991 UNIX Systems Laboratories, ISBN 0-13-880410-9
- *Programming Microcontroller in C*, Ted Van Sickle, ISBN 1878707140
- *C Programming for Embedded Systems*, Kirk Zurell, ISBN 1929629044
- *Programming Embedded Systems in C and C ++*, Michael Barr, ISBN 1565923545
- *Embedded C*, Michael J. Pont, ISBN 020179523X

Chapter 2 Introduction

This chapter explains how to use CodeWarrior tools to build programs. CodeWarrior build tools translate source code into object code then organize that object code to create a program that is ready to execute. CodeWarrior build tools run on the *host* system to generate software that runs on the *target* system. Sometimes the host and target are the same system. Usually, these systems are different.

This chapter covers the CodeWarrior compiler and its linker, versions 10.x and higher.

This chapter consists of the following topics:

- Compiler Architecture

2.1 Compiler Architecture

From a programmer's point of view, the CodeWarrior compiler translates source code into object code. Internally, however, the CodeWarrior compiler organizes its work between its front-end and back-end, each end taking several steps. The following figure shows the steps the compiler takes.

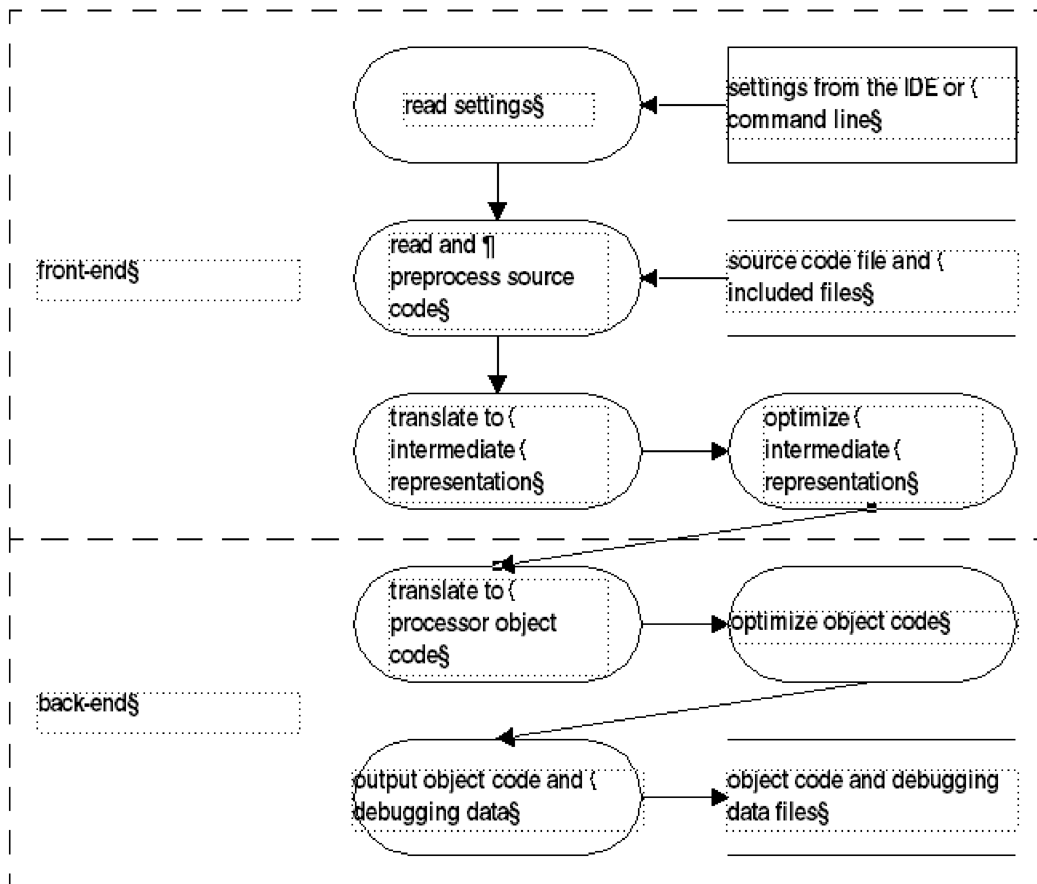

Figure 2-1. CodeWarrior Compiler Steps

Front-end steps:

- **read settings** : retrieves your settings from the host's integrated development environment (IDE) or the command line to configure how to perform subsequent steps
- **read and preprocess source code** : reads your program's source code files and applies preprocessor directives
- **translate to intermediate representation** : translates your program's preprocessed source code into a platform-independent intermediate representation
- **optimize intermediate representation** : rearranges the intermediate representation to reduce your program's size, improve its performance, or both

Back-end steps:

- **translate to processor object code** : converts the optimized intermediate representation into native object code, containing data and instructions, for the target processor

- **optimize object code** : rearranges the native object code to reduce its size, improve performance, or both
- **output object code and diagnostic data** : writes output files on the host system, ready for the linker and diagnostic tools such as a debugger or profiler

Chapter 3
Creating Project

This chapter covers the primary method to create a project with the CodeWarrior Development Studio for Microcontrollers, Version 10.x.

For information on creating and compiling a project using the CodeWarrior IDE, refer to the Creating and Managing Project Using CodeWarrior IDE section of this chapter.

NOTE

Information on the other build tools can be found in *User Guide*s included with the CodeWarrior Suite and are located in the `Help` folder for the CodeWarrior installation. The default location of this folder is: `C:\Freescale\CW MCU v10.x\MCU\Help`

3.1 Creating and Managing Project Using CodeWarrior IDE

You can create an Microcontrollers project and generate the basic project files using the **New Bareboard Project** wizard in the CodeWarrior IDE. You can use the **CodeWarrior Projects** view in the CodeWarrior IDE to manage files in the project.

3.1.1 Creating Project Using New Bareboard Project Wizard

The steps below create an example Microcontrollers project that uses C language for its source code.

1. Select **Start > Programs > Freescale CodeWarrior > CW for MCU v10.x > CodeWarrior**.

   The **Workspace Launcher** dialog box appears. The dialog box displays the default workspace directory. For this example, the default workspace is `workspace_MCU`.

2. Click **OK** to accept the default location. To use a workspace different from the default, click **Browse** and specify the desired workspace.

The CodeWarrior IDE launches.

3. Select **File > New > Bareboard Project** from the IDE menu bar.

   The **Create an MCU Bareboard Project** page of the **New Bareboard Project** wizard appears.

4. Enter the name of the project in the **Project name** text box. For example, type in `S12Z_Project`.
5. Click **Next**.

   The **Devices** page appears.

6. Select the desired CPU derivative for the project.
7. Click **Next**.

   The **Connections** page appears.

8. Select the connection(s) appropriate for your project.
9. Click **Next**.

   The **Language and Build Tools Options** page appears.

10. Select the options appropriate for your project.
11. Click **Next**.

    The **Rapid Application Development** page appears.

12. Select the options appropriate for your project.
13. Click **Finish**.

NOTE

For detailed descriptions of the options available in the **New Bareboard Project** wizard pages, refer to the *Microcontrollers V10.x Targeting Manual*.

The Wizard automatically generates the startup and initialization files for the specific microcontroller derivative, and assigns the entry point into your ANSI-C project (the `main()` function). The `S12Z_Project` project appears in the **CodeWarrior Projects** view in the Workbench window.

By default, the project is not built. To do so, select **Project > Build Project** from the IDE menu bar. Expand the `S12Z_Project` tree control in the **CodeWarrior Projects** view to display its supporting directories and files.

NOTE

To configure the IDE, so that it automatically builds the project when a project is created, select **Window > Preferences** to

open the **Preferences** window. Expand the **General** node and select **Workspace**. In the **Workspace** panel, check the **Build Automatically** checkbox and click **OK**.

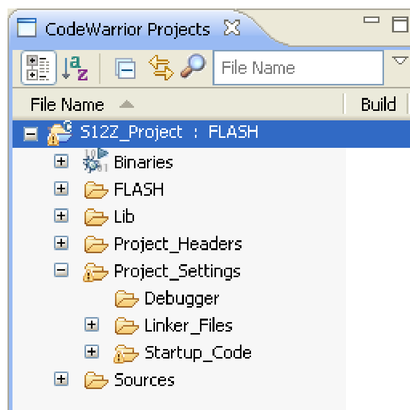

Figure 3-1. CodeWarrior Projects View

The expanded view displays the logical arrangement of the project files. At this stage, you can safely close the project and reopen it later, if desired.

The following is the list of the default groups and files displayed in the **CodeWarrior Projects** view.

- `Binaries` is a link to the generated binary (`.elf`) file.
- `FLASH` is the directory that contains all of the files used to build the application for `S12Z_Project`. This includes the source, lib, the makefiles that manage the build process, and the build settings.
- `Lib` is the directory that contains a `c` source code file that describes the chosen MCU derivative's registers and the symbols used to access them.
- `Project_Headers` is the directory that contains any MCU-specific header files.
- `Project_Settings` group consists of the following folders:
  - `Debugger`: Consists of any initialization and memory configuration files that prepare the hardware target for debugging. It also stores the launch configuration used for the debugging session.
  - `Linker_Files`: Stores the linker command file (`.prm`) and the burner command file (`.bbl`).
  - `Startup_Code`: Contains a C file that initializes the MCU's stack and critical registers when the program launches.
- `Sources` contains the source code files for the project. For this example, the wizard has created only `main.c`, which contains the `main()` function.

The CodeWarrior compiler allows you to compile the C-source code files separately, simultaneously, or in other combinations.

Examine the project folder that the IDE generates when you create the project. To do this, right-click on the project's name (`S12Z_Project : FLASH`) in the **CodeWarrior Projects** view, and select **Show In Windows Explorer**. The workspace folder containing the project folder, `S12Z_Project` appears.

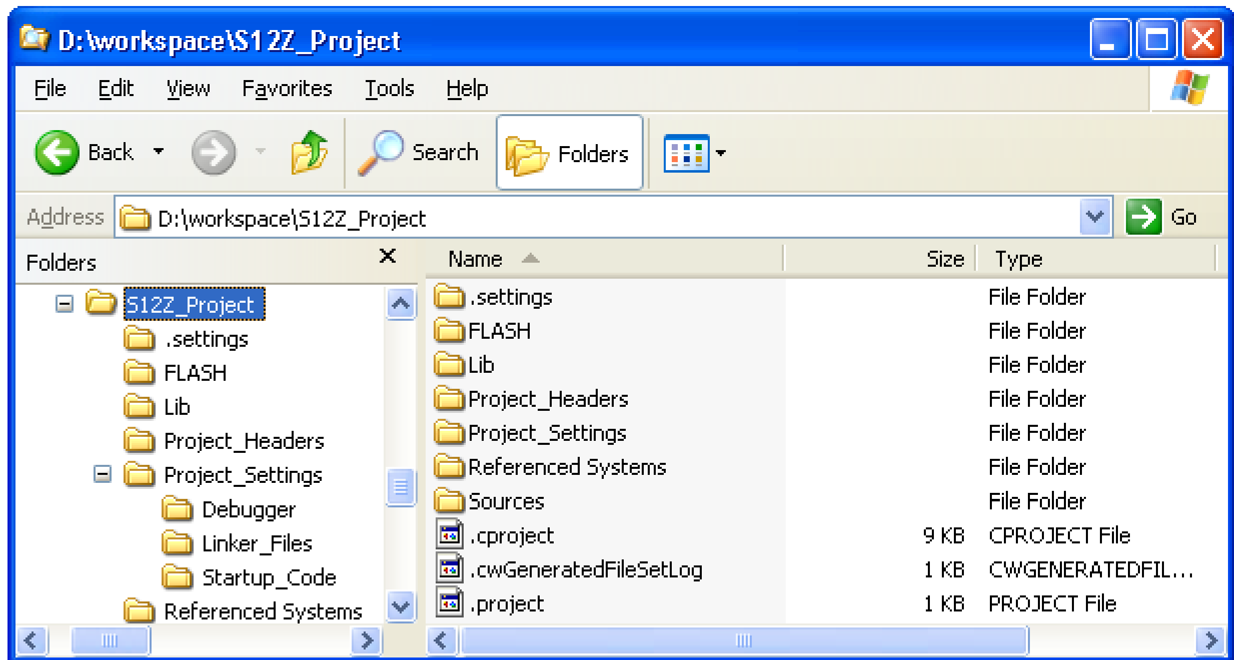

Figure 3-2. Contents of S12Z_Project Directory

These are the actual folders and files generated for your project. When working with standalone tools, you may need to specify the paths to these files, so you should know their locations.

NOTE

The files (`.project`, `.cproject`) store critical information about the project's state. The **CodeWarrior Projects** view does not display these files, but they should not be deleted.

3.1.2 Analysis of Groups in CodeWarrior Projects View

In the **CodeWarrior Projects** view, the project files are distributed into five major groups, each with their own folder within the `S12Z_Project` folder.

The default groups and their usual functions are:

- `FLASH`

The `FLASH` group contains all of the files that the CodeWarrior IDE uses to build the program. It also stores any files generated by the build process, such as any binaries (`.o`, `.obj`, and `.elf`), and a map file (`.map`). The CodeWarrior IDE uses this directory to manage the build process, so you should not tamper with anything in this directory. This directory's name is based on the build configuration, so if you switch to a different build configuration, its name changes.

- `Lib`

  The `Lib` group contains the C-source code file for the chosen MCU derivative. For example, the `mc9s12zvh64.c` file supports the `mc9s12zvh64` derivative. This file defines symbols that you use to access the MCU's registers and bits within a register. It also defines symbols for any on-chip peripherals and their registers. After the first build, you can expand this file to see all of the symbols that it defines.

- `Project_Headers`

  The `Project_Headers` group contains the derivative-specific header files required by the MCU derivative file in the `Lib` group.

- `Project_Settings`

  The `Project_Settings` group consists of the following sub-folders:

  - `Debugger`

    This group contains the files used to manage a debugging session.

  - `Linker_Files`

    This group contains the linker file.

  - `Startup_Code`

    This group contains the source code that manages the MCU's initialization and startup functions. For S12Z derivatives, these functions appear in the source file `starts12z.c`.

- `Sources`

  This group contains the user's C source code files. The **New Bareboard Project** wizard generates a default `main.c` file for this group. You can add your own source files to this folder. You can double-click on these files to open them in the IDE's editor. You can right-click on the source files and select **Resource Configurations > Exclude from Build** to prevent the build tools from compiling them.

3.1.3 Analysis of Files in CodeWarrior Projects View

Expand the groups in the **CodeWarrior Projects** view to display all the default files generated by the **New Bareboard Project** wizard.

The wizard generates following three C source code files, located in their respective folders in the project directory:

- `main.c`,

  located in <*project_directory*>`\Sources`
- `starts12z.c`, and

  located in <*project_directory*>`\Project_Settings\Startup_Code`
- `mc9s12zvh64.c`

  a target specific `.c` file, in this case `mc9s12zvh64.c`, located in <*project_directory*>`\Lib`

At this time, the project should be configured correctly and the source code free of syntactical errors. If the project has been built already, you should see a link to the project's binary files, and the `FLASH` folder in the **CodeWarrior Projects** view.

To understand what the IDE does while building a project, clean the project and re-build the project.

1. Select **Project > Clean** from the IDE menu bar.

   The **Clean** dialog box appears.

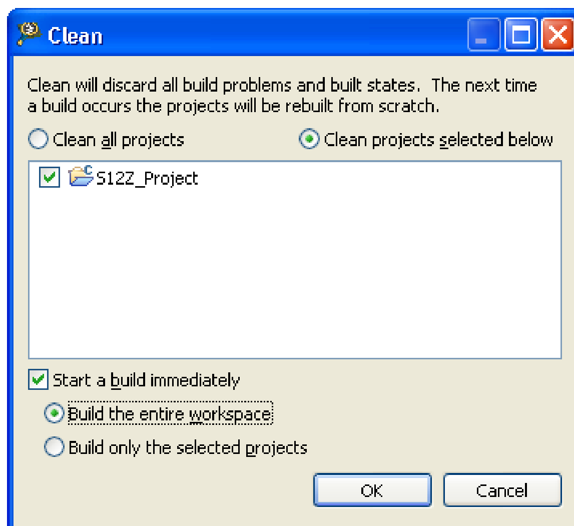

Figure 3-3. Clean Dialog Box

2. Select the **Clean projects selected below** option and select the project you want to re-build.

3. Clear the **Start a build immediately** checkbox. Click **OK**.

   The `Binaries` link disappears, and the `FLASH` folder is deleted.

4. Select **Project > Build Project** from the IDE menu bar.

   The **Console** view displays the statements that direct the build tools to compile and link the project. The `Binaries` link appears, and so does the `FLASH` folder.

During a project build, the C source code is compiled, the object files are linked together, and the CPU derivative's ROM and RAM area are allocated by the linker according to the settings in the linker command file. When the build is complete, the `FLASH` folder contains the `S12Z_Project.elf` file.

The Linker Map file (`S12Z_Project.map`) file indicates the memory areas allocated for the program and contains other useful information.

To examine the source file, `main.c`, double click on the `main.c` file in the `Sources` group. The default `main.c` file opens in the editor area, as the following figure shows.

Figure 3-4. Default main.c File

Use the integrated editor to write your C source files (`*.c` and `*.h`) and add them to your project. During development, you can test your source code by building and simulating/ debugging your application.

3.2 Highlights

The CodeWarrior build tools provide the following features:

- Powerful User Interface
- Online Help
- Flexible Type Management
- Flexible Message Management
- 32-bit Application
- Support for Encrypted Files
- High-Performance Optimizations
- Conforms to ANSI/ISO 9899-1990

3.3 CodeWarrior Integration of Build Tools

All required CodeWarrior plug-ins are installed together with the Eclipse IDE. The program that launches the IDE with the CodeWarrior tools, `cwide.exe`, is installed in the `eclipse` directory (usually `C:\Freescale\CW MCU V10.x\eclipse`). The plug-ins are installed in the `eclipse\plugins` directory.

3.3.1 Combined or Separated Installations

The installation script enables you to install several CPUs along one single installation path. This saves disk space and enables switching from one processor family to another without leaving the IDE.

NOTE

It is possible to have separate installations on one machine. There is only one point to consider: The IDE uses COM files, and for COM the IDE installation path is written into the Windows Registry. This registration is done in the installation setup. However, if there is a problem with the COM registration using several installations on one machine, the COM registration is done by starting a small batch file located in the `bin` (usually `C:\Freescale\CW MCU V10.x\MCU\bin`) directory. To do this, start the `regservers.bat` batch file.

3.3.2 S12Z Compiler Build Settings Panels

The following sections describe the settings panels that configure the build tool options. These panels are part of the project's build properties settings, which are managed in the **Properties for <project>** dialog box. To access these panels, proceed as follows:

1. Select the project for which you want to set the build properties, in the **CodeWarrior Projects** view.
2. Select **Project > Properties** from the IDE menu bar.

   A **Properties for <project>** dialog box appears.

3. Expand the **C/C++ Build** tree control, and then select the **Settings** option.

   The settings for the build tools are displayed in the right panel of the **Properties for <project>** dialog box. If not, click on the **Tool Settings** tab.

The options are grouped by tool, such as **S12Z Burner** options, **S12Z Linker** options, **S12Z Compiler** options, and so on. Scroll down the list, and click on the option whose settings you wish to examine and modify.

The following table lists the build properties specific to developing software for S12Z.

The properties that you specify in these panels, apply to the selected build tool on the **Tool Settings** page of the **Properties for <project>** dialog box.

Table 3-1. Build Properties for S12Z

Build Tool	Build Properties Panels
S12Z Compiler	S12Z Compiler > Input
	S12Z Compiler > Access Paths
	S12Z Compiler > Warnings
	S12Z Compiler > Code Generation
	S12Z Compiler > Optimization
	S12Z Compiler > Language
	S12Z Compiler > Messages
	S12Z Compiler > General

3.3.2.1 S12Z Compiler

Use this panel to specify the command, options, and expert settings for the build tool compiler. Additionally, the S12Z Compiler tree control includes the general and the file search path settings.

The following table lists and describes the compiler options for S12Z.

Table 3-2. Tool Settings - Compiler Options

Option	Description
Command	Shows the location of the compiler executable file. Default value is: `"${S12Z_ToolsDir}/mwccs12lisa"` You can specify additional command line options for the compiler; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the compiler will be called with.
Expert Settings Command line pattern	Shows the command line pattern; default is `${COMMAND} -c ${FLAGS} ${OUTPUT_FLAG} ${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`.

3.3.2.2 S12Z Compiler > Input

Use this panel to specify file search paths and any additional include files the **S12Z Compiler** should use. You can specify multiple search paths and the order in which you want to perform the search.

The IDE first looks for an include file in the current directory, or the directory that you specify in the `INCLUDE` directive. If the IDE does not find the file, it continues searching the paths shown in this panel. The IDE keeps searching paths until it finds the `#include` file or finishes searching the last path at the bottom of the Include File Search Paths list. The IDE appends to each path the string that you specify in the `INCLUDE` directive.

NOTE

The IDE displays an error message if a header file is in a different directory from the referencing source file. Sometimes, the IDE also displays an error message if a header file is in the same directory as the referencing source file.

For example, if you see the message `Could not open source file myfile.h`, you must add the path for `myfile.h` to this panel.

The following table lists and describes the input options for S12Z Compiler.

Table 3-3. Tool Settings - S12Z Compiler > Input Options

Option	Description
Prefix File	This option allows you to specify the prefix file or precompiled header file search path.
Source File Encoding	This option allows you to select the source file encoding. The options available are: • ASCII (default) • Auto-Detect (multibyte encoding) • System (use system locale) • UTF-8 • Shift-JIS • EUC-JP • ISO-2022-JP
Allow Macro Redefinition	This option allows macro redefinitions without an error or warning.
Defined Macros	Use this option to specify the defined macros.
Undefined Macros	Use this option to specify the undefined macros.

3.3.2.3 S12Z Compiler > Access Paths

Use this panel to specify access path options for the S12Z Compiler.

The following table lists and describes the access paths options for S12Z.

Table 3-4. Tool Settings - S12Z Compiler > Access Paths Options

Option	Description
Do Not use MWCIncludes Variable	This option inhibits the usage of MWCInclude variables. By default, this checkbox is checked.
Always Search User Paths	Use this option to enable the usage of the search paths.
Search User Paths (#include "...")	Use this option to specify the user paths.
Search User Paths Recursively	Use this option to specify the user paths recursively.
Search System Paths (#include <...>)	Use this option to specify the system paths.
Search System Paths Recursively	Use this option to specify the system paths recursively.

3.3.2.4 S12Z Compiler > Warnings

Use this panel to specify the warnings settings for S12Z compiler.

The following table lists and describes the **Warnings** options for S12Z compiler.

Table 3-5. Tool Settings - S12Z Compiler > Warnings Options

Option	Description
Treat All Warnings as Errors	Check to treat all warnings as errors. The compiler will stop if it generates a warning message.
Enable Warnings	Select the level of warnings you want reported from the compiler. Custom lets you to select individual warnings. Other settings select a pre-defined set of warnings. The available options are as follows: • Custom • Off • Most • All • Full (likely to generate spurious warnings)
Illegal #pragmas (most)	Check to notify the presence of illegal pragmas.
Possible Unwanted Effects (most)	Check to notify the presence of illegal pragmas.
Extended Error Checks (most)	Check if you want to do an extended error checking.
Hidden Virtual Functions (most)	Check to generate a warning message if you declare a non-virtual member function that prevents a virtual function, that was defined in a superclass, from being called and is equivalent to pragma `warn_hidevirtual` and the command-line option `-warnings hidevirtual`.
Implicit Arithmetic Conversions (all)	Check to warn of implict arithmetic conversions.
Implicit Signed/Unsigned Conversions (all)	Check to enable warning of implict conversions between signed and unsigned variables.
Implicit Float to Integer Conversions (all)	Check to warn of implict conversions of a floating-point variable to integer type.
Implicit Integer to Float Conversions (all)	Check to warn of implict conversion of an integer variable to floating-point type.
Pointer/Integer Conversions (most)	Check to enable warnings of conversions between pointer and integers.
Unused Arguments (most)	Check to warn of unused arguments in a function.
Unused Variables (most)	Check to warn of unused variables in the code.
Unused Result from Non-Void-Returning Function (full)	Check to warn of unused result from nonvoid-returning functions.
Missing 'return' Value in Non-Void-Returning Function (most)	Check to warn of when a function lacks a return statement.
Expression Has No Side Effect (most)	Check to issue a warning message if a source statement does not change the program's state. This is equivalent to the pragma `warn_no_side_effect`, and the command-line option `-warnings unusedexpr`.
Extra Commas (most)	Check to issue a warning message if a list in an enumeration terminates with a comma. The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 ("C99") standard and is equivalent to pragma `warn_extracomma` and the command-line option `-warnings extracomma`.
Empty Declarations (most)	Check to warn of empty declarations.
Inconsistent 'class'/'struct' Usage (most)	Check to warn of inconsistent usage of class or struct.

Table continues on the next page...

Table 3-5. Tool Settings - S12Z Compiler > Warnings Options (continued)

Option	Description
Incorrect Capitalization in #include "..." (most)	Check to issue a warning message if the name of the file specified in a `#include "file"` directive uses different letter case from a file on disk and is equivalent to pragma `warn_filenamecaps` and the commandline option `-warnings filecaps`.
Incorrect Capitalization in System #include <...> (most)	Check to issue a warning message if the name of the file specified in a `#include <file>` directive uses different letter case from a file on disk and is equivalent to pragma `warn_filenamecaps_system` and the command-line option `-warnings sysfilecaps`.
Pad Bytes Added (full)	Check to issue a warning message when the compiler adjusts the alignment of components in a data structure and is equivalent to pragma `warn_padding` and the command-line option `-warnings padding`.
Undefined Macro in #if/#elif (full)	Check to issues a warning message if an undefined macro appears in `#if` and `#elif` directives and is equivalent to pragma `warn_undefmacro` and the command-line option `-warnings undefmacro`.
Non-Inlined Functions (full)	Check to issue a warning message if a call to a function defined with the `inline`, `__inline__`, `or __inline` keywords could not be replaced with the function body and is equivalent to pragma `warn_notinlined` and the command-line option `-warnings notinlined`.
Token Not Formed by ## Operator (most)	Check to enable warnings for the illegal uses of the preprocessor's token concatenation operator (##). It is equivalent to the pragma `warn_illtokenpasting on`.

3.3.2.5 S12Z Compiler > Code Generation

Use this panel to control the code generation for S12Z Compiler.

The following table lists and describes the **Code Generation** options for S12Z compiler.

Table 3-6. Tool Settings - S12Z Compiler > Code Generation Options

Option	Description
Memory Model	This option allows to specify the memory model. The options available are: • Small • Medium (default) • Large
Bit-field gap limit (0 - 127 or 255 (0xff) as -1)	The bitfield allocation tries to avoid crossing a byte boundary whenever possible. To achieve optimized accesses, the compiler may insert some padding or gap bits to reach this. Use this option to affect the maximum number of gap bits allowed.

Table continues on the next page...

Table 3-6. Tool Settings - S12Z Compiler > Code Generation Options (continued)

Option	Description
Bit-field byte allocation from LSB to MSB (right-to-left)	Use this option to produces less code overhead in the case of partially allocated byte bitfields.
Bit-field type size reduction	Type-size reduction means that the compiler reduces the type of an `int` bitfield to a `char` bitfield if the `int` bitfield fits into a character. Use this option to allow the compiler to allocate memory only for one byte instead of for an integer.

3.3.2.6 S12Z Compiler > Optimization

Use this panel to control compiler optimizations. The compiler's optimizer can apply any of its optimizations in either global or non-global optimization mode. You can apply global optimization at the end of the development cycle, after compiling and optimizing all source files individually or in groups.

The following table lists and describes the **Optimization** options for S12Z compiler.

Table 3-7. Tool Settings - S12Z Compiler > Optimization Options

Option	Description
Optimization Level	Specify the optimizations that you want the compiler to apply to the generated object code. The options available are as follows: **Off (default)** - Disables optimizations. This setting is equivalent to specifying the `-O0` command-line option. The compiler generates unoptimized, linear assembly-language code. **1** - The compiler performs all target-independent (that is, non-parallelized) optimizations, such as function inlining. This setting is equivalent to specifying the `-O1` command-line option. The compiler omits all target-specific optimizations and generates linear assembly-language code. **2** - The compiler performs all optimizations (both target-independent and target-specific). This setting is equivalent to specifying the `-O2` command-line option. The compiler outputs optimized, non-linear, parallelized assembly-language code. **3** - The compiler performs all the level 2 optimizations, then the low-level optimizer performs global-algorithm register allocation. This setting is equivalent to specifying the `-O3` command-line option. At this optimization level, the compiler generates code that is usually faster than the code generated from level 2 optimizations. By default, the optimization is disabled for S12Z Compiler.
Speed Vs Size	Use to specify an Optimization Level greater than 0.

Table continues on the next page...

Table 3-7. Tool Settings - S12Z Compiler > Optimization Options (continued)

Option	Description
	• **Speed** - The compiler optimizes object code at the specified Optimization Level such that the resulting binary file has a faster execution speed, as opposed to a smaller executable code size. • **Size(default)** - The compiler optimizes object code at the specified Optimization Level such that the resulting binary file has a smaller executable code size, as opposed to a faster execution speed. This setting is equivalent to specifying the `-Os` command-line option.
Inline Level	Enables inline expansion. If there is a #pragma INLINE before a function definition, all calls of this function are replaced by the code of this function, if possible. The options available are: • Off • Smart (default) • 1 • 2 • 3 • 4 • 5 • 6 • 7 • 8
Auto Inline	Check to enable the auto inlining.
Bottom-Up Inlining	Check to control the bottom-up function inlining method. When active, the compiler inlines function code starting with the last function in the chain of functions calls, to the first one.
Array index expressions do not overflow the index type (this will enable array accesses optimizations)	Check to enable the array accesses optimizations.

3.3.2.7 S12Z Compiler > Language

Use this panel to specify the language settings for S12Z compiler.

The following table lists and describes the **Language** options for S12Z compiler.

Table 3-8. Tool Settings - S12Z Compiler > Language Options

Option	Description
Require Function Prototypes	Check to enforce the requirement of function prototypes. The compiler generates an error message if you define a previously referenced function that does not have a prototype. If you define the function before it is referenced but do not give it a prototype, this setting causes the compiler to issue a warning message.

Table continues on the next page...

Table 3-8. Tool Settings - S12Z Compiler > Language Options (continued)

Option	Description
Enable C++ 'bool' type, 'true' and 'false' Constants	Check to enable the C++ compiler to recognize the bool type and its true and false values specified in the ISO/IEC 14882-2003 C++ standard.
ISO C++ Template Parser	Check to follow the ISO/IEC 14882-2003 standard for C++ to translate templates, enforcing more careful use of the typename and template keywords. The compiler also follows stricter rules for resolving names during declaration and instantiation.
Use Instance Manager	Check to reduce compile time by generating any instance of a C++ template (or noninlined inline) function only once.
Force C++ Compilation	Check to enable the forced C++ compilation.
Enable GCC Extensions	Check to recognize language features of the GNU Compiler Collection (GCC) C compiler that are supported by CodeWarrior compilers; is equivalent to pragma `gcc_extensions` and the command-line option `-gcc_extensions`.
Enable C99 Extensions	Check to recognize ISO/IEC 9899-1999 ("C99") language features; is equivalent to pragma `c99` and the command-line option `-dialect c99`.
Enable C++ Exceptions	Check to generate executable code for C++ exceptions; is equivalent to pragma `exceptions` and the command-line option `-cpp_exceptions`.
Enable RTTI	Check to allow the use of the C++ runtime type information (RTTI) capabilities, including the `dynamic_cast` and typeid operators; is equivalent to `pragma RTTI` and the command-line option `-RTTI`.
Enable wchar_t Support	Check to enable C++ compiler recognize the `wchar_t` data type specified in the ISO/IEC 14882-2003 C++ standard; is equivalent to `pragma wchar_type` and the command-line option `-wchar_t`.
ANSI Strict	Check to enable C compiler operate in strict ANSI mode. In this mode, the compiler strictly applies the rules of the ANSI/ ISO specification to all input files. This setting is equivalent to specifying the - ansi command-line option. The compiler issues a warning for each ANSI/ISO extension it finds.
ANSI Keywords Only	Check to generate an error message for all non-standard keywords (ISO/IEC 9899-1990 C, §6.4.1). If you must write source code that strictly adheres to the ISO standard, enable this setting; is equivalent to pragma `only_std_keywords` and the command-line option `-stdkeywords`.
Expand Trigraphs	Check to recognize trigraph sequences (ISO/ IEC 9899-1990 C, §5.2.1.1); is equivalent to pragma `trigraphs` and the commandline option `-trigraphs`.
Legacy for-scoping	Check to generate an error message when the compiler encounters a variable scope usage that the ISO/IEC 14882-2003 C++ standard disallows, but is allowed in the C++ language specified in The Annotated C++ Reference Manual ("ARM"); is equivalent to pragma `ARM_scoping` and the command-line option `-for_scoping`.

Table continues on the next page...

Table 3-8. Tool Settings - S12Z Compiler > Language Options (continued)

Option	Description
Enum Always Int	Check to use signed integers to represent enumerated constants and is equivalent to pragma `enumsalwaysint` and the command-line option `-enum`.
Use Unsigned Chars	Check to treat char declarations as unsigned char declarations and is equivalent to pragma `unsigned_char` and the command-line option `-char unsigned`.
Reuse Strings	Check to store only one copy of identical string literals and is equivalent to opposite of the pragma `dont_reuse_strings` and the command-line option `-string reuse`.
Pool Strings	Check to collect all string constants into a single data section in the object code it generates and is equivalent to pragma `pool_strings` and the command-line option `-strings pool`.

3.3.2.8 S12Z Compiler > Messages

Use this panel to specify the messages settings for S12Z compiler.

The following table lists and describes the **Messages** options for S12Z compiler.

Table 3-9. Tool Settings - S12Z Compiler > Messages Options

Option	Description
Message Style	Use this option to set the message style. The options available are: • GCC • MPW • Standard • IDE • Parseable (default) • Enterprise-IDE
Maximum Number of Errors	This option allows you to specify the maximum number of error messages to be displayed.
Maximum Number of Warnings	This option allows you to specify the maximum number of warning messages to be displayed.

3.3.2.9 S12Z Compiler > General

Use this panel to specify the general compiler behavior.

The following table lists and describes the general compiler options for S12Z architecture.

Table 3-10. Tool Settings - Linker > General Options

Option	Description
Generate Debug Information	This option allows the compiler to generate the debug information. By default, this checkbox is checked.
Other Flags	Specify additional command line options for the linker; type in custom flags that are not otherwise available in the UI.

3.3.3 CodeWarrior Tips and Tricks

The *CodeWarrior Tips and Tricks* are as follows:

- If the Simulator or Debugger cannot be launched, check the settings in the *project's launch configuration*. For more information on launch configurations and their settings, refer to the *Mircocontrollers Version 10.x Targeting Manual*.

  NOTE

  You can view and modify the project's launch configurations from the IDE's **Run Configurations** or **Debug Configurations** dialog box. To open these dialog boxes, select **Run > Run Configurations** or **Run > Debug Configurations**.

- If a file cannot be added to the project, its file extension may not be available in the File Types panel. To resolve the issue, add the file's extension to the list in the **File Types** panel.
  a. Select **Project > Properties** from the IDE menu bar.

     The **Properties for <project>** dialog box appears.

  b. Expand the **C/C++ General** tree control and select the **File Types** option.
  c. Select the **Use project settings** option.
  d. Click **New**, enter the required file type, and click **OK**.
  e. Click **OK** to save the changes and close the properties window.

Chapter 4
Using Build Tools on Command Line

CodeWarrior build tools may be invoked from the command-line. These command-line tools operate almost identically to their counterparts in an integrated development environment (IDE). CodeWarrior command-line compilers and assemblers translate source code files into object code files. CodeWarrior command-line linkers then combine one or more object code files to produce an executable image file, ready to load and execute on the target platform. Each command-line tool has options that you configure when you invoke the tool.

- Configuring Command-Line Tools
- Invoking Command-Line Tools
- Getting Help
- File Name Extensions

4.1 Configuring Command-Line Tools

This topic contains the following sections:

- Setting CodeWarrior Environment Variables
- Setting the PATH Environment Variable

4.1.1 Setting CodeWarrior Environment Variables

Use environment variables on the host system to specify to the CodeWarrior command line tools where to find CodeWarrior files for compiling and linking. The following table describes these environment variables.

Table 4-1. Environment Variables for CodeWarrior Command-line Tools

This environment variable...	specifies this information
`MWCIncludes`	Directories on the host system for system header files for the CodeWarrior compiler.
`MWLibraries`	Directories on the host system for system libraries for the CodeWarrior linker.
`CWFolder`	CodeWarrior installation path on the host system.

A system header file is a header file that is enclosed with the less than (`<`) and greater than (`>`) characters in include directives. For example

```
#include <stdlib.h> /* stdlib.h system header. */
```

Typically, you define the `MWCIncludes` and `MWLibraries` environment variables to refer to the header files and libraries in the subdirectories of your CodeWarrior software.

To specify more than one directory for the `MWCIncludes` and `MWLibraries` variables, use the conventional separator for your host operating system command-line shell.

Listing: Setting Environment Variables in Microsoft® Windows® Operating Systems

```
rem Use ; to separate directory paths
set CWFolder=C:\Freescale\CW MCU v10.x
set MWCIncludes=%CWFolder%\MCU\S12lisa_Support\s12lisac\Include
set MWCIncludes=%MWCIncludes%;%CWFolder%\MCU\S12lisa_Support\s12lisac\Include
set MWCIncludes=%MWCIncludes%;%CWFolder%\MCU\S12lisa_Support\s12lisac\src
set MWLibraries=%CWFolder%\MCU\S12lisa_Support\s12lisac\lib
```

4.1.2 Setting PATH Environment Variable

The `PATH` variable should include the paths for your CodeWarrior tools, shown in the following listing. *Toolset* represents the name of the folder that contains the command line tools for your build target.

Listing: Example of setting PATH

```
set CWFolder=C:\Freescale\CW MCU v10.x
set PATH=%PATH%;%CWFolder%\MCU\Bin;%CWFolder%\MCU\Command_Line_Tools
```

4.2 Invoking Command-Line Tools

To compile, assemble, link, or perform some other programming task with the CodeWarrior command-line tools, you type a command at a command line's prompt. This command specifies the tool you want to run, what options to use while the tool runs, and what files the tool should operate on.

The form of a command to run a command-line tool is

```
tool options files
```

where *tool*is the name of the CodeWarrior command-line tool to invoke, *options* is a list of zero or more options that specify to the tool what operation it should perform and how it should be performed, and *files* is a list of files zero or more files that the tool should operate on.

The CodeWarrior command-line tools for S12Z architectures are as follows:

- `as12lisa.exe` (assembler) - the assembler translate asm files into object files
- `mwccs12lisa.exe` (compiler) - the compiler translates C/C++ compilation units into object files.
- `linker.exe` (smart linker) - the linker binds object files to create executables or libraries.

Which options and files you should specify depend on what operation you want the tool to perform.

The tool then performs the operation on the files you specify. If the tool is successful it simply finishes its operation and a new prompt appears at the command line. If the tool encounters problems it reports these problems as text messages on the command-line before a new prompt appears.

Scripts that automate the process to build a piece of software contain commands to invoke command-line tools. For example, the `make` tool, a common software development tool, uses scripts to manage dependencies among source code files and invoke command-line compilers, assemblers and linkers as needed, much like the CodeWarrior IDE's project manager.

4.3 Getting Help

To show short descriptions of a tool's options, type this command at the command line:

```
tool -help
```

where *tool* is the name of the CodeWarrior build tool.

To show only a few lines of help information at a time, pipe the tool's output to a pager program. For example,

```
tool -help | more
```

will use the `more` pager program to display the help information.

Enter the following command in a **Command Prompt** window to see a list of specifications that describe how options are formatted:

```
tool -help usage
```

where *tool* is the name of the CodeWarrior build tool.

4.3.1 Parameter Formats

Parameters in an option are formatted as follows:

- A parameter included in brackets " `[]`" is optional.
- Use of the ellipsis " `...`" character indicates that the previous type of parameter may be repeated as a list.

4.3.2 Option Formats

The options are formatted as follows:

- For most options, the option and the parameters are separated by a space as in "`-xxx param`".

  When the option's name is "`-xxx+`", however, the parameter must directly follow the option, without the "`+`" character (as in "`-xxx45`") and with no space separator.
- An option given as "`-[no]xxx`" may be issued as "`-xxx`" or "`-noxxx`".

  The use of "`-noxxx`" reverses the meaning of the option.
- When an option is specified as "`-xxx | yy[y] | zzz`", then either "`-xxx`", "`-yy`", "`-yyy`", or "`-zzz`" matches the option.
- The symbols "`,`" and "`=`" separate options and parameters unconditionally; to include one of these symbols in a parameter or filename, escape it (e.g., as "`\,`" in `mwcc file.c \,v`).

4.3.3 Common Terms

These common terms appear in many option descriptions:

- A *cased* option is considered case-sensitive. By default, no options are case-sensitive.
- The *compatibility* indicates that the option is borrowed from another vendor's tool and its behavior may only approximate its counterpart.
- A *global* option has an effect over the entire command line and is parsed before any other options. When several global options are specified, they are interpreted in order.
- A *deprecated* option will be eliminated in the future and should no longer be used. An alternative form is supplied.
- An *ignored* option is accepted by the tool but has no effect.
- A *meaningless* option is accepted by the tool but probably has no meaning for the target operating system.
- An *obsolete* option indicates a deprecated option that is no longer available.
- A *substituted* option has the same effect as another option. This points out a preferred form and prevents confusion when similar options appear in the help.
- Use of *default* in the help text indicates that the given value or variation of an option is used unless otherwise overridden.

This tool calls the linker (unless a compiler option such as `-c` prevents it) and understands linker options - use "`-help tool=other`" to see them. Options marked "passed to linker" are used by the compiler and the linker; options marked "for linker" are used only by the linker. When using the compiler and linker separately, you must pass the common options to both.

4.4 File Name Extensions

Files specified on the command line are identified by contents and file extension, as in the CodeWarrior IDE.

The command-line version of the CodeWarrior C/C++ compiler accepts non-standard file extensions as source code but also emits a warning message. By default, the compiler assumes that a file with any extensions besides `.c`, `.h`, `.pch` is C++ source code. The linker ignores all files that it can not identify as object code, libraries, or command files.

Linker command files must end in `.prm`. They may be simply added to the link line, for example refer the following listing.

Listing: Example of using linker command files

```
mwldtarget file.o lib.a commandfile.prm
```

NOTE

For more information on linker command files, refer to the *CodeWarrior Development Studio for Microcontrollers V10.x Targeting Manual* for your platform.

Chapter 5
Command-Line Options for Standard C Conformance

The chapter lists command-line options for standard C conformance.

5.1 -ansi

Controls the ISO/IEC 9899-1990 ("C90") conformance options, overriding the given settings.

Syntax

```
-ansi keyword
```

The arguments for `keyword` are:

`off`

Turns ISO conformance off. Same as

```
-stdkeywords off -enum min -strict off
```

`on | relaxed`

Turns ISO conformance on in relaxed mode. Same as

```
-stdkeywords on -enum min -strict on
```

`strict`

Turns ISO conformance on in strict mode. Same as

```
-stdkeywords on -enum int -strict on
```

5.2 -stdkeywords

Controls the use of ISO/IEC 9899-1990 ("C90") keywords.

Syntax

```
-stdkeywords on | off
```

Remarks

Default setting is `off`.

5.3 -strict

Controls the use of non-standard ISO/IEC 9899-1990 ("C90") language features.

Syntax

```
-strict on | off
```

Remarks

If this option is `on`, the compiler generates an error message when it encounters some CodeWarrior extensions to the C language defined by the ISO/IEC 9899-1990 ("C90") standard:

- C++-style comments
- unnamed arguments in function definitions
- non-standard keywords

The default setting is `off`.

Chapter 6
Command-Line Options for Standard C++ Conformance

The chapter lists command-line options for standard C++ conformance.

6.1 -ARM

Deprecated. Use -for_scoping instead.

6.2 -bool

Controls the use of `true` and `false` keywords for the C++ `bool` data type.

Syntax

```
-bool on | off
```

Remarks

When `on`, the compiler recognizes the `true` and `false` keywords in expressions of type `bool`. When off, the compiler does not recognizes the keywords, forcing the source code to provide definitions for these names. The default is `on`.

6.3 -Cpp_exceptions

Controls the use of C++ exceptions.

Syntax

```
-cpp_exceptions on | off
```

Remarks

When `on`, the compiler recognizes the `try`, `catch`, and `throw` keywords and generates extra executable code and data to handle exception throwing and catching.

NOTE

The S12Z C++ does not support exceptions. The option is automatically set to `OFF` and cannot be activated.

6.4 -dialect

Specifies the source language.

Syntax

```
-dialect keyword
```

```
-lang keyword
```

The arguments for *keyword* are:

`c`

Expect source code to conform to the language specified by the ISO/IEC 9899-1990 ("C90") standard.

`c99`

Expect source code to conform to the language specified by the ISO/IEC 9899-1999 ("C99") standard.

`c++ | cplus`

Always treat source as the C++ language.

`ec++`

Generate error messages for use of C++ features outside the Embedded C++ subset. Implies `-dialect cplus`.

`objc`

Always treat source as the Objective-C language.

6.5 -for_scoping

Controls legacy scope behavior in for loops.

Syntax

```
-for_scoping
```

Remarks

When enabled, variables declared in `for` loops are visible to the enclosing scope; when disabled, such variables are scoped to the loop only. The default is `off`.

6.6 -instmgr

Controls whether the instance manager for templates is active.

Syntax

```
-inst[mgr] keyword [,...]
```

The options for *keyword* are:

`off`

Turns off the C++ instance manager. This is the default.

`on`

Turns on the C++ instance manager.

`file=`*`path`*

Specify the path to the database used for the C++ instance manager. Unless specified the default database is `cwinst.db`.

Remarks

This command is global. The default setting is `off`.

6.7 -iso_templates

Controls whether the ISO/IEC 14882-2003 standard C++ template parser is active.

Syntax

```
-iso_templates on | off
```

Remarks

Default setting is `on`.

6.8 -RTTI

Controls the availability of runtime type information (RTTI).

Syntax

```
-RTTI on | off
```

Remarks

Default setting is `on`.

6.9 -som

Obsolete. This option is no longer available.

6.10 -som_env_check

Obsolete. This option is no longer available.

6.11 -wchar_t

Controls the use of the `wchar_t` data type in C++ source code.

Syntax

```
-wchar_t on | off
```

Remarks

The `-wchar on` option tells the C++ compiler to recognize the `wchar_t` type as a built-in type for wide characters. The `-wchar off` option tells the compiler not to allow this built-in type, forcing the user to provide a definition for this type. Default setting is `on`.

Chapter 7
Command-Line Options for Language Translation

The chapter lists command-line options for language translation.

7.1 -char

Controls the default sign of the `char` data type.

Syntax

```
-char keyword
```

The arguments for *keyword* are:

`signed`

`char` data items are signed.

`unsigned`

`char` data items are unsigned.

Remarks

The default is `unsigned`.

7.2 -encoding

Specifies the default source encoding used by the compiler.

Syntax

```
-enc[oding] keyword
```

The options for *keyword* are:

`ascii`

American Standard Code for Information Interchange (ASCII) format. This is the default.

`autodetect | multibyte | mb`

Scan file for multibyet encoding.

`system`

Uses local system format.

`UTF[8 | -8]`

Unicode Transformation Format (UTF).

`SJIS | Shift-JIS | ShiftJIS`

Shift Japanese Industrial Standard (Shift-JIS) format.

`EUC[JP | -JP]`

Japanese Extended UNIX Code (EUCJP) format.

`ISO[2022JP | -2022-JP]`

International Organization of Standards (ISO) Japanese format.

Remarks

The compiler automatically detects UTF-8 (Unicode Transformation Format) header or UCS-2/UCS-4 (Uniform Communications Standard) encodings regardless of setting. The default setting is `ascii`.

7.3 -flag

Specifies compiler `#pragma` as either `on` or `off`.

Syntax

```
-fl[ag] [no-]pragma
```

Remarks

For example, this option setting

```
-flag require_prototypes
```

is equivalent to

```
#pragma require_prototypes on
```

This option setting

```
-flag no-require_prototypes
```

is the same as

```
#pragma require_prototypes off
```

7.4 -mapcr

Swaps the values of the `\n` and `\r` escape characters.

Syntax

```
-mapcr
-nomapcr
```

Remarks

The `-mapcr` option tells the compiler to treat the `\n` character as ASCII 13 and the `\r` character as ASCII 10. The `-nomapcr` option tells the compiler to treat these characters as ASCII 10 and 13, respectively.

(for Macintosh MPW compatability)

7.5 -pragma

Defines a pragma for the compiler.

Syntax

```
-pragma "name [setting]"
```

The arguments are:

`name`

Name of the pragma.

`setting`

Arguments to give to the pragma

Remarks

For example, this command-line option

```
-pragma "c99 on"
```

is equivalent to inserting this directive in source code

```
#pragma c99 on
```

7.6 -requireprotos

Controls whether or not the compiler should expect function prototypes.

Syntax

```
-r[equireprotos]
```

7.7 -trigraphs

Controls the use of trigraph sequences specified by the ISO/IEC standards for C and C++.

Syntax

```
-trigraphs on | off
```

Remarks

Default setting is `off`.

7.8 -nolonglong

Disables the 'long long' support.

Syntax

```
-nolonglong
```

Chapter 8
Command-Line Options for Diagnostic Messages

The chapter lists command-line options for diagnostic messages.

8.1 -help

Lists descriptions of the CodeWarrior tool's command-line options.

Syntax

```
-help [keyword [,...]]
```

The options for *keyword* are:

`all`

Show all standard options

`group=`*`keyword`*

Show help for groups whose names contain *keyword* (case-sensitive).

`[no]compatible`

Use `compatible` to show options compatible with this compiler. Use `nocompatible` to show options that do not work with this compiler.

`[no]deprecated`

Shows deprecated options

`[no]ignored`

Shows ignored options

`[no]meaningless`

Shows options meaningless for this target

`[no]normal`

Shows only standard options

`[no]obsolete`

Shows obsolete options

`[no]spaces`

Inserts blank lines between options in printout.

`opt[ion]=`*`name`*

Shows help for a given option; for *name*, maximum length 63 chars

`search=`*`keyword`*

Shows help for an option whose name or help contains *keyword* (case-sensitive), maximum length 63 chars

`tool=keyword[ all | this | other | skipped | both ]`

Categorizes groups of options by tool; default.

- `all`-show all options available in this tool
- `this`-show options executed by this tool; default
- `other | skipped`-show options passed to another tool
- `both`-show options used in all tools

`usage`

Displays usage information.

8.2 -maxerrors

Specifies the maximum number of errors messages to show.

Syntax

```
-maxerrors max
```

`max`

Use `max` to specify the number of error messages. Common values are:

- `0` (zero) - disable maximum count, show all error messages. This is default.

8.3 -maxwarnings

Specify the maximum number of warning messages to show.

Syntax

```
-maxwarnings max
```

`max`

Use `max` to specify the number of warning messages. Common value is:

- `0` (zero) - disable maximum count, show all warning messages. This is default.

8.4 -msgstyle

Controls the style used to show error and warning messages.

Syntax

```
-msgstyle keyword
```

The options for *keyword* are:

`gcc`

Uses the message style that the GNU Compiler Collection tools use.

`ide`

Uses CodeWarrior's Integrated Development Environment (IDE) message style.

`mpw`

Uses Macintosh Programmer's Workshop (MPW®) message style.

`parseable`

Uses context-free machine parseable message style.

`std`

Uses standard message style. This is the default.

`enterpriseIDE`

Uses Enterprise-IDE message style.

8.5 -progress

Shows progress and version information.

Syntax

```
-progress
```

8.6 -stderr

Uses the standard error stream to report error and warning messages.

Syntax

```
-stderr
```

```
-nostderr
```

Remarks

The `-stderr` option specifies to the compiler, and other tools that it invokes, that error and warning messages should be sent to the standard error stream.

The `-nostderr` option specifies that error and warning messages should be sent to the standard output stream.

8.7 -verbose

Tells the compiler to provide extra, cumulative information in messages.

Syntax

```
-v[erbose]
```

Remarks

This option also gives progress and version information.

8.8 -version

Displays version, configuration, and build data.

Syntax

```
-v[ersion]
```

8.9 -timing

Shows the amount of time that the tool used to perform an action.

Syntax

```
-timing
```

8.10 -warnings

Specifies which warning messages the command-line tool issues. This command is global.

Syntax

```
-w[arning] keyword [,...]
```

The options for `keyword` are:

`off`

Turns off all warning messages. Passed to all tools. Equivalent to

```
#pragma warning off
```

`on`

Turns on most warning messages. Passed to all tools. Equivalent to

```
#pragma warning on
```

`[no]cmdline`

Passed to all tools.

`[no]err[or] | [no]iserr[or]`

Treats warnings as errors. Passed to all tools. Equivalent to

```
#pragma warning_errors
```

`all`

Turns on all warning messages and require prototypes.

`[no]pragmas | [no]illpragmas`

Issues warning messages on invalid pragmas. Equivalent to

```
#pragma warn_illpragma
```

`[no]empty[decl]`

Issues warning messages on empty declarations. Equivalent to

```
#pragma warn_emptydecl
```

`[no]possible | [no]unwanted`

Issues warning messages on possible unwanted effects. Equivalent to

```
#pragma warn_possunwanted
```

`[no]unusedarg`

Issues warning messages on unused arguments. Equivalent to

```
#pragma warn_unusedarg
```

`[no]unusedvar`

Issues warning messages on unused variables. Equivalent to

```
#pragma warn_unusedvar
```

`[no]unused`

Same as

```
-w [no]unusedarg,[no]unusedvar
```

`[no]extracomma | [no]comma`

Issues warning messages on extra commas in enumerations. The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 ("C99") standard. Equivalent to

```
#pragma warn_extracomma
```

`[no]pedantic | [no]extended`

Pedantic error checking.

`[no]hidevirtual | [no]hidden[virtual]`

Issues warning messages on hidden virtual functions. Equivalent to

```
#pragma warn_hidevirtual
```

`[no]implicit[conv]`

Issues warning messages on implicit arithmetic conversions. Implies

```
-warn impl_float2int,impl_signedunsigned
```

`[no]impl_int2float`

Issues warning messages on implicit integral to floating conversions. Equivalent to

```
#pragma warn_impl_i2f_conv
```

`[no]impl_float2int`

Issues warning messages on implicit floating to integral conversions. Equivalent to

```
#pragma warn_impl_f2i_conv
```

`[no]impl_signedunsigned`

Issues warning messages on implicit signed/unsigned conversions.

`[no]relax_i2i_conv`

Issues warnings for implicit integer to integer arithmetic conversions (off for full, on otherwise).

`[no]notinlined`

Issues warning messages for functions declared with the `inline` qualifier that are not inlined. Equivalent to

```
#pragma warn_notinlined
```

`[no]largeargs`

Issues warning messages when passing large arguments to unprototyped functions. Equivalent to

```
#pragma warn_largeargs
```

`[no]structclass`

Issues warning messages on inconsistent use of `class` and `struct`. Equivalent to

```
#pragma warn_structclass
```

`[no]padding`

Issue warning messages when padding is added between `struct` members. Equivalent to

```
#pragma warn_padding
```

`[no]notused`

Issues warning messages when the result of non-void-returning functions are not used. Equivalent to

```
#pragma warn_resultnotused
```

`[no]missingreturn`

Issues warning messages when a return without a value in non-void-returning function occurs. Equivalent to

```
#pragma warn_missingreturn
```

`[no]unusedexpr`

Issues warning messages when encountering the use of expressions as statements without side effects. Equivalent to

```
#pragma warn_no_side_effect
```

`[no]ptrintconv`

Issues warning messages when lossy conversions occur from pointers to integers.

`[no]anyptrintconv`

Issues warning messages on any conversion of pointers to integers. Equivalent to

```
#pragma warn_ptr_int_conv
```

`[no]undef[macro]`

Issues warning messages on the use of undefined macros in `#if` and `#elif` conditionals. Equivalent to

```
#pragma warn_undefmacro
```

`[no]filecaps`

Issues warning messages when `# include ""` directives use incorrect capitalization. Equivalent to

```
#pragma warn_filenamecaps
```

`[no]sysfilecaps`

Issues warning messages when `# include <>` statements use incorrect capitalization. Equivalent to

```
#pragma warn_filenamecaps_system
```

`[no]tokenpasting`

Issues warning messages when token is not formed by the `##` preprocessor operator. Equivalent to

```
#pragma warn_illtokenpasting
```

`[no]alias_ptr_conv`

Generates the warnings for potentially dangerous pointer casts.

`display | dump`

Displays list of active warnings.

8.11 -wraplines

Controls the word wrapping of messages.

Syntax

```
-wraplines
```

```
-nowraplines
```

Chapter 9
Command-Line Options for Preprocessing

The chapter lists command-line options for preprocessing.

9.1 -convertpaths

Instructs the compiler to interpret `# include` file paths specified for a foreign operating system. This command is global.

Syntax

```
-[no]convertpaths
```

Remarks

The CodeWarrior compiler can interpret file paths from several different operating systems. Each operating system uses unique characters as path separators. These separators include:

- Mac OS® - colon " `:`" (`:sys:stat.h`)
- UNIX - forward slash " `/`" (`sys/stat.h`)
- Windows® operating systems - backward slash " `\`" (`sys\stat.h`)

When `convertpaths` is enabled, the compiler can correctly interpret and use paths like `<sys/stat.h>` or `<:sys:stat.h>`. However, when enabled, (`/`) and (`:`) separate directories and cannot be used in filenames.

NOTE

> This is not a problem on Windows systems since these characters are already disallowed in file names. It is safe to leave this option on.

When `noconvertpaths` is enabled, the compiler can only interpret paths that use the Windows form, like `<\sys\stat.h>`.

9.2 -cwd

Controls where a search begins for `# include` files.

Syntax

```
-cwd keyword
```

The options for *keyword* are:

`explicit`

No implicit directory. Search `-I` or `-ir` paths.

`include`

Begins searching in directory of referencing file.

`proj`

Begins searching in current working directory (default).

`source`

Begins searching in directory that contains the source file.

Remarks

The path represented by *keyword* is searched before searching access paths defined for the build target.

9.3 -D+

Same as the `-define` option.

Syntax

```
-D+name
```

The parameters are:

`name`

The symbol name to define. Symbol is set to 1.

9.4 -define

Defines a preprocessor symbol.

Syntax

```
-d[efine]name[=value]
```

The parameters are:

`name`

The symbol name to define.

`value`

The value to assign to symbol name. If no value is specified, set symbol value equal to 1.

9.5 -E

Tells the command-line tool to preprocess source files.

Syntax

```
-E
```

Remarks

This option is global and case sensitive.

9.6 -EP

Tells the command-line tool to preprocess source files that are stripped of `#line` directives.

Syntax

```
-EP
```

Remarks

This option is global and case sensitive.

9.7 -gccincludes

Controls the compilers use of GCC `#include` semantics.

Syntax

```
-gccinc[ludes]
```

Remarks

Use `-gccincludes` to control the CodeWarrior compiler understanding of GNU Compiler Collection (GCC) semantics. When enabled, the semantics include:

- Adds `-I-` paths to the systems list if `-I-` is not already specified
- Search referencing file's directory first for `#include` files (same as `-cwd include`) The compiler and IDE only search access paths, and do not take the currently `#include` file into account.

This command is global.

9.8 -gccdepends

Writes dependency file.

Syntax

```
-[no]gccdep[ends]
```

Remarks

When set, writes the dependency file (`-MD`, `-MMD`) with name and location based on output file (compatible with gcc 3.x); else base filename on the source file and write to the current directory (legacy MW behavior).

This command is global.

9.9 -I-

Changes the build target's search order of access paths to start with the system paths list.

Syntax

```
-I-
-i-
```

Remarks

The compiler can search `#include` files in several different ways. Implies `-cwdexplicit`. Use `-I-` to set the search order as follows:

- For include statements of the form `#include "xyz"`, the compiler first searches user paths, then the system paths
- For include statements of the form `#include <xyz>`, the compiler searches only system paths

This command is global.

9.10 -I+

Appends a non-recursive access path to the current `#include` list.

Syntax

```
-I+path
```

```
-i path
```

The parameters are:

`path`

The non-recursive access path to append.

Remarks

This command is global and case-sensitive.

9.11 -include

Defines the name of the text file or precompiled header file to add to every source file processed.

Syntax

```
-include file
```

file

Name of text file or precompiled header file to prefix to all source files.

Remarks

With the command line tool, you can add multiple prefix files all of which are included in a meta-prefix file.

9.12 -ir

Appends a recursive access path to the current `#include` list. This command is global.

Syntax

```
-ir path
```

path

The recursive access path to append.

9.13 -P

Preprocesses the source files without generating object code, and send output to file.

Syntax

```
-P
```

Remarks

This option is global and case-sensitive.

9.14 -precompile

Precompiles a header file from selected source files.

Syntax

```
-precompile file | dir | ""
```

file

If specified, the precompiled header name.

`dir`

If specified, the directory to store the header file.

`""`

If "" is specified, write header file to location specified in source code. If neither argument is specified, the header file name is derived from the source file name.

Remarks

The driver determines whether to precompile a file based on its extension. The option

```
-precompile filesource
```

is equivalent to

```
-c -o filesource
```

9.15 -preprocess

Preprocesses the source files. This command is global.

Syntax

```
-preprocess
```

9.16 -ppopt

Specifies options affecting the preprocessed output.

Syntax

```
-ppopt keyword [,...]
```

The arguments for *keyword* are:

`[no]break`

Emits file and line breaks. This is the default.

`[no]line`

Controls whether #line directives are emitted or just comments. The default is `line`.

`[no]full[path]`

Controls whether full paths are emitted or just the base filename. The default is `fullpath`.

`[no]pragma`

Controls whether #pragma directives are kept or stripped. The default is `pragma`.

`[no]comment`

Controls whether comments are kept or stripped.

`[no]space`

Controls whether whitespace is kept or stripped. The default is `space`.

Remarks

The default settings is `break`.

9.17 -prefix

Adds contents of a text file or precompiled header as a prefix to all source files.

Syntax

```
-prefix file
```

9.18 -noprecompile

Do not precompile any source files based upon the filename extension.

Syntax

```
-noprecompile
```

9.19 -nosyspath

Performs a search of both the user and system paths, treating `#include` statements of the form `#include <xyz>` the same as the form `#include "xyz"`.

Syntax

```
-nosyspath
```

Remarks

This command is global.

9.20 -stdinc

Uses standard system include paths as specified by the environment variable `%MWCIncludes%`.

Syntax

```
-stdinc
```

```
-nostdinc
```

Remarks

Add this option after all system `-I` paths.

9.21 -U+

Same as the `-undefine` option.

Syntax

```
-U+name
```

9.22 -undefine

Undefines the specified symbol name.

Syntax

```
-u[ndefine] name
```

```
-U+name
```

```
name
```

The symbol name to undefine.

Remarks

This option is case-sensitive.

9.23 -allow_macro_redefs

Allows macro redefinitions without an error or warning.

Syntax

```
-allow_macro_redefs
```

Chapter 10
Command-Line Options for Object Code

The chapter lists command-line options for object code.

10.1 -c

Instructs the compiler to compile without invoking the linker to link the object code.

Syntax

```
-c
```

Remarks

This option is global.

10.2 -codegen

Instructs the compiler to compile without generating object code.

Syntax

```
-codegen
```

```
-nocodegen
```

Remarks

This option is global.

10.3 -enum

Specifies the default size for enumeration types.

Syntax

```
-enum keyword
```

The arguments for *keyword* are:

`int`

Uses `int` size for enumerated types.

`min`

Uses minimum size for enumerated types. This is the default.

10.4 -min_enum_size

Specifies the size, in bytes, of enumerated types.

Syntax

```
-min_enum_size 1 | 2 | 4
```

Remarks

Specifying this option also invokes the `-enum min` option by default.

10.5 -ext

Specifies which file name extension to apply to object files.

Syntax

```
-ext extension
```

`extension`

The extension to apply to object files. Use these rules to specify the extension:

- Limited to a maximum length of 14 characters

- Extensions specified without a leading period replace the source file's extension. For example, if *extension* is " `o`" (without quotes), then `source.cpp` becomes `source.o`.
- Extensions specified with a leading period (`.`*extension*) are appended to the object files name. For example, if `extension` is " `.o`" (without quotes), then `source.cpp` becomes `source.cpp.o`.

Remarks

This command is global. The default is none.

Chapter 11
Command-Line Options for Optimization

The chapter lists command-line options for optimization.

11.1 -inline

Specifies inline options. Default settings are `smart`, `noauto`.

Syntax

```
-inline keyword
```

The options for *keyword* are:

`off | none`

Turns off inlining.

`on | smart`

Turns on inlining for functions declared with the `inline` qualifier. This is the default.

`auto`

Attempts to inline small functions even if they are declared with `inline`.

`noauto`

Does not auto-inline. This is the default auto-inline setting.

`deferred`

Refrains from inlining until a file has been translated. This allows inlining of functions in both directions.

`level=n`

Inlines functions up to *n* levels deep. Level 0 is the same as `-inline on`. For `n`, enter 1 to 8 levels. This argument is case-sensitive.

`all`

Turns on aggressive inlining. This option is the same as `-inlineon`, `-inlineauto`.

`[no]bottomup`

Inlines bottom-up, starting from the leaves of the call graph rather than the top-level function. This is default.

11.2 -O

Sets optimization settings to `-opt level=2`.

Syntax

```
-O
```

Remarks

Provided for backwards compatibility.

11.3 -O+

Controls optimization settings.

Syntax

```
-O+keyword [,...]
```

The *keyword* arguments are:

`0`

Equivalent to `-opt off`.

`1`

Equivalent to `-opt level=1`.

`2`

Equivalent to `-opt level=2`.

`3`

Equivalent to `-opt level=3`.

`p`

Equivalent to `-opt speed`.

`s`

Equivalent to `-opt space`.

Remarks

Options can be combined into a single command. Command is case-sensitive.

11.4 -opt

Specifies code optimization options to apply to object code.

Remarks

```
-optkeyword [,...]
```

The *keyword* arguments are:

`off | none`

Suppresses all optimizations. This is the default.

`on`

Same as `-opt level=2`

`all | full`

Same as `-opt speed,level=2`

`l[evel]=num`

Sets a specific optimization level. The options for *num* are:

- `0` - Suppress all optimizations
- `1` - High Level Optimizations, Branch Optimization
- `2` - Constant Propagation, Copy Propagation, Dead Code Elimination, Register Allocation, Peephole, Common Subexpression Elimination, Branch Tail Merge
- `3` - Constant Propagation, Copy Propagation, Dead Code Elimination, Register Allocation, Peephole, Common Subexpression Elimination, Branch Tail Merge

For `num` options 0 through 3 inclusive, the default is 0.

`[no]space`

Optimizes object code for size. Equivalent to `#pragma optimize_for_size on`.

`[no]speed`

Optimizes object code for speed. Equivalent to `#pragma optimize_for_size off`.

`[no]cse | [no]commonsubs`

Common subexpression elimination. Equivalent to `#pragma opt_common_subs`.

`[no]deadcode`

Removes dead code. Equivalent to `#pragma opt_dead_code`.

`[no]deadstore`

Removes dead assignments. Equivalent to `#pragma opt_dead_assignments`.

`[no]lifetimes`

Computes variable lifetimes. Equivalent to `#pragma opt_lifetimes`.

`[no]loop[invariants]`

Removes loop invariants. Equivalent to `#pragma opt_loop_invariants`.

`[no]prop[agation]`

Propagation of constant and copy assignments. Equivalent to `#pragma opt_propagation`.

`[no]strength`

Strength reduction. Reducing multiplication by an array index variable to addition. Equivalent to `#pragma opt_strength_reduction`.

`[no]dead`

Same as `-opt [no]deadcode` and `[no]deadstore`. Equivalent to `#pragma opt_dead_code on|off` and `#pragma opt_dead_assignments`.

`display | dump`

Displays complete list of active optimizations.

Chapter 12
S12Z Command-Line Options

This chapter describes how to use the command-line tools to generate, examine, and manage the source and object code for the S12Z processors.

The command line compiler is `mwccs12lisa.exe`. This tool translates the C and C++ source code to a S12Z object code using the `ELF32` format.

- Diagnostic Command-Line Options
- Code Generation Command-Line Options
- Bit Field Command-Line Options
- Configuration Command-Line Options

12.1 Diagnostic Command-Line Options

This topic lists the following diagnostic command-line options:

- -g
- -sym

12.1.1 -g

Generates the debugging information.

Syntax

```
-g
```

Remarks

This option is global and case-sensitive. It is equivalent to

```
-sym full
```

12.1.2 -sym

Specifies the global debugging options.

Syntax

```
-sym keyword[,...]
```

The options for the *keyword* are:

`off`

Does not generate the debugging information. This option is the *default*.

`on`

Generates the debugging information.

`full[path]`

Stores the absolute paths of the source files instead of the relative paths.

12.2 Code Generation Command-Line Options

This section lists the following code generation command-line options:

- -model
- -peephole
- -coloring

12.2.1 -model

Sets the memory model for the application.

Syntax

```
-model SMALL | MEDIUM | LARGE
```

Remarks

With the SMALL memory model, the compiler generates 14-bit accesses to global non-constant data. With the MEDIUM memory model, it generates 18-bit accesses. With the LARGE memory model, it generates 24-bit accesses. For more information, refer to the topic Memory Models.

The default is `MEDIUM`.

12.2.2 -peephole

Uses the peephole optimization.

Syntax

```
-[no]peephole
```

Remarks

This is default.

12.2.3 -coloring

Uses register coloring for the locals.

Syntax

```
-[no]coloring
```

Remarks

This is default.

12.3 Bit Field Command-Line Options

This section lists the following bit field command-line options:

- -bfield_gap_limit
- -[no]bfield_lsbit_first
- -[no]bfield_reduce_type

12.3.1 -bfield_gap_limit

Controls the maximum number of gap bits allowed.

Syntax

```
-bfield_gap_limit <number>
```

where,

`<number>` is the maximum number of bits in the gap

Default

```
0
```

Remarks

When performing bitfield allocation, the compiler tries to avoid crossing a byte boundary whenever possible. In doing so, it may insert some padding(gap) bits.

This command-line is equivalent to the pragma #pragma bfield_gap_limit <number>

The following example demonstrates how this option works on bitfield allocation. Suppose the specified maximum gap is 2, that is,

```
-bfield_gap_limit 2
```

Listing: Example -bfield_gap_limit

```
typedef struct
{

        unsigned char a: 7;

        unsigned char b: 5;

        unsigned char c: 4;

} SomeStruct;
```

The compiler allocates struct B with 3 bytes. First, the compiler allocates the 7 bits of 'a'. Then the compiler tries to allocate the 5 bits of 'b', but this would cross a byte boundary. Because the gap of 1 bit is smaller than the specified gap of 2 bits, 'b' should be allocated in the next byte. However, if the compiler uses byte as the allocation unit and 'b' is allocated in the next byte, there would be 3 bits left after its allocation.

Since the maximum gap has been set to 2, and the required gap (if 'c' is to be allocated in the next byte) would be 3, the compiler will in fact use a 16-bit word as the allocation unit. Both 'b' and 'c' will be allocated within this word.

Assuming we initialize an instance of `SomeStruct` as below:

```
SomeStruct s = {2, 7, 5},
```

we get the following memory layouts for `s`:

```
-bfield_gap_limit 0 : 53 82

-bfield_gap_limit 2 : 02 00 A7

-bfield_gap_limit 3 : 02 07 05
```

NOTE

The option can also be used to force word as the allocation unit for bitfields. To achieve this, either enable the option with argument `0xFF` (or any other value that would represent a negative number on 8-bit precision) or use the equivalent pragma with a negative argument. This behavior is supported for the sake of backward compatibility with older S12 versions.

The following example shows how a negative gap limit works.

Listing: Example - Negative Gap Limit

```
typedef struct {
        unsigned char b0: 1;

        unsigned char b1: 1;

        unsigned char b2: 1;

        unsigned char b3: 1;

        unsigned char b4: 1;

        unsigned char b5: 1;

        unsigned char b6: 1;

        unsigned char b7: 1;

        unsigned char b8: 1;

        unsigned char b9: 1;

        unsigned char b10: 1;

        unsigned char b11: 1;

        unsigned char b12: 1;

        unsigned char b13: 1;

        unsigned char b14: 1;

        unsigned char b15: 1;

} SomeStruct;
```

Assuming we initialize an instance of SomeOtherStruct as below:

```
SomeStruct s = {1,0,1,0,0,0,0,0,1,1,1,0,0,0,0,0};
```

we get the following memory layouts for `s`:

```
-bfield_gap_limit 0  (default)  : 05 07

-bfield_gap_limit 0xFF          : 07 05
```

For more information, refer to the pragma bfield_gap_limit.

12.3.2 -[no]bfield_lsbit_first

Changes the default allocation order.

Syntax

```
-[no]bfield_lsbit_first
```

Default

```
-bfield_lsbit_first
```

Defines

```
__BITFIELD_LSBIT_FIRST__

__BITFIELD_MSBIT_FIRST__
```

Remarks

By default, during bitfield allocation, bits are allocated from the least significant to the most significant one, in the order of declaration (that is, the first declared bitfield will be leftmost in the allocation unit).

This option is equivalent to the pragma,

#pragma bfield_lsbit_first on | off | reset

NOTE

Allocation units (either bytes or words) are always allocated in memory from the smallest to the highest address (that is, the first allocation unit will be at the smallest address in memory).

The following example demonstrates how this option works:

Listing: Example bfield_lsbit_first

```
typedef struct {
        unsigned char b0: 1;

        unsigned char b1: 1;

        unsigned char b2: 1;

        unsigned char b3: 1;

        unsigned char b4: 1;
```

```
        unsigned char b5: 1;
        unsigned char b6: 1;
        unsigned char b7: 1;
        unsigned char b8: 1;
        unsigned char b9: 1;
        unsigned char b10: 1;
        unsigned char b11: 1;
        unsigned char b12: 1;
        unsigned char b13: 1;
        unsigned char b14: 1;
        unsigned char b15: 1;
} SomeStruct;
```

Assuming we initialize an instance of `SomeStruct` as below:

```
SomeStruct s = {1,0,1,0,0,0,0,0,1,1,1,0,0,0,0,0};
```

we get the following memory layouts for `s`:

```
-bfield_lsbit_first (default)   : 05 07 (
[b7|b6|b5|b4|b3|b2|b1|b0]
[b15|b14|b13|b12|b11|b10|b9|b8] )


-nobfield_lsbit_first           : A0 E0 (
[b0|b1|b2|b3|b4|b5|b6|b7]
[b8|b9|b10|b11|b12|b13|b14|b15] )
```

For more information, refer to the pragma bfield_lsbit_first.

12.3.3 -[no]bfield_reduce_type

Controls whether or not the compiler uses type-size reduction for bitfields.

Syntax

```
-[no]bfield_reduce_type
```

Default

```
-bfield_reduce_type
```

Defines

```
__BITFIELD_TYPE_SIZE_REDUCTION__
```

```
__BITFIELD_NO_TYPE_SIZE_REDUCTION__
```

Remarks

Type-size reduction means that the compiler can reduce the type of a bitfield from int to char if that bitfield fits into a character. Thus, memory will be allocated for a byte instead of an integer.

This option is equivalent to the pragma,

#pragma bfield_reduce_type on | off | reset

The following example demonstrates how this option works.

Listing: Example bfield_reduce_type

```
typedef struct {
long a: 4;

long b: 4;

} SomeStruct;
```

Assuming we initialize an instance of `SomeStruct` as below:

```
SomeStruct s = {2,7};
```

we get the following memory layouts for `s`:

```
-bfield_reduce_type (default):   72          ( [b][a]] )

-nobfield_reduce_type         :   72 00 00    ( [b][a]]
[0|0|0|0|0|0|0|0] [0|0|0|0|0|0|0|0] )
```

For more information, refer to the pragma bfield_reduce_type.

12.4 Configuration Command-Line Options

This section lists the following configuration command-line options for S12Z compiler:

- -schar_size
- -uchar_size
- -short_size
- -int_size
- -long_size
- -llong_size
- -float_size
- -double_size

- -ldouble_size
- -lldouble_size

12.4.1 -schar_size

Allows changing the format for signed char (by default 8-bit).

Syntax

```
-schar_size 1 | 2 | 4
```

The *arguments* are as follows:

`1`: 8-bit

`2`: 16-bit

`4`: 32-bit

Default

`1`

Defines

```
__CHAR_IS_8BIT__

__CHAR_IS_16BIT__

__CHAR_IS_32BIT__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(char) <= sizeof(short)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.4.2 -uchar_size

Allows changing the format for unsigned char (by default 8-bit).

Syntax

```
-uchar_size 1 | 2 | 4
```

The *arguments* are as follows:

`1` : 8-bit

`2` : 16-bit

`4` : 32-bit

Default

`1`

Defines

```
__CHAR_IS_8BIT__
__CHAR_IS_16BIT__
__CHAR_IS_32BIT__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(char) <= sizeof(short)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.4.3 -short_size

Allows changing the format for signed/unsigned short (by default 16-bit).

Syntax

```
-short_size 1 | 2 | 4
```

The *arguments* are as follows:

`1` : 8-bit

`2` : 16-bit

`4` : 32-bit

Default

`2`

Defines

```
__SHORT_IS_8BIT__
__SHORT_IS_16BIT__
__SHORT_IS_32BIT__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(char) <= sizeof(short) <= sizeof(int)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.4.4 -int_size

Allows changing the format for signed/unsigned int (by default 16-bit).

Syntax

```
-int_size 1 | 2 | 4
```

The *arguments* are as follows:

`1` : 8-bit

`2` : 16-bit

`4` : 32-bit

Default

`2`

Defines

```
__INT_IS_8BIT__

__INT_IS_16BIT__

__INT_IS_32BIT__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(short) <= sizeof(int) <= sizeof(long)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.4.5 -long_size

Allows changing the format for signed/unsigned long (by default 32-bit).

Syntax

```
-long_size 1 | 2 | 4
```

The *arguments* are as follows:

`1` : 8-bit

`2` : 16-bit

`4` : 32-bit

Default

`4`

Defines

```
__LONG_IS_8BIT__
```

```
__LONG_IS_16BIT__
```

```
__LONG_IS_32BIT__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(int) <= sizeof(long) <= sizeof(long long)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.4.6 -llong_size

Allows changing the format for signed/unsigned long long (by default 32-bit).

Syntax

```
-llong_size 1 | 2 | 4
```

The *arguments* are as follows:1(8-bit)

`2` : 16-bit

`4` : 32-bit

Default

`4`

Defines

```
__LONG_LONG_IS_8BIT__
```

```
__LONG_LONG_IS_16BIT__
```

```
__LONG_LONG_IS_32BIT__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(long) <= sizeof(long long)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.4.7 -float_size

Allows changing the format for float (by default IEEE32).

Syntax

```
-float_size 4 | 8
```

The *arguments* are as follows:

`4` : IEEE32

`8` : IEEE64

Default

`4`

Defines

```
__FLOAT_IS_IEEE32__
```

```
__FLOAT_IS_IEEE64__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(float) <= sizeof(double)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.4.8 -double_size

Allows changing the format for double (by default IEEE32).

Syntax

```
-double_size 4 | 8
```

The *arguments* are as follows:

`4` : IEEE32

`8` : IEEE64

Default

`4`

Defines

```
__DOUBLE_IS_IEEE32__
```

```
__DOUBLE_IS_IEEE64__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(float) <= sizeof(double) <= sizeof(long double)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.4.9 -ldouble_size

Allows changing the format for long double (by default IEEE32).

Syntax

```
-ldouble_size 4 | 8
```

The *arguments* are as follows:

`4` : IEEE32

`8` : IEEE64

Default

`4`

Defines

```
__LONG_DOUBLE_IS_IEEE32__
```

```
__LONG_DOUBLE_IS_IEEE64__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(double) <= sizeof(long double) <= sizeof(long long double)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.4.10 -lldouble_size

Allows changing the format for long long double (by default IEEE32).

Syntax

```
-lldouble_size 4 | 8
```

The *arguments* are as follows:

`4` : IEEE32

`8` : IEEE64

Default

`4`

Defines

```
__LONG_LONG_DOUBLE_IS_IEEE32__
```

```
__LONG_LONG_DOUBLE_IS_IEEE64__
```

Remarks

For integrity and compliance to ANSI, the following must hold:

```
sizeof(long double) <= sizeof(long long double)
```

NOTE

Be careful if you change type sizes. Type sizes must be consistent over the whole application. The libraries delivered with the compiler are compiled with the standard type settings. Also, underflows or overflows might occur following type configuration, for example when you assign a value too large to an object which is now smaller.

For more information, refer to the topic Data Types.

12.5 Alignment Options

This chapter consists of the following topics:

- align_globals
- align_structs
- align_stack

12.5.1 align_globals

Enables alignment of global data.

Syntax

```
-[no]align_globals
```

Arguments

None

Default

```
-align_globals
```

Define

None

Scope

This option is enabled by default when optimizing for speed (-opt speed or -opt nospace in the compiler command line). In order to disable it, use -noalignglobals. If enabled when optimizing for size (-opt space or -opt nospeed), the option has no effect.

Description

This option enables alignment of all global data.

The S12Z compiler can optimize either for code size (default) or for execution speed. Since the S12Z CPU data bus operates on 32-bit address boundaries, one way to improve execution time is to ensure that accesses to global data are such that the corresponding data transfer need not be split into two consecutive bus accesses. This can be accomplished by aligning all 16-bit or wider global variables as follows:

- 16-bit: on a 2-byte boundary;
- 24-bit or more: on a 4-byte boundary.

12.5.2 align_structs

Enables alignment of struct members.

Syntax

```
-[no]align_structs
```

Arguments

None

Default

```
-noalign_structs
```

Define

None

Scope

You should only enable this option when optimizing for speed (-opt speed or -opt nospace in the compiler command line). If enabled when optimizing for size (-opt space or -opt nospeed), the option has no effect.

Description

This option enables alignment of struct members.

he S12Z compiler can optimize either for code size (default) or for execution speed. Since the S12Z CPU data bus operates on 32-bit address boundaries, one way to improve execution time is to ensure that accesses to struct members are such that, if the struct itself is aligned to a multiple of 4, the corresponding data transfer need not be split into two consecutive bus accesses. This can be accomplished by aligning the struct members, i.e. making sure that:

- 64-bit members are aligned to 4 (the address is a multiple of 4);
- no 16-bit member is aligned to “4 + 3” (the address is not a ‘<multiple of 4> + 3’);
- no 24-bit member is aligned to either “4 + 2” or “4 + 3” (the address is not a ‘<multiple of 4> + 2’ or a ‘<multiple of 4> + 3’).

NOTE

If this option is enabled, the offsets of the struct members might change. If your application relies on the default offset values, it will no longer work as expected.

12.5.3 align_stack

Enables stack alignment.

Syntax

```
-[no]align_stack
```

Arguments

None

Default

```
-noalign_stack
```

Define

None

Scope

This option should only be enabled when optimizing for speed (-opt speed or -opt nospace in the compiler command line). If enabled when optimizing for size (-opt space or -opt nospeed), the option has no effect.

Description

This option enables stack alignment.

The S12Z compiler can optimize either for code size (default) or for execution speed. Since the S12Z CPU data bus operates on 32-bit address boundaries, one way to improve execution time is to ensure that no stack access is such that the corresponding data transfer needs to be split into two consecutive bus accesses. This can be accomplished by aligning the stack, i.e. making sure that:

- all stack accesses to 64-bit data are aligned to 4 (the address is a multiple of 4);
- no stack access to 16-bit data is aligned to “4 + 3” (the address is not a ‘<multiple of 4> + 3’);
- no stack access to 24-bit data is aligned to either “4 + 2” or “4 + 3” (the address is not a ‘<multiple of 4> + 2’ or a ‘<multiple of 4> + 3’).

NOTE

When stack alignment is enabled, function parameters are no longer passed according to the default calling convention. Padding will be inserted, if necessary, in order to (1) align the parameters that must be passed on stack and (2) make sure that, following the function call, the stack pointer will be aligned to 4 (i.e. the stack effect is a multiple of 4). A linker warning will be reported when mixing user code and library code that was not compiled with the same stack alignment setting (on/off).

Chapter 13
C Compiler

This chapter explains the CodeWarrior implementation of the C programming language:

- Extensions to Standard C
- C99 Extensions
- GCC Extensions

13.1 Extensions to Standard C

The CodeWarrior C compiler adds extra features to the C programming language. These extensions make it easier to port source code from other compilers and offer some programming conveniences. Note that some of these extensions do not conform to the ISO/IEC 9899-1990 C standard ("C90").

- Controlling Standard C Conformance
- C++-style Comments
- Unnamed Arguments
- Extensions to the Preprocessor
- Non-Standard Keywords

13.1.1 Controlling Standard C Conformance

The compiler offers settings that verify how closely your source code conforms to the ISO/IEC 9899-1990 C standard ("C90"). Enable these settings to check for possible errors or improve source code portability.

Some source code is too difficult or time-consuming to change so that it conforms to the ISO/IEC standard. In this case, disable some or all of these settings.

The following table shows how to control the compiler's features for ISO conformance.

Table 13-1. Controlling conformance to the ISO/IEC 9899-1990 C language

To control this option from here...	use this setting
CodeWarrior IDE	**ANSI Strict** and **ANSI Keywords Only** in the **C/C++ Build > S12Z Compiler > Language** panel.
source code	`#pragma ANSI_strict` `#pragma only_std_keywords`
command line	`-ansi`

13.1.2 C++-style Comments

When ANSI strictness is off, the C compiler allows C++-style comments. The following listing shows an example.

Listing: C++ Comments

```
a = b; // This is a C++-style comment.
c = d; /* This is a regular C-style comment. */
```

13.1.3 Unnamed Arguments

When ANSI strictness is off, the C compiler allows unnamed arguments in function definitions. The following listing shows an example.

Listing: Unnamed Arguments

```
void f(int) {}  /* OK if ANSI Strict is disabled. */
void f(int i) {} /* Always OK. */
```

13.1.4 Extensions to Preprocessor

When ANSI strictness is off, the C compiler allows a `#` to prefix an item that is not a macro argument. It also allows an identifier after an `#endif` directive. The following listings shows the examples:

Listing: Using # in Macro Definitions

```
#define add1(x) #x #1

    /* OK, if ANSI_strict is disabled,but probably not what you wanted:
add1(abc) creates "abc"#1 */

#define add2(x) #x "2"

    /* Always OK: add2(abc) creates "abc2". */
```

Listing: Identifiers After #endif

```
#ifdef __CWCC__

  /* . . . */

#endif __CWCC__ /* OK if ANSI_strict is disabled. */

#ifdef __CWCC__

  /* . . . */

#endif /*__CWCC__*/ /* Always OK. */
```

13.1.5 Non-Standard Keywords

When the ANSI keywords setting is off, the C compiler recognizes non-standard keywords that extend the language. This section covers the following topics:

- Global Variable Address Modifier (@address)
- Variable Allocation using @ "SegmentName"
- interrupt Keyword
- Binary Constants (0b)
- Hexadecimal Constants ($)
- #warning Directive

13.1.5.1 Global Variable Address Modifier (@address)

Use the global variable address modifier to assign global variables to specific addresses and to access memory-mapped I/O ports. These variables, called absolute variables, have the following syntax:

```
Declaration = <TypeSpec><Declarator>[@<Address>|@"<Section>"][= <Initializer>]
```

- `<TypeSpec>` is the type specifier, for example, int, char
- `<Declarator>` is the identifier of the global object, for example, i, glob
- `<Address>` is the absolute address of the object, for example, 0xff04, 0x00+8
- `<Initializer>` is the value to which the global variable is initialized.

The frontend creates a segment for each global object specified with an absolute address. This address must not be inside any address range in the SECTIONS entries of the link parameter file, or a linker error (overlapping segments) occurs.

Listing: Using the Global Variable Address Modifier

```
int glob @0x0500 = 10; // OK, global variable "glob" is
// at 0x0500, initialized with 10
void g() @0x40c0; // error (the object is a function)
void f() {
int i @0x40cc; // error (the object is a local variable)
}
```

13.1.5.2 Variable Allocation using @ "SegmentName"

The following listing shows a method of directly allocating variables in a named segment, rather than using a #pragma.

Listing: Allocation of Variables in Named Segments

```
#pragma DATA_SEG dseg
int i@"dseg";
```

With some pragmas in a common header file and with another macro definition, you can allocate variables depending on the macro.

```
Declaration = <TypeSpec><Declarator>[@"<Section>"][=<Initializer>];
```

Variables declared and defined with the @"section" syntax behave exactly like variables declared after their respective pragmas.

- `<TypeSpec>` is the type specifier, for example, int or char
- `<Declarator>` is the identifier of your global object, for example, i, glob
- `<Section>` is the section name. Define the section name in the link parameter file as well. For example, `"MyDataSection"`.
- `<Initializer>` is the value to which the global variable is initialized.

13.1.5.3 interrupt Keyword

The `__interrupt` keyword is a synonym for interrupt, which is allowed when using the `"-ansi off"` compiler option.

Use the non-standard interrupt keyword like any other type qualifier. The keyword specifies a function as an interrupt routine. It is followed by a number specifying the entry in the interrupt vector that contains the address of the interrupt routine

Listing: Examples of the Interrupt Keyword

```
interrupt void f(); // OK
// set the entry number in the prm-file
interrupt 2 int g();
// The 2nd entry (number 2) gets the address of func g().
```

13.1.5.4 Binary Constants (0b)

It is as well possible to use the binary notation for constants instead of hexadecimal constants or normal constants. Note that binary constants are not allowed if the -Ansi: Strict ANSI compiler option is switched on. Binary constants start with the 0b prefix, followed by a sequence of zeros or ones.

Listing: Demonstration of a binary constant

```
#define myBinaryConst 0b01011

int i;

void main(void) {

i = myBinaryConst;

}
```

13.1.5.5 Hexadecimal Constants ($)

It is possible to use Hexadecimal constants inside HLI (High-Level Inline) Assembly. For example, instead of `0x1234` you can use `$1234`. This is valid only for inline assembly.

13.1.5.6 #warning Directive

The `#warning` directive is used as it is similar to the `#error` directive as the following listing shows:

Listing: #warning directive

```
#ifndef MY_MACRO
#warning "MY_MACRO set to default"
#define MY_MACRO 1234
#endif
```

13.2 C99 Extensions

The CodeWarrior C compiler accepts the enhancements to the C language specified by the ISO/IEC 9899-1999 standard, commonly referred to as "C99."

- Controlling C99 Extensions
- Trailing Commas in Enumerations
- Compound Literal Values
- Designated Initializers
- Predefined Symbol __func__
- Implicit Return From main()
- Non-constant Static Data Initialization
- Variable Argument Macros
- Extra C99 Keywords
- C++-Style Comments
- C++-Style Digraphs
- Empty Arrays in Structures
- Hexadecimal Floating-Point Constants
- Variable-Length Arrays
- Unsuffixed Decimal Literal Values
- C99 Complex Data Types

13.2.1 Controlling C99 Extensions

The following table shows how to control C99 extensions.

Table 13-2. Controlling C99 extensions to the C language

To control this option from here...	use this setting
CodeWarrior IDE	**Enable C99 Extensions** (-lang c99) in the **C/C++ Build > S12Z Compiler > Language** settings panel.
source code	`#pragma c99`
command line	`-lang c99`

13.2.2 Trailing Commas in Enumerations

When the C99 extensions setting is on, the compiler allows a comma after the final item in a list of enumerations. The following listing shows an example.

Listing: Trailing comma in enumeration example

```
enum
{
    violet,
    blue
    green,
    yellow,
    orange,
    red, /* OK: accepted if C99 extensions setting is on. */
};
```

13.2.3 Compound Literal Values

When the C99 extensions setting is on, the compiler allows literal values of structures and arrays. The following listing shows an example.

Listing: Example of a Compound Literal

```
#pragma c99 on
struct my_struct {
  int i;
  char c[2];
} my_var;
my_var = ((struct my_struct) {x + y, 'a', 0});
```

13.2.4 Designated Initializers

When the C99 extensions setting is on, the compiler allows an extended syntax for specifying which structure or array members to initialize. The following listing shows an example.

Listing: Example of Designated Initializers

```
#pragma c99 on

struct X {

    int a,b,c;

} x = { .c = 3, .a = 1, 2 };

union U {

    char a;

    long b;

} u = { .b = 1234567 };
```

```
int arr1[6] = { 1,2, [4] = 3,4 };

int arr2[6] = { 1, [1 ... 4] = 3,4 }; /* GCC only, not part of C99. */
```

13.2.5 Predefined Symbol __func__

When the C99 extensions setting is on, the compiler offers the `__func__` predefined variable. The following listing shows an example.

Listing: Predefined symbol __func__

```
void abc(void)
{
    puts(__func__); /* Output: "abc" */
}
```

13.2.6 Implicit Return from main()

When the C99 extensions setting is on, the compiler inserts the line listed below, at the end of a program's `main()` function if the function does not return a value:

```
return 0;
```

13.2.7 Non-constant Static Data Initialization

When the C99 extensions setting is on, the compiler allows static variables to be initialized with non-constant expressions.

13.2.8 Variable Argument Macros

When the C99 extensions setting is on, the compiler allows macros to have a variable number of arguments. The following listing shows an example.

Listing: Variable argument macros example

```
#define MYLOG(...) fprintf(myfile, __VA_ARGS__)
#define MYVERSION 1
#define MYNAME "SockSorter"
int main(void)
{
```

```
    MYLOG("%d %s\n", MYVERSION, MYNAME);
    /* Expands to: fprintf(myfile, "%d %s\n", 1, "SockSorter"); */
    return 0;
}
```

13.2.9 Extra C99 Keywords

When the C99 extensions setting is on, the compiler recognizes extra keywords and the language features they represent. The following table lists these keywords.

Table 13-3. Extra C99 Keywords

This keyword or combination of keywords...	represents this language feature
`_Bool`	boolean data type
`long long`	integer data type
`restrict`	type qualifier
`inline`	function qualifier
`_Complex`	complex number data type
`_Imaginary`	imaginary number data type

13.2.10 C++-Style Comments

When the C99 extensions setting is on, the compiler allows C++-style comments as well as regular C comments. A C++-style comment begins with

```
//
```

and continues till the end of a source code line.

A C-style comment begins with

```
/*
```

ends with

```
*/
```

and may span more than one line.

13.2.11 C++-Style Digraphs

When the C99 extensions setting is on, the compiler recognizes C++-style two-character combinations that represent single-character punctuation. The following table lists these digraphs.

Table 13-4. C++-Style Digraphs

This digraph	is equivalent to this character
<:	[
:>	]
<%	{
%>	}
%:	#
%:%:	##

13.2.12 Empty Arrays in Structures

When the C99 extensions setting is on, the compiler allows an empty array to be the last member in a structure definition. The following listing shows an example.

Listing: Example of an Empty Array as the Last struct Member

```
struct {
  int r;
  char arr[];
} s;
```

13.2.13 Hexadecimal Floating-Point Constants

Precise representations of constants specified in hexadecimal notation to ensure an accurate constant is generated across compilers and on different hosts. The compiler generates a warning message when the mantissa is more precise than the host floating point format. The compiler generates an error message if the exponent is too wide for the host float format.

Examples:

```
  0x2f.3a2p3
  0xEp1f
```

```
0x1.8p0L
```

The standard library supports printing values of type `float` in this format using the "`%a`" and "`%A`" specifiers.

13.2.14 Variable-Length Arrays

Variable length arrays are supported within local or function prototype scope, as required by the ISO/IEC 9899-1999 ("C99") standard. The following listing shows an example.

Listing: Example of C99 Variable Length Array usage

```
#pragma c99 on
void f(int n) {
   int arr[n];
   /* ... */
}
```

While the example shown in the following figure generates an error message.

Listing: Bad Example of C99 Variable Length Array usage

```
#pragma c99 on
int n;
int arr[n];
// ERROR: variable length array
// types can only be used in local or
// function prototype scope.
```

A variable length array cannot be used in a function template's prototype scope or in a local template `typedef`, as shown in the following listing.

Listing: Bad Example of C99 usage in Function Prototype

```
#pragma c99 on
template<typename T> int f(int n, int A[n][n]);
{
};
// ERROR: variable length arrays
// cannot be used in function template prototypes
// or local template variables
```

13.2.15 Unsuffixed Decimal Literal Values

The following listing shows an example of specifying decimal literal values without a suffix to specify the literal's type.

Listing: Examples of C99 Unsuffixed Constants

```
#pragma c99 on  // Note: ULONG_MAX == 4294967295
sizeof(4294967295)  == sizeof(long long)
sizeof(4294967295u) == sizeof(unsigned long)
#pragma c99 off
sizeof(4294967295)  == sizeof(unsigned long)
sizeof(4294967295u) == sizeof(unsigned long)
```

13.2.16 C99 Complex Data Types

The compiler supports the C99 complex and imaginary data types when the `C99 extensions` option is enabled. The following listing shows an example.

Listing: C99 Complex Data Type

```
#include <complex.h>
complex double cd = 1 + 2*I;
```

NOTE

This feature is currently not available for all targets. Use `#if __has_feature(C99_COMPLEX)` to check if this feature is available for your target.

13.3 GCC Extensions

The CodeWarrior compiler accepts many of the extensions to the C language that the GCC (GNU Compiler Collection) tools allow. Source code that uses these extensions does not conform to the ISO/IEC 9899-1990 C ("C90") standard.

- Controlling GCC Extensions
- Initializing Automatic Arrays and Structures
- The sizeof() Operator
- Statements in Expressions
- Redefining Macros
- The typeof() Operator
- Void and Function Pointer Arithmetic
- The __builtin_constant_p() Operator
- Forward Declarations of Static Arrays
- Omitted Operands in Conditional Expressions
- The __builtin_expect() Operator
- Void Return Statements
- Minimum and Maximum Operators
- Local Labels

13.3.1 Controlling GCC Extensions

The following table shows how to turn GCC extensions on or off.

Table 13-5. Controlling GCC extensions to the C language

To control this option from here...	use this setting
CodeWarrior IDE	**Enable GCC Extensions (-gccext on)** in the **C/C++ Build > S12Z Compiler > Language** settings panel.
source code	`#pragma gcc_extensions`
command line	`-gcc_extensions`

13.3.2 Initializing Automatic Arrays and Structures

When the GCC extensions setting is on, array and structure variables that are local to a function and have the automatic storage class may be initialized with values that do not need to be constant. The following listing shows an example.

Listing: Initializing Arrays and Structures with Non-constant Values

```
void f(int i)
{
    int j = i * 10; /* Always OK. */
    /* These initializations are only accepted when GCC extensions
     * are on. */
    struct { int x, y; } s = { i + 1, i + 2 };
    int a[2] = { i, i + 2 };
}
```

13.3.3 sizeof() Operator

When the GCC extensions setting is on, the `sizeof()` operator computes the size of function and void types. In both cases, the `sizeof()` operator evaluates to 1. The ISO/IEC 9899-1990 C Standard ("C90") does not specify the size of the `void` type and functions. The following listing shows an example.

Listing: Using the sizeof() Operator with void and Function Types

```
int f(int a)
{
   return a * 10;
}
void g(void)
{
    size_t voidsize = sizeof(void); /* voidsize contains 1 */
    size_t funcsize = sizeof(f); /* funcsize contains 1 */
}
```

13.3.4 Statements in Expressions

When the GCC extensions setting is on, expressions in function bodies may contain statements and definitions. To use a statement or declaration in an expression, enclose it within braces. The last item in the brace-enclosed expression gives the expression its value. The following listing shows an example.

Listing: Using Statements and Definitions in Expressions

```
#define POW2(n) ({ int i,r; for(r=1,i=n; i>0; --i) r *= 2; r;})
int main()
{
    return POW2(4);
}
```

13.3.5 Redefining Macros

When the GCC extensions setting is on, macros may be redefined with the `#define` directive without first undefining them with the `#undef` directive. The following listing shows an example.

Listing: Redefining Macro Without Undefining First

```
#define SOCK_MAXCOLOR 100
#undef SOCK_MAXCOLOR
#define SOCK_MAXCOLOR 200 /* OK: this macro is previously undefined. */
#define SOCK_MAXCOLOR 300
```

13.3.6 typeof() Operator

When the GCC extensions setting is on, the compiler recognizes the `typeof()` operator. This compile-time operator returns the type of an expression. You may use the value returned by this operator in any statement or expression where the compiler expects you to specify a type. The compiler evaluates this operator at compile time. The `__typeof()__` operator is the same as this operator. The following listing shows an example.

Listing: Using typeof() Operator

```
int *ip;
/* Variables iptr and jptr have the same type. */
typeof(ip) iptr;
int *jptr;
/* Variables i and j have the same type. */
typeof(*ip) i;
int j;
```

13.3.7 Void and Function Pointer Arithmetic

The ISO/IEC 9899-1990 C Standard does not accept arithmetic expressions that use pointers to `void` or functions. With GCC extensions on, the compiler accepts arithmetic manipulation of pointers to `void` and functions.

13.3.8 __builtin_constant_p() Operator

When the GCC extensions setting is on, the compiler recognizes the `__builtin_constant_p()` operator. This compile-time operator takes a single argument and returns 1 if the argument is a constant expression or 0 if it is not.

13.3.9 Forward Declarations of Static Arrays

When the GCC extensions setting is on, the compiler will not issue an error when you declare a static array without specifying the number of elements in the array if you later declare the array completely. The following listing shows an example.

Listing: Forward Declaration of Empty Array

```
static int a[]; /* Allowed only when GCC extensions are on. */

/* ... */

static int a[10]; /* Complete declaration. */
```

13.3.10 Omitted Operands in Conditional Expressions

When the GCC extensions setting is on, you may skip the second expression in a conditional expression. The default value for this expression is the first expression. The following listing shows an example.

Listing: Using Shorter Form of Conditional Expression

```
void f(int i, int j)
{
    int a = i ? i : j;
    int b = i ?: j; /* Equivalent to int b = i ? i : j; */
    /* Variables a and b are both assigned the same value. */
}
```

13.3.11 __builtin_expect() Operator

When the GCC extensions setting is on, the compiler recognizes the `__builtin_expect()` operator. Use this compile-time operator in an `if` or `while` statement to specify to the compiler how to generate instructions for branch prediction.

This compile-time operator takes two arguments:

- the first argument must be an integral expression
- the second argument must be a literal value

The second argument is the most likely result of the first argument. The following listing shows an example.

Listing: Example for __builtin_expect() Operator

```
void search(int *array, int size, int key)
{
    int i;
    for (i = 0; i < size; ++i)
    {
          /* We expect to find the key rarely. */
          if (__builtin_expect(array[i] == key, 0))
         {
             rescue(i);
         }
    }
}
```

13.3.12 Void Return Statements

When the GCC extensions setting is on, the compiler allows you to place expressions of type `void` in a `return` statement. The following listing shows an example.

Listing: Returning void

```
void f(int a)
{
    /* ... */
    return; /* Always OK. */
}
void g(int b)
{
    /* ... */
    return f(b); /* Allowed when GCC extensions are on. */
}
```

13.3.13 Minimum and Maximum Operators

When the GCC extensions setting is on, the compiler recognizes built-in minimum (`<?`) and maximum (`>?`) operators.

Listing: Example of Minimum and Maximum Operators

```
int a = 1 <? 2; // 1 is assigned to a.
int b = 1 >? 2; // 2 is assigned to b.
```

13.3.14 Local Labels

When the GCC extensions setting is on, the compiler allows labels limited to a block's scope. A label declared with the `__label__` keyword is visible only within the scope of its enclosing block. The following listing shows an example.

Listing: Example of Using Local Labels

```
void f(int i)
{
  if (i >= 0)
  {
    __label__ again; /* First again. */
    if (--i > 0)
      goto again; /* Jumps to first again. */
  }
  else
  {
    __label__ again; /* Second again. */
    if (++i < 0)
      goto again; /* Jumps to second again. */
```

```
    }
}
```

Chapter 14
Precompiling

Each time you invoke the CodeWarrior compiler to translate a source code file, it *preprocesses* the file to prepare its contents for translation. Preprocessing tasks include expanding macros, removing comments, and including header files. If many source code files include the same large or complicated header file, the compiler must preprocess it each time it is included. Repeatedly preprocessing this header file can take up a large portion of the time that the compiler operates.

To shorten the time spent compiling a project, CodeWarrior compilers can *precompile* a file once instead of preprocessing it every time it is included in project source files. When it precompiles a header file, the compiler converts the file's contents into internal data structures, then writes this internal data to a precompiled file. Conceptually, precompiling records the compiler's state after the preprocessing step and before the translation step of the compilation process.

This section shows you how to use and create precompiled files:

- What can be Precompiled
- Using Precompiled File
- Creating Precompiled File

14.1 What can be Precompiled

A file to be precompiled does not have to be a header file (`.h` or `.hpp` files, for example), but it must meet these requirements:

- The file must be a source code file in text format.

  You cannot precompile libraries or other binary files.

- Precompiled files must have a `.mch` filename extension.
- The file must not contain any statements that generate data or executable code.

However, the file may define static data.

- Precompiled header files for different IDE build targets are not interchangeable.

14.2 Using Precompiled File

To use a precompiled file, simply include it in your source code files like you would any other header file:

- A source file may include only one precompiled file.
- A file may not define any functions or variables before including a precompiled file.
- Typically, a source code file includes a precompiled file before anything else (except comments).

The following listing shows an example.

Listing: Using a precompiled file

```
/* sock_main.c */
#include "sock.mch" /* Precompiled header file. */
#include "wool.h /* Regular header file. */
/* ... */
```

14.3 Creating Precompiled File

This section shows how to create and manage precompiled files:

- Precompiling File on the Command Line
- Updating Precompiled File Automatically
- Preprocessor Scope in Precompiled Files

14.3.1 Precompiling File on Command Line

To precompile a file on the command line, follow these steps:

1. Start the command line shell.
2. Issue this command

```
mwcch_file.pch -precompile p_file.mch
```

where *mwcc* is the name of the CodeWarrior compiler tool, *h_file* is the name of the header to precompile, and *p_file* is the name of the resulting precompiled file.

The file is precompiled.

14.3.2 Updating Precompiled File Automatically

Use the CodeWarrior IDE's project manager to update a precompiled header automatically. The IDE creates a precompiled file from a source code file during a compile, update, or make operation if the source code file meets these criteria:

- The text file name ends with `.pch`.
- The file is in a project's build target.
- The file uses the `precompile_target` pragma.
- The file, or files it depends on, have been modified.

The IDE uses the build target's settings to preprocess and precompile files.

14.3.3 Preprocessor Scope in Precompiled Files

When precompiling a header file, the compiler preprocesses the file too. In other words, a precompiled file is preprocessed in the context of its precompilation, not in the context of its later compilation.

The preprocessor also tracks macros used to guard `#include` files to reduce parsing time. If a file's contents are surrounded with

```
 #ifndef MYHEADER_H
 #define MYHEADER_H
 /* file contents */
#endif
```

the compiler will not load the file twice, saving some time in the process.

Pragma settings inside a precompiled file affect only the source code within that file. The pragma settings for an item declared in a precompiled file (such as data or a function) are saved, then restored when the precompiled header file is included.

For example, the source code in the following listing specifies that the variable `xxx` is a `far` variable.

Listing: Pragma Settings in Precompiled Header

```
/* my_pch.pch */
/* Generate a precompiled header named pch.mch. */
#pragma precompile_target "my_pch.mch"
#pragma far_data on
extern int xxx;
```

The source code in the following listing includes the precompiled version of the listing displayed above.

Listing: Pragma Settings in Included Precompiled File

```
/* test.c */
/* Far data is disabled. */
#pragma far_data off
/* This precompiled file sets far_data on. */
#include "my_pch.mch"
/* far_data is still off but xxx is still a far variable. */
```

The pragma setting in the precompiled file is active within the precompiled file, even though the source file including the precompiled file has a different setting.

Chapter 15
C++ Compiler

This chapter explains the CodeWarrior implementation of the C++ programming language:

- C++ Compiler Performance
- Extensions to Standard C++
- Implementation-Defined Behavior
- GCC Extensions

15.1 C++ Compiler Performance

Some options affect the C++ compiler's performance. This section explains how to improve compile times when translating C++ source code:

- Precompiling C++ Source Code
- Using the Instance Manager

15.1.1 Precompiling C++ Source Code

The CodeWarrior C++ compiler has these requirements for precompiling source code:

- C source code may not include precompiled C++ header files and C++ source code may not include precompiled C header files.
- C++ source code can contain inline functions.
- C++ source code may contain constant variable declarations.
- A C++ source code file that will be automatically precompiled must have a `.pch++` file name extension.

15.1.2 Using Instance Manager

The instance manager reduces compile time by generating a single instance of some kinds of functions:

- template functions
- functions declared with the `inline` qualifier that the compiler was not able to insert in line

The instance manager reduces the size of object code and debug information but does not affect the linker's output file size, though, since the compiler is effectively doing the same task as the linker in this mode.

The following table shows how to control the C++ instance manager.

Table 15-1. Controlling C++ Instance Manager

To control this option from here...	use this setting
CodeWarrior IDE	**Use Instance Manager** (`-instmgr on`) in the **C/C++ Build > S12Z Compiler > Language** settings panel.
source code	`#pragma instmgr_file`
command line	`-instmgr`

15.2 Extensions to Standard C++

The CodeWarrior C++ compiler has features and capabilities that are not described in the ISO/IEC 14882-2003 C++ standard:

- __PRETTY_FUNCTION__ Identifier
- Standard and Non-Standard Template Parsing

15.2.1 __PRETTY_FUNCTION__ Identifier

The `__PRETTY_FUNCTION__` predefined identifier represents the qualified (unmangled) C++ name of the function being compiled.

15.2.2 Standard and Non-Standard Template Parsing

CodeWarrior C++ has options to specify how strictly template declarations and instantiations are translated. When using its strict template parser, the compiler expects the `typename` and `template` keywords to qualify names, preventing the same name in different scopes or overloaded declarations from being inadvertently used. When using its regular template parser, the compiler makes guesses about names in templates, but may guess incorrectly about which name to use.

A qualified name that refers to a type and that depends on a template parameter must begin with `typename` (ISO/IEC 14882-2003 C++, §14.6). The following listing shows an example.

Listing: Using typename Keyword

```
template <typename T> void f()
{
  T::name *ptr; // ERROR: an attempt to multiply T::name by ptr
  typename T::name *ptr; // OK
}
```

The compiler requires the `template` keyword at the end of "`.`" and "`->`" operators, and for qualified identifiers that depend on a template parameter. The following listing shows an example.

Listing: Using template Keyword

```
template <typename T> void f(T* ptr)
{
  ptr->f<int>(); // ERROR: f is less than int
  ptr->template f<int>(); // OK
}
```

Names referred to inside a template declaration that are not dependent on the template declaration (that do not rely on template arguments) must be declared before the template's declaration. These names are bound to the template declaration at the point where the template is defined. Bindings are not affected by definitions that are in scope at the point of instantiation. The following listing shows an example.

Listing: Binding Non-dependent Identifiers

```
void f(char);
template <typename T> void tmpl_func()
{
  f(1); // Uses f(char); f(int), below, is not defined yet.
  g(); // ERROR: g() is not defined yet.
}
void g();
void f(int);
```

Names of template arguments that are dependent in base classes must be explicitly qualified (ISO/IEC 14882-2003 C++, §14.6.2). See the following listing.

Listing: Qualifying Template Arguments in Base Classes

```
template <typename T> struct Base
{
  void f();
}
template <typename T> struct Derive: Base<T>
{
 void g()
  {
    f(); // ERROR: Base<T>::f() is not visible.
    Base<T>::f(); // OK
  }
}
```

When a template contains a function call in which at least one of the function's arguments is type-dependent, the compiler uses the name of the function in the context of the template definition (ISO/IEC 14882-2003 C++, §14.6.2.2) and the context of its instantiation (ISO/IEC 14882-2003 C++, §14.6.4.2). The following listing shows an example.

Listing: Function Call with Type-dependent Argument

```
void f(char);
template <typename T> void type_dep_func()
{
  f(1); // Uses f(char), above; f(int) is not declared yet.
  f(T()); // f() called with a type-dependent argument.
}
void f(int);
struct A{};
void f(A);
int main()
{
 type_dep_func<int>(); // Calls f(char) twice.
  type_dep_func<A>(); // Calls f(char) and f(A);
  return 0;
}
```

The compiler only uses external names to look up type-dependent arguments in function calls. See the following listing.

Listing: Function Call with Type-dependent Argument and External Names

```
static void f(int); // f() is internal.
template <typename T> void type_dep_fun_ext()
{
  f(T()); // f() called with a type-dependent argument.
}
int main()
{
  type_dep_fun_ext<int>(); // ERROR: f(int) must be external.
}
```

The compiler does not allow expressions in inline assembly statements that depend on template parameters. See the following listing.

Listing: Unsupported Template Parameters

```
template <typename T> void asm_tmpl()
{
```

```
  asm { move #sizeof(T), D0 ); // ERROR: Not supported.
}
```

The compiler also supports the address of template-id rules. See the following listing.

Listing: Address of Template-id Supported

```
template <typename T> void funcA(T) {}
template <typename T> void funcB(T) {}
...

funcA{ &funcB<int> );     // now accepted
```

15.3 Implementation-Defined Behavior

Annex A of the ISO/IEC 14882-2003 C++ Standard lists compiler behaviors that are beyond the scope of the standard, but which must be documented for a compiler implementation. This annex also lists minimum guidelines for these behaviors, although a conforming compiler is not required to meet these minimums.

The CodeWarrior C++ compiler has these implementation quantities listed in the following table, based on the ISO/IEC 14882-2003 C++ Standard, Annex A.

NOTE

The term *unlimited* in the table listed below, means that a behavior is limited only by the processing speed or memory capacity of the computer on which the CodeWarrior C++ compiler is running.

Table 15-2. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882-2003 C++, §A)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
Nesting levels of compound statements, iteration control structures, and selection control structures	256	Unlimited
Nesting levels of conditional inclusion	256	256
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration	256	Unlimited
Nesting levels of parenthesized expressions within a full expression	256	Unlimited
Number of initial characters in an internal identifier or macro name	1024	Unlimited
Number of initial characters in an external identifier	1024	Unlimited
External identifiers in one translation unit	65536	Unlimited

Table continues on the next page...

Table 15-2. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882-2003 C++, §A) (continued)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
Identifiers with block scope declared in one block	1024	Unlimited
Macro identifiers simultaneously defined in one translation unit	65536	Unlimited
Parameters in one function definition	256	Unlimited
Arguments in one function call	256	Unlimited
Parameters in one macro definition	256	256
Arguments in one macro invocation	256	256
Characters in one logical source line	65536	Unlimited
Characters in a character string literal or wide string literal (after concatenation)	65536	Unlimited
Size of an object	262144	2 GB
Nesting levels for `# include` files	256	256
`Case` labels for a `switch` statement (excluding those for any nested `switch` statements)	16384	Unlimited
Data members in a single class, structure, or union	16384	Unlimited
Enumeration constants in a single enumeration	4096	Unlimited
Levels of nested class, structure, or union definitions in a single struct-declaration-list	256	Unlimited
Functions registered by `atexit()`	32	64
Direct and indirect base classes	16384	Unlimited
Direct base classes for a single class	1024	Unlimited
Members declared in a single class	4096	Unlimited
Final overriding virtual functions in a class, accessible or not	16384	Unlimited
Direct and indirect virtual bases of a class	1024	Unlimited
Static members of a class	1024	Unlimited
Friend declarations in a class	4096	Unlimited
Access control declarations in a class	4096	Unlimited
Member initializers in a constructor definition	6144	Unlimited
Scope qualifications of one identifier	256	Unlimited
Nested external specifications	1024	Unlimited
Template arguments in a template declaration	1024	Unlimited
Recursively nested template instantiations	17	64 (adjustable upto 30000 using #pragma template_depth(<n>))

Table continues on the next page...

Table 15-2. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882-2003 C++, §A) (continued)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
Handlers per try block	256	Unlimited
Throw specifications on a single function declaration	256	Unlimited

15.4 GCC Extensions

The CodeWarrior C++ compiler recognizes some extensions to the ISO/IEC 14882-2003 C++ standard that are also recognized by the GCC (GNU Compiler Collection) C++ compiler.

The compiler allows the use of the :: operator, of the form *class*::*member*, in a class declaration.

Listing: Using :: Operator in Class Declarations

```
class MyClass {
   int MyClass::getval();
};
```

Chapter 16
Inline Assembler

The Inline Assembler provides means to make full use of the properties of the target processor right within a C program. There is no need to write a separate assembly file, assemble it and later link it with the rest of the application written in ANSI-C/C++ with the inline assembler. The Compiler does all that work for you. The following provides you details.

- Syntax
- Reserved Words
- Pseudo-Opcodes
- Accessing Variables
- Constant Expressions
- Addresses of Variables
- Pure Inline Assembly Functions
- Enforce Operators

16.1 Syntax

Inline assembly statements can appear anywhere a C statement can appear (an __asm statement must be inside a C function). Inline assembly statements take one of the following forms.

Listing: Inline Assembly - Version #1

```
__asm <Assembly Instruction> ; [/* Comment */]
__asm <Assembly Instruction> ; [// Comment]
```

Listing: Inline Assembly - Version #2

```
__asm ( <Assembly Instruction> ) ; [// Comment]
```

Listing: Inline Assembly - Version #3

```
__asm {
<Assembly Instruction> [; Comment] \n

..................

}
```

If you use the first form, multiple __asm statements are contained on one line and comments are delimited like regular C or C++ comments. If you use the third form, one to several assembly instructions are contained within the __asm block, but only one assembly instruction per line is possible and the semicolon starts an assembly comment.

16.2 Reserved Words

The inline assembler knows a couple of reserved words, which must not collide with user defined identifiers such as variable names. These reserved words are:

- All opcodes
- All register names

For these reserved words, the inline assembler is not case-sensitive, i.e., LD is the same as ld or even Ld. For all other identifiers (labels, variable names, and so on) the inline assembler is case-sensitive.

16.3 Pseudo-Opcodes

The inline assembler provides some pseudo opcodes to put constant bytes into the instruction stream.

```
DC.B 1 ; Byte constant 1
DC.B 0 ; Byte constant 0
DC.W 12 ; Word constant 12
DC.L 20,23 ; Longword constants
```

16.4 Accessing Variables

The inline assembler allows accessing local and global variables declared in C by using their names in the instruction.

16.5 Constant Expressions

Constant expressions may be used anywhere an IMMEDIATE value is expected. They may contain the binary operators for addition ("+"), subtraction ("-"), multiplication ("*"), and division ("/"). Also, the unary operator "-" is allowed. Round brackets may be used to force an evaluation order other than the normal one. The syntax of numbers is the same as in ANSI-C.

16.6 Addresses of Variables

A constant expression may also be the address of a global variable or the offset of a local variable.

```
AddrOfVar = "#"<Variable.
```

As examples:

```
LD X, #g ; Load X with address of global variable

LD Y, #l ; Load Y with frame offset of local variable or parameter
```

It is also possible to access the fields of a struct or a union by using the normal

```
ANSI-C notation.

LD D0 r.f ; Load D0 with content of field f.
```

The inline assembler enables you to specify an offset from the address of a variable in order to access the low word of a long or a float variable:

```
Offset = "+" ConstExpr.

Variable = Ident {"." Ident}.
```

Below are some examples (assuming all variables are long):

```
LD Y,  #g+2   ; Load Y with ((address of g) + 2)

LD X,  g+2    ; Load X with the value stored there

LD D0, r.f+2 ; Load D0 with low word of field f.
```

This feature may also be used to access array elements with a constant index:

```
int a[30]    ;

LD  D2, a+24 ; Load a[12] into D
```

16.7 Pure Inline Assembly Functions

One can define a whole function to be written in inline assembly using the syntax

```
__asm <return_type> function_name(<parameters_list>)
{
local vars list>
<Assembly Instruction>
...
}
```

In pure inline assembly functions there can be also declared and used local variables.

The declaration part is written as in C, while the usage of local variables is done in assembly instructions.

The compiler default behavior is to generate prologue and epilogue code for pure inline assembly functions, especially for handling parameters, return value and local variables.

This behavior can be altered using specific pragmas to avoid prologue, epilogue generation:

```
#pragma NO_ENTRY
#pragma NO_EXIT
#pragma NO_RETURN
```

If the function uses these pragmas, the user should be aware that the local variables declaration will be impacted (usually when using these pragmas, the stack management is the user full responsability). The pure inline assembly functions use the usual calling convention, unless the user is defining own parameter handling.

16.8 Enforce Operators

This section includes the following topics:

- Syntax
- Keyworks

- Examples
- Special Cases

16.8.1 Syntax

Listed below is the syntax for enforce operator:

```
<enforce_keyword> (<forced_operand>)
```

The following addressing modes are subject to the enforce operators:

Immediates:

```
#<enforce_keyword>(<immediate_value>)
```

Indexed addressing modes:

- Direct indexed

```
(<enforce_keyword>(<immediate_value>), REG)
```

- Indirect indexed

```
[<enforce_keyword>(<immediate_value>), REG]
```

REG is X, Y, SP, PC or Di register, according to the addressing mode

Extended addressing modes:

- direct

```
<enforce_keyword>(<var_name>)
<enforce_keyword>(<address_value>)
```

- indirect

```
[<enforce_keyword>(<var_name>)]
[<enforce_keyword>(<address_value>)]
```

NOTE

When an offset is specified for a variable, the syntax is:

```
<enforce_keyword>(<var_name> + <offset_value>)
```

Conclusion: The enforce operators apply on a variable, an address or an immediate value.

16.8.2 Keyworks

This feature adds some keywords besides those mentioned in Reserved Words chapter:

Keyword	Corresponding addressing mode
opru14	EXT1 – u14 Short Extended

Table continues on the next page...

Keyword	Corresponding addressing mode
opru18	EXT2 – u18 Extended
opr24	EXT3 – 24b Extended
opr24a	
opru4	IDX – u4 Constant offset from xys
oprs9	IDX1 – s9 Constant offset from xysp (direct or indirect)
opr24	IDX – 24b Constant offset from xysp (direct or indirect)
opr8i	IMM1 (8-bit signed offset)
opr16i	IMM2 (16-bit signed offset)
opr24i	IMM3 (24-bit signed offset)
opr32i	IMM4 (32-bit signed offset)
sxe4i	IMMe4 – Short immediate (-1, 1, 2, … 14, 15)
opr5i	5-bit immediate value
opr7i	REL – 7-bit relative offset
opr15i	REL1 – 15-bit relative offset

All keywords are case insensitive in Inline Assembler.

16.8.3 Examples

The following examples cover different addressing modes:

LD D1, **#opr8i**(0x123)	The immediate value is truncated to 8-bit number.
ADC D7 , **#sxe4i**(11)	The immediate value is forced to IMMe4 postbyte addressing mode. Another alternative, more size consuming is `ADC D32, #imm4`.
ADC D0 , (**opru4**(7), X)	Ensures that the shortest encoding is used (another alternative is `ADC D0, (oprs9, xysp))`.
ADC D0 , **opru14**(var1)	Force to 14-bit address. Useful when compiling with medium or large memory model.
ADC D0 , **opru18**(var2)	Force to 18-bit address. Useful when compiling with other memory model than medium.
ADC D0 , **opr24**(var3)	Force to 24-bit address. Useful for `const var`, when the application is compiled with small memory model.
BEQ **opr7i**(label0)	The user manually computed the value for `label0` and forced the compiler to use `opr7i`.

An instruction may have as many enforce operators as the number of operands which can accept enforce operators. Example,

```
MODS.BB D4 , (opru4(14), X) ,(opru24(20013), D0)
MOV.W #opr16i(1361) ,  (oprs9(86), Y)
```

16.8.4 Special Cases

This topic lists the following special cases for enforce operators:

1. **Constants which do not fit.** Constants which do not fit can be split in 2 case:
   - Constants which are truncated and an warning may be raised:

     Example, `ASR.B D1 , opr24(var3) ,(oprs9(0x3E8), Y)`

     The number 0x3E8 doesn't fit into 9-bit field. This value is truncated to 0xE8
   - Constants which can't be adapted to the wished addressing mode (selected by the enforce operator) and the compiler raise an error. Here we have 2 special cases:
     - sxe4

       Example, `ADD D1, #sxe4i(-7)`
     - Unsigned operators

       Example, `ADD D2, (opru4(-6), X)`
2. **Immediate forced to wrong size.** In this case the compiler exits with an error. Example,

```
LD D7, #opr8i(0x500)
```

3. **Variable address forced to unaccepted addressing mode for that instruction.** This case also ends with a compiler error.Example,

```
ADD D2, opr24a(var)
```

4. **Shift instructions – when the number of bit positions to shift the operand is forced to #sxe4i(value).** This addressing mode is accepted, but is transformed into #imm5i according to the architecture manual. Example,

```
ASL.B D3 , (-X)  ,#sxe4i(2) is transformed in the compiler into
ASL.B D3 , (-X)  ,#opr5i(2)
```

Chapter 17
Supported Intrinsic Functions

When mixing high level C/C++ code with low level code for better performance, the user can either use inline assembly code or intrinsic function calls. S12Z compiler provides 14 intrinsic that can be used by the user to instruct the compiler to use a specific machine instruction.

17.1 Functions

When mixing high level C/C++ code with low level code for better performance, the user can either use inline assembly code or intrinsic function calls. S12Z compiler provides 14 intrinsic that can be used by the user to instruct the compiler to use a specific machine instruction.

- __abs8
- __abs16
- __abs32
- __qmuls8
- __qmuls16
- __qmuls32
- __qmuls32_16_16
- __qmulu8
- __qmulu16
- __qmulu32
- __qmulu32_16_16
- __sat8
- __sat16
- __sat32

17.1.1 __abs8

Absolute value

Prototype

```
signed char __abs8(const signed char op);
```

Mapped Instruction

```
ABS D8_bit
```

17.1.2 __abs16

Absolute value

Prototype

```
signed int __abs16(const signed int op);
```

Mapped Instruction

```
ABS D16_bit
```

17.1.3 __abs32

Absolute value

Prototype

```
signed long __abs32(const signed long op);
```

Mapped Instruction

```
ABS D32_bit
```

17.1.4 __qmuls8

Fractional multiply

Prototype

```
signed char __qmuls8(const signed char op1, const signed char op2);
```

Mapped Instruction

```
QMULS.BB D8_bit, op1, op2
```

17.1.5 __qmuls16

Fractional multiply

Prototype

```
signed int __qmuls16(const signed int op1, const signed int op2);
```

Mapped Instruction

```
QMULS.WW D16_bit, op1, op2
```

17.1.6 __qmuls32

Fractional multiply

Prototype

```
signed long __qmuls32(const signed long op1, const signed long op2);
```

Mapped Instruction

```
QMULS.LL D32_bit, op1, op2
```

17.1.7 __qmuls32_16_16

Fractional multiply

Prototype

```
signed long __qmuls32_16_16(const signed int op1, const signed int op2);
```

Mapped Instruction

```
QMULS.WW D32_bit, op1, op2
```

17.1.8 __qmulu8

Fractional multiply

Prototype

```
unsigned char __qmulu8(const unsigned char op1, const unsigned char op2);
```

Mapped Instruction

```
QMULU.BB D8_bit, op1, op2
```

17.1.9 __qmulu16

Fractional multiply

Prototype

```
unsigned int __qmulu16(const unsigned int op1, const unsigned int op2);
```

Mapped Instruction

```
QMULU.WW D16_bit, op1, op2
```

17.1.10 __qmulu32

Fractional multiply

Prototype

```
unsigned long __qmulu32(const unsigned long op1, const unsigned long op2);
```

Mapped Instruction

```
QMULU.LL D32_bit, op1, op2
```

17.1.11 __qmulu32_16_16

Fractional multiply

Prototype

```
unsigned long __qmulu32_16_16(const unsigned int op1, const unsigned int op2);
```

Mapped Instruction

```
QMULU.WW D32_bit, op1, op2
```

17.1.12 __sat8

Saturate

Prototype

```
signed char __sat8(const signed char op);
```

Mapped Instruction

```
SAT D8_bit
```

17.1.13 __sat16

Saturate

Prototype

```
signed int __sat16(const signed int op);
```

Mapped Instruction

```
SAT D16_bit
```

17.1.14 __sat32

Saturate

Prototype

```
signed long __sat32(const signed long op);
```

Mapped Instruction

```
SAT D32_bit
```

Chapter 18
Addressing

This chapters covers the following topics:

- Memory Models
- Segmentation
- Data Types
- Calling Convention

18.1 Memory Models

In order to allow for efficient use of addressing modes, the S12Z compiler supports three different memory models: SMALL, MEDIUM and LARGE - with MEDIUM being the compiler default. Depending on the memory model, the compiler generates 14-bit, 18-bit or 24-bit accesses to memory. Since code and constant data reside in Flash and, for each of the available S12Z devices, Flash memory is placed at high addresses and, therefore, requires 24 bits to access, memory models only apply to global non-constant data.

NOTE

The compiler option for setting the memory model is such that, by default, the compiler uses MEDIUM. However, the actual memory model to be used with an S12Z application depends on the derivative for which that application is being developed. The memory model should be set according to the derivative memory map.

This section covers the following topics:

- SMALL Memory Model
- MEDIUM Memory Model
- LARGE Memory Model

18.1.1 SMALL Memory Model

If the current memory model is SMALL, the compiler generates 14-bit accesses to global non-constant data. All global non-constant data must fit into 16KB of RAM.

Use option `-model small` to select the SMALL memory model. For more information, refer to the topic -model.

18.1.2 MEDIUM Memory Model

If the current memory model is MEDIUM, the compiler generates 18-bit accesses to global non-constant data. All global non-constant data must fit into 256KB of RAM.

Use option `-model medium` to select the MEDIUM memory model. For more information, refer to the topic -model.

18.1.3 LARGE Memory Model

If the current memory model is LARGE, the compiler generates 24-bit accesses to global non-constant data. Use option `-model large` to select the LARGE memory model. For more information, refer to the topic -model.

18.2 Segmentation

The linker memory space may be partitioned into several segments. The compiler allows associating certain global variables or functions with a peculiar segment. Those variables/ functions will then be allocated by the linker into that segment. Where the segment actually lies is determined by a placement entry in the linker parameter file.

There are three types of segments, code segments, data segments and constant data segments, each with a matching pragma:

- `#pragma CODE_SEG`

- `#pragma DATA_SEG`
- `#pragma CONST_SEG`

Each such pragma is valid until the next pragma of the same type is encountered. If no pragma is specified, the compiler assumes the following default segments:

- `DEFAULT_ROM`, for code
- `DEFAULT_RAM`, for data
- `ROM_VAR`, for constant data

To explicitly set the current segment to the default segment, use segment name `DEFAULT`:

- `#pragma CODE_SEG DEFAULT`
- `#pragma DATA_SEG DEFAULT`
- `#pragma CONST_SEG DEFAULT`

For more information, refer to the following topics:

- CODE_SEG
- DATA_SEG
- CONST_SEG

18.3 Data Types

This section describes the set of implemented ANSI-C basic types. The S12Z compiler supports flexible type configuration, several command line options are available, that allow configuring each of these basic types:

- `schar_size`
- `uchar_size`
- `short_size`
- `int_size`
- `long_size`
- `llong_size`
- `float_size`
- `double_size`
- `ldouble_size`
- `lldouble_size`

The following data types are covered here:

- Scalar Types
- Floating Point Types

- Pointer Types
- Bitfields

18.3.1 Scalar Types

Type `char` is by default `unsigned` (that is, `char` is the same as `unsigned char`). The other scalar types are all signed by default (for example, `int` is the same as `signed int`).

The following table specifies the possible formats for each of the supported scalar types:

Table 18-1. Scalar Types

Type	Default Format	Default Value Range		Alternative Formats (flexible type configuration)
		Min	Max	
char (unsigned)	8-bit	0	255	8-, 16-, & 32-bit
signed char	8-bit	-128	127	8-, 16-, & 32-bit
unsigned char	8-bit	0	255	8-, 16-, & 32-bit
signed short	16-bit	-32,768	32,767	8-, 16-, & 32-bit
unsigned short	16-bit	0	65,535	8-, 16-, & 32-bit
enum (unsigned)	16-bit	0	65,535	8-, 16-, & 32-bit
signed int	16-bit	-32,768	32,767	8-, 16-, & 32-bit
unsigned int	16-bit	0	65,535	8-, 16-, & 32-bit
signed long	32-bit	-2,147,483,648	2,147,483,647	8-, 16-, & 32-bit
unsigned long	32-bit	0	4,294,967,295	8-, 16-, & 32-bit
signed long long	32-bit	-2,147,483,648	2,147,483,647	8-, 16-, & 32-bit
unsigned long long	32-bit	0	4,294,967,295	8-, 16-, & 32-bit

For more information, refer to the following topics:

- -schar_size
- -uchar_size
- -short_size
- -int_size
- -long_size
- -llong_size

18.3.2 Floating Point Types

The compiler supports the two IEEE standard formats (32-bit and 64-bit) for floating point types. The default format, for both float and double, is IEEE32.

The following table specifies the possible formats for each of the supported floating point types.

Table 18-2. Floating Point Types

Type	Default Format	Default Value Range		Alternative Formats (flexible type configuration)
		Min	Max	
float	IEEE32	-1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64
double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64
long double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64
long long double	IEEE32	1.17549435E-38F	3.402823466E+38F	IEEE32, IEEE64

For more information, refer to the following topics:

- -float_size
- -double_size
- -ldouble_size
- -lldouble_size

18.3.3 Pointer Types

The size of a pointer type depends on what kind of pointer it designates (constant data, non-constant data, or function) and on the memory model.

The following table shows an overview:

Table 18-3. Pointer Types

Type	Example	Memory Model		
		SMALL	MEDIUM	LARGE
pointer to non-constant data	int *	14	18	24
pointer to constant data	const int *	24	24	24
pointer to function	void (*) (int)	24	24	24

18.3.4 Bitfields

The maximum width of bitfields is 32 bits. The default allocation unit is a byte. The compiler uses words only if a bitfield is wider than 8 bits, or if using bytes would cause a gap bigger than the limit specified by the `-bfield_gap_limit` option. The default allocation order within the allocation unit is from the least significant bit up to the most significant one in the order of bitfield declaration (that is, the first bitfield in the order of declaration will be leftmost in the allocation unit). Allocation units (either bytes or words) are always allocated in memory from the smallest to the highest address (that is, the first allocation unit will be at the smallest address in memory).

The following figure illustrates this allocation scheme:

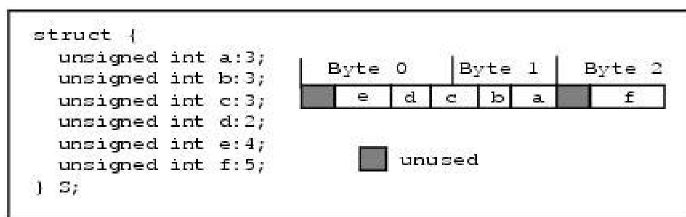

Figure 18-1. Allocation of Six Bitfields

The compiler also supports unnamed bitfields - which can be used for padding. Unnamed bitfield of width 0 forces alignment of the next bitfield to word boundary (no further bits will be placed in the current word), even if the current allocation unit is a byte.

NOTE

Unnamed bitfields cannot be initialized.

The following example demonstrates how to use unnamed bitfields for padding:

Listing: Example - Using Unnamed Bitfields for Padding

```
typedef struct {
        unsigned char a : 3;

        unsigned char : 2;

        unsigned char c: 3;

} SomeStruct;

SomeStruct s;
```

Assuming we initialize the fields of `s` as follows:

```
  s.a = 6; s.c = 7;
```

we get this memory layout for `s: E6`.

The example below shows how zero-width unnamed bitfields work.

Listing: Example - Zero-width Unnamed Bitfields

```
typedef struct {
        unsigned char a : 3;

        unsigned char : 0;

        unsigned char c: 3;

} SomeStruct;

SomeStruct s;
```

Assuming we initialize the fields of `s` as follows:

```
  s.a = 6; s.c = 7;
```

we get this memory layout for `s: 00 06 07`.

For more information, refer to the following topics:

- -bfield_gap_limit
- -[no]bfield_lsbit_first
- -[no]bfield_reduce_type

18.4 Calling Convention

This section describes the S12Z calling convention, covering the following topics:

- Argument Passing
- Return Values
- Entry and Exit Code

18.4.1 Argument Passing

The same calling convention is used for a fixed as well as a variable number of arguments:

- Arguments which have primitive types are either passed in registers or pushed onto stack. In order to decide which approach to take, the compiler evaluates them from left to right. Depending on the size of the argument and whether a register is still available for that size, the argument will be passed in one of the following registers:
  - D0 (1 byte)
  - D2, D3 (2 bytes)
  - X (3 bytes)
  - D6 (4 bytes)

or, alternatively, on the stack, if the size of the argument is greater than 4 bytes or there is no register left.

- All arguments with aggregate types are passed on stack.

The actual passing order is from right to left (that is, the first argument pushed is the rightmost one).

The following example demonstrates argument passing for a simple function with a fixed number of arguments:

Listing: Argument Passing for Simple Function with Fixed Number of Arguments

```
void func(char a, char b, int c, long d, long e)
{

g1 = a;

g2 = b;

g3 = c;

g4 = d;

g5 = e;

}

...

void main(void)

{

...

func(0x12, 0x34, 0x5678, 0x1234ABCD, 0x5678ABCD);

...

}
```

Assuming inlining has been turned off, the compiler generates code as listed below:

Listing: Compiler Output when Inlining is Turned Off

```
29:   func(0x12, 0x34, 0x5678, 0x1234ABCD, 0x5678ABCD);
LD        D6,#1450748877

PSH       D6

LD        D0,#52

PSH       D0

LD        D6,#305441741

LD        D2,#22136

LD        D0,#18

JSR       func
```

18.4.2 Return Values

If the function return type is a primitive type with its size at most 4 bytes, the result is returned in a register, as detailed below:

- D0 (1 byte)
- D2 (2 bytes)
- X (3 bytes)
- D6 (4 bytes)

If the function return type is an aggregate type or a primitive type with its size greater than 4, the function is called with an additional (leftmost one), hidden argument which contains the address to which the result should be copied.

The example listed below shows how value returning works when the function return type is 'int' (primitive type, 2-byte size).

Listing: Example for Value Return when Return Type is int

```
int func(int p)
{

return ++p;

}

...

void main(void)

{

...

x = func(0x1234);

...

}
```

Assuming inlining has been turned off, the compiler generates the following code for the callee:

Listing: Compiler Output for Callee when Inlining is Off

```
19: int func(int p)
  LEA       S,(#-2,S)

ST        D2,(0,S)

21: return ++p;

INC.W      (0,S)
```

```
LD        D2,(0,S)

22: }
```

and the follwing code for the caller:

Listing: Compiler Output for Caller when Inlining is Off

```
31:   x = func(0x1234);
LD        D2,#4660

JSR       func

ST        D2,x
```

The example listed below shows how value returning works when the function return type is a struct type.

Listing: Example for Value Return when Return Type is struct

```
typedef struct
{

int a;

int b;

} SomeStruct;

SomeStruct s;

...

SomeStruct func(int p1, int p2)

{

SomeStruct r;

r.a = p1;

r.b = p2;

return r;

}

...

void main(void)

{

...

s = func(1, 2);

...

}
```

Assuming inlining has been turned off, the compiler generates the following code for the callee:

Listing: Compiler Output for Callee when Inlining is Off

```
32: SomeStruct func(int p1, int p2)
LEA        S,(#-11,S)

ST         X,(0,S)

ST         D2,(3,S)

ST         D3,(5,S)

35: r.a = p1;

MOV.W      (3,S),(7,S)

36: r.b = p2;

LEA        X,(7,S)

LEA        X,(2,X)

LD         D2,(5,S)

ST         D2,(0,X)

37: return r;
```

and the follwing code for the caller:

Listing: Compiler Output for Caller when Inlining is Off

```
 48:   s = func3(1, 2);
LD         D3,#2

LD         D2,#1

LD         X,@s

JSR        func
```

Notice how the first (leftmost) argument to function 'func', passed in register X, is the address of global variable 's' of type 'SomeStruct'.

18.4.3 Entry and Exit Code

The entry code of a function consists of a sequence of instructions responsible for handling the actual parameters, saving registers, and allocating stack space for that function. Entry code generation can be disabled if the parameters, the registers and the stack space are handled explicitly from inline-assembly code or if they are not required at all in the function.

The exit code of a function is responsible with restoring registers and deallocating stack space for that function. Exit code generation can be disabled if the registers and the stack space are handled explicitly from inline-assembly code or if they are not required at all in the function.

In order to disable entry/exit code generation, you need to use the NO_ENTRY/ NO_EXIT pragma, however not by itself, but in pair with the NO_EXIT/ NO_ENTRY pragma, since the stack frame can get corrupted otherwise.

The following example demonstrates incorrect use of the entry/exit code pragmas.

Listing: Example - Incorrect use of Entry/Exit Code Pragmas

```
long func(char p1, int p2)
{

int a;

long b, c;

a = p1;

b = p2;

c = a + b;

return c;

}
```

Without #pragma NO_ENTRY and #pragma NO_EXIT, the compiler generates this code:

Listing: Compiler Output without #pragma NO_ENTRY and #pragma NO_EXIT

```
42: long func(char p1, int p2)
LEA        S,(#-13,S)   ; entry code : allocates stack space (sets the
stack pointer)

ST         D0,(0,S)     ; entry code : stores the first argument ('p1'),
which was passed in D0

ST         D2,(1,S)     ; entry code : stores the second argument ('p2'),
which was passed in D2

46: a = p1;

LD         D0,(0,S)

TFR        D0,D2

ST         D2,(11,S)

47: b = p2;

LD         D2,(1,S)

SEX        D2,D6

ST         D6,(7,S)

48: c = a + b;

LD         D2,(11,S)

SEX        D2,D6

ADD        D6,(7,S)

ST         D6,(3,S)
```

```
   49: return c;

LD        D6,(3,S)   ; exit code: returns the result value in D6
(according to the calling convention)

   50: }

LEA       S,(13,S)   ; exit code: deallocates stack space (restores the
stack pointer)

RTS
```

With `#pragma NO_ENTRY` and `#pragma NO_EXIT`, the compiler generates this code:

Listing: Compiler Output with #pragma NO_ENTRY and #pragma NO_EXIT

```
42: long func(char p1, int p2)
TFR       D0,D3

ST        D3,(8,S)

47: b = p2;

SEX       D2,D6

ST        D6,(4,S)

48: c = a + b;

LD        D2,(8,S)

SEX       D2,D6

ADD       D6,(4,S)

ST        D6,(0,S)

49: return c;

LD        D6,(0,S)

   50: }

RTS
```

which will not work, because it does not set the stack pointer as it should.

Chapter 19
Intermediate Analysis and Optimizations

After it translates a program's source code into its intermediate representation, the compiler optionally applies optimizations that reduce the program's size, improve its execution speed, or both. The topics in this chapter explain these optimizations and how to apply them:

- Interprocedural Analysis
- Intermediate Optimizations
- Inlining

19.1 Interprocedural Analysis

Most compiler optimizations are applied only within a function. The compiler analyzes a function's flow of execution and how the function uses variables. It uses this information to find shortcuts in execution and reduce the number of registers and memory that the function uses. These optimizations are useful and effective but are limited to the scope of a function.

The CodeWarrior compiler has a special optimization that it applies at a greater scope. Widening the scope of an optimization offers the potential to greatly improve performance and reduce memory use. *Interprocedural analysis* examines the flow of execution and data within entire files.

- Invoking Interprocedural Analysis
- Function-Level Optimization
- File-Level Optimization

19.1.1 Invoking Interprocedural Analysis

The following table explains how to control interprocedural analysis.

Table 19-1. Controlling interprocedural analysis

Turn control this option from here...	use this setting
source code	`#pragma ipa file | on | off`
command line	`-ipa file`

19.1.2 Function-Level Optimization

Interprocedural analysis may be disabled by setting it to either `off` or `function`. If IPA is disabled, the compiler generates instructions and data as it reads and analyzes each function. This setting is equivalent to the "no deferred codegen" mode of older compilers.

19.1.3 File-Level Optimization

When interprocedural analysis is set to optimize at the file level, the compiler reads and analyzes an entire file before generating instructions and data.

At this level, the compiler generates more efficient code for inline function calls and C++ exception handling than when interprocedural analysis is off. The compiler is also able to increase character string reuse and pooling, reducing the size of object code. This is equivalent to the " deferred inlining" and "deferred codegen" options of older compilers.

The compiler also safely removes static functions and variables that are not referred to within the file, which reduces the amount of object code that the linker must process, resulting in better linker performance.

19.2 Intermediate Optimizations

After it translates a function into its intermediate representation, the compiler may optionally apply some optimizations. The result of these optimizations on the intermediate representation will either reduce the size of the executable code, improve the executable code's execution speed, or both.

- Dead Code Elimination
- Expression Simplification
- Common Subexpression Elimination
- Copy Propagation
- Dead Store Elimination
- Live Range Splitting
- Loop-Invariant Code Motion
- Strength Reduction
- Loop Unrolling

19.2.1 Dead Code Elimination

The dead code elimination optimization removes expressions that are not accessible or are not referred to. This optimization reduces size and increases execution speed.

The following table explains how to control the optimization for dead code elimination.

Table 19-2. Controlling dead code elimination

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 1**, **Level 2**, **Level 3**, or **Level 4** in the **OptimizationsLevel** settings panel.
source code	`#pragma opt_dead_code on | off | reset`
command line	`-opt [no]deadcode`

In the following listing, the call to `func1()` will never execute because the `if` statement that it is associated with will never be true.

Listing: Before dead code elimination

```
void func_from(void)
{
    if (0)
    {
        func1();
    }
    func2();
}
```

Consequently, the compiler can safely eliminate the call to `func1()`, as shown in the following listing.

Listing: After dead code elimination

```
void func_to(void)
{
    func2();
}
```

19.2.2 Expression Simplification

The expression simplification optimization attempts to replace arithmetic expressions with simpler expressions. Additionally, the compiler also looks for operations in expressions that can be avoided completely without affecting the final outcome of the expression. This optimization reduces size and increases speed.

The following table explains how to control the optimization for expression simplification.

Table 19-3. Controlling expression simplification

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 1** , **Level 2** , **Level 3** , or **Level 4** in the **Optimization Level** settings panel.
source code	There is no pragma to control this optimization.
command line	`-opt level=1`, `-opt level=2`, `-opt level=3`, `-opt level=4`

For example, the following listing contains a few assignments to some arithmetic expressions:

- addition to zero
- multiplication by a power of 2
- subtraction of a value from itself
- arithmetic expression with two or more literal values

Listing: Before expression simplification

```
void func_from(int* result1, int* result2, int* result3, int* result4, int x)
{
    *result1 = x + 0;
    *result2 = x * 2;
    *result3 = x - x;
    *result4 = 1 + x + 4;
}
```

The following listing shows source code that is equivalent to expression simplification. The compiler has modified these assignments to:

- remove the addition to zero
- replace the multiplication of a power of 2 with bit-shift operation
- replace a subtraction of x from itself with 0
- consolidate the additions of 1 and 4 into 5

Listing: After expression simplification

```
void func_to(int* result1, int* result2, int* result3, int* result4, int x)
{
    *result1 = x;
    *result2 = x << 1;
    *result3 = 0;
    *result4 = 5 + x;
}
```

19.2.3 Common Subexpression Elimination

Common subexpression elimination replaces multiple instances of the same expression with a single instance. This optimization reduces size and increases execution speed.

The following table explains how to control the optimization for common subexpression elimination.

Table 19-4. Controlling common subexpression elimination

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 2** , **Level 3** , or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_common_subs on | off | reset`
command line	`-opt [no]cse`

For example, in the following listing, the subexpression `x * y` occurs twice.

Listing: Before common subexepression elimination

```
void func_from(int* vec, int size, int x, int y, int value)
{
    if (x * y < size)
    {
        vec[x * y - 1] = value;
    }
}
```

The following listing shows equivalent source code after the compiler applies common subexpression elimination. The compiler generates instructions to compute `x * y` and stores it in a hidden, temporary variable. The compiler then replaces each instance of the subexpression with this variable.

Listing: After common subexpression elimination

```
void func_to(int* vec, int size, int x, int y, int value)
{
    int temp = x * y;
    if (temp < size)
    {
        vec[temp - 1] = value;
    }
}
```

19.2.4 Copy Propagation

Copy propagation replaces variables with their original values if the variables do not change. This optimization reduces runtime stack size and improves execution speed.

The following table explains how to control the optimization for copy propagation.

Table 19-5. Controlling copy propagation

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 2**, **Level 3**, or **Level 4** in the **Global Optimizations** settings panel.
source code	`#pragma opt_propagation on | off | reset`
command line	`-opt [no]prop[agation]`

For example, in the following listing, the variable `j` is assigned the value of `x`.

Listing: Before copy propagation

```
void func_from(int* a, int x)
{
    int i;
    int j;
    j = x;
    for (i = 0; i < j; i++)
    {
        a[i] = j;
    }
}
```

But `j`'s value is never changed, so the compiler replaces later instances of `j` with `x`, as shown in the following listing:

Listing: After copy propagation

```
void func_to(int* a, int x)
{
    int i;
    int j;
   j = x;
    for (i = 0; i < x; i++)
    {
        a[i] = x;
    }
}
```

By propagating `x`, the compiler is able to reduce the number of registers it uses to hold variable values, allowing more variables to be stored in registers instead of slower memory. Also, this optimization reduces the amount of stack memory used during function calls.

19.2.5 Dead Store Elimination

Dead store elimination removes unused assignment statements. This optimization reduces size and improves speed.

The following table explains how to control the optimization for dead store elimination.

Table 19-6. Controlling dead store elimination

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_dead_assignments on	off	reset`
command line	`-opt [no]deadstore`

For example, in the following listing the variable `x` is first assigned the value of `y * y`. However, this result is not used before `x` is assigned the result returned by a call to `getresult()`.

Listing: Before dead store elimination

```
void func_from(int x, int y)
{
    x = y * y;
    otherfunc1(y);
    x = getresult();
    otherfunc2(y);
}
```

In the following listing the compiler can safely remove the first assignment to `x` since the result of this assignment is never used.

Listing: After dead store elimination

```
void func_to(int x, int y)
{
    otherfunc1(y);
    x = getresult();
    otherfunc2(y);
}
```

19.2.6 Live Range Splitting

Live range splitting attempts to reduce the number of variables used in a function. This optimization reduces a function's runtime stack size, requiring fewer instructions to invoke the function. This optimization potentially improves execution speed.

The following table explains how to control the optimization for live range splitting.

Table 19-7. Controlling live range splitting

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	There is no pragma to control this optimization.
command line	`-opt level=3`, `-opt level=4`

For example, in the following listing three variables, `a`, `b`, and `c`, are defined. Although each variable is eventually used, each of their uses is exclusive to the others. In other words, `a` is not referred to in the same expressions as `b` or `c`, `b` is not referred to with `a` or `c`, and `c` is not used with `a` or `b`.

Listing: Before live range splitting

```
void func_from(int x, int y)

{

    int a;

    int b;

    int c;

    a = x * y;

    otherfunc(a);

    b = x + y;

    otherfunc(b);

    c = x - y;

    otherfunc(c);

}
```

In the following listing, the compiler has replaced `a`, `b`, and `c`, with a single variable. This optimization reduces the number of registers that the object code uses to store variables, allowing more variables to be stored in registers instead of slower memory. This optimization also reduces a function's stack memory.

Listing: After live range splitting

```
void func_to(int x, int y)
{
    int a_b_or_c;
    a_b_or_c = x * y;
    otherfunc(temp);
    a_b_or_c = x + y;
    otherfunc(temp);
    a_b_or_c = x - y;
    otherfunc(temp);
}
```

19.2.7 Loop-Invariant Code Motion

Loop-invariant code motion moves expressions out of a loop if the expressions are not affected by the loop or the loop does not affect the expression. This optimization improves execution speed.

The following table explains how to control the optimization for loop-invariant code motion.

Table 19-8. Controlling loop-invariant code motion

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_loop_invariants on | off | reset`
command line	`-opt [no]loop[invariants]`

For example, in the following listing, the assignment to the variable `circ` does not refer to the counter variable of the `for` loop, `i`. But the assignment to `circ` will be executed at each loop iteration.

Listing: Before loop-invariant code motion

```
void func_from(float* vec, int max, float val)
{
    float circ;
    int i;
    for (i = 0; i < max; ++i)
    {
        circ = val * 2 * PI;
        vec[i] = circ;
    }
}
```

The following listing shows source code that is equivalent to how the compiler would rearrange instructions after applying this optimization. The compiler has moved the assignment to `circ` outside the `for` loop so that it is only executed once instead of each time the `for` loop iterates.

Listing: After loop-invariant code motion

```
void func_to(float* vec, int max, float val)
{
    float circ;
    int i;
    circ = val * 2 * PI;
    for (i = 0; i < max; ++i)
    {
        vec[i] = circ;
    }
}
```

19.2.8 Strength Reduction

Strength reduction attempts to replace slower multiplication operations with faster addition operations. This optimization improves execution speed but increases code size.

The following table explains how to control the optimization for strength reduction.

Table 19-9. Controlling strength reduction

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_strength_reduction on | off | reset`
command line	`-opt [no]strength`

For example, in the following listing, the assignment to elements of the `vec` array use a multiplication operation that refers to the `for` loop's counter variable, `i`.

Listing: Before strength reduction

```
void func_from(int* vec, int max, int fac)
{
    int i;
    for (i = 0; i < max; ++i)
    {
        vec[i] = fac * i;
    }
}
```

In the code listed below, the compiler has replaced the multiplication operation with a hidden variable that is increased by an equivalent addition operation. Processors execute addition operations faster than multiplication operations.

Listing: After strength reduction

```
void func_to(int* vec, int max, int fac)
{
    int i;
    int strength_red;
    hidden_strength_red = 0;
    for (i = 0; i < max; ++i)
    {
        vec[i] = hidden_strength_red;
        hidden_strength_red = hidden_strength_red + i;
    }
}
```

19.2.9 Loop Unrolling

Loop unrolling inserts extra copies of a loop's body in a loop to reduce processor time executing a loop's overhead instructions for each iteration of the loop body. In other words, this optimization attempts to reduce the ratio of the time that the processor takes to execute a loop's completion test and the branching instructions, compared to the time the processor takes to execute the loop's body. This optimization improves execution speed but increases code size.

The following table explains how to control the optimization for loop unrolling.

Table 19-10. Controlling loop unrolling

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_unroll_loops on	off	reset`
command line	`-opt level=3`, `-opt level=4`

For example, in the following listing, the `for` loop's body is a single call to a function, `otherfunc()`. For each time the loop's completion test executes

```
for (i = 0; i < MAX; ++i)
```

the function executes the loop body only once.

Listing: Before loop unrolling

```
const int MAX = 100;

void func_from(int* vec)

{

    int i;

    for (i = 0; i < MAX; ++i)

    {

        otherfunc(vec[i]);

    }

}
```

In the following code, the compiler has inserted another copy of the loop body and rearranged the loop to ensure that variable `i` is incremented properly. With this arrangement, the loop's completion test executes once for every 2 times that the loop body executes.

Listing: After loop unrolling

```
const int MAX = 100;
void func_to(int* vec)
{
    int i;
    for (i = 0; i < MAX;)
    {
        otherfunc(vec[i]);
        ++i;
        otherfunc(vec[i]);
        ++i;
    }
}
```

19.3 Inlining

Inlining replaces instructions that call a function and return from it with the actual instructions of the function being called. Inlining function makes your program faster because it executes the function code immediately without the overhead of a function call and return. However, inlining can also make your program larger because the compiler may insert the function's instructions many times throughout your program.

The rest of this section explains how to specify which functions to inline and how the compiler performs the inlining:

- Choosing Which Functions to Inline
- Inlining Techniques

19.3.1 Choosing Which Functions to Inline

The compiler offers several methods to specify which functions are eligible for inlining.

To specify that a function is eligible to be inlined, precede its definition with the `inline`, `__inline__`, or `__inline` keyword. To allow these keywords in C source code, turn off **ANSI Keywords Only** in the CodeWarrior IDE's **C/C++ Language** settings panel or turn off the `only_std_keywords` pragma in your source code.

To verify that an eligible function has been inlined or not, use the **Non-Inlined Functions** option in the IDE's **C/C++ Warnings** panel or the `warn_notinlined` pragma. The following listing shows an example.

Listing: Specifying to the compiler that a function may be inlined

```
#pragma only_std_keywords off
inline int attempt_to_inline(void)
{
    return 10;
}
```

To specify that a function must never be inlined, follow its definition's specifier with `__attribute__((never_inline))`. The following listing shows an example.

Listing: Specifying to the compiler that a function must never be inlined

```
int never_inline(void) __attribute__((never_inline))
{
   return 20;
}
```

To specify that no functions in a file may be inlined, including those that are defined with the `inline`, `__inline__`, or `__inline` keywords, use the `dont_inline` pragma. The following listing shows an example.

Listing: Specifying that no functions may be inlined

```
#pragma dont_inline on
/* Will not be inlined. */
inline int attempt_to_inline(void)
{
    return 10;
}
/* Will not be inlined. */
int never_inline(void) __attribute__((never_inline))
{
   return 20;
}
#pragma dont_inline off
/* Will be inlined, if possible. */
inline int also_attempt_to_inline(void)
{
    return 10;
}
```

The kind of functions that are never inlined are as follows:

- functions with variable argument lists
- functions defined with `__attribute__((never_inline))`

- functions compiled with `#pragma optimize_for_size on` or the **Optimize For Size** setting in the IDE's **Optimization Level** panel.
- functions which have their addresses stored in variables

The compiler will not inline these functions, even if they are defined with the `inline`, `__inline__`, or `__inline` keywords.

The following functions also should never be inlined:

- functions that return class objects that need destruction
- functions with class arguments that need destruction

It can inline such functions if the class has a trivial empty constructor as in this case:

```
struct X {
        int n;
      X(int a) { n = a; }
      ~X() {}
  };
  inline X f(X x) { return X(x.n + 1); }
  int main()
  {
      return f(X(1)).n;
  }
```

This does not depend on "ISO C++ templates".

19.3.2 Inlining Techniques

The depth of inlining explains how many levels of function calls the compiler will inline. The **Inlining** setting in the IDE's **C/C++ Build > Settings > S12Z Compiler > Optimization** panel and the `inline_depth` pragma control inlining depth.

Normally, the compiler only inlines an eligible function if it has already translated the function's definition. In other words, if an eligible function has not yet been compiled, the compiler has no object code to insert. To overcome this limitation, the compiler can perform interprocedural analysis (IPA) either in file or program mode. This lets the compiler evaluate all the functions in a file or even the entire program before inlining the code. The **IPA** setting in the IDE's **C/C++ Build > Settings > S12Z Compiler Language** settings panel and the `ipa` pragma control this capability.

The compiler normally inlines functions from the first function in a chain of function calls to the last function called. Alternately, the compiler may inline functions from the last function called to the first function in a chain of function calls. The **Bottom-up**

Inlining option in the IDE's **C/C++ Build > Settings > S12Z Compiler > Optimization** settings panel and the `inline_bottom_up` and `inline_bottom_up_once` pragmas control this reverse method of inlining.

Some functions that have not been defined with the `inline`, `__inline__`, or `__inline` keywords may still be good candidates to be inlined. Automatic inlining allows the compiler to inline these functions in addition to the functions that you explicitly specify as eligible for inlining. The **Auto-Inline** option in the IDE's **C/C++ Build > Settings > S12Z Compiler > Optimization** panel and the `auto_inline` pragma control this capability.

When inlining, the compiler calculates the complexity of a function by counting the number of statements, operands, and operations in a function to determine whether or not to inline an eligible function. The compiler does not inline functions that exceed a maximum complexity. The compiler uses three settings to control the extent of inlined functions:

- maximum auto-inlining complexity: the threshold for which a function may be auto-inlined
- maximum complexity: the threshold for which any eligible function may be inlined
- maximum total complexity: the threshold for all inlining in a function

The `inline_max_auto_size`, `inline_max_size`, and `inline_max_total_size` pragmas control these thresholds, respectively.

Chapter 20
Alignment

The S12Z compiler provides support for aligning:

- global data
- struct members
- stack

NOTE

For code alignment, use the ALIGN linker directive in the linker parameter file.

NOTE

For information on this directive, refer to the topic 'Defining an Alignment Rule' in the Build Tools Utilities manual.

20.1 Alignment of Global Data

Alignment of global data can be achieved using one of the following:

- `__attribute__ ((aligned (<n>)))` - to specify an alignment value for the associated data object or type definition;
- `option 'align_globals'` - to align _all_ the global variables;
- `pragma 'align_globals' (on/off/reset)` - to align all the global variables in its scope.

The rest of this topic lists the alignment approaches.

20.1.1 Alignment Attributes

Use the __attribute__ ((aligned(...))) directive to specify to the compiler on what memory boundary to store data objects.

The syntax of the directive is:

```
__attribute__ ((aligned(<n>))
```

where <n> is a decimal number of a power of 2 from 1 to 4096.

20.1.1.1 Alignment Attribute and Global Variable Declarations

Use the alignment attribute to specify the alignment of a global variable.

Example

- The following declaration aligns global variable 'foo' on a 4-byte boundary: `int foo __attribute__ ((aligned (4)))`;
- The following declaration aligns global variable 'bar' on a 16-byte boundary. `int bar[4] __attribute__ ((aligned (16)));`

20.1.1.2 Alignment Attribute and typedef Declarations

Use the alignment attribute to specify how objects of a specific type should be aligned.

Example

- The following typedef declaration aligns all objects of type SignedLong on an 8-byte boundary.

  ```
  typedef long SignedLong __attribute__ ((aligned (8)));
  SignedLong sl1; /* aligned on a 8-byte boundary */
  SignedLong sl2; /* aligned on a 8-byte boundary */
  ```

  However, if an array is declared that contains elements of type SignedLong, such as, for instance: SignedLong v[5]; the alignment attribute will only have effect upon the array itself, not its individual elements. Variable 'v' will be aligned on an 8-byte boundary, but its elements will not (the size of the array will be 20, not 40). If the declaration of the array also uses an alignment attribute, specifying a different alignment than that of the type of the elements. For instance:

  ```
  SignedLong v[5] __attribute__ ((aligned (64)));
  ```

  the array attribute will prevail. Variable 'v' will be aligned on a 64-byte boundary. The following typedef declaration aligns all instances of struct SimpleStruct on a 4-byte boundary.
- The following typedef declaration aligns all instances of struct SimpleStruct on a 4-byte boundary.

```
typedef struct
{
char x;
char y;
long z;
} SimpleStruct __attribute__ ((aligned(8)));
SimpleStruct s1; /* aligned on an 8-byte boundary */
SimpleStruct s2; /* aligned on an 8-byte boundary */
```

20.1.1.3 Alignment Attribute and Struct Members

The alignment attribute should not be used with a struct member. A warning will be reported if this happens.

In order to align struct members for optimal execution speed, use option -align_structs.

See Also

align_structs

20.1.2 Alignment of Struct Members

Use option -align_structs for optimal alignment of all the members of a given struct.

For information on this option, refer to topic: align_structs

20.1.3 Stack Alignment

Use option -align_stack to enable alignment of stack accesses.

For information on this option, refer to topic: align_stack.

Chapter 21
Predefined Macros

The compiler preprocessor has predefined macros (some refer to these as predefined symbols). The compiler simulates the variable definitions, that describe the compile-time environment and the properties of the target processor.

This chapter lists the predefined macros that all CodeWarrior compilers make available.

- __COUNTER__
- __cplusplus
- __CWCC__
- __DATE__
- __embedded_cplusplus
- __FILE__
- __func__
- __FUNCTION__
- __ide_target()
- __LINE__
- __MWERKS__
- __PRETTY_FUNCTION__
- __profile__
- __STDC__
- __TIME__
- __optlevelx

21.1 __COUNTER__

Preprocessor macro that expands to an integer.

Syntax

```
__COUNTER__
```

Remarks

The compiler defines this macro as an integer that has an intial value of 0 incrementing by 1 every time the macro is used in the translation unit.

The value of this macro is stored in a precompiled header and is restored when the precompiled header is used by a translation unit.

21.2 __cplusplus

Preprocessor macro defined if compiling C++ source code.

Syntax

```
__cplusplus
```

Remarks

The compiler defines this macro when compiling C++ source code. This macro is undefined otherwise.

21.3 __CWCC__

Preprocessor macro defined as the version of the CodeWarrior compiler.

Syntax

```
__CWCC__
```

Remarks

CodeWarrior compilers issued after 2006 define this macro with the compiler's version. For example, if the compiler version is 4.2.0, the value of `__CWCC__` is `0x4200`.

CodeWarrior compilers issued prior to 2006 used the pre-defined macro `__MWERKS__`. The `__MWERKS__` predefined macro is still functional as an alias for `__CWCC__`.

The ISO standards do not specify this symbol.

21.4 __DATE__

Preprocessor macro defined as the date of compilation.

Syntax

```
__DATE__
```

Remarks

The compiler defines this macro as a character string representation of the date of compilation. The format of this string is

```
"Mmm dd yyyy"
```

where *Mmm* is the three-letter abbreviation of the month, *dd* is the day of the month, and *yyyy* is the year.

21.5 __embedded_cplusplus

Defined as 1 when compiling embedded C++ source code, undefined otherwise.

Syntax

```
__embedded_cplusplus
```

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the Embedded C++ proposed standard. The compiler does not define this macro otherwise.

21.6 __FILE__

Preprocessor macro of the name of the source code file being compiled.

Syntax

```
__FILE__
```

Remarks

The compiler defines this macro as a character string literal value of the name of the file being compiled, or the name specified in the last instance of a `#line` directive.

21.7 __func__

Predefined variable of the name of the function being compiled.

Prototype

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

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__func__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body. This variable is also undefined when C99 (ISO/IEC 9899-1999) or GCC (GNU Compiler Collection) extension settings are off.

21.8 __FUNCTION__

Predefined variable of the name of the function being compiled.

Prototype

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

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__FUNCTION__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body. This variable is also undefined when C99 (ISO/IEC 9899-1999) or GCC (GNU Compiler Collection) extension settings are off.

21.9 __ide_target()

Preprocessor operator for querying the IDE about the active build target.

Syntax

```
__ide_target("target_name")
```

`target-name`

The name of a build target in the active project in the CodeWarrior IDE.

Remarks

Expands to `1` if *target_name* is the same as the active build target in the CodeWarrior IDE's active project, otherwise, expands to `0`. The ISO standards do not specify this symbol.

21.10 __LINE__

Preprocessor macro of the number of the line of the source code file being compiled.

Syntax

```
__LINE__
```

Remarks

The compiler defines this macro as an integer value of the number of the line of the source code file that the compiler is translating. The `#line` directive also affects the value that this macro expands to.

21.11 __MWERKS__

Deprecated. Preprocessor macro defined as the version of the CodeWarrior compiler.

Syntax

```
__MWERKS__
```

Remarks

Replaced by the built-in preprocessor macro `__CWCC__`.

CodeWarrior compilers issued after 1995 define this macro with the compiler's version. For example, if the compiler version is 4.0, the value of `__MWERKS__` is `0x4000`.

This macro is defined as `1` if the compiler was issued before the CodeWarrior CW7 that was released in 1995.

The ISO standards do not specify this symbol.

21.12 __PRETTY_FUNCTION__

Predefined variable containing a character string of the "unmangled" name of the C++ function being compiled.

Syntax

Prototype

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

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__PRETTY_FUNCTION__`. This name, *function-name*, is the same identifier that appears in source code, not the "mangled" identifier that the compiler and linker use. The C++ compiler "mangles" a function name by appending extra characters to the function's identifier to denote the function's return type and the types of its parameters.

The ISO/IEC 14882-2003 C++ standard does not specify this symbol. This implicit variable is undefined outside of a function body. This symbol is only defined if the GCC extension setting is on.

21.13 __profile__

Preprocessor macro that specifies whether or not the compiler is generating object code for a profiler.

Syntax

```
__profile__
```

Remarks

Defined as 1 when generating object code that works with a profiler, otherwise, undefined. The ISO standards does not specify this symbol.

21.14 __STDC__

Defined as 1 when compiling ISO/IEC Standard C source code, otherwise, undefined.

Syntax

```
__STDC__
```

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the ISO/IEC 9899-1990 and ISO/IEC 9899-1999 standards. Otherwise, the compiler does not define this macro.

21.15 __TIME__

Preprocessor macro defined as a character string representation of the time of compilation.

Syntax

```
__TIME__
```

Remarks

The compiler defines this macro as a character string representation of the time of compilation. The format of this string is

```
"hh:mm:ss"
```

where *hh* is a 2-digit hour of the day, *mm* is a 2-digit minute of the hour, and *ss* is a 2-digit second of the minute.

21.16 __optlevelx

Optimization level exported as a predefined macro.

Syntax

```
__optlevel0

__optlevel1

__optlevel2

__optlevel3

__optlevel4
```

Remarks

Using these macros, user can conditionally compile code for a particular optimization level. The following table lists the level of optimization provided by the `__optlevelx` macro.

Table 21-1. Optimization Levels

Macro	Optimization Level
__optlevel0	O0
__optlevel1	O1
__optlevel2	O2
__optlevel3	O3
__optlevel4	O4

Example

The listing below shows an example of `__optlevelx` macro usage.

Listing: Example usage of __optlevel macro

```
int main()
{
#if __optlevel0
... // This code compiles only if this code compiled with Optimization
level 0
#elif __optlevel1
... // This code compiles only if this code compiled with Optimization
level 1
#elif __optlevel2
... // This code compiles only if this code compiled with Optimization
level 2
#elif __optlevel3
... // This code compiles only if this code compiled with Optimization
level 3
#elif __optlevel4
... // This code compiles only if this code compiled with Optimization
level 4
#endif
}
```

Chapter 22
S12Z Predefined Symbols

The compiler preprocessor has predefined macros and the compiler simulates variable definitions that describe the compile-time environment and properties of the target processor.

This chapter describes the predefined symbols made available by the CodeWarrior compiler for S12Z processors. The following listed are the predefined symbols for S12Z compiler:

Table 22-1. S12Z Predefined Symbols

__S12LISA__	__INT_IS_32BIT__	__WCHAR_T_IS_UCHAR__
__S12Z__	__LONG_IS_8BIT__	__WCHAR_T_IS_USHORT__
__HC12__	__LONG_IS_16BIT__	__WCHAR_T_IS_ULONG__
__CHAR_IS_UNSIGNED__	__LONG_IS_32BIT__	__SIZE_T_IS_UCHAR__
__CHAR_IS_SIGNED__	__LONG_LONG_IS_8BIT__	__SIZE_T_IS_USHORT__
__CHAR_IS_8BIT__	__LONG_LONG_IS_16BIT__	__SIZE_T_IS_UINT__
__CHAR_IS_16BIT__	__LONG_LONG_IS_32BIT__	__SIZE_T_IS_ULONG__
__CHAR_IS_32BIT__	__FLOAT_IS_IEEE32__	__PTRDIFF_T_IS_CHAR__
__SHORT_IS_8BIT__	__FLOAT_IS_IEEE64__	__PTRDIFF_T_IS_SHORT__
__SHORT_IS_16BIT__	__DOUBLE_IS_IEEE32__	__PTRDIFF_T_IS_INT__
__SHORT_IS_32BIT__	__DOUBLE_IS_IEEE64__	__PTRDIFF_T_IS_LONG__
__INT_IS_8BIT__	__LONG_DOUBLE_IS_IEEE32__	
__INT_IS_16BIT__	__LONG_DOUBLE_IS_IEEE64__	

22.1 __S12LISA__

Preprocessor macro defined to describe the target processor.

Syntax

```
#define __S12LISA__ 1
```

Remarks

The compiler defines this macro to be 1 for any S12Z derivative.

22.2 __S12Z__

Preprocessor macro defined to describe the target processor.

Syntax

```
#define __S12Z__ 1
```

Remarks

The compiler defines this macro to be 1 for any S12Z derivative.

22.3 __HC12__

Preprocessor macro defined to describe the target processor.

Syntax

```
#define __HC12__  1
```

Remarks

The compiler defines this macro to be 1 for any S12Z derivative.

22.4 __CHAR_IS_UNSIGNED__

Preprocessor macro defined to describe whether the char is interpreted as signed or unsigned number.

Syntax

```
#define __CHAR_IS_UNSIGNED__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the compiler is set to interpret char as an unsigned number, 0 otherwise.

22.5 __CHAR_IS_SIGNED__

Preprocessor macro defined to describe whether the char is interpreted as signed or unsigned number.

Syntax

```
#define __CHAR_IS_SIGNED__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the compiler is set to interpret char as a signed number, 0 otherwise.

22.6 __CHAR_IS_8BIT__

Preprocessor macro defined to describe if the size of char is 8-bit.

Syntax

```
#define __CHAR_IS_8BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of char is 8-bit, 0 otherwise.

22.7 __CHAR_IS_16BIT__

Preprocessor macro defined to describe if the size of char is 16-bit.

Syntax

```
#define __CHAR_IS_16BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of char is 16-bit, 0 otherwise.

22.8 __CHAR_IS_32BIT__

Preprocessor macro defined to describe if the size of char is 32-bit.

Syntax

```
#define __CHAR_IS_32BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of char is 32-bit, 0 otherwise.

22.9 __SHORT_IS_8BIT__

Preprocessor macro defined to describe if the size of short is 8-bit.

Syntax

```
#define __SHORT_IS_8BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of short is 8-bit, 0 otherwise.

22.10 __SHORT_IS_16BIT__

Preprocessor macro defined to describe if the size of short is 16-bit.

Syntax

```
#define __SHORT_IS_16BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of short is 16-bit, 0 otherwise.

22.11 __SHORT_IS_32BIT__

Preprocessor macro defined to describe if the size of short is 32-bit.

Syntax

```
#define __SHORT_IS_32BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of short is 32-bit, 0 otherwise.

22.12 __INT_IS_8BIT__

Preprocessor macro defined to describe if the size of int is 8-bit.

Syntax

```
#define __INT_IS_8BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of int is 8-bit, 0 otherwise.

22.13 __INT_IS_16BIT__

Preprocessor macro defined to describe if the size of int is 16-bit.

Syntax

```
#define __INT_IS_16BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of int is 16-bit, 0 otherwise.

22.14 __INT_IS_32BIT__

Preprocessor macro defined to describe if the size of int is 32-bit.

Syntax

```
#define __INT_IS_32BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of int is 32-bit, 0 otherwise.

22.15 __LONG_IS_8BIT__

Preprocessor macro defined to describe if the size of long is 8-bit.

Syntax

```
#define __LONG_IS_8BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of long is 8-bit, 0 otherwise.

22.16 __LONG_IS_16BIT__

Preprocessor macro defined to describe if the size of long is 16-bit.

Syntax

```
#define __LONG_IS_16BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of long is 16-bit, 0 otherwise.

22.17 __LONG_IS_32BIT__

Preprocessor macro defined to describe if the size of long is 32-bit.

Syntax

```
#define __LONG_IS_32BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of long is 32-bit, 0 otherwise.

22.18 __LONG_LONG_IS_8BIT__

Preprocessor macro defined to describe if the size of long long is 8-bit.

Syntax

```
#define __LONG_LONG_IS_8BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of long long is 8-bit, 0 otherwise.

22.19 __LONG_LONG_IS_16BIT__

Preprocessor macro defined to describe if the size of long long is 16-bit.

Syntax

```
#define __LONG_LONG_IS_16BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of long long is 16-bit, 0 otherwise.

22.20 __LONG_LONG_IS_32BIT__

Preprocessor macro defined to describe if the size of long long is 32-bit.

Syntax

```
#define __LONG_LONG_IS_32BIT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the size of long long is 32-bit, 0 otherwise.

22.21 __FLOAT_IS_IEEE32__

Preprocessor macro defined to describe if the float is represented on 32 bits.

Syntax

```
#define __FLOAT_IS_IEEE32__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the float is represented on 32 bits, 0 otherwise.

22.22 __FLOAT_IS_IEEE64__

Preprocessor macro defined to describe if the float is represented on 64 bits.

Syntax

```
#define __FLOAT_IS_IEEE64__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the float is represented on 64 bits, 0 otherwise.

22.23 __DOUBLE_IS_IEEE32__

Preprocessor macro defined to describe if the double is represented on 32 bits.

Syntax

```
#define __DOUBLE_IS_IEEE32__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the double is represented on 32 bits, 0 otherwise.

22.24 __DOUBLE_IS_IEEE64__

Preprocessor macro defined to describe if the double is represented on 64 bits.

Syntax

```
#define __DOUBLE_IS_IEEE64__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the double is represented on 64 bits, 0 otherwise.

22.25 __LONG_DOUBLE_IS_IEEE32__

Preprocessor macro defined to describe if the long double is represented on 32 bits.

Syntax

```
#define __LONG_DOUBLE_IS_IEEE32__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the long double is represented on 32 bits, 0 otherwise.

22.26 __LONG_DOUBLE_IS_IEEE64__

Preprocessor macro defined to describe if the long double is represented on 64 bits.

Syntax

```
#define __LONG_DOUBLE_IS_IEEE64__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the long double is represented on 64 bits, 0 otherwise.

22.27 __WCHAR_T_IS_UCHAR__

Preprocessor macro defined to describe `wchar_t` settings.

Syntax

```
#define __WCHAR_T_IS_UCHAR__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `wchar_t` is set to unsigned char, 0 otherwise.

22.28 __WCHAR_T_IS_USHORT__

Preprocessor macro defined to describe `wchar_t` settings.

Syntax

```
#define __WCHAR_T_IS_USHORT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `wchar_t` is set to unsigned short, 0 otherwise.

22.29 __WCHAR_T_IS_ULONG__

Preprocessor macro defined to describe `wchar_t` settings.

Syntax

```
#define __WCHAR_T_IS_ULONG__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `wchar_t` is set to unsigned long, 0 otherwise.

22.30 __SIZE_T_IS_UCHAR__

Preprocessor macro defined to describe `size_t` settings.

Syntax

```
#define __SIZE_T_IS_UCHAR__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `size_t` is set to unsigned char, 0 otherwise.

22.31 __SIZE_T_IS_USHORT__

Preprocessor macro defined to describe `size_t` settings.

Syntax

```
#define __SIZE_T_IS_USHORT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `size_t` is set to unsigned short, 0 otherwise.

22.32 __SIZE_T_IS_UINT__

Preprocessor macro defined to describe `size_t` settings.

Syntax

```
#define __SIZE_T_IS_UINT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `size_t` is set to unsigned int, 0 otherwise.

22.33 __SIZE_T_IS_ULONG__

Preprocessor macro defined to describe `size_t` settings.

Syntax

```
#define __SIZE_T_IS_ULONG__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `size_t` is set to unsigned long, 0 otherwise.

22.34 __PTRDIFF_T_IS_CHAR__

Preprocessor macro defined to describe `ptrdiff_t` settings.

Syntax

```
#define __PTRDIFF_T_IS_CHAR__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `ptrdiff_t` is set to char, 0 otherwise.

22.35 __PTRDIFF_T_IS_SHORT__

Preprocessor macro defined to describe `ptrdiff_t` settings.

Syntax

```
#define __PTRDIFF_T_IS_SHORT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `ptrdiff_t` is set to short, 0 otherwise.

22.36 __PTRDIFF_T_IS_INT__

Preprocessor macro defined to describe `ptrdiff_t` settings.

Syntax

```
#define __PTRDIFF_T_IS_INT__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `ptrdiff_t` is set to int, 0 otherwise.

22.37 __PTRDIFF_T_IS_LONG__

Preprocessor macro defined to describe `ptrdiff_t` settings.

Syntax

```
#define __PTRDIFF_T_IS_LONG__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the `ptrdiff_t` is set to long, 0 otherwise.

Chapter 23
Using Pragmas

The #pragma preprocessor directive specifies option settings to the compiler to control the compiler and linker's code generation.

- Checking Pragma Settings
- Saving and Restoring Pragma Settings
- Determining Which Settings Are Saved and Restored
- Invalid Pragmas
- Pragma Scope

23.1 Checking Pragma Settings

The preprocessor function __option() returns the state of pragma settings at compile-time. The syntax is

```
__option(setting-name)
```

where *setting-name* is the name of a pragma that accepts the on, off, and reset arguments.

If *setting-name* is on, __option(*setting-name*) returns 1. If *setting-name* is off, __option(*setting-name*) returns 0. If *setting-name* is not the name of a pragma, __option(*setting-name*) returns false. If *setting-name* is the name of a pragma that does not accept the on, off, and reset arguments, the compiler issues a warning message.

The following listing shows an example.

Listing: Using the __option() preprocessor function

```
#if __option(ANSI_strict)

#include "portable.h" /* Use the portable declarations. */

#else

#include "custom.h" /* Use the specialized declarations. */
```

```
#endif
```

23.2 Saving and Restoring Pragma Settings

There are some occasions when you would like to apply pragma settings to a piece of source code independently from the settings in the rest of the source file. For example, a function might require unique optimization settings that should not be used in the rest of the function's source file.

Remembering which pragmas to save and restore is tedious and error-prone. Fortunately, the compiler has mechanisms that save and restore pragma settings at compile time. Pragma settings may be saved and restored at two levels:

- all pragma settings
- some individual pragma settings

Settings may be saved at one point in a compilation unit (a source code file and the files that it includes), changed, then restored later in the same compilation unit. Pragma settings cannot be saved in one source code file then restored in another unless both source code files are included in the same compilation unit.

Pragmas `push` and `pop` save and restore, respectively, most pragma settings in a compilation unit. Pragmas `push` and `pop` may be nested to unlimited depth. The following listing shows an example.

Listing: Using push and pop to save and restore pragma settings

```
/* Settings for this file. */
#pragma opt_unroll_loops on
#pragma optimize_for_size off
void fast_func_A(void)
{
/* ... */
}
/* Settings for slow_func(). */
#pragma push /* Save file settings. */
#pragma optimization_size 0
void slow_func(void)
{
/* ... */
}
#pragma pop /* Restore file settings. */
void fast_func_B(void)
{
/* ... */
}
```

Pragmas that accept the `reset` argument perform the same actions as pragmas `push` and `pop`, but apply to a single pragma. A pragma's `on` and `off` arguments save the pragma's current setting before changing it to the new setting. A pragma's `reset` argument restores the pragma's setting. The `on`, `off`, and `reset` arguments may be nested to an unlimited depth. The following listing shows an example.

Listing: Using the reset option to save and restore a pragma setting

```
/* Setting for this file. */
#pragma opt_unroll_loops on
void fast_func_A(void)
{
/* ... */
}
/* Setting for smallslowfunc(). */
#pragma opt_unroll_loops off
void small_func(void)
{
/* ... */
}
/* Restore previous setting. */
#pragma opt_unroll_loops reset
void fast_func_B(void)
{
/* ... */
}
```

23.3 Determining Which Settings are Saved and Restored

Not all pragma settings are saved and restored by pragmas `push` and `pop`. Pragmas that do not change compiler settings are not affected by `push` and `pop`. For example, pragma `message` cannot be saved and restored.

The following listing shows an example that checks if the `ANSI_strict` pragma setting is saved and restored by pragmas `push` and `pop`.

Listing: Testing if pragmas push and pop save and restore a setting

```
/* Preprocess this source code. */

#pragma ANSI_strict on

#pragma push

#pragma ANSI_strict off

#pragma pop

#if __option(ANSI_strict)

#error "Saved and restored by push and pop."

#else

#error "Not affected by push and pop."

#endif
```

23.4 Invalid Pragmas

If you enable the compiler's setting for reporting invalid pragmas, the compiler issues a warning when it encounters a pragma it does not recognize. For example, the pragma statements in The following listing generate warnings with the invalid pragmas setting enabled.

Listing: Invalid Pragmas

```
#pragma silly_data off       // WARNING: silly_data is not a pragma.

#pragma ANSI_strict select  // WARNING: select is not defined

#pragma ANSI_strict on      // OK
```

The following table shows how to control the recognition of invalid pragmas.

Table 23-1. Controlling invalid pragmas

To control this option from here...	use this setting
CodeWarrior IDE	**Illegal Pragmas** in the **C/C++ Build > Settings > S12Z Compiler > Warnings panel**
source code	`#pragma warn_illpragma`
command line	`-warnings illpragmas`

23.5 Pragma Scope

The scope of a pragma setting is limited to a compilation unit (a source code file and the files that it includes).

At the beginning of compilation unit, the compiler uses its default settings. The compiler then uses the settings specified by the CodeWarrior IDE's build target or in command-line options.

The compiler uses the setting in a pragma beginning at the pragma's location in the compilation unit. The compiler continues using this setting:

- until another instance of the same pragma appears later in the source code
- until an instance of pragma `pop` appears later in the source code
- until the compiler finishes translating the compilation unit

Chapter 24
Pragmas for Standard C Conformance

This chapter lists the following pragmas for standard C conformance:

- ANSI_strict
- c99
- c9x
- ignore_oldstyle
- only_std_keywords
- require_prototypes

24.1 ANSI_strict

Controls the use of non-standard language features.

Syntax

```
#pragma ANSI_strict on | off | reset
```

Remarks

If you enable the pragma `ANSI_strict`, the compiler generates an error message if it encounters some CodeWarrior extensions to the C language defined by the ISO/IEC 9899-1990 ("C90") standard:

- C++-style comments
- unnamed arguments in function definitions
- non-standard keywords

24.2 c99

Controls the use of a subset of ISO/IEC 9899-1999 ("C99") language features.

Syntax

```
#pragma c99 on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts many of the language features described by the ISO/IEC 9899-1999 standard:

- More rigid type checking.
- Trailing commas in enumerations.
- GCC/C99-style compound literal values.
- Designated initializers.
- `__func__` predefined symbol.
- Implicit `return 0;` in `main()`.
- Non- `const` static data initializations.
- Variable argument macros (`__VA_ARGS__`).
- `_Bool` support.
- `long long` support (separate switch).
- `restrict` support.
- `//` comments.
- `inline` support.
- Digraphs.
- `_Complex` and `_Imaginary` (treated as keywords but not supported).
- Empty arrays as last struct members.
- Designated initializers
- Hexadecimal floating-point constants.
- Variable length arrays are supported within local or function prototype scope (as required by the C99 standard).
- Unsuffixed decimal constant rules.
- `++bool--` expressions.
- `(T) (int-list)` are handled/parsed as cast-expressions and as literals.
- `__STDC_HOSTED__` is `1`.

24.3 c9x

Equivalent to `#pragma c99`.

24.4 ignore_oldstyle

Controls the recognition of function declarations that follow the syntax conventions used before ISO/IEC standard C (in other words, "K&R" style).

Syntax

```
#pragma ignore_oldstyle on | off | reset
```

Remarks

If you enable this pragma, the compiler ignores old-style function declarations and lets you prototype a function any way you want. In old-style declarations, you specify the types of arguments on separate lines instead of the function's argument list. For example, the code in the following listing defines a prototype for a function with an old-style definition.

Listing: Mixing Old-style and Prototype Function Declarations

```
int f(char x, short y, float z);
#pragma ignore_oldstyle on
f(x, y, z)
char x;
short y;
float z;
{
 return (int)x+y+z;
}
#pragma ignore_oldstyle reset
```

This pragma does not correspond to any panel setting. By default, this setting is disabled.

24.5 only_std_keywords

Controls the use of ISO/IEC keywords.

Syntax

```
#pragma only_std_keywords on | off | reset
```

Remarks

The compiler recognizes additional reserved keywords. If you are writing source code that must follow the ISO/IEC C standards strictly, enable the pragma `only_std_keywords`.

24.6 require_prototypes

Controls whether or not the compiler should expect function prototypes.

Syntax

```
#pragma require_prototypes on | off | reset
```

Remarks

This pragma affects only non-static functions.

If you enable this pragma, the compiler generates an error message when you use a function that does not have a preceding prototype. Use this pragma to prevent error messages, caused by referring to a function before you define it. For example, without a function prototype, you might pass data of the wrong type. As a result, your code might not work as you expect even though it compiles without error.

In the following listing, the `main()` function calls `PrintNum()` with an integer argument, even though the `PrintNum()` takes an argument of the type `float`.

Listing: Unnoticed Type-mismatch

```
#include <stdio.h>
void main(void)
{
 PrintNum(1); /* PrintNum() tries to interpret the
 integer as a float. Prints 0.000000. */
}
void PrintNum(float x)
{
 printf("%f\n", x);
}
```

When you run this program, you could get this result:

```
0.000000
```

Although the compiler does not complain about the type mismatch, the function does not give the result you desired. Since `PrintNum()` does not have a prototype, the compiler does not know how to generate the instructions to convert the integer, to a floating-point number before calling `PrintNum()`. Consequently, the function interprets the bits it received as a floating-point number and prints nonsense.

A prototype for `PrintNum()`, as in the following listing, gives the compiler sufficient information about the function to generate instructions to properly convert its argument to a floating-point number. The function prints what you expected.

Listing: Using a Prototype to Avoid Type-mismatch

```
#include <stdio.h>
void PrintNum(float x); /* Function prototype. */
void main(void)
```

```
{
 PrintNum(1); /* Compiler converts int to float.
} Prints 1.000000. */
void PrintNum(float x)
{
 printf("%f\n", x);
}
```

In other situations where automatic conversion is not possible, the compiler generates an error message if an argument does not match the data type required by a function prototype. Such a mismatched data type error is easier to locate at compile time than at runtime.

Chapter 25
Pragmas for C++

This chapter lists the following pragmas for C++:

- access_errors
- always_inline
- arg_dep_lookup
- ARM_conform
- ARM_scoping
- array_new_delete
- auto_inline
- bool
- cplusplus
- cpp1x
- cpp_extensions
- debuginline
- def_inherited
- defer_codegen
- defer_defarg_parsing
- direct_destruction
- direct_to_som
- dont_inline
- ecplusplus
- exceptions
- extended_errorcheck
- inline_bottom_up
- inline_bottom_up_once
- inline_depth
- inline_max_auto_size
- inline_max_size
- inline_max_total_size
- internal
- iso_templates
- new_mangler
- no_conststringconv

- no_static_dtors
- nosyminline
- old_friend_lookup
- old_pods
- old_vtable
- opt_classresults
- parse_func_templ
- parse_mfunc_templ
- RTTI
- suppress_init_code
- template_depth
- thread_safe_init
- warn_hidevirtual
- warn_no_explicit_virtual
- warn_no_typename
- warn_notinlined
- warn_structclass
- wchar_type

25.1 access_errors

Controls whether or not to change invalid access errors to warnings.

Syntax

```
#pragma access_errors on | off | reset
```

Remarks

If you enable this pragma, the compiler issues an error message instead of a warning when it detects invalid access to protected or private class members.

This pragma does not correspond to any IDE panel setting. By default, this pragma is `on`.

25.2 always_inline

Controls the use of inlined functions.

Syntax

```
#pragma always_inline on | off | reset
```

Remarks

This pragma is deprecated. We recommend that you use the `inline_depth()` pragma instead.

25.3 arg_dep_lookup

Controls C++ argument-dependent name lookup.

Syntax

```
#pragma arg_dep_lookup on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler uses argument-dependent name lookup.

This pragma does not correspond to any IDE panel setting. By default, this setting is `on`.

25.4 ARM_conform

This pragma is no longer available. Use `ARM_scoping` instead.

25.5 ARM_scoping

Controls the scope of variables declared in the expression parts of `if`, `while`, `do`, and `for` statements.

Syntax

```
#pragma ARM_scoping on | off | reset
```

Remarks

If you enable this pragma, any variables you define in the conditional expression of an `if`, `while`, `do`, or `for` statement remain in scope until the end of the block that contains the statement. Otherwise, the variables only remain in scope until the end of that statement. The following listing shows an example.

Listing: Example of Using Variables Declared in for Statement

```
for(int i=1; i<1000; i++) { /* . . . */ }
return i; // OK if ARM_scoping is on, error if ARM_scoping is off.
```

25.6 array_new_delete

Enables the operator `new[]` and `delete[]` in array allocation and deallocation operations, respectively.

Syntax

```
#pragma array_new_delete on | off | reset
```

Remarks

By default, this pragma is `on`.

25.7 auto_inline

Controls which functions to inline.

Syntax

```
#pragma auto_inline on | off | reset
```

Remarks

If you enable this pragma, the compiler automatically chooses functions to inline for you, in addition to functions declared with the `inline` keyword.

Note that if you enable the `dont_inline` pragma, the compiler ignores the setting of the `auto_inline` pragma and does not inline any functions.

25.8 bool

Determines whether or not `bool`, `true`, and `false` are treated as keywords in C++ source code.

Syntax

```
#pragma bool on | off | reset
```

Remarks

If you enable this pragma, you can use the standard C++ `bool` type to represent `true` and `false`. Disable this pragma if `bool`, `true`, or `false` are defined in your source code.

Enabling the `bool` data type and its `true` and `false` values is not equivalent to defining them in source code with `typedef`, `enum`, or `#define`. The C++ `bool` type is a distinct type defined by the ISO/IEC 14882-2003 C++ Standard. Source code that does not treat `bool` as a distinct type might not compile properly.

25.9 cplusplus

Controls whether or not to translate subsequent source code as C or C++ source code.

Syntax

```
#pragma cplusplus on | off | reset
```

Remarks

If you enable this pragma, the compiler translates the source code that follows as C++ code. Otherwise, the compiler uses the suffix of the filename to determine how to compile it. If a file name ends in `.c`, `.h`, or `.pch`, the compiler automatically compiles it as C code, otherwise as C++. Use this pragma only if a file contains both C and C++ code.

NOTE

The CodeWarrior C/C++ compilers do not distinguish between uppercase and lowercase letters in file names and file name extensions except on UNIX-based systems.

25.10 cpp1x

Controls whether or not to enable support to experimental features made available in the 1x version of C++ standard.

Syntax

```
#pragma cpp1x on | off | reset
```

Remarks

If you enable this pragma, you can use the following extensions to the 1x or 05 version of the C++ standard that would otherwise be invalid:

- Enables support for `__alignof__`.
- Enables support for `__decltype__`, which is a reference type preserving typeof.
- Enables support for `nullptr`.
- Enables support to allow `>>` to terminate nested template argument lists.
- Enables support for `__static_assert`.

NOTE

This pragma enables support to experimental and unvalidated implementations of features that may or may not be available in the final version of the C++ standard. The features should not be used for critical or production code.

25.11 cpp_extensions

Controls language extensions to ISO/IEC 14882-2003 C++.

Syntax

```
#pragma cpp_extensions on | off | reset
```

Remarks

If you enable this pragma, you can use the following extensions to the ISO/IEC 14882-2003 C++ standard that would otherwise be invalid:

- Anonymous `struct` & `union` objects. The following listing shows an example.

  Listing: Example of Anonymous struct & union Objects

```
#pragma cpp_extensions on
void func()
{
 union {
 long hilo;
 struct { short hi, lo; }; // anonymous struct
 };
 hi=0x1234;
 lo=0x5678; // hilo==0x12345678
}
```

- Unqualified pointer to a member function. The following listing shows an example.

  Listing: Example of an Unqualified Pointer to a Member Function

```
#pragma cpp_extensions on
struct RecA { void f(); }
void RecA::f()
```

```
{
 void (RecA::*ptmf1)() = &RecA::f; // ALWAYS OK
 void (RecA::*ptmf2)() = f; // OK if you enable cpp_extensions.
}
```

- Inclusion of `const` data in precompiled headers.

25.12 debuginline

Controls whether the compiler emits debugging information for expanded inline function calls.

Syntax

```
#pragma debuginline on | off | reset
```

Remarks

If the compiler emits debugging information for inline function calls, then the debugger can step to the body of the inlined function. This behavior more closely resembles the debugging experience for un-inlined code.

NOTE

Since the actual "call" and "return" instructions are no longer present when stepping through inline code, the debugger will immediately jump to the body of an inlined function and "return" before reaching the return statement for the function. Thus, the debugging experience of inlined functions may not be as smooth as debugging un-inlined code.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

25.13 def_inherited

Controls the use of `inherited`.

Syntax

```
#pragma def_inherited on | off | reset
```

Remarks

The use of this pragma is deprecated. It lets you use the non-standard `inherited` symbol in C++ programming by implicitly adding

```
typedef base inherited;
```

as the first member in classes with a single base class.

NOTE

The ISO/IEC 14882-2003 C++ standard does not support the `inherited` symbol. Only the CodeWarrior C++ language implements the `inherited` symbol for single inheritance.

25.14 defer_codegen

Obsolete pragma. Replaced by interprocedural analysis options. For more information, refer to the topic Interprocedural Analysis.

25.15 defer_defarg_parsing

Defers the parsing of default arguments in member functions.

Syntax

```
#pragma defer_defarg_parsing on | off
```

Remarks

To be accepted as valid, some default expressions with template arguments will require additional parenthesis. For example, the following listing results in an error message.

Listing: Deferring parsing of default arguments

```
template<typename T,typename U> struct X { T t; U u; };
struct Y {
 // The following line is not accepted, and generates
 // an error message with defer_defarg_parsing on.
 void f(X<int,int> = X<int,int>());
};
```

The following listing does not generate an error message.

Listing: Correct default argument deferral

```
template<typename T,typename U> struct X { T t; U u; };
struct Y {
 // The following line is OK if the default
 // argument is parenthesized.
 void f(X<int,int> = (X<int,int>()) );
};
```

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

25.16 direct_destruction

This pragma is obsolete. It is no longer available.

25.17 direct_to_som

This pragma is obsolete. It is no longer available.

25.18 dont_inline

Controls the generation of inline functions.

Syntax

```
#pragma dont_inline on | off | reset
```

Remarks

If you enable this pragma, the compiler does not inline any function calls, even those declared with the `inline` keyword or within a class declaration. Also, it does not automatically inline functions, regardless of the setting of the `auto_inline` pragma, described in the topic auto_inline. If you disable this pragma, the compiler expands all inline function calls, within the limits you set through other inlining-related pragmas.

25.19 ecplusplus

Controls the use of embedded C++ features.

Syntax

```
#pragma ecplusplus on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler disables the non-EC++ features of ISO/IEC 14882-2003 C++ such as templates, multiple inheritance, and so on.

25.20 exceptions

Controls the availability of C++ exception handling.

Syntax

```
#pragma exceptions on | off | reset
```

Remarks

If you enable this pragma, you can use the `try` and `catch` statements in C++ to perform exception handling. If your program does not use exception handling, disable this setting to make your program smaller.

You can throw exceptions across any code compiled by the CodeWarrior C/C++ compiler with `#pragma exceptions on`.

You cannot throw exceptions across libraries compiled with `#pragma exceptions off`. If you throw an exception across such a library, the code calls `terminate()` and exits.

This pragma corresponds to the **Enable C++ Exceptions** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is `on`.

25.21 extended_errorcheck

Controls the issuing of warning messages for possible unintended logical errors.

Syntax

```
#pragma extended_errorcheck on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler generates a warning message for the possible unintended logical errors.

It also issues a warning message when it encounters a delete operator for a class or structure that has not been defined yet. The following listing shows an example.

Listing: Attempting to delete an undefined structure

```
#pragma extended_errorcheck on

struct X;
```

```
int func(X *xp)
{
     delete xp;    // Warning: deleting incomplete type X
}
```

- An empty `return` statement in a function that is not declared `void`. For example, the following listing results in a warning message.

Listing: A non-void function with an empty return statement

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err != -1)
  {
    err = GetMoreResources();
  }
  return; /* WARNING: empty return statement */
}
```

The following listing shows how to prevent this warning message.

Listing: A non-void function with a proper return statement

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err != -1)
 {
    err = GetMoreResources();
 }
  return err; /* OK */
}
```

25.22 inline_bottom_up

Controls the bottom-up function inlining method.

Syntax

```
#pragma inline_bottom_up on | off | reset
```

Remarks

Bottom-up function inlining tries to expand up to eight levels of inline leaf functions. The maximum size of an expanded inline function and the caller of an inline function can be controlled by the pragmas shown in the following listings.

Listing: Maximum Complexity of an Inlined Function

```
// Maximum complexity of an inlined function
#pragma inline_max_size(
max
 ) // default
max
 == 256
```

Listing: Maximum Complexity of a Function that Calls Inlined Functions

```
// Maximum complexity of a function that calls inlined functions
#pragma inline_max_total_size(
max
 ) // default
max
 == 10000
```

where *max* loosely corresponds to the number of instructions in a function.

If you enable this pragma, the compiler calculates inline depth from the last function in the call chain up to the first function that starts the call chain. The number of functions the compiler inlines from the bottom depends on the values of `inline_depth`, `inline_max_size`, and `inline_max_total_size`. This method generates faster and smaller source code for some (but not all) programs with many nested inline function calls.

If you disable this pragma, top-down inlining is selected, and the `inline_depth` setting determines the limits for top-down inlining. The `inline_max_size` and `inline_max_total_size` pragmas do not affect the compiler in top-down mode.

25.23 inline_bottom_up_once

Performs a single bottom-up function inlining operation.

Syntax

```
#pragma inline_bottom_up_once on | off | reset
```

Remarks

By default, this pragma is `off`.

25.24 inline_depth

Controls how many passes are used to expand inline function calls.

Syntax

```
#pragma inline_depth(n)

#pragma inline_depth(smart)
```

Parameters

`n`

Sets the number of passes used to expand inline function calls. The number *n* is an integer from `0` to `1024` or the `smart` specifier. It also represents the distance allowed in the call chain from the last function up. For example, if `d` is the total depth of a call chain, then functions below a depth of `d-n` are inlined if they do not exceed the following size settings:

```
#pragma inline_max_size(n);

#pragma inline_max_total_size(n);
```

The first pragma sets the maximum function size to be considered for inlining; the second sets the maximum size to which a function is allowed to grow after the functions it calls are inlined. Here, *n* is the number of statements, operands, and operators in the function, which turns out to be roughly twice the number of instructions generated by the function. However, this number can vary from function to function. For the `inline_max_size` pragma, the default value of *n* is 256; for the `inline_max_total_size` pragma, the default value of *n* is 10000.

`smart`

The `smart` specifier is the default mode, with four passes where the passes 2-4 are limited to small inline functions. All inlineable functions are expanded if `inline_depth` is set to 1-1024.

25.25 inline_max_auto_size

Determines the maximum complexity for an auto-inlined function.

Syntax

```
#pragma inline_max_auto_size ( complex )
```

Parameters

`complex`

The `complex` value is an approximation of the number of statements in a function, the current default value is 15. Selecting a higher value will inline more functions, but can lead to excessive code bloat.

Remarks

This pragma does not correspond to any panel setting.

25.26 inline_max_size

Sets the maximum number of statements, operands, and operators used to consider the function for inlining.

Syntax

```
#pragma inline_max_size ( size )
```

Parameters

`size`

The maximum number of statements, operands, and operators in the function to consider it for inlining, up to a maximum of 256.

Remarks

This pragma does not correspond to any panel setting.

25.27 inline_max_total_size

Sets the maximum total size a function can grow to when the function it calls is inlined.

Syntax

```
#pragma inline_max_total_size ( max_size )
```

Parameters

`max_size`

The maximum number of statements, operands, and operators the inlined function calls that are also inlined, up to a maximum of 7000.

Remarks

This pragma does not correspond to any panel setting.

25.28 internal

Controls the internalization of data or functions.

Syntax

```
#pragma internal on | off | reset
```

```
#pragma internal list name1 [, name2 ]*
```

Remarks

When using the `#pragma internal on` format, all data and functions are automatically internalized.

Use the `#pragma internal list` format to tag specific data or functions for internalization. It applies to all names if it is used on an overloaded function. You cannot use this pragma for C++ member functions or static class members.

The following listing shows an example:

Listing: Example of an Internalized List

```
extern int f(), g;
#pragma internal list f,g
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

25.29 iso_templates

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler and issue warning messages for missing typenames.

Syntax

```
#pragma iso_templates on | off | reset
```

Remarks

This pragma combines the functionality of pragmas parse_func_templ, parse_mfunc_templ and warn_no_typename.

This pragma ensures that your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions. The compiler issues a warning message if a typenames required by the C++ standard is missing but can still be determined by the compiler based on the context of the surrounding C++ syntax.

By default, this pragma is `on`.

25.30 new_mangler

Controls the inclusion or exclusion of a template instance's function return type, to the mangled name of the instance.

Syntax

```
#pragma new_mangler on | off | reset
```

Remarks

The C++ standard requires that the function return type of a template instance to be included in the mangled name, which can cause incompatibilities. Enabling this pragma within a prefix file resolves those incompatibilities.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

25.31 no_conststringconv

Disables the deprecated implicit const string literal conversion (ISO/IEC 14882-2003 C++, §4.2).

Syntax

```
#pragma no_conststringconv on | off | reset
```

Remarks

When enabled, the compiler generates an error message when it encounters an implicit const string conversion.

Listing: Example of const string conversion

```
#pragma no_conststringconv on
char *cp = "Hello World"; /* Generates an error message. */
```

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

25.32 no_static_dtors

Controls the generation of static destructors in C++.

Syntax

```
#pragma no_static_dtors on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate destructor calls for static data objects. Use this pragma to generate smaller object code for C++ programs that never exit (and consequently never need to call destructors for static objects).

This pragma does not correspond to any panel setting. By default, this setting is disabled.

25.33 nosyminline

Controls whether debug information is gathered for inline/template functions.

Syntax

```
#pragma nosyminline on | off | reset
```

Remarks

When on, debug information is not gathered for inline/template functions.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

25.34 old_friend_lookup

Implements non-standard C++ friend declaration behavior that allows friend declarations to be visible in the enclosing scope.

```
#pragma old_friend_lookup on | off | reset
```

Example

This example shows friend declarations that are invalid without #pragma old_friend_lookup.

Listing: Valid and invalid declarations without #pragma old_friend_lookup

```
class C2;
void f2();
struct S {
 friend class C1;
 friend class C2;
 friend void f1();
 friend void f2();
};
C1 *cp1; // error, C1 is not visible without namespace declaration
C2 *cp2; // OK
int main()
{
 f1(); // error, f1() is not visible without namespace declaration
 f2(); // OK
}
```

25.35 old_pods

Permits non-standard handling of classes, structs, and unions containing pointer-to-pointer members

Syntax

```
#pragma old_pods on | off | reset
```

Remarks

According to the ISO/IEC 14882:2003 C++ Standard, classes/structs/unions that contain pointer-to-pointer members are now considered to be plain old data (POD) types.

This pragma can be used to get the old behavior.

25.36 old_vtable

This pragma is no longer available.

25.37 opt_classresults

Controls the omission of the copy constructor call for class return types if all return statements in a function return the same local class object.

Syntax

```
#pragma opt_classresults on | off | reset
```

Remarks

The following listing shows an example.

Listing: Example #pragma opt_classresults

```
#pragma opt_classresults on
struct X {
 X();
 X(const X&);
 // ...
};
X f() {
 X x; // Object x will be constructed in function result buffer.
 // ...
 return x; // Copy constructor is not called.
}
```

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

25.38 parse_func_templ

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler.

Syntax

```
#pragma parse_func_templ on | off | reset
```

Remarks

If you enable this pragma, your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions.

This option actually corresponds to the **ISO C++ Template Parser** option (together with pragmas `parse_func_templ` and warn_no_typename). By default, this pragma is disabled.

25.39 parse_mfunc_templ

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler for member function bodies.

Syntax

```
#pragma parse_mfunc_templ on | off | reset
```

Remarks

If you enable this pragma, member function bodies within your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

25.40 RTTI

Controls the availability of runtime type information.

Syntax

```
#pragma RTTI on | off | reset
```

Remarks

If you enable this pragma, you can use runtime type information (or RTTI) features such as `dynamic_cast` and `typeid`.

NOTE

Note that `*type_info::before(const type_info&)` is not implemented.

25.41 suppress_init_code

Controls the suppression of static initialization object code.

Syntax

```
#pragma suppress_init_code on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate any code for static data initialization such as C++ constructors.

Warning

Using this pragma can produce erratic or unpredictable behavior in your program.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

25.42 template_depth

Controls how many nested or recursive class templates you can instantiate.

```
#pragma template_depth(n)
```

Remarks

This pragma lets you increase the number of nested or recursive class template instantiations allowed. By default, *n* equals 64; it can be set from 1 to 30000. You should always use the default value unless you receive the error message

```
template too complex or recursive
```

This pragma does not correspond to any panel setting.

25.43 thread_safe_init

Controls the addition of extra code in the binary to ensure that multiple threads cannot enter a static local initialization at the same time.

Syntax

```
#pragma thread_safe_init on | off | reset
```

Remarks

A C++ program that uses multiple threads and static local initializations introduces the possibility of contention over which thread initializes static local variable first. When the pragma is `on`, the compiler inserts calls to mutex functions around each static local initialization to avoid this problem. The C++ runtime library provides these mutex functions.

Listing: Static local initialization example

```
int func(void) {
 // There may be synchronization problems if this function is
 // called by multiple threads.
 static int countdown = 20;
 return countdown--;
}
```

NOTE

This pragma requires runtime library functions which may not be implemented on all platforms, due to the possible need for operating system support.

The following listing shows another example.

Listing: Example thread_safe_init

```
#pragma thread_safe_init on
void thread_heavy_func()
{
 // Multiple threads can now safely call this function:
 // the static local variable will be constructed only once.
 static std::string localstring = thread_unsafe_func();
}
```

NOTE

When an exception is thrown from a static local initializer, the initializer is retried by the next client that enters the scope of the local.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

25.44 warn_hidevirtual

Controls the recognition of a non-virtual member function that hides a virtual function in a superclass.

Syntax

```
#pragma warn_hidevirtual on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if you declare a non-virtual member function that hides a virtual function in a superclass. One function hides another if it has the same name but a different argument type. The following listing shows an example.

Listing: Hidden Virtual Functions

```
class A {
 public:
 virtual void f(int);
 virtual void g(int);
};
class B: public A {
 public:
 void f(char); // WARNING: Hides A::f(int)
 virtual void g(int); // OK: Overrides A::g(int)
};
```

The ISO/IEC 14882-2003 C++ Standard does not require this pragma.

NOTE

A warning message normally indicates that the pragma name is not recognized, but an error indicates either a syntax problem or that the pragma is not valid in the given context.

25.45 warn_no_explicit_virtual

Controls the issuing of warning messages if an overriding function is not declared with a virtual keyword.

Syntax

```
#pragma warn_no_explicit_virtual on | off | reset
```

Remarks

The following listing shows an example.

Listing: Example of warn_no_explicit_virtual pragma

```
#pragma warn_no_explicit_virtual on
struct A {
 virtual void f();
};
struct B {
 void f();
 // WARNING: override B::f() is declared without virtual keyword
}
```

Tip

This warning message is not required by the ISO/IEC 14882-2003 C++ standard, but can help you track down unwanted overrides.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

25.46 warn_no_typename

Controls the issuing of warning messages for missing `typenames`.

Syntax

```
#pragma warn_no_typename on | off | reset
```

Remarks

The compiler issues a warning message if a `typenames` required by the C++ standard is missing but can still be determined by the compiler based on the context of the surrounding C++ syntax.

This pragma does not correspond to any panel setting. This pragma is enabled by the ISO/IEC 14882-2003 C++ template parser.

25.47 warn_notinlined

Controls the issuing of warning messages for functions the compiler cannot inline.

Syntax

```
#pragma warn_notinlined on | off | reset
```

Remarks

The compiler issues a warning message for non-inlined, (i.e., on those indicated by the `inline` keyword or in line in a class declaration) inline function calls.

25.48 warn_structclass

Controls the issuing of warning messages for the inconsistent use of the `class` and `struct` keywords.

Syntax

```
#pragma warn_structclass on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if you use the `class` and `struct` keywords in the definition and declaration of the same identifier.

Listing: Inconsistent use of <codeph>class</codeph> and <codeph>struct</codeph>

```
class X;
struct X { int a; }; // WARNING
```

Use this warning when using static or dynamic libraries to link with object code produced by another C++ compiler that distinguishes between class and structure variables in its name " mangling."

25.49 wchar_type

Controls the availability of the wchar_t data type in C++ source code.

Syntax

```
#pragma wchar_type on | off | reset
```

Remarks

If you enable this pragma, `wchar_t` is treated as a built-in type. Otherwise, the compiler does not recognize this type.

This pragma corresponds to the **Enable wchar_t Support** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is enabled.

Chapter 26
Pragmas for Language Translation

The chapter lists the following pragmas for language translation:

- asmsemicolcomment
- gcc_extensions
- mpwc_newline
- mpwc_relax
- multibyteaware
- multibyteaware_preserve_literals
- text_encoding
- trigraphs
- unsigned_char

26.1 asmsemicolcomment

Controls whether the " `;` " symbol is treated as a comment character in inline assembly.

Syntax

```
#pragma asmsemicolcomment on | off | reset
```

Remarks

Some targets may have additional comment characters, and may treat these characters as comments even when

```
#pragma asmsemicolcomment off
```

is used.

Using this pragma may interfere with the assembly language of a specific target.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

26.2 gcc_extensions

Controls the acceptance of GNU C language extensions.

Syntax

```
#pragma gcc_extensions on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts GNU C extensions in C source code. This includes the following non-ANSI C extensions:

- Initialization of automatic `struct` or `array` variables with non- `const` values.
- Illegal pointer conversions
- `sizeof( void ) == 1`
- `sizeof( function-type ) == 1`
- Limited support for GCC statements and declarations within expressions.
- Macro redefinitions without a previous `#undef`
- The GCC keyword `typeof`
- Function pointer arithmetic supported
- `void*` arithmetic supported
- Void expressions in return statements of void
- `__builtin_constant_p (`*expr*`)` supported
- Forward declarations of arrays of incomplete type
- Forward declarations of empty static arrays
- Pre-C99 designated initializer syntax (deprecated)
- shortened conditional expression (*c* `?:` *y*)
- `long __builtin_expect (long exp, long c)` now accepted

26.3 mpwc_newline

Controls the use of newline character convention.

Syntax

```
#pragma mpwc_newline on | off | reset
```

Remarks

If you enable this pragma, the compiler translates `'\n'` as a Carriage Return (0x0D) and `'\r'` as a Line Feed (0x0A). Otherwise, the compiler uses the ISO standard conventions for these characters.

If you enable this pragma, use ISO standard libraries that were compiled when this pragma was enabled.

If you enable this pragma and use the standard ISO standard libraries, your program will not read and write `'\n'` and `'\r'` properly. For example, printing `'\n'` brings your program's output to the beginning of the current line instead of inserting a newline.

This pragma does not correspond to any IDE panel setting. By default, this pragma is disabled.

26.4 mpwc_relax

Controls the compatibility of the `char*` and `unsigned char*` types.

Syntax

```
#pragma mpwc_relax on | off | reset
```

Remarks

If you enable this pragma, the compiler treats `char*` and `unsigned char*` as the same type. Use this setting to compile source code written before the ISO C standards. Old source code frequently uses these types interchangeably.

This setting has no effect on C++ source code.

NOTE

Turning this option on may prevent the compiler from detecting some programming errors. We recommend not turning on this option.

The following listing shows how to use this pragma to relax function pointer checking.

Listing: Relaxing function pointer checking

```
#pragma mpwc_relax on
extern void f(char *);
/* Normally an error, but allowed. */
extern void(*fp1)(void *) = &f;
/* Normally an error, but allowed. */
extern void(*fp2)(unsigned char *) = &f;
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

26.5 multibyteaware

Controls how the **Source encoding** option in the IDE is treated

Syntax

```
#pragma multibyteaware on | off | reset
```

Remarks

This pragma is deprecated. See `#pragma text_encoding` for more details.

26.6 multibyteaware_preserve_literals

Controls the treatment of multibyte character sequences in narrow character string literals.

Syntax

```
#pragma multibyteaware_preserve_literals on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

26.7 text_encoding

Identifies the character encoding of source files.

Syntax

```
#pragma text_encoding ( "name" | unknown | reset [, global] )
```

Parameters

`name`

The IANA or MIME encoding name or an OS-specific string that identifies the text encoding. The compiler recognizes these names and maps them to its internal decoders:

```
system US-ASCII ASCII ANSI_X3.4-1968
```

```
ANSI_X3.4-1968 ANSI_X3.4 UTF-8 UTF8 ISO-2022-JP

CSISO2022JP ISO2022JP CSSHIFTJIS SHIFT-JIS

SHIFT_JIS SJIS EUC-JP EUCJP UCS-2 UCS-2BE

UCS-2LE UCS2 UCS2BE UCS2LE UTF-16 UTF-16BE

UTF-16LE UTF16 UTF16BE UTF16LE UCS-4 UCS-4BE

UCS-4LE UCS4 UCS4BE UCS4LE 10646-1:1993

ISO-10646-1 ISO-10646 unicode

global
```

Tells the compiler that the current and all subsequent files use the same text encoding. By default, text encoding is effective only to the end of the file.

Remarks

By default, `#pragmatext_encoding` is only effective through the end of file. To affect the default text encoding assumed for the current and all subsequent files, supply the "global" modifier.

26.8 trigraphs

Controls the use trigraph sequences specified in the ISO standards.

Syntax

```
#pragma trigraphs on | off | reset
```

Remarks

If you are writing code that must strictly adhere to the ANSI standard, enable this pragma.

Table 26-1. Trigraph table

Trigraph	Character
??=	#
??/	\
??'	^
??(	[
??)	]

Table continues on the next page...

Table 26-1. Trigraph table (continued)

Trigraph	Character
??!	|
??<	{
??>	}
??-	~

> **NOTE**
> Use of this pragma may cause a portability problem for some targets.

Be careful when initializing strings or multi-character constants that contain question marks.

Listing: Example of Pragma trigraphs

```
char c = '????'; /* ERROR: Trigraph sequence expands to '??^ */
char d = '\?\?\?\?'; /* OK */
```

26.9 unsigned_char

Controls whether or not declarations of type `char` are treated as `unsigned char`.

Syntax

```
#pragma unsigned_char on | off | reset
```

Remarks

If you enable this pragma, the compiler treats a `char` declaration as if it were an `unsigned char` declaration.

> **NOTE**
> If you enable this pragma, your code might not be compatible with libraries that were compiled when the pragma was disabled. In particular, your code might not work with the ISO standard libraries included with CodeWarrior.

Chapter 27
Pragmas for Diagnostic Messages

The chapter lists the following pragmas for diagnostic messages:

- extended_errorcheck
- maxerrorcount
- message
- showmessagenumber
- show_error_filestack
- suppress_warnings
- sym
- unused
- warning
- warning_errors
- warn_any_ptr_int_conv
- warn_emptydecl
- warn_extracomma
- warn_filenamecaps
- warn_filenamecaps_system
- warn_hiddenlocals
- warn_illpragma
- warn_illtokenpasting
- warn_illunionmembers
- warn_impl_f2i_conv
- warn_impl_i2f_conv
- warn_impl_s2u_conv
- warn_implicitconv
- warn_largeargs
- warn_missingreturn
- warn_no_side_effect
- warn_padding
- warn_pch_portability
- warn_possunwant
- warn_ptr_int_conv

- warn_resultnotused
- warn_undefmacro
- warn_uninitializedvar
- warn_possiblyuninitializedvar
- warn_unusedarg
- warn_unusedvar

27.1 extended_errorcheck

Controls the issuing of warning messages for possible unintended logical errors.

Syntax

```
#pragma extended_errorcheck on | off | reset
```

Remarks

If you enable this pragma, the compiler generates a warning message (not an error) if it encounters some common programming errors:

- An integer or floating-point value assigned to an `enum` type. The following listing shows an example.

  Listing: Assigning to an Enumerated Type

  ```
  enum Day { Sunday, Monday, Tuesday, Wednesday,
             Thursday, Friday, Saturday } d;
  d = 5; /* WARNING */
  d = Monday; /* OK */
  d = (Day)3; /* OK */
  ```

- An empty `return` statement in a function that is not declared `void`. For example, the following listing results in a warning message.

  Listing: A non-void function with an empty return statement

  ```
  int MyInit(void)
  {
    int err = GetMyResources();
    if (err != -1)
    {
      err = GetMoreResources();
    }
    return; /* WARNING: empty return statement */
  ```

```
}
```

The following listing shows how to prevent this warning message.

Listing: A non-void function with a proper return statement

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err != -1)
  {
    err = GetMoreResources();
  }
  return err; /* OK */
}
```

27.2 maxerrorcount

Limits the number of error messages emitted while compiling a single file.

Syntax

```
#pragma maxerrorcount(num | off )
```

Parameters

num

Specifies the maximum number of error messages issued per source file.

`off`

Does not limit the number of error messages issued per source file.

Remarks

The total number of error messages emitted may include one final message:

```
Too many errors emitted
```

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

27.3 message

Tells the compiler to issue a text message to the user.

Syntax

```
#pragma message(msg)
```

Parameter

msg

Actual message to issue. Does not have to be a string literal.

Remarks

On the command line, the message is sent to the standard error stream.

27.4 showmessagenumber

Controls the appearance of warning or error numbers in displayed messages.

Syntax

```
#pragma showmessagenumber on | off | reset
```

Remarks

When enabled, this pragma causes messages to appear with their numbers visible. You can then use the `warning` pragma with a warning number to suppress the appearance of specific warning messages.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

27.5 show_error_filestack

Controls the appearance of the current `# include` file stack within error messages occurring inside deeply-included files.

Syntax

```
#pragma show_error_filestack on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

27.6 suppress_warnings

Controls the issuing of warning messages.

Syntax

```
#pragma suppress_warnings on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate warning messages, including those that are enabled.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

27.7 sym

Controls the generation of debugger symbol information for subsequent functions.

Syntax

```
#pragma sym on | off | reset
```

Remarks

The compiler pays attention to this pragma only if you enable the debug marker for a file in the IDE project window. If you disable this pragma, the compiler does not put debugging information into the source file debugger symbol file (SYM or DWARF) for the functions that follow.

The compiler always generates a debugger symbol file for a source file that has a debug diamond next to it in the IDE project window. This pragma changes only which functions have information in that symbol file.

This pragma does not correspond to any panel setting. By default, this pragma is enabled.

27.8 unused

Controls the suppression of warning messages for variables and parameters that are not referenced in a function.

Syntax

```
#pragma unused ( var_name [, var_name ]... )
```

var_name

The name of a variable.

Remarks

This pragma suppresses the compile time warning messages for the unused variables and parameters specified in its argument list. You can use this pragma only within a function body. The listed variables must be within the scope of the function.

In C++, you cannot use this pragma with functions defined within a class definition or with template functions.

Listing: Example of Pragma unused() in C

```
#pragma warn_unusedvar on

#pragma warn_unusedarg on

static void ff(int a)

{

  int b;

#pragma unused(a,b)

/* Compiler does not warn that a and b are unused. */

}
```

Listing: Example of Pragma unused() in C++

```
#pragma warn_unusedvar on

#pragma warn_unusedarg on

static void ff(int /* No warning */)

{

  int b;

#pragma unused(b)

/* Compiler does not warn that b is unused. */

}
```

This pragma does not correspond to any CodeWarrior IDE panel setting.

27.9 warning

Controls which warning numbers are displayed during compiling.

Syntax

```
#pragma warning on | off | reset (num [, ...])
```

This alternate syntax is allowed but ignored (message numbers do not match):

```
#pragma warning(warning_type : warning_num_list [, warning_type : warning_num_list, ...])
```

Parameters

num

The number of the warning message to show or suppress.

warning_type

Specifies one of the following settings:

- `default`
- `disable`
- `enable`

warning_num_list

The *warning_num_list* is a list of warning numbers separated by spaces.

Remarks

Use the pragma `showmessagenumber` to display warning messages with their warning numbers.

This pragma only applies to CodeWarrior front-end warnings. Using the pragma for the Power Architecture back-end warnings returns invalid message number warning.

The CodeWarrior compiler allows, but ignores, the alternative syntax for compatibility with Microsoft® compilers.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

27.10 warning_errors

Controls whether or not warnings are treated as errors.

Syntax

```
#pragma warning_errors on | off | reset
```

Remarks

If you enable this pragma, the compiler treats all warning messages as though they were errors and does not translate your file until you resolve them.

27.11 warn_any_ptr_int_conv

Controls if the compiler generates a warning message when an integral type is explicitly converted to a pointer type or vice versa.

Syntax

```
#pragma warn_any_ptr_int_conv on | off | reset
```

Remarks

This pragma is useful to identify potential 64-bit pointer portability issues. An example is shown in.

Listing: Example of warn_any_ptr_int_conv

```
#pragma warn_ptr_int_conv on
short i, *ip
void func() {
 i = (short)ip;
 /* WARNING: short type is not large enough to hold pointer. */
}
#pragma warn_any_ptr_int_conv on
void bar() {
 i = (int)ip; /* WARNING: pointer to integral conversion. */
 ip = (short *)i; /* WARNING: integral to pointer conversion. */
}
```

27.12 warn_emptydecl

Controls the recognition of declarations without variables.

Syntax

```
#pragma warn_emptydecl on | off | reset
```

Remarks

If you enable this pragma, the compiler displays a warning message when it encounters a declaration with no variables.

Listing: Examples of empty declarations in C and C++

```
#pragma warn_emptydecl on
int ; /* WARNING: empty variable declaration. */
int i; /* OK */
long j;; /* WARNING */
long j; /* OK */
```

Listing: Example of empty declaration in C++

```
#pragma warn_emptydecl on
extern "C" {
}; /* WARNING */
```

27.13 warn_extracomma

Controls the recognition of superfluous commas in enumerations.

Syntax

```
#pragma warn_extracomma on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a trailing comma in enumerations. For example, the following listing is acceptable source code but generates a warning message when you enable this setting.

Listing: Warning about extra commas

```
#pragma warn_extracomma on
enum { mouse, cat, dog, };
/* WARNING: compiler expects an identifier after final comma. */
```

The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 ("C99") standard.

This pragma corresponds to the **Extra Commas** setting in the CodeWarrior IDE's **C/C++ Warnings** settings panel. By default, this pragma is disabled.

27.14 warn_filenamecaps

Controls the recognition of conflicts involving case-sensitive filenames within user includes.

Syntax

```
#pragma warn_filenamecaps on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when an `#include` directive capitalizes a filename within a user include differently from the way the filename appears on a disk. It also detects use of "8.3" DOS filenames in Windows® operating systems when a long filename is available. Use this pragma to avoid porting problems to operating systems with case-sensitive file names.

By default, this pragma only checks the spelling of user includes such as the following:

```
#include "file"
```

For more information on checking system includes, refer to the topic warn_filenamecaps_system.

27.15 warn_filenamecaps_system

Controls the recognition of conflicts involving case-sensitive filenames within system includes.

Syntax

```
#pragma warn_filenamecaps_system on | off | reset
```

Remarks

If you enable this pragma along with `warn_filenamecaps`, the compiler issues a warning message when an `#include` directive capitalizes a filename within a system include differently from the way the filename appears on a disk. It also detects use of "8.3" DOS filenames in Windows® systems when a long filename is available. This pragma helps avoid porting problems to operating systems with case-sensitive file names.

To check the spelling of system includes such as the following:

```
#include <file>
```

Use this pragma along with the warn_filenamecaps pragma.

NOTE

Some SDKs (Software Developer Kits) use "colorful" capitalization, so this pragma may issue a lot of unwanted messages.

27.16 warn_hiddenlocals

Controls the recognition of a local variable that hides another local variable.

Syntax

```
#pragma warn_hiddenlocals on | off | reset
```

Remarks

When `on`, the compiler issues a warning message when it encounters a local variable that hides another local variable. An example appears in the following listing.

Listing: Example of hidden local variables warning

```
#pragma warn_hiddenlocals on
void func(int a)
{
 {
 int a; /* WARNING: this 'a' obscures argument 'a'.
 }
}
```

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this setting is `off`.

27.17 warn_illpragma

Controls the recognition of invalid pragma directives.

Syntax

```
#pragma warn_illpragma on | off | reset
```

Remarks

If you enable this pragma, the compiler displays a warning message when it encounters a pragma it does not recognize.

27.18 warn_illtokenpasting

Controls whether or not to issue a warning message for improper preprocessor token pasting.

Syntax

```
#pragma warn_illtokenpasting on | off | reset
```

Remarks

An example of this is shown below:

```
#define PTR(x) x##* / PTR(y)
```

Token pasting is used to create a single token. In this example, `y` and `x` cannot be combined. Often the warning message indicates the macros uses " `##`" unnecessarily.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

27.19 warn_illunionmembers

Controls whether or not to issue a warning message for invalid union members, such as unions with reference or non-trivial class members.

Syntax

```
#pragma warn_illunionmembers on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

27.20 warn_impl_f2i_conv

Controls the issuing of warning messages for implicit `float`-to- `int` conversions.

Syntax

```
#pragma warn_impl_f2i_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting floating-point values to integral values. The following listing provides an example.

Listing: Example of Implicit <codeph>float</codeph>-to-<codeph>int</codeph> Conversion

```
#pragma warn_impl_f2i_conv on
float f;
signed int si;
int main()
{
 f = si; /* WARNING */
#pragma warn_impl_f2i_conv off
 si = f; /* OK */
}
```

27.21 warn_impl_i2f_conv

Controls the issuing of warning messages for implicit `int`-to- `float` conversions.

Syntax

```
#pragma warn_impl_i2f_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting integral values to floating-point values. The following listing shows an example.

Listing: Example of implicit int-to-float conversion

```
#pragma warn_impl_i2f_conv on
float f;
signed int si;
int main()
{
 si = f; /* WARNING */
#pragma warn_impl_i2f_conv off
 f = si; /* OK */
}
```

27.22 warn_impl_s2u_conv

Controls the issuing of warning messages for implicit conversions between the `signed int` and `unsigned int` data types.

Syntax

```
#pragma warn_impl_s2u_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting either from `signed int` to `unsigned int` or vice versa. The following listing provides an example.

Listing: Example of implicit conversions between signed int and unsigned int

```
#pragma warn_impl_s2u_conv on
signed int si;
unsigned int ui;
int main()
{
 ui = si; /* WARNING */
 si = ui; /* WARNING */
#pragma warn_impl_s2u_conv off
 ui = si; /* OK */
 si = ui; /* OK */
}
```

27.23 warn_implicitconv

Controls the issuing of warning messages for all implicit arithmetic conversions.

Syntax

```
#pragma warn_implicitconv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message for all implicit arithmetic conversions when the destination type might not represent the source value. The following listing provides an example.

Listing: Example of Implicit Conversion

```
#pragma warn_implicitconv on
float f;
signed int si;
unsigned int ui;
int main()
{
 f = si; /* WARNING */
 si = f; /* WARNING */
 ui = si; /* WARNING */
 si = ui; /* WARNING */
}
```

NOTE

This option "opens the gate" for the checking of implicit conversions. The sub-pragmas `warn_impl_f2i_conv`, `warn_impl_i2f_conv`, and `warn_impl_s2u_conv` control the classes of conversions checked.

27.24 warn_largeargs

Controls the issuing of warning messages for passing non-"int" numeric values to unprototyped functions.

Syntax

```
#pragma warn_largeargs on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if you attempt to pass a non-integer numeric value, such as a `float` or `long long`, to an unprototyped function when the `require_prototypes` pragma is disabled.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

27.25 warn_missingreturn

Issues a warning message when a function that returns a value is missing a `return` statement.

Syntax

```
#pragma warn_missingreturn on | off | reset
```

Remarks

An example is shown in the following figure.

Listing: Example of warn_missingreturn pragma

```
#pragma warn_missingreturn on
int func()
{
 /* WARNING: no return statement. */
}
```

27.26 warn_no_side_effect

Controls the issuing of warning messages for redundant statements.

Syntax

```
#pragma warn_no_side_effect on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a statement that produces no side effect. To suppress this warning message, cast the statement with `(void)`. The following listing provides an example.

Listing: Example of Pragma warn_no_side_effect

```
#pragma warn_no_side_effect on
void func(int a,int b)
{
 a+b; /* WARNING: expression has no side effect */
 (void)(a+b); /* OK: void cast suppresses warning. */
}
```

27.27 warn_padding

Controls the issuing of warning messages for data structure padding.

Syntax

```
#pragma warn_padding on | off | reset
```

Remarks

If you enable this pragma, the compiler warns about any bytes that were implicitly added after an ANSI C `struct` member to improve memory alignment. Refer to the appropriate *Targeting* manual for more information on how the compiler pads data structures for a particular processor or operating system.

This pragma corresponds to the **Pad Bytes Added** setting in the CodeWarrior IDE's **C/C++ Warnings** settings panel. By default, this setting is `off`.

27.28 warn_pch_portability

Controls whether or not to issue a warning message when `#pragmaonceon` is used in a precompiled header.

Syntax

```
#pragma warn_pch_portability on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when you use `#pragma once on` in a precompiled header. This helps you avoid situations in which transferring a precompiled header from machine to machine causes the precompiled header to produce different results. For more information, see pragma `once`.

This pragma does not correspond to any panel setting. By default, this setting is `off`.

27.29 warn_possunwant

Controls the recognition of possible unintentional logical errors.

Syntax

```
#pragma warn_possunwant on | off | reset
```

Remarks

If you enable this pragma, the compiler checks for common, unintended logical errors:

- An assignment in either a logical expression or the conditional portion of an `if`, `while`, or `for` expression. This warning message is useful if you use `=` when you mean to use `==`. The following listing shows an example.
  Listing: Confusing = and == in Comparisons

  ```
  if (a=b) f(); /* WARNING: a=b is an assignment. */
  if ((a=b)!=0) f(); /* OK: (a=b)!=0 is a comparison. */
  if (a==b) f(); /* OK: (a==b) is a comparison. */
  ```
- An equal comparison in a statement that contains a single expression. This check is useful if you use `==` when you meant to use `=`. The following listing shows an example.
  Listing: Confusing = and == Operators in Assignments

  ```
  a == 0; // WARNING: This is a comparison.
  a = 0; // OK: This is an assignment, no warning
  ```
- A semicolon (`;`) directly after a `while`, `if`, or `for` statement.

  For example, The following listing generates a warning message.

  Listing: Empty statement

  ```
  i = sockcount();
  while (--i); /* WARNING: empty loop. */
   matchsock(i);
  ```

If you intended to create an infinite loop, put white space or a comment between the `while` statement and the semicolon. The statements in the following suppress the above error or warning messages.

Listing: Intentional empty statements

```
while (i++) ; /* OK: White space separation. */
while (i++) /* OK: Comment separation */ ;
```

27.30 warn_ptr_int_conv

Controls the recognition the conversion of pointer values to incorrectly-sized integral values.

Syntax

```
#pragma warn_ptr_int_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if an expression attempts to convert a pointer value to an integral type that is not large enough to hold the pointer value.

Listing: Example for #pragma warn_ptr_int_conv

```
#pragma warn_ptr_int_conv on
char *my_ptr;
char too_small = (char)my_ptr; /* WARNING: char is too small. */
```

See also warn_any_ptr_int_conv.

27.31 warn_resultnotused

Controls the issuing of warning messages when function results are ignored.

Syntax

```
#pragma warn_resultnotused on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a statement that calls a function without using its result. To prevent this, cast the statement with `(void)`. The following listing provides an example.

Listing: Example of Function Calls with Unused Results

```
#pragma warn_resultnotused on
extern int bar();
void func()
{
 bar(); /* WARNING: result of function call is not used. */
 void(bar()); /* OK: void cast suppresses warning. */
}
```

This pragma does not correspond to any panel setting. By default, this pragma is off.

27.32 warn_undefmacro

Controls the detection of undefined macros in `#if` and `#elif` directives.

Syntax

```
#pragma warn_undefmacro on | off | reset
```

Remarks

The following listing provides an example.

Listing: Example of Undefined Macro

```
#if BADMACRO == 4 /* WARNING: undefined macro. */
```

Use this pragma to detect the use of undefined macros (especially expressions) where the default value 0 is used. To suppress this warning message, check if macro defined first.

NOTE

A warning message is only issued when a macro is evaluated. A short-circuited "`&&`" or "`||`" test or unevaluated "`?:`" will not produce a warning message.

27.33 warn_uninitializedvar

Controls the compiler to perform some dataflow analysis and emits a warning message whenever there is a usage of a local variable and no path exists from any initialization of the same local variable.

Usages will not receive a warning if the variable is initialized along any path to the usage, even though the variable may be uninitialized along some other path. `warn_possiblyunitializedvar` pragma is introduced for such cases. Refer to pragma warn_possiblyuninitializedvar for more details.

Syntax

```
#pragma warn_uninitializedvar on | off | reset
```

Remarks

This pragma has no corresponding setting in the CodeWarrior IDE. By default, this pragma is `on`.

27.34 warn_possiblyuninitializedvar

It is a distinct pragma from `warn_uninitializedvar`, which uses a slightly different process to detect the uninitialized variables.

It will give a warning whenever local variables are used before being initialized along any path to the usage. As a result, you get more warnings.

However, some of the warnings will be false ones. The warnings will be false when all of the paths with uninitialized status turn out to be paths that can never actually be taken.

Syntax

```
#pragma warn_possiblyuninitializedvar on | off | reset
```

Remarks

This pragma has no corresponding setting in the CodeWarrior IDE. By default, this pragma is `off`.

NOTE

`warn_possiblyuninitializedvar` is superset of `warn_uninitializedvar`.

27.35 warn_unusedarg

Controls the recognition of unreferenced arguments.

Syntax

```
#pragma warn_unusedarg on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters an argument you declare but do not use.

This check helps you find arguments that you either misspelled or did not use in your program. The following listing shows an example.

Listing: Warning about unused function arguments

```
void func(int temp, int error);
{
 error = do_something(); /* WARNING: temp is unused. */
}
```

To prevent this warning, you can declare an argument in a few ways:

- Use the pragma `unused`, as in the following listing.
  Listing: Using pragma unused() to prevent unused argument messages

  ```
  void func(int temp, int error)
  {
   #pragma unused (temp)
   /* Compiler does not warn that temp is not used. */
   error=do_something();
  }
  ```
- Do not give the unused argument a name. The following listing shows an example.

  The compiler allows this feature in C++ source code. To allow this feature in C source code, disable ANSI strict checking.

  Listing: Unused, Unnamed Arguments

  ```
  void func(int /* temp */, int error)
  {
   /* Compiler does not warn that "temp" is not used. */
   error=do_something();
  }
  ```

27.36 warn_unusedvar

Controls the recognition of unreferenced variables.

Syntax

```
#pragma warn_unusedvar on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a variable you declare but do not use.

This check helps you find variables that you either misspelled or did not use in your program. The following listing shows an example.

Listing: Unused Local Variables Example

```
int error;
void func(void)
{
 int temp, errer; /* NOTE: errer is misspelled. */
 error = do_something(); /* WARNING: temp and errer are unused. */
}
```

If you want to use this warning but need to declare a variable that you do not use, include the pragma `unused`, as in the following listing.

Listing: Suppressing Unused Variable Warnings

```
void func(void)
{
 int i, temp, error;
 #pragma unused (i, temp) /* Do not warn that i and temp */
 error = do_something(); /* are not used */
}
```

Chapter 28
Pragmas for Preprocessing

This chapter lists pragmas for preprocessing.

- check_header_flags
- faster_pch_gen
- flat_include
- fullpath_file
- fullpath_prepdump
- keepcomments
- line_prepdump
- macro_prepdump
- msg_show_lineref
- msg_show_realref
- notonce
- old_pragma_once
- once
- pop, push
- pragma_prepdump
- precompile_target
- simple_prepdump
- space_prepdump
- srcrelincludes
- syspath_once

28.1 check_header_flags

Controls whether or not to ensure that a precompiled header's data matches a project's target settings.

Syntax

```
#pragma check_header_flags on | off | reset
```

Remarks

This pragma affects precompiled headers only.

If you enable this pragma, the compiler verifies that the precompiled header's preferences for `double` size, `int` size, and floating point math correspond to the build target's settings. If they do not match, the compiler generates an error message.

If your precompiled header file depends on these settings, enable this pragma. Otherwise, disable it.

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `off`.

28.2 faster_pch_gen

Controls the performance of precompiled header generation.

Syntax

```
#pragma faster_pch_gen on | off | reset
```

Remarks

If you enable this pragma, generating a precompiled header can be much faster, depending on the header structure. However, the precompiled file can also be slightly larger.

This pragma does not correspond to any panel setting. By default, this setting is `off`.

28.3 flat_include

Controls whether or not to ignore relative path names in `#include` directives.

Syntax

```
#pragma flat_include on | off | reset
```

Remarks

For example, when `on`, the compiler converts this directive

```
#include <sys/stat.h>
```

to

```
#include <stat.h>
```

Use this pragma when porting source code from a different operating system, or when a CodeWarrior IDE project's access paths cannot reach a given file.

By default, this pragma is `off`.

28.4 fullpath_file

Controls if `__FILE__` macro expands to a full path or the base file name.

Syntax

```
#pragma fullpath_file on | off | reset
```

Remarks

When this pragma `on`, the `__FILE__` macro returns a full path to the file being compiled, otherwise it returns the base file name.

28.5 fullpath_prepdump

Shows the full path of included files in preprocessor output.

Syntax

```
#pragma fullpath_prepdump on | off | reset
```

Remarks

If you enable this pragma, the compiler shows the full paths of files specified by the `#include` directive as comments in the preprocessor output. Otherwise, only the file name portion of the path appears.

28.6 keepcomments

Controls whether comments are emitted in the preprocessor output.

Syntax

```
#pragma keepcomments on | off | reset
```

Remarks

This pragma corresponds to the **Keep comments** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

28.7 line_prepdump

Shows `#line` directives in preprocessor output.

Syntax

```
#pragma line_prepdump on | off | reset
```

Remarks

If you enable this pragma, `#line` directives appear in preprocessing output. The compiler also adjusts line spacing by inserting empty lines.

Use this pragma with the command-line compiler's `-E` option to make sure that `#line` directives are inserted in the preprocessor output.

This pragma corresponds to the **Use #line** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

28.8 macro_prepdump

Controls whether the compiler emits `#define` and `#undef` directives in preprocessing output.

Syntax

```
#pragma macro_prepdump on | off | reset
```

Remarks

Use this pragma to help unravel confusing problems like macros that are aliasing identifiers or where headers are redefining macros unexpectedly.

28.9 msg_show_lineref

Controls diagnostic output involving `#line` directives to show line numbers specified by the `#line` directives in error and warning messages.

Syntax

```
#pragma msg_show_lineref on | off | reset
```

Remarks

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `on`.

28.10 msg_show_realref

Controls diagnostic output involving `#line` directives to show actual line numbers in error and warning messages.

Syntax

```
#pragma msg_show_realref on | off | reset
```

Remarks

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `on`.

28.11 notonce

Controls whether or not the compiler lets included files be repeatedly included, even with `#pragma once on`.

Syntax

```
#pragma notonce
```

Remarks

If you enable this pragma, files can be repeatedly `#include`d, even if you have enabled `#pragma once on`. For more information, refer to the topic once.

This pragma does not correspond to any CodeWarrior IDE panel setting.

28.12 old_pragma_once

This pragma is no longer available.

28.13 once

Controls whether or not a header file can be included more than once in the same compilation unit.

Syntax

```
#pragma once [ on ]
```

Remarks

Use this pragma to ensure that the compiler includes header files only once in a source file. This pragma is especially useful in precompiled header files.

There are two versions of this pragma:

```
#pragma once
```

and

```
#pragma once on
```

Use `#pragma once` in a header file to ensure that the header file is included only once in a source file. Use `#pragma once on` in a header file or source file to ensure that *any* file is included only once in a source file.

Beware that when using `#pragma once on`, precompiled headers transferred from one host machine to another might not give the same results during compilation. This inconsistency is because the compiler stores the full paths of included files to distinguish between two distinct files that have identical file names but different paths. Use the `warn_pch_portability` pragma to issue a warning message when you use `#pragma once on` in a precompiled header.

Also, if you enable the `old_pragma_once on` pragma, the `once` pragma completely ignores path names.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

28.14 pop, push

Saves and restores pragma settings.

Syntax

```
#pragma push

#pragma pop
```

Remarks

The pragma `push` saves all the current pragma settings. The pragma `pop` restores all the pragma settings that resulted from the last `push` pragma. For example, see in the following listing.

Listing: push and pop example

```
#pragma ANSI_strict on
#pragma push /* Saves all compiler settings. */
#pragma ANSI_strict off
#pragma pop /* Restores ANSI_strict to on. */
```

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

Tip

Pragmas directives that accept `on` | `off` | `reset` already form a stack of previous option values. It is not necessary to use `#pragma pop` or `#pragma push` with such pragmas.

28.15 pragma_prepdump

Controls whether pragma directives in the source text appear in the preprocessing output.

Syntax

```
#pragma pragma_prepdump on | off | reset
```

28.16 precompile_target

Specifies the file name for a precompiled header file.

Syntax

```
#pragma precompile_target filename
```

Parameters

filename

A simple file name or an absolute path name. If *filename* is a simple file name, the compiler saves the file in the same folder as the source file. If *filename* is a path name, the compiler saves the file in the specified folder.

Remarks

If you do not specify the file name, the compiler gives the precompiled header file the same name as its source file.

The following listing shows sample source code from a precompiled header source file. By using the predefined symbols `__cplusplus` and the pragma `precompile_target`, the compiler can use the same source code to create different precompiled header files for C and C++.

Listing: Using #pragma precompile_target

```
#ifdef __cplusplus
 #pragma precompile_target "MyCPPHeaders"
#else
 #pragma precompile_target "MyCHeaders"
#endif
```

This pragma does not correspond to any panel setting.

28.17 simple_prepdump

Controls the suppression of comments in preprocessing output.

Syntax

```
#pragma simple_prepdump on | off | reset
```

Remarks

By default, the compiler adds comments about the current include file being in preprocessing output. Enabling this pragma disables these comments.

28.18 space_prepdump

Controls whether or not the compiler removes or preserves whitespace in the preprocessor's output.

Syntax

```
#pragma space_prepdump on | off | reset
```

Remarks

This pragma is useful for keeping the starting column aligned with the original source code, though the compiler attempts to preserve space within the line. This pragma does not apply to expanded macros.

28.19 srcrelincludes

Controls the lookup of `#include` files.

Syntax

```
#pragma srcrelincludes on | off | reset
```

Remarks

When `on`, the compiler looks for `#include` files relative to the previously included file (not just the source file). When `off`, the compiler uses the CodeWarrior IDE's access paths or the access paths specified with the `-ir` option.

Use this pragma when multiple files use the same file name and are intended to be included by another header file in that directory. This is a common practice in UNIX programming.

28.20 syspath_once

Controls how included files are treated when `#pragmaonce` is enabled.

Syntax

```
#pragma syspath_once on | off | reset
```

Remarks

When this pragma and pragma `once` are set to `on`, the compiler distinguishes between identically-named header files referred to in `#include <file-name>` and `#include "file-name"`.

When this pragma is `off` and pragma once is `on`, the compiler will ignore a file that uses a

```
#include <file-name>
```

directive if it has previously encountered another directive of the form

```
#include "file-name"
```

for an identically-named header file.

The following listing shows an example.

This pragma does not correspond to any panel setting. By default, this setting is `on`.

Listing: Pragma syspath_once example

```
#pragma syspath_once off
#pragma once on /* Include all subsequent files only once. */
#include "sock.h"
#include <sock.h> /* Skipped because syspath_once is off. */
```

Chapter 29
Pragmas for Code Generation

This chapter lists the pragmas for code generation.

- dont_reuse_strings
- enumsalwaysint
- explicit_zero_data
- float_constants
- longlong
- longlong_enums
- min_enum_size
- readonly_strings
- safe_index_expr
- common_sub_expr_elim
- const_propag
- copy_propag
- dead_store_elim
- no_register_coloring
- branch_tail_merge

29.1 dont_reuse_strings

Controls whether or not to store identical character string literals separately in object code.

Syntax

```
#pragma dont_reuse_strings on | off | reset
```

Remarks

Normally, C and C++ programs should not modify character string literals. Enable this pragma if your source code follows the unconventional practice of modifying them.

If you enable this pragma, the compiler separately stores identical occurrences of character string literals in a source file.

If this pragma is disabled, the compiler stores a single instance of identical string literals in a source file. The compiler reduces the size of the object code it generates for a file if the source file has identical string literals.

The compiler always stores a separate instance of a string literal that is used to initialize a character array. The following listing shows an example.

Although the source code contains 3 identical string literals, `"cat"`, the compiler will generate 2 instances of the string in object code. The compiler will initialize `str1` and `str2` to point to the first instance of the string and will initialize `str3` to contain the second instance of the string.

Using `str2` to modify the string it points to also modifies the string that `str1` points to. The array `str3` may be safely used to modify the string it points to without inadvertently changing any other strings.

Listing: Reusing string literals

```
#pragma dont_reuse_strings off
void strchange(void)
{
 const char* str1 = "cat";
 char* str2 = "cat";
 char str3[] = "cat";
 *str2 = 'h'; /* str1 and str2 point to "hat"! */
 str3[0] = 'b';
 /* OK: str3 contains "bat", *str1 and *str2 unchanged.
}
```

29.2 enumsalwaysint

Specifies the size of enumerated types.

Syntax

```
#pragma enumsalwaysint on | off | reset
```

Remarks

If you enable this pragma, the C/C++ compiler makes an enumerated type the same size as an `int`. If an enumerated constant is larger than `int`, the compiler generates an error message. Otherwise, the compiler makes an enumerated type the size of any integral type. It chooses the integral type with the size that most closely matches the size of the largest enumerated constant. The type could be as small as a `char` or as large as a `long long`.

The following listing shows an example.

Listing: Example of Enumerations the Same as Size as int

```
enum SmallNumber { One = 1, Two = 2 };
 /* If you enable enumsalwaysint, this type is
 the same size as an int. Otherwise, this type is
 the same size as a char. */
enum BigNumber
 { ThreeThousandMillion = 3000000000 };
 /* If you enable enumsalwaysint, the compiler might
 generate an error message. Otherwise, this type is
 the same size as a long long. */
```

29.3 explicit_zero_data

Controls the placement of zero-initialized data.

Syntax

```
#pragma explicit_zero_data on | off | reset
```

Remarks

Places zero-initialized data into the initialized data section instead of the BSS section when `on`.

By default, this pragma is `off`.

29.4 float_constants

Controls how floating pointing constants are treated.

Syntax

```
#pragma float_constants on | off | reset
```

Remarks

If you enable this pragma, the compiler assumes that all unqualified floating point constant values are of type `float`, not `double`. This pragma is useful when porting source code for programs optimized for the " `float`" rather than the " `double`" type.

When you enable this pragma, you can still explicitly declare a constant value as double by appending a "D" suffix.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

29.5 longlong

Controls the availability of the `long long` type.

Syntax

```
#pragma longlong on | off | reset
```

Remarks

When this pragma is enabled and the compiler is translating "C90" source code (ISO/IEC 9899-1990 standard), the compiler recognizes a data type named `long long`. The `long long` type holds twice as many bits as the `long` data type.

This pragma does not correspond to any CodeWarrior IDE panel setting.

By default, this pragma is on for processors that support this type. It is `off` when generating code for processors that do not support, or cannot turn on, the `long long` type.

29.6 longlong_enums

Controls whether or not enumerated types may have the size of the `long long` type.

Syntax

```
#pragma longlong_enums on | off | reset
```

Remarks

This pragma lets you use enumerators that are large enough to be `long long`integers. It is ignored if you enable the `enumsalwaysint` pragma (described in the topic enumsalwaysint).

This pragma does not correspond to any panel setting. By default, this setting is enabled.

29.7 min_enum_size

Specifies the size, in bytes, of enumeration types.

Syntax

```
#pragma min_enum_size 1 | 2 | 4
```

Remarks

Turning on the `enumsalwaysint` pragma overrides this pragma. The default is 1.

29.8 readonly_strings

Controls whether string objects are placed in a read-write or a read-only data section.

Syntax

```
#pragma readonly_strings on | off | reset
```

Remarks

If you enable this pragma, literal strings used in your source code are output to the read-only data section instead of the global data section. In effect, these strings act like `constchar*`, even though their type is really `char*`.

This pragma does not correspond to any IDE panel setting.

29.9 safe_index_expr

Controls whether the index expressions are type-safe. This pragma enable front-end optimization on consecutive array accesses.

Syntax

```
#pragma safe_index_expr on | off | reset
```

Remarks

If you enable this pragma, the compiler assumes that the index expressions are type-safe (no overflow on index computations). The compiler enables front-end optimization on consecutive array accesses.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

29.10 common_sub_expr_elim

Controls the use of Common Subexpression Elimination optimization at low-level.

Syntax

```
#pragma common_sub_expr_elim on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces similar redundant expressions with a single expression.

This pragma corresponds to `-common_sub_expr_elim` command line option.

29.11 const_propag

Controls the use of constant propagation optimization at low-level.

Syntax

```
#pragma const_propag on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces the use of variable with the constants assigned to the variable.

This pragma corresponds to `-const_propag` command line option.

29.12 copy_propag

Controls the use of copy propagation optimization at low-level.

Syntax

```
#pragma copy_propag on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces the use of variable with its previous value.

This pragma corresponds to `-copy_propag` command line option.

29.13 dead_store_elim

Controls the use of dead store elimination optimization at low-level.

Syntax

```
#pragma dead_store_elim on | off | reset
```

Remarks

If you enable this pragma, the compiler removes the store instructions that does not have any use.

This pragma corresponds to `-dead_store_elim` command line option.

29.14 no_register_coloring

Controls the register coloring optimization.

Syntax

```
#pragma no_register_coloring on | off | reset
```

Remarks

If you enable this pragma, the compiler enables register coloring.

This pragma corresponds to `-no_register_coloring` command line option.

29.15 branch_tail_merge

Controls the branch tail merge optimization.

Syntax

```
#pragma branch_tail_merge on | off | reset
```

Remarks

If you enable this pragma, the compiler enables common code optimization in blocks.

This pragma corresponds to `-branch_tail_merge` command line option.

Chapter 30
Pragmas for Optimization

This chapter lists the pragmas for optimization.

- global_optimizer
- opt_common_subs
- opt_dead_assignments
- opt_dead_code
- opt_lifetimes
- opt_loop_invariants
- opt_propagation
- opt_strength_reduction
- opt_strength_reduction_strict
- opt_unroll_loops
- optimization_level
- optimize_for_size
- strictheaderchecking

30.1 global_optimizer

Controls whether the Global Optimizer is invoked by the compiler.

Syntax

```
#pragma global_optimizer on | off | reset
```

Remarks

In most compilers, this #pragma determines whether the Global Optimizer is invoked (configured by options in the panel of the same name). If disabled, only simple optimizations and back-end optimizations are performed.

NOTE

This is not the same as `#pragmaoptimization_level`. The Global Optimizer is invoked even at `optimization_level0` if `#pragmaglobal_optimizer` is enabled.

30.2 opt_common_subs

Controls the use of common subexpression optimization.

Syntax

```
#pragma opt_common_subs on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces similar redundant expressions with a single expression. For example, if two statements in a function both use the expression

```
a * b * c + 10
```

the compiler generates object code that computes the expression only once and applies the resulting value to both statements.

The compiler applies this optimization to its own internal representation of the object code it produces.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer pragma.

30.3 opt_dead_assignments

Controls the use of dead store optimization.

Syntax

```
#pragma opt_dead_assignments on | off | reset
```

Remarks

If you enable this pragma, the compiler removes assignments to unused variables before reassigning them.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

30.4 opt_dead_code

Controls the use of dead code optimization.

Syntax

```
#pragma opt_dead_code on | off | reset
```

Remarks

If you enable this pragma, the compiler removes a statement that other statements never execute or call.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

30.5 opt_lifetimes

Controls the use of lifetime analysis optimization.

Syntax

```
#pragma opt_lifetimes on | off | reset
```

Remarks

If you enable this pragma, the compiler uses the same processor register for different variables that exist in the same routine but not in the same statement.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

30.6 opt_loop_invariants

Controls the use of loop invariant optimization.

Syntax

```
#pragma opt_loop_invariants on | off | reset
```

Remarks

If you enable this pragma, the compiler moves all computations that do not change inside a loop to outside the loop, which then runs faster.

This pragma does not correspond to any panel setting.

30.7 opt_propagation

Controls the use of copy and constant propagation optimization.

Syntax

```
#pragma opt_propagation on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces multiple occurrences of one variable with a single occurrence.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

30.8 opt_strength_reduction

Controls the use of strength reduction optimization.

Syntax

```
#pragma opt_strength_reduction on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces array element arithmetic instructions with pointer arithmetic instructions to make loops faster.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

30.9 opt_strength_reduction_strict

Uses a safer variation of strength reduction optimization.

Syntax

```
#pragma opt_strength_reduction_strict on | off | reset
```

Remarks

Like the opt_strength_reduction pragma, this setting replaces multiplication instructions that are inside loops with addition instructions to speed up the loops. However, unlike the regular strength reduction optimization, this variation ensures that the optimization is only applied when the array element arithmetic is not of an unsigned type that is smaller than a pointer type.

This pragma does not correspond to any panel setting. The default varies according to the compiler.

30.10 opt_unroll_loops

Controls the use of loop unrolling optimization.

Syntax

```
#pragma opt_unroll_loops on | off | reset
```

Remarks

If you enable this pragma, the compiler places multiple copies of a loop's statements inside a loop to improve its speed.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

30.11 optimization_level

Controls global optimization.

Syntax

```
#pragma optimization_level 0 | 1 | 2 | 3 | reset
```

Remarks

This pragma specifies the degree of optimization that the global optimizer performs.

To select optimizations, use the pragma `optimization_level` with an argument from `0` to `3`. The higher the argument, the more optimizations performed by the compiler. The `reset` argument specifies the previous optimization level.

For more information on the optimization the compiler performs for each optimization level, refer to the *Targeting* manual for your target platform.

30.12 optimize_for_size

Controls optimization to reduce the size of object code.

```
#pragma optimize_for_size on | off | reset
```

Remarks

This setting lets you choose what the compiler does when it must decide between creating small code or fast code. If you enable this pragma, the compiler creates smaller object code at the expense of speed. It also ignores the `inline` directive and generates function calls to call any function declared `inline`. If you disable this pragma, the compiler creates faster object code at the expense of size.

30.13 strictheaderchecking

Controls how strict the compiler checks headers for standard C library functions.

Syntax

```
#pragma strictheaderchecking on | off | reset
```

Remarks

The 3.2 version compiler recognizes standard C library functions. If the correct prototype is used, and, in C++, if the function appears in the " `std`" or root namespace, the compiler recognizes the function, and is able to optimize calls to it based on its documented effects.

When this #pragma is on (default), in addition to having the correct prototype, the declaration must also appear in the proper standard header file (and not in a user header or source file).

This pragma does not correspond to any panel setting. By default, this pragma is on.

Chapter 31
S12Z Pragmas

This chapter lists pragmas for the CodeWarrior compiler for S12Z architectures. The following listed are the pragmas covered:

Table 31-1. S12Z Pragmas

CODE_SEG	safe_index_expr	branch_tail_merge
DATA_SEG	common_sub_expr_elim	peephole
CONST_SEG	const_propag	bfield_gap_limit
NO_ENTRY	copy_propag	bfield_lsbit_first
NO_EXIT	dead_code_elim	bfield_reduce_type
NO_RETURN	dead_store_elim	

31.1 CODE_SEG

Sets the current code segment.

Syntax

```
#pragma CODE_SEG <name>|DEFAULT
```

Parameters

`<name>`

The name of the code segment being defined. This segment must be explicitly allocated within the `PLACEMENT` block in the link parameter file. For more information, refer to the chapter *Linker Issues* of the *Build Tools Utilities* reference manual.

Default

`DEFAULT`

Remarks

This pragma sets the current code segment, either the default code segment (that is, DEFAULT_ROM), or a user-defined segment. All the functions subsequently defined are allocated into that segment by the linker.

NOTE

The CODE_SEG pragma affects function declarations as well as definitions. Ensure that, for a given function, the declaration and definition are in the same segment.

The following listing exemplifies correct usage of the CODE_SEG pragma:

Listing: Using pragma CODE_SEG

```
/* p.h */
#pragma CODE_SEG MY_ROM

extern void func1();

#pragma CODE_SEG DEFAULT

extern void func2();

/* p.c */

#pragma CODE_SEG MY_ROM

void func1()

{

c = a + b;

}

#pragma CODE_SEG DEFAULT

void func2()

{

c = a - b;

}
```

The following listing contains examples of improper CODE_SEG pragma usage:

Listing: Improper Usage of CODE_SEG pragma

```
#pragma CONST_SEG MY_ROM
#pragma CODE_SEG MY_ROM

/** incorrect: the same segment name used with different segment types
**/

#pragma CODE_SEG MY_ROM

extern void func1();

#pragma CODE_SEG DEFAULT

extern void func1();

/** incorrect: function 'func1' is declared in two different segments
```

```
**/

/* p.h */

#pragma CODE_SEG MY_ROM

extern void func1();

#pragma CODE_SEG DEFAULT

extern void func2();

#pragma CODE_SEG MY_ROM

extern void func3();

#pragma CODE_SEG DEFAULT

/* p.c */

#pragma CODE_SEG MY_ROM

void func1()

{

z = x + y;

}

#pragma CODE_SEG DEFAULT

void func2()

{

z = x - y;

}

void func3()

{

z = x * y;

}

/** incorrect: 'func3' is declared in segment MY_ROM and defined in
segment DEFAULT **/
```

31.2 DATA_SEG

Sets the current data segment.

Syntax

```
#pragma DATA_SEG <name>|DEFAULT
```

Parameters

`<name>`

The name of the data segment being defined.This segment must be explicitly allocated within the PLACEMENT block in the link parameter file. For more information, refer to the chapter *Linker Issues* of the *Build Tools Utilities* reference manual.

Default

DEFAULT

Remarks

This pragma sets the current data segment: either the default data segment (that is, DEFAULT_RAM), or a user-defined segment. All the variables subsequently declared are allocated into that segment by the linker.

NOTE

The DATA_SEG pragma affects data definitions as well as declarations. Ensure that, for a given variable, all declarations and the definition are in the same segment.

The following listing exemplifies the correct usage of the DATA_SEG pragma:

Listing: Using pragma DATA_SEG

```
/* p.h */
#pragma DATA_SEG MY_RAM

extern int x;

#pragma DATA_SEG DEFAULT

extern int y;

/* p.c */

#pragma DATA_SEG MY_RAM

int x;

#pragma DATA_SEG DEFAULT

int y;

/* main.c */

#include "p.h"

void main(void)

{

x = 5;

y = 10;

}
```

The following listing contains examples of improper DATA_SEG pragma usage:

Listing: Improper Usage of DATA_SEG pragma

```
#pragma CONST_SEG MY_RAM
#pragma DATA_SEG  MY_RAM

/** incorrect: the same segment name used with different segment types
**/

#pragma DATA_SEG MY_RAM

extern int x;

#pragma DATA_SEG DEFAULT

extern int x;

/** incorrect: variable 'x' is declared in two different segments **/

/* p.h */

#pragma DATA_SEG MY_RAM

extern int x;

#pragma DATA_SEG DEFAULT

extern int y;

#pragma DATA_SEG MY_RAM

extern int z;

#pragma DATA_SEG DEFAULT

/* p.c */

#pragma DATA_SEG MY_RAM

int x;

#pragma DATA_SEG DEFAULT

int y, z;

/** incorrect: variable 'z' is declared in segment MY_RAM and defined
in segment DEFAULT **/
```

31.3 CONST_SEG

Sets the current constant data segment.

Syntax

```
#pragma CONST_SEG <name>|DEFAULT
```

Parameters

```
<name>
```

The name of the data segment being defined. This segment must be explicitly allocated within the PLACEMENT block in the link parameter file. For more information, refer to the chapter *Linker Issues* of the *Build Tools Utilities* reference manual.

Default

`DEFAULT`

Remarks

This pragma sets the current constant data segment: either the default constant data segment (that is, `ROM_VAR`), or a user-defined segment. All the `<keyword>const</keyword>` variables subsequently declared are allocated into that segment by the linker.

NOTE

The `CONST_SEG` pragma affects constant data definitions as well as declarations. Ensure that, for a given constant variable, all declarations and the definition are in the same segment.

The following listing exemplifies the correct usage of the `CONST_SEG` pragma:

Listing: Using pragma CONST_SEG

```
/* p.h */
#pragma CONST_SEG MY_ROM

extern const int cx;

#pragma CONST_SEG DEFAULT

extern const int y;

/* p.c */

#pragma CONST_SEG MY_RAM

const int cx = 1;

#pragma CONST_SEG DEFAULT

const int cy = 2;

/* main.c */

#include "p.h"

void main(void)

{

s = cx + cy;

}
```

The following listing shows the examples of improper `CONST_SEG` pragma usage:

Listing: Improper Usage of CONST_SEG pragma

```
#pragma CONST_SEG MY_ROM
#pragma CODE_SEG  MY_ROM

/** incorrect: the same segment name used with different segment types
**/

#pragma CONST_SEG MY_ROM

extern const int cx;

#pragma CONST_SEG DEFAULT

extern const int cx;

/** incorrect: constant variable 'cx' is declared in two different
segments **/

/* p.h */

#pragma CONST_SEG MY_ROM

extern const int cx;

#pragma CONST_SEG DEFAULT

extern const int cy;

#pragma CONST_SEG MY_ROM

extern const int cz;

#pragma CONST_SEG DEFAULT

/* p.c */

#pragma CONST_SEG MY_ROM

const int cx = 1;

#pragma CONST_SEG DEFAULT

const int cy = 2;

const int cz = 3;

/** incorrect: constant variable 'cz' is declared in segment MY_RAM and
defined in segment DEFAULT **/
```

31.4 NO_ENTRY

Disables the `ENTRY` code generation.

Syntax

```
#pragma NO_ENTRY
```

Remarks

This pragma disables the ENTRY code generation (the ENTRY code is responsible with managing the received parameters and allocating stack space for the current function). The ENTRY code generation can be disabled if the parameters and stack space are handled explicitly from inline-assembly code or if they are not required at all in the function.

This pragma should be used in pair with the NO_EXIT pragma since the stack frame can get corrupted otherwise

31.5 NO_EXIT

Disables the EXIT code generation.

Syntax

```
#pragma NO_EXIT
```

Remarks

This pragma disables the EXIT code generation (the EXIT code is responsible with managing the stack space deallocating for the current function).

The EXIT code generation can be disabled if the return value and stack space are handled explicitly from inline-assembly code or if they are not required at all in the function.

NOTE

This pragma should be used in pair with the NO_ENTRY pragma since the stack frame can get corrupted otherwise.

31.6 NO_RETURN

Disables the RETURN instruction generation.

Syntax

```
#pragma NO_RETURN
```

Remarks

This pragma disables the RETURN instruction generation. The RETURN generation can be disabled if the function is known that it never returns, or if it returns in any other manner than using the RETURN instruction, or if the returning sequence is handled from inline-assembly.

31.7 align_globals

Enables/disables alignment of all global data in its scope.

Syntax

```
#pragma align_globals on | off | reset
```

Default

```
off
```

Remarks

This pragma enables/disables alignment of all global variables in its scope. It should not be used when optimizing for size (i.e. -opt size or -opt nospeed).

31.8 safe_index_expr

Optimizes the access to the elements of an array.

Syntax

```
#pragma safe_index_expr on | off | reset
```

Default

```
off
```

Remarks

It is applicable to the *optimization level O3*.

The compiler uses this pragma to decide whether it can optimize access to several elements of the same array in expressions such as, `some_array[i] + some_array[i + <offset>]`.

Optimization is safe and performed only if this pragma is on. Only enable this pragma if the index expression cannot produce an overflow with respect to its type.

The following listing displays an example of correct pragma usage:

Listing: Correct Usage of pragma safe_index_expr

```
unsigned int index = 0xF;
char value;

#pragma safe_index_expr on

void func(char *array)

{

value = array[index] + array[index + 2]; /* 'index + 2' does not
overflow wrt 'unsigned int' */

}

void main(void)

{

char array[] = {0, 1, 2, 3, 4, 5};

func(array);

}
```

The following listing shows an example of incorrect pragma usage:

Listing: Incorrect Usage of pragma safe_index_expr

```
unsigned int index = 0xFFFF;
char value;

#pragma safe_index_expr on

void func(char *array)

{

value = array[index] + array[index + 2]; /* '0xFFFF + 2' overflows wrt
'unsigned int' */

}

void main(void)

{

char array[] = {0, 1, 2, 3, 4, 5};

func(array);

}
```

31.9 common_sub_expr_elim

Enables/disables the low-level Common Subexpression Elimination.

Syntax

```
#pragma common_sub_expr_elim on | off | reset
```

Default

on

Remarks

This pragma enables/disables low-level Common Subexpression Elimination. It is applicable to the *optimization level O2 or above*.

31.10 const_propag

Enables/disables the low-level Constant Propagation.

Syntax

```
#pragma const_propag on | off | reset
```

Default

on

Remarks

This pragma enables/disables the low-level Constant Propagation. It is applicable to the *optimization level O2 or above*.

31.11 copy_propag

Enables/disables the low-level Copy Propagation.

Syntax

```
#pragma copy_propag on | off | reset
```

Default

on

Remarks

This pragma enables/disables the low-level Copy Propagation. It is applicable to the *optimization level O2 or above*.

31.12 dead_code_elim

Enables/disables the Dead Code Elimination.

Syntax

```
#pragma dead_code_elim on | off | reset
```

Default

`on`

Remarks

This pragma enables/disables the Dead Code Elimination. It is applicable to the *optimization level O2 or above*.

31.13 dead_store_elim

Enables/disables the Dead Store Elimination.

Syntax

```
#pragma dead_store_elim on | off | reset
```

Default

`on`

Remarks

This pragma enables/disables the Dead Store Elimination. It is applicable to the *optimization level O2 or above*.

31.14 branch_tail_merge

Enables/disables the Branch Tail Merging.

Syntax

```
#pragma branch_tail_merge on | off | reset
```

Default

`on`

Remarks

This pragma enables/disables the Branch Tail Merging. It is applicable to the *optimization level O2 or above*.

31.15 peephole

Enables/disables the peephole optimizations.

Syntax

```
#pragma peephole on | off | reset
```

Default

`on`

Remarks

This pragma enables/disables the peephole optimizations. It is applicable to the *optimization level O2 or above*.

31.16 bfield_gap_limit

Controls the maximum number of gap bits allowed.

Syntax

```
#pragma bfield_gap_limit <number>
```

Default

`0`

Remarks

When performing bitfield allocation, the compiler tries to avoid crossing a byte boundary whenever possible. In doing so, it may insert some padding(gap) bits.

Its functionality is equivalent to that of command line option,

`-bfield_gap_limit`

This pragma can also be used to force word as the allocation unit for bitfields. To achieve this, use the pragma with a negative argument, for example:

```
#pragma bfield_gap_limit -1
```

For more information, refer to the topic -bfield_gap_limit.

31.17 bfield_lsbit_first

Changes the default allocation order.

Syntax

```
#pragma bfield_lsbit_first on | off | reset
```

Default

```
on
```

Remarks

By default, during bitfield allocation, bits are allocated from the least significant to the most significant one, in the order of declaration (that is, the first declared bitfield will be leftmost in the allocation unit).

Its functionality is equivalent to that of command line option,

```
-[no]bfield_lsbit_first
```

For more information, refer to the topic -[no]bfield_lsbit_first.

31.18 bfield_reduce_type

Controls whether or not the compiler uses type-size reduction for bitfields.

Syntax

```
#pragma bfield_reduce_type on | off | reset
```

Default

```
on
```

Remarks

Type-size reduction means that the compiler can reduce the type of a bitfield from int to `char` if that bitfield fits into a character. Thus, memory will be allocated for a byte instead of an integer.

The functionality of this pragma is equivalent to that of command line option,

```
-[no]bfield_reduce_type
```

For more information, refer to the topic -[no]bfield_reduce_type.

Index

__abs16 154
__abs32 154
__abs8 154
__asm 145
__builtin_constant_p() 129
__builtin_expect() 130
__CHAR_IS_16BIT__ 203
__CHAR_IS_32BIT__ 204
__CHAR_IS_8BIT__ 203
__CHAR_IS_SIGNED__ 203
__CHAR_IS_UNSIGNED__ 202
__COUNTER__ 193
__cplusplus 194
__CWCC__ 194
__DATE__ 194
__DOUBLE_IS_IEEE32__ 208
__DOUBLE_IS_IEEE64__ 208
__embedded_cplusplus 195
__FILE__ 195
__FLOAT_IS_IEEE32__ 207
__FLOAT_IS_IEEE64__ 208
__func__ 122, 196
__FUNCTION__ 196
__HC12__ 202
__ide_target() 196
__inline 186
__inline__ 186
__INT_IS_16BIT__ 205
__INT_IS_32BIT__ 205
__INT_IS_8BIT__ 205
__LINE__ 197
__LONG_DOUBLE_IS_IEEE32__ 209
__LONG_DOUBLE_IS_IEEE64__ 209
__LONG_IS_16BIT__ 206
__LONG_IS_32BIT__ 206
__LONG_IS_8BIT__ 206
__LONG_LONG_IS_16BIT__ 207
__LONG_LONG_IS_32BIT__ 207
__LONG_LONG_IS_8BIT__ 207
__MWERKS__ 197
__option(setting-name) 213
__optlevelx 199
__PRETTY_FUNCTION__ 138, 198
__profile__ 198
__PTRDIFF_T_IS_CHAR__ 211
__PTRDIFF_T_IS_INT__ 212
__PTRDIFF_T_IS_LONG__ 212
__PTRDIFF_T_IS_SHORT__ 212
__qmuls16 155
__qmuls32 155
__qmuls32_16_16 155
__qmuls8 154
__qmulu16 156
__qmulu32 156
__qmulu32_16_16 156
__qmulu8 156
__S12LISA__ 201
__S12Z__ 202
__sat16 157
__sat32 157
__sat8 157
__SHORT_IS_16BIT__ 204
__SHORT_IS_32BIT__ 204
__SHORT_IS_8BIT__ 204
__SIZE_T_IS_UCHAR__ 210
__SIZE_T_IS_UINT__ 211
__SIZE_T_IS_ULONG__ 211
__SIZE_T_IS_USHORT__ 211
__STDC__ 198
__TIME__ 199
__WCHAR_T_IS_UCHAR__ 209
__WCHAR_T_IS_ULONG__ 210
__WCHAR_T_IS_USHORT__ 210
_Bool 123
_Complex 123
_Imaginary 123
(@address) 117
-[no]bfield_lsbit_first 100
-[no]bfield_reduce_type 101
@ "SegmentName" 118
#warning 119
-allow_macro_redefs 86
-ansi 55
-ARM 57
-bfield_gap_limit 98
-bool 57
-c 87
-char 63
-codegen 87
-coloring 97
-convertpaths 77
-Cpp_exceptions 57
-cwd 78
-D+ 78
-define 79
-dialect 58
-double_size 109
-E 79
-encoding 63
-enum 88
-EP 79
-ext 88
-flag 64
-float_size 108
-for_scoping 59
-g 95
-gccdepends 80

-gccincludes 80
-help 51, 67
-I- 80
-I+ 81
-include 81
-inline 91
-instmgr 59, 138
-int_size 105
-ir 82
-iso_templates 59
-ldouble_size 110
-lldouble_size 110
-llong_size 107
-long_size 106
-mapcr 65
-maxerrors 68
-maxwarnings 69
-min_enum_size 88
-model 96
-msgstyle 69
-nolonglong 66
-noprecompile 84
-nosyspath 84
-O 92
-O+ 92
-opt 93
-P 82
-peephole 97
-ppopt 83
-pragma 65
-precompile 82
-prefix 84
-preprocess 83
-progress 70
-requireprotos 66
-RTTI 60
-schar_size 103
-short_size 104
-som 60
-som_env_check 60
-stderr 70
-stdinc 85
-stdkeywords 56
-strict 56
-sym 96
-timing 71
-trigraphs 66
-U+ 85
-uchar_size 104
-undefine 85
-verbose 70
-version 71
-warnings 71
-wchar_t 60
-wraplines 75

A

access_errors 224
Access Paths Options
    Always Search User Paths 41
    Do Not use MWCIncludes Variable 41
    Search System Paths (#include <...>) 41
    Search System Paths Recursively 41
    Search User Paths (#include "...") 41
    Search User Paths Recursively 41
addressing 159
Address Modifier 117
AddrOfVar 147
align_globals 111, 311
align_stack 113
align_structs 112
alignment 189–191
always_inline 224
Analysis 173
ANSI_strict 215, 217
arg_dep_lookup 225
Argument 165
Arguments 116
Arithmetic 129
ARM_conform 225
ARM_scoping 225
array_new_delete 226
Arrays 125
as12lisa.exe 51
asmsemicolcomment 249
Assembly Functions 148
attribute 191
attributes 189, 190
auto_inline 226
Automatic Arrays 127

B

bfield_gap_limit 315
bfield_lsbit_first 316
bfield_reduce_type 316
Binary 119
Bitfields 163
bool 226
branch_tail_merge 293, 314
build settings panels
    S12Z Compiler
        Access Paths 41
        Code Generation 43
        General 47
        Input 40
        Language 45
        Messages 47
        Optimization 44
        Warnings 41
build tools 38
Build Tools 49

C

C++-style 116
C++-Style 123
c99 217
C99 120, 123, 126
c9x 218
Calling 165
C Compiler 115
check_header_flags 277
CODE_SEG 303
Code Generation 287
Code Generation Options
    Bit-field byte allocation from LSB to MSB (right-to-left) 44
    Bit-field gap limit (0 - 127 or 255 (0xff) as -1) 43
    Bit-field type size reduction 44
    Memory Model 43
combined 38
Command Line 49
Command-Line Options
    Bit Field 97
    Code Generation 96
    Configuration 102
    Diagnostic 95
    Diagnostic Messages 67
    Language Translation 63
    Object Code 87
    Optimization 91
    Preprocessing 77
    S12Z 95
    Standard C++ Conformance 57
    Standard C Conformance 55
Command-Line Tools 49, 51
Comments 116, 123
common_sub_expr_elim 292, 312
Common Subexpression 177
common terms
    cased 53
    compatibility 53
    default 53
    deprecated 53
    global 53
    ignored 53
    meaningless 53
    obsolete 53
    substituted 53
compiler architecture 27
Compiler Performance 137
Complex 126
compound literal values 121
Conditional Expressions 130
Conformance 115
const_propag 292, 313
CONST_SEG 307
constant expressions 147
Constants
Constants *(index-continued-string)*
    $ 119
    0b 119
Convention 165
copy_propag 292, 313
Copy Propagation 178
cplusplus 227
cpp_extensions 228
cpp1x 227
CWFolder 50
cwide.exe 38

D

DATA_SEG 305
Data Initialization 122
Data Types
    double_size 161
    float_size 161
    int_size 161
    ldouble_size 161
    lldouble_size 161
    llong_size 161
    long_size 161
    schar_size 161
    short_size 161
    uchar_size 161
dead_code_elim 314
dead_store_elim 293, 314
Dead Code 175
Dead Store 179
Debugger 35
debuginline 229
Decimal 125
declaration 190
def_inherited 229
DEFAULT_RAM 161
DEFAULT_ROM 161
defer_codegen 230
defer_defarg_parsing 230
deferred codegen 174
deferred inlining 174
Diagnostic Messages 255
Digraphs 123
direct_destruction 231
direct_to_som 231
Directive 119
dont_inline 231
dont_reuse_strings 287
double 163

E

ecplusplus 231
Elimination 175, 177, 179
Empty Arrays 124
enforce_keyword 149

enforce operators 148
Entry 169
Enumerations 120
enumsalwaysint 288
Environment Variables 49
exceptions 232
Exit 169
explicit_zero_data 289
Expression 176
Expressions 128
extended_errorcheck 232, 256
Extensions 115, 116, 126, 127

F

faster_pch_gen 278
File-Level 174
file name extensions 53
FLASH 34
flat_include 278
float 163
float_constants 289
Floating-Point Constants 124
forced_operand 149
Forward Declarations 129
fullpath_file 279
fullpath_prepdump 279
Function-Level 174
Function Pointer 129
functions 153
Functions 185

G

GCC 126, 127, 143
gcc_extensions 250
General Options
    Generate Debug Information 48
    Other Flags 48
global_optimizer 295
global data 189
Global Variable 117

H

Hexadecimal 119, 124
highlights 38

I

ignore_oldstyle 219
immediate_value 149
implementation-defined behavior 141
Implicit Return 122
Initializers 121
inline 123, 186
Inline 185
inline_bottom_up 233
inline_bottom_up_once 234
inline_depth 235
inline_max_auto_size 235
inline_max_size 236
inline_max_total_size 236
Inline Assembler
    Syntax 145
Inlining 185
Inlining Techniques 187
Input Options
    Allow Macro Redefinition 41
    Defined Macros 41
    Prefix File 41
    Source File Encoding 41
    Undefined Macros 41
installations 38
Instance Manager 138
instmgr_file 138
integration 38
Intermediate 173, 174
internal 237
Interprocedural 173, 174
interrupt 118
intrinsic 153
Invalid Pragmas 216
iso_templates 237

K

keepcomments 279
Keyword 118
Keywords 117

L

Language Options
    ANSI Keywords Only 46
    ANSI Strict 46
    Enable C++ 'bool' type, 'true' and 'false' Constants 46
    Enable C++ Exceptions 46
    Enable C99 Extensions 46
    Enable GCC Extensions 46
    Enable RTTI 46
    Enable wchar_t Support 46
    Enum Always Int 47
    Expand Trigraphs 46
    Force C++ Compilation 46
    ISO C++ Template Parser 46
    Legacy for-scoping 46
    Pool Strings 47
    Require Function Prototypes 45
    Reuse Strings 47
    Use Instance Manager 46
    Use Unsigned Chars 47
Language Translation 249

Lib 35
line_prepdump 280
Linker_Files 35
linker.exe 51
Literal Values 125
Live Range 180
Local Labels 131
long double 163
longlong 290
long long 123
longlong_enums 290
long long double 163
Loop-Invariant 181
Loop Unrolling 184

M

macro_prepdump 280
Macros 122, 128, 193
main.c 36, 37
main() 122
maxerrorcount 257
Maximum Operators 131
mc9s12zvh64.c 36
member 191
Memory Model
    LARGE 160
    MEDIUM 160
    SMALL 160
Memory Models 159
message 258
Messages Options
    Maximum Number of Errors 47
    Maximum Number of Warnings 47
    Message Style 47
min_enum_size 290
Minimum Operators 131
Motion 181
mpwc_newline 250
mpwc_relax 251
msg_show_lineref 280
msg_show_realref 281
multibyteaware 252
multibyteaware_preserve_literals 252
mwccs12lisa.exe 51
MWCIncludes 50
MWLibraries 50

N

new_mangler 238
no_conststringconv 238
NO_ENTRY 170, 309
NO_EXIT 170, 310
no_register_coloring 293
NO_RETURN 310
no_static_dtors 239
Non-constant 122
Non-Standard 117
nosyminline 239
notonce 281

O

old_friend_lookup 239
old_pods 240
old_pragma_once 282
old_vtable 240
Omitted Operands 130
once 282
only_std_keywords 219
opr15i 150
opr16i 150
opr24 150
opr24a 150
opr24i 150
opr32i 150
opr5i 150
opr7i 150
opr8i 150
oprs9 150
opru14 149
opru18 150
opru4 150
opt_classresults 241
opt_common_subs 296
opt_dead_assignments 296
opt_dead_code 297
opt_lifetimes 297
opt_loop_invariants 297
opt_propagation 298
opt_strength_reduction 298
opt_strength_reduction_strict 299
opt_unroll_loops 299
Optimization 174, 295
optimization_level 299
Optimization Options
    Array index expressions do not overflow the index type 45
    Auto Inline 45
    Bottom-Up Inlining 45
    Inline Level 45
    Optimization Level 44
    Speed Vs Size 44
Optimizations 173, 174
optimize_for_size 300
option formats 52

P

parameter formats 52
parse_func_templ 241
parse_mfunc_templ 242
PATH Environment Variable 50

peephole 315
Pointer 163
pop, 283
pragma_prepdump 283
Pragmas 213
Pragma Scope 216
Pragma Settings 213
precompile_target 135, 283
Precompiled 133, 134
Precompiled File 135
Precompiled Files 134, 135
Precompiling 133, 134, 137
Predefined 122, 193, 201
Preprocessing 277
Preprocessor 116
Preprocessor Scope 135
Project_Headers 35
Project_Settings 35
Pseudo-Opcodes 146
Pure Inline 148
push 283

R

readonly_strings 291
require_prototypes 220
reserved words 146
Restoring 214
restrict 123
Return 167
ROM_VAR 161
RTTI 242

S

S12Z_Project.elf 37
S12Z_Project.map 37
safe_index_expr 291, 311
Scalar 162
Segmentation 160
separated 38
show_error_filestack 258
showmessagenumber 258
simple_prepdump 284
Simplification 176
sizeof() 127
Sources 35
space_prepdump 285
special cases 151
Splitting 180
srcrelincludes 285
stack 191
Standard 115
Standard C Conformance 217
starts12z.c 36
Startup_Code 35
Statements 128
Static 122
Static Arrays 129
Strength Reduction 182
strictheaderchecking 300
struct 191
Structures 127
suppress_init_code 242
suppress_warnings 259
sxe4i 150
sym 259
Symbol 122
Symbols 201
syspath_once 285

T

template_depth 243
Template Parsing 139
text_encoding 252
thread_safe_init 243
Tips 48
tool 51
Trailing Commas 120
Tricks 48
trigraphs 253
typedef 190
typeof() 128

U

Unnamed 116
unsigned_char 254
Unsuffixed 125
unused 259

V

Values 167
variable 190
Variable Allocation 118
Variable Argument 122
Variable-Length 125
Void 129, 131

W

warn_any_ptr_int_conv 262
warn_emptydecl 262
warn_extracomma 263
warn_filenamecaps 263
warn_filenamecaps_system 264
warn_hiddenlocals 265
warn_hidevirtual 244
warn_illpragma 265
warn_illtokenpasting 265
warn_illunionmembers 266
warn_impl_f2i_conv 266

warn_impl_i2f_conv *267*
warn_impl_s2u_conv *267*
warn_implicitconv *268*
warn_largeargs *269*
warn_missingreturn *269*
warn_no_explicit_virtual *245*
warn_no_side_effect *269*
warn_no_typename *246*
warn_notinlined *246*
warn_padding *270*
warn_pch_portability *270*
warn_possiblyuninitializedvar *274*
warn_possunwant *271*
warn_ptr_int_conv *272*
warn_resultnotused *272*
warn_structclass *246*
warn_undefmacro *273*
warn_uninitializedvar *273*
warn_unusedarg *274*
warn_unusedvar *275*
warning *261*
warning_errors *261*
Warnings Options
    Empty Declarations (most) *42*
    Enable Warnings *42*
    Expression Has No Side Effect (most) *42*
    Extended Error Checks (most) *42*
    Extra Commas (most) *42*
    Hidden Virtual Functions (most) *42*
    Illegal #pragmas (most) *42*
    Implicit Arithmetic Conversions (all) *42*
    Implicit Float to Integer Conversions (all) *42*
    Implicit Integer to Float Conversions (all) *42*
    Implicit Signed/Unsigned Conversions (all) *42*
    Inconsistent 'class'/'struct' Usage (most) *42*
    Incorrect Capitalization in #include "..." (most) *43*
    Incorrect Capitalization in System #include <...> (most) *43*
    Missing 'return' Value in Non-Void-Returning Function (most) *42*
    Non-Inlined Functions (full) *43*
    Pad Bytes Added (full) *43*
    Pointer/Integer Conversions (most) *42*
    Possible Unwanted Effects (most) *42*
    Token Not Formed by ## Operator (most) *43*
    Treat All Warnings as Errors *42*
    Undefined Macro in #if/#elif (full) *43*
    Unused Arguments (most) *42*
    Unused Result from Non-Void-Returning Function (full) *42*
    Unused Variables (most) *42*
wchar_type *247*

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Information in this document is provided solely to enable system and software implementers to use Freescale products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits based on the information in this document. Freescale reserves the right to make changes without further notice to any products herein.

Freescale makes no warranty, representation, or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale data sheets and/or specifications can and do vary in different applications, and actual performance may vary over time. All operating parameters, including "typicals," must be validated for each customer application by customer's technical experts. Freescale does not convey any license under its patent rights nor the rights of others. Freescale sells products pursuant to standard terms and conditions of sale, which can be found at the following address: freescale.com/SalesTermsandConditions.

Freescale and the Freescale logo are trademarks of Freescale Semiconductor, Inc., Reg. U.S. Pat. & Tm. Off. All other product or service names are the property of their respective owners. ARM is the registered trademark of ARM Limited. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© 2010–2014 Freescale Semiconductor, Inc.

Document Number CWMCUS12ZCMPREF
Revision 10.6, 02/2014

freescale™