Freescale™ and the Freescale logo are trademarks of Freescale Semiconductor, Inc. CodeWarrior is a trademark or registered trademark of Freescale Semiconductor, Inc. in the United States and/or other countries. All other product or service names are the property of their respective owners.

Copyright © 2006 by Freescale Semiconductor, Inc. All rights reserved.

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor 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 Semiconductor 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 Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

How to Contact Us

<table>
<thead>
<tr>
<th>Corporate Headquarters</th>
<th>Freescale Semiconductor, Inc.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>7700 West Parmer Lane</td>
</tr>
<tr>
<td></td>
<td>Austin, TX 78729</td>
</tr>
<tr>
<td></td>
<td>U.S.A.</td>
</tr>
</tbody>
</table>

World Wide Web         | http://www.freescale.com/codewarrior |
Technical Support       | http://www.freescale.com/support  |
# Table of Contents

1 Introduction 19
   CodeWarrior Build Tools Versions ........................................ 19
   Compiler Architecture ..................................................... 19
   Linker Architecture ...................................................... 20

2 Using Build Tools with the CodeWarrior IDE 23
   Invoking CodeWarrior Compilers and Linkers .......................... 23
   Specifying File Locations ................................................ 23
   IDE Options and Pragmas .................................................. 24
   IDE Settings Panels ....................................................... 24
   C/C++ Language Settings Panel ......................................... 24
   C/C++ Preprocessor Panel ................................................ 30
   C/C++ Warnings Panel ..................................................... 32

3 Using Build Tools on the Command Line 39
   Configuring Command-Line Tools ........................................ 39
   CWFolder Environment Variable ......................................... 40
   Setting the PATH Environment Variable ............................... 40
   Invoking Command-Line Tools .......................................... 40
   Getting Help ................................................................... 41
   Help Guidelines ................................................................ 41
   File Name Extensions ....................................................... 43

4 Command-Line Standard C Conformance 45
   -ansi ............................................... 45
   -stdkeywords ........................................ 45
   -strict ............................................... 46

5 Command-Line Standard C++ Conformance 47
   -ARM ............................................... 47
   -bool ............................................... 47
   -Cpp_exceptions ........................................ 47
# Table of Contents

- **-dialect** ................................................................. 48  
- **-for_scoping** ......................................................... 48  
- **-instmgr** ............................................................... 49  
- **-iso_templates** ...................................................... 49  
- **-RTTI** ................................................................... 50  
- **-som** ................................................................... 50  
- **-som_env_check** ...................................................... 50  
- **-wchar_t** ................................................................. 50  

## 6 Command-Line Language Translation 51

- **-char** ................................................................. 51  
- **-defaults** .............................................................. 51  
- **-encoding** .............................................................. 52  
- **-flag** ................................................................... 53  
- **-gccext** ................................................................. 53  
- **-gcc_extensions** .................................................... 53  
- **-M** ................................................................... 54  
- **-make** ................................................................... 54  
- **-mapcr** ................................................................... 54  
- **-MM** ................................................................... 55  
- **-MD** ................................................................... 55  
- **-MMD** ................................................................... 55  
- **-msext** ................................................................. 56  
- **-multibyteaware** ................................................... 56  
- **-once** ................................................................... 56  
- **-pragma** ............................................................. 57  
- **-relax_pointers** .................................................... 57  
- **-requireprotos** ..................................................... 57  
- **-search** ............................................................... 58  
- **-trigraphs** ............................................................ 58  

## 7 Command-Line Diagnostic Messages 59

- **-disassemble** ......................................................... 59  
- **-help** ................................................................... 59  
- **-maxerrors** ............................................................ 60
Table of Contents

-maxwarnings .......................................................... 61
-msgstyle ............................................................... 61
-nofail ................................................................. 62
-progress .............................................................. 62
-S ................................................................. 62
-stderr ................................................................. 62
-verbose ............................................................... 63
-version ............................................................... 63
-timing ................................................................. 63
-warnings ............................................................. 64
-wraplines ............................................................ 67

8 Command-Line Preprocessing and Precompilation 69
-convertpaths .......................................................... 69
-cwd ................................................................. 70
-D+ ................................................................. 70
-define ............................................................... 70
-E ................................................................. 71
-EP ................................................................. 71
-gccincludes .......................................................... 71
-I- ................................................................. 72
-I+ ................................................................. 72
-include ............................................................. 73
-ir ................................................................. 73
-P ................................................................. 73
-precompile ........................................................... 74
-preprocess ........................................................... 74
-ppopt ............................................................... 75
-prefix ............................................................... 75
-noprecompile ........................................................ 76
-nosyspath ............................................................ 76
-stdinc .............................................................. 76
-U+ ................................................................. 76
-undefine ............................................................. 77
# Table of Contents

## 9 Command-Line Library and Linking

- -keepobjects .......................................................... 79
- -nolink ............................................................... 79
- -o ................................................................. 80

## 10 Command-Line Object Code

- -c ................................................................. 81
- -codegen ............................................................ 81
- -enum ............................................................... 81
- -min_enum_size ..................................................... 82
- -ext ............................................................... 82
- -strings ........................................................... 83

## 11 Command-Line for Optimization

- -inline .............................................................. 85
- -O ................................................................. 86
- -O+ ................................................................. 86
- -opt ............................................................... 87

## 12 Linker

Specifying Link Order in the IDE .................................. 91
Defining Sections in Source Code .................................. 92
Using a Linker Command File ....................................... 92
  Dead-Stripping ..................................................... 93
Defining the Target’s Memory Map ................................ 93
Defining Sections in the Output File .............................. 94
Associating Input Sections With Output Sections ............... 95
Controlling Alignment ............................................... 96
Specifying Memory Area Locations and Sizes .................... 97
Linker Command File Syntax ....................................... 98
Commands, Directives, and Keywords ............................. 102
  . (location counter) ............................................... 102
ADDR .............................................................. 103
ALIGN ............................................................. 104
## Table of Contents

ALIGNALL  ................................................................. 104
EXCEPTION ................................................................. 105
EXPORTSTRTAB .......................................................... 105
EXPORTSYMTAB .......................................................... 106
FORCE_ACTIVE ........................................................... 107
IMPORTSTRTAB .......................................................... 107
IMPORTSYMTAB .......................................................... 108
INCLUDE ................................................................. 109
KEEP_SECTION ........................................................... 109
MEMORY ................................................................. 110
OBJECT ................................................................. 111
REF_INCLUDE ............................................................ 112
SECTIONS ............................................................... 112
SIZEOF ................................................................. 113
SIZEOF_ROM ............................................................. 114
WRITEB ................................................................. 114
WRITEH ................................................................. 114
WRITEW ................................................................. 115
WRITES0COMMENT ....................................................... 115
ZERO_FILL_UNINITIALIZED ............................................. 116

13 ColdFire Linker ......................................................... 119

Deadstripping ............................................................ 119
Executable files in Projects .......................................... 120
S-Record Comments ...................................................... 120
LCF Structure ............................................................ 120
Memory Segment ........................................................ 120
Closure Segments ........................................................ 121
Sections Segment ........................................................ 122
LCF Syntax ............................................................... 123
Variables, Expressions, and Integrals .............................. 123
Arithmetic, Comment Operators ...................................... 124
Alignment ................................................................. 125
Specifying Files and Functions ...................................... 126
Stack and Heap .......................................................... 127
# Table of Contents

- Static Initializers ............................................................. 127
- Exception Tables ............................................................... 128
- Position-Independent Code and Data .................................... 128
- ROM-RAM Copying ............................................................... 129
- Writing Data Directly to Memory ......................................... 130

## 14 C Compiler 133

- Extensions to Standard C .................................................... 133
- Controlling Standard C Conformance ..................................... 133
- C++-style Comments ............................................................ 134
- Unnamed Arguments ............................................................ 134
- Extensions to the Preprocessor ............................................. 134
- Non-Standard Keywords ...................................................... 135
- C99 Extensions ................................................................. 135
- Controlling C99 Extensions .................................................. 136
- Trailing Commas in Enumerations ......................................... 136
- Compound Literal Values ..................................................... 137
- Designated Initializers ....................................................... 137
- Predefined Symbol __func__ .................................................. 137
- Implicit Return From main() ............................................... 138
- Non-constant Static Data Initialization ................................... 138
- Variable Argument Macros .................................................. 138
- Extra C99 Keywords ............................................................ 139
- C++-Style Comments ........................................................... 139
- C++-Style Digraphs ............................................................. 140
- Empty Arrays in Structures .................................................. 140
- Hexadecimal Floating-Point Constants .................................... 140
- Variable-Length Arrays ....................................................... 141
- Unsuffixed Decimal Literal Values ........................................ 142

- GCC Extensions ................................................................. 142
- Controlling GCC Extensions ................................................ 143
- Initializing Automatic Arrays and Structures ......................... 143
- The sizeof() Operator ........................................................ 143
- Statements in Expressions ................................................... 144
- Redefining Macros ............................................................. 144
Table of Contents

The typeof() Operator .............................................. 145
Void and Function Pointer Arithmetic ............................ 145
The __builtin_constant_p() Operator ........................... 145
Forward Declarations of Static Arrays ......................... 145
Omitted Operands in Conditional Expressions .................. 146
The __builtin_expect() Operator ............................... 146
Void Return Statements ............................................. 147
Minimum and Maximum Operators ................................. 147

15 C++ Compiler .................................................. 149
C++ Compiler Performance ......................................... 149
Precompiling C++ Source Code .................................... 149
Using the Instance Manager ....................................... 149
Extensions to Standard C++ ........................................ 150
__PRETTY_FUNCTION__ Identifier ................................. 150
Standard and Non-Standard Template Parsing .................... 150
Implementation-Defined Behavior .................................. 153
GCC Extensions .................................................. 156
Using the :: Operator in Class Declarations .................... 156
Embedded C++ ..................................................... 157
Activating EC++ ................................................... 157
Differences Between ISO C++ and EC++ ........................... 157
EC++ Specifications ............................................... 158

16 Tool Performance .............................................. 161
Precompiling ..................................................... 161
When to Use Precompiled Files ................................... 161
What Can be Precompiled ........................................ 162
Using a Precompiled Header File .................................. 162
Preprocessing and Precompiling ................................... 163
Pragma Scope in Precompiled Files ................................. 164
Precompiling a File in the CodeWarrior IDE ...................... 164
Updating a Precompiled File Automatically ....................... 165

Table of Contents

17 Intermediate Optimizations

- Interprocedural Analysis .................................................. 167
- Invoking Interprocedural Analysis .................................... 168
- File-Level Optimizations .................................................. 168
- Program-Level Optimizations .......................................... 168
- Program-Level Requirements ............................................ 168
- Intermediate Optimizations ............................................. 171
  - Dead Code Elimination ............................................... 171
  - Expression Simplification ........................................... 172
  - Common Subexpression Elimination ................................. 174
  - Copy Propagation .................................................... 175
  - Dead Store Elimination ............................................. 176
  - Live Range Splitting ................................................ 177
  - Loop-Invariant Code Motion ....................................... 178
  - Strength Reduction .................................................. 180
  - Loop Unrolling ........................................................ 181
- Inlining ................................................................. 182
  - Choosing Which Functions to Inline ............................. 182
  - Inlining Techniques .................................................. 184

18 Inline Assembly

- Inline Assembly Syntax .................................................. 187
  - Statements .............................................................. 187
  - Additional Syntax Rules ............................................ 188
  - Preprocessor Features ............................................... 189
  - Local Variables and Arguments .................................. 189
  - Returning From a Routine .......................................... 191
- Inline Assembly Directives ............................................ 191
  - dc ................................................................. 192
  - ds ................................................................. 192
  - entry .............................................................. 193
  - fralloc ............................................................ 194
  - frfree ............................................................. 194
  - machine ........................................................... 195
# Table of Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>naked</td>
<td>195</td>
</tr>
<tr>
<td>opword</td>
<td>196</td>
</tr>
<tr>
<td>return</td>
<td>196</td>
</tr>
</tbody>
</table>

## 19 ColdFire Code Generation | 197
- Code Generation Limits | 197
- Integer Representation | 197
- Calling Conventions | 199
- Variable Allocation | 200
- Register Variables | 200
- Position-Independent Code | 201
- Cryptographic Acceleration Instructions | 201

## 20 ColdFire Runtime Libraries | 205
- MSL for ColdFire Development | 205
- Customizing MSL Libraries | 206
- Using MSL for ColdFire | 206
- Serial I/O and UART Libraries | 217
- Reduced Working Set Libraries | 218
- Memory, Heaps, and Other Libraries | 219
- Runtime Libraries | 220
- Position-Independent Code | 224
- Board Initialization Code | 225
- Custom Modifications | 225

## 21 Predefined Symbols | 227
- __cplusplus | 227
- __DATE__ | 227
- __embedded_cplusplus | 228
- __FILE__ | 228
- __func__ | 228
- __FUNCTION__ | 229
- __ide_target() | 229
- __LINE__ | 230
- __MWERKS__ | 230
# Table of Contents

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>PRETTY_FUNCTION</strong></td>
<td>230</td>
</tr>
<tr>
<td><strong>profile</strong></td>
<td>231</td>
</tr>
<tr>
<td><strong>STDC</strong></td>
<td>231</td>
</tr>
<tr>
<td><strong>TIME</strong></td>
<td>232</td>
</tr>
<tr>
<td><strong>BACKENDVERSION</strong></td>
<td>233</td>
</tr>
<tr>
<td><strong>COLDIRE</strong></td>
<td>233</td>
</tr>
<tr>
<td><strong>STDABI</strong></td>
<td>234</td>
</tr>
<tr>
<td><strong>REGABI</strong></td>
<td>235</td>
</tr>
</tbody>
</table>

## 22 ColdFire Predefined Symbols

- __BACKENDVERSION__ .......................... 233
- __COLDIRE__ ...................... 233
- __STDABI__ ............................. 234
- __REGABI__ ...................... 235

## 23 Using Pragmas

- Checking Pragma Settings .................... 237
- Saving and Restoring Pragma Settings ......... 238
- Determining Which Settings Are Saved and Restored 239
- Illegal Pragmas .......................... 240
- Pragma Scope ............................ 240

## 24 Pragmas for Standard C Conformance

- ANSI_strict ................................ 243
- c99 ........................................ 243
- ignore_oldstyle ................................ 245
- only_std_keywords .......................... 245
- require_prototypes ........................ 246

## 25 Pragmas for C++

- access_errors ............................ 249
- always_inline ................................ 249
- arg_dep_lookup ............................ 249
- ARM_conform ................................ 250
- ARM_scoping ................................ 250
- array_new_delete .......................... 251
- auto_inline ................................ 251
- bool ........................................ 251
- cplusplus .................................. 252
# Table of Contents

- cpp_extensions .......................................................... 252
- debuginline .............................................................. 253
- def_inherited ............................................................ 254
- defer_codegen ............................................................ 255
- defer_defarg_parsing .................................................... 255
- direct_destruction ....................................................... 256
- direct_to_som ............................................................ 256
- dont_inline ............................................................... 256
- ecplusplus ................................................................. 256
- exceptions ............................................................... 257
- extended_errorcheck ..................................................... 257
- inline_bottom_up ........................................................ 259
- inline_bottom_up_once ................................................... 260
- inline_depth .............................................................. 260
- inline_max_auto_size .................................................... 261
- inline_max_size .......................................................... 261
- inline_max_total_size ................................................... 262
- internal ................................................................. 262
- new_mangler ............................................................... 263
- no_conststringconv ...................................................... 263
- no_static_dtors .......................................................... 264
- nosyminline .............................................................. 264
- old_pods ................................................................. 265
- old_vtable ............................................................... 265
- opt_classresults ........................................................ 265
- parse_func_tpl ............................................................ 266
- parse_mfunc_tpl .......................................................... 266
- RTTI .............................................................. 267
- suppress_init_code ........................................................ 267
- template_depth ........................................................... 268
- thread_safe_init ........................................................ 268
- warn_hidevirtual ........................................................ 269
- warn_no_explicit_virtual ................................................. 270
- warn_no_typename ........................................................ 271
- warn_notinlined ........................................................ 271
Table of Contents

warn_structclass ........................................... 272
wchar_type .................................................. 272

26 Pragmas for Language Translation 275
asmpoundcomment ........................................... 275
asmsemicoloncomment ....................................... 275
const_strings ............................................... 276
dollar_identifiers .......................................... 276
gcc_extensions .............................................. 277
mark ........................................................ 278
mpwc_newline ................................................ 278
mpwc_relax .................................................. 279
multibyteaware .............................................. 279
multibyteaware_preserve_literals ...................... 280
text_encoding ................................................ 280
trigraphs ..................................................... 281
unsigned_char .............................................. 282

27 Pragmas for Diagnostic Messages 283
extended_errorcheck ........................................ 283
maxerrorcount ............................................... 284
message ....................................................... 285
showmessagenumber ........................................ 285
show_error_filestack ...................................... 286
suppress_warnings .......................................... 286
sym .......................................................... 286
unused ......................................................... 287
warning ....................................................... 288
warning_errors .............................................. 289
warn_any_ptr_int_conv .................................... 289
warn_emptydecl .............................................. 290
warn_extracomma ............................................ 291
warn_filenamecaps .......................................... 291
warn_filenamecaps_system .................................. 292
warn_hiddenlocals .......................................... 293
Table of Contents

warn_illpragma ...................................................... 293
warn_illtokenpasting .............................................. 294
warn_illunionmembers ............................................. 294
warn_impl_f2i_conv ................................................ 295
warn_impl_i2f_conv ................................................ 295
warn_impl_s2u_conv ............................................... 296
warn_implicitconv ................................................ 297
warn_largeargs ..................................................... 298
warn_missingreturn ................................................ 298
warn_no_side_effect .............................................. 299
warn_padding ....................................................... 299
warn_pch_portability ............................................. 300
warn_possunwant .................................................. 300
warn_ptr_int_conv ................................................ 302
warn_resultnotused ............................................... 302
warn.Undefmacro .................................................. 303
warn_uninitializedvar ............................................ 304
warn_unusedarg ..................................................... 304
warn_unusedvar ..................................................... 305

28 Pragmas for Preprocessing and Precompilation 307

check_header_flags .............................................. 307
faster_pch_gen ..................................................... 307
flat_include ....................................................... 308
fullpath_file ...................................................... 308
fullpath_prepdump ................................................. 309
keepcomments ...................................................... 309
line_prepdump ..................................................... 309
macro_prepdump ................................................... 310
msg_show_lineref .................................................. 310
msg_show_realref .................................................. 310
notonce ............................................................. 311
oldPragma_once .................................................... 311
once ................................................................. 311
pop, push ........................................................... 312
Table of Contents

pragma_prepdump .......................................................313
precompile_target ...................................................313
simple_prepdump .....................................................314
space_prepdump ......................................................314
srcrelincludes .......................................................315
syspath_once ........................................................315

29 Pragmas for Library and Linking 317
always_import .......................................................317
export ...............................................................317
force_active .........................................................318
import ..............................................................318
lib_export ..........................................................319

30 Pragmas for Code Generation 321
dont_reuse_strings ..................................................321
enumsalwaysint .......................................................322
erro_name ...........................................................323
explicit_zero_data ...................................................323
float_constants .......................................................324
instmgr_file ........................................................324
longlong ..............................................................325
longlong Enums .......................................................325
min_enum_size ........................................................326
options ...............................................................326
pool_strings ........................................................327
readonly_strings ....................................................328
reverse_bitfields ....................................................328
store_object_files ..................................................329

31 Pragmas for Optimization 331
global_optimizer .....................................................331
ipa .................................................................331
opt_common_subs ....................................................332
opt_dead_assignments ...............................................332
# Table of Contents

<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>opt_dead_code</td>
<td>333</td>
</tr>
<tr>
<td>opt_lifetimes</td>
<td>333</td>
</tr>
<tr>
<td>opt_loop_invariants</td>
<td>333</td>
</tr>
<tr>
<td>opt_propagation</td>
<td>334</td>
</tr>
<tr>
<td>opt_strength_reduction</td>
<td>334</td>
</tr>
<tr>
<td>opt_strength_reduction_strict</td>
<td>335</td>
</tr>
<tr>
<td>opt_unroll_loops</td>
<td>335</td>
</tr>
<tr>
<td>opt_vectorize_loops</td>
<td>335</td>
</tr>
<tr>
<td>optimization_level</td>
<td>336</td>
</tr>
<tr>
<td>optimize_for_size</td>
<td>336</td>
</tr>
<tr>
<td>optimize_withasm</td>
<td>337</td>
</tr>
<tr>
<td>strict_header_checking</td>
<td>337</td>
</tr>
</tbody>
</table>

## Pragmas for ColdFire

| ColdFire Diagnostic Pragmas                                        | 339  |
| SDS_debug_support                                                  | 339  |
| ColdFire Library and Linking Pragmas                              | 339  |
| define_section                                                     | 339  |
| ColdFire Code Generation Pragmas                                   | 341  |
| codeColdFire                                                       | 341  |
| const_multiply                                                     | 342  |
| emac                                                                | 342  |
| explicit_zero_data                                                 | 342  |
| inline_intrinsics                                                  | 343  |
| interrupt                                                          | 343  |
| readonly_strings                                                   | 344  |
| section                                                            | 344  |
| ColdFire Optimization Pragmas                                      | 345  |
| opt_unroll_count                                                   | 345  |
| opt_unroll_instr_count                                             | 345  |
| profile                                                            | 345  |
Introduction

This reference describes how to use CodeWarrior build 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 often run on a different platform than the programs they generate. Build tools run on the host platform to generate software that runs on the target platform.

This chapter describes what this reference covers and the processes that CodeWarrior build tools use to create software:
- CodeWarrior Build Tools Versions
- Compiler Architecture
- Linker Architecture

CodeWarrior Build Tools Versions

This reference covers the CodeWarrior compiler version 4.0 and its related linker.

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 into several steps.

Figure 1.1 shows the steps the compiler takes to coordinate its front-end and back-end to translate source code into object code.

- reading settings: the compiler retrieves your settings from the CodeWarrior IDE or the command line to determine what files to translate and how they should be translated in subsequent steps
- preprocessing: reads your program’s source code files then preprocesses them
- front-end translation: translates your program’s preprocessed source code into a platform-independent intermediate representation
- front-end optimization: rearranges the intermediate representation to reduce your program’s size or improve its performance while preserving its logic
Introduction

Linker Architecture

- back-end translation: converts the optimized intermediate representation into native object code, containing data and instructions, for the target processor
- back-end optimization: specific to a target platform, rearranges the native object code to reduce its size or improve performance
- output: writes object code and other data, ready for linking

Figure 1.1 CodeWarrior compiler steps

Linker Architecture

A linker combines and arranges data and instructions from one or more object code files into a single file, or image. This image is ready to execute on the target platform. The CodeWarrior linker uses settings from the CodeWarrior IDE or command line to...
determine how to generate the image file. The linker also uses an optional linker command file. A linker command file allows you to specify precise details of how data and instructions should be arranged in the linker’s output file.

Figure 1.2 shows the steps the CodeWarrior linker takes to build an executable image.

**Figure 1.2 CodeWarrior linker steps**

- **start linker**
- **read settings**
  - settings from the IDE or command line
- **read linker command file**
  - linker command file
- **read object code**
  - object code files
- **resolve references among objects**
  - delete unused objects ("deadstripping")
- **output link map and image files**
  - link map and executable image files
Using Build Tools with the CodeWarrior IDE

The CodeWarrior Integrated Development Environment (IDE) uses settings in a project’s build target to choose which compilers and linkers to invoke, which files those compilers and linkers will process, and which options the compilers and linkers will use.

This chapter describes how to use CodeWarrior compilers and linkers with the CodeWarrior IDE:

- **Invoking CodeWarrior Compilers and Linkers**
- **Specifying File Locations**
- **IDE Options and Pragmas**
- **IDE Settings Panels**

### Invoking CodeWarrior Compilers and Linkers

The IDE uses settings in the **Target Settings** panel of the build-target Settings window, where **build-target** is the name of the current build target, to determine which compilers and linkers to use. The **Linker** option in this settings panel specifies the platform or processor to build for. From this option, the IDE also determines which compilers, pre-linkers, and post-linkers to use.

The IDE uses the settings in the **File Mappings** panel of the build-target Settings window to determine which types of files may be added to a project’s build target and which compiler should be invoked to process each file. The menu of compilers in the **Compiler** option of this panel is determined by the **Linker** setting in the Target Settings panel.

### Specifying File Locations

The IDE uses the settings in a build target’s **Access Paths** and **Source Trees** panels to choose the source code and object code files to dispatch to the CodeWarrior build tools. See the **IDE User’s Guide** for more information on these panels.
IDE Options and Pragmas

The build tools determine their settings by IDE settings and directives in source code. The CodeWarrior compiler follows these steps to determine the settings to apply to each file that the compiler translates under the IDE:

- before translating the source code file, the compiler gets option settings from the IDE’s settings panels in the current build target
- the compiler updates the settings for pragmas that correspond to panel settings
- the compiler translates the source code in the **Prefix Text** field of the build target’s **C/C++ Preprocessor** panel

The compiler applies pragma directives and updates their settings as pragmas directives are encountered in this source code.

- the compiler translates the source code file and the files that it includes
  The compiler applies pragma settings as it encounters them.

IDE Settings Panels

A build target that uses a CodeWarrior C or C++ compiler has these settings panels to control the compiler:

- **C/C++ Language Settings Panel**
- **C/C++ Preprocessor Panel**
- **C/C++ Warnings Panel**

C/C++ Language Settings Panel

This settings panel controls compiler language features and some object code storage features for the current build target.

- **Force C++ Compilation**
- **ISO C++ Template Parser**
- **Use Instance Manager**
- **Enable C++ Exceptions**
- **Enable RTTI**
- **Enable bool Support**
- **Enable wchar_t Support**
- **EC++ Compatibility Mode**
Using Build Tools with the CodeWarrior IDE

IDE Settings Panels

- Inline Depth
- Auto-Inline
- Bottom-up Inlining
- ANSI Strict
- ANSI Keywords Only
- Expand Trigraphs
- Legacy for-scoping
- Require Function Prototypes
- Enable C99 Extensions
- Enable GCC Extensions
- Enums Always Int
- Use Unsigned Chars
- Pool Strings
- Reuse Strings

Force C++ Compilation

When on, translates all C source files as C++ source code. When off, the IDE uses the file name’s extension to determine whether to use the C or C++ compiler. The entries in the IDE’s File Mappings settings panel specify the suffixes that the compiler assigns to each compiler.

This setting corresponds to the pragma `cplusplus` and the command-line option `-lang c++`.

ISO C++ Template Parser

When on, follows the ISO/IEC 14882-1998 standard for C++ to translate templates, enforcing more careful use of the `typename` and `template` keywords. When on, the compiler also follows stricter rules for resolving names during declaration and instantiation. When off, the C++ compiler does not expect template source code to follow the ISO C++ standard as closely.

This setting corresponds to the `parse_func_templ` pragma. It corresponds to the command-line option `-iso_templates`.

CodeWarrior Build Tools Reference ColdFire™ Architectures Edition
Using Build Tools with the CodeWarrior IDE
IDE Settings Panels

Use Instance Manager
When on, reduces compile time by generating any instance of a C++ template (or non-inlined inline) function only once. When off, generates a new instance of a template or non-inlined function each time it appears in source code.

You can control where the instance database is stored using the #pragma instmgr_file. This setting corresponds to the command-line option -instmgr.

Enable C++ Exceptions
When on, generates executable code for C++ exceptions. When off, generates smaller, faster executable code.

Enable the Enable C++ Exceptions setting if you use the try, throw, and catch statements specified in the ISO/IEC 14882-1998 C++ standard. Otherwise, disable this setting to generate smaller and faster code.

This setting corresponds to the pragma exceptions and the command-line option -cpp_exceptions.

Enable RTTI
When on, allows the use of the C++ runtime type information (RTTI) capabilities, including the dynamic_cast and typeid operators. When off, the compiler generates smaller, faster object code but does not allow runtime type information operations.

This setting corresponds to the pragma RTTI and the command-line option -RTTI.

Enable bool Support
When on, the C++ compiler recognizes the bool type and its true and false values specified in the ISO/IEC 14882-1998 C++ standard. When off, the compiler does not recognize this type or its values.

This setting corresponds to the pragma bool and the command-line option -bool.

Enable wchar_t Support
When on, the C++ compiler recognizes the wchar_t data type specified in the ISO/IEC 14882-1998 C++ standard. When off, the compiler does not recognize this type.

Turn off this option when compiling source code that defines its own wchar_t type.
This setting corresponds to the pragma wchar_type and the command-line option -wchar_t.
EC++ Compatibility Mode

When on, expects C++ source code files to contain Embedded C++ source code. When off, the compiler expects regular C++ source code in C++ source files.

This setting corresponds to the pragma `ecplusplus` and the command-line option `-dialect ec++`.

Inline Depth

Specifies the policy to follow to determine the level of function calls to replace with function bodies. These policies are listed in Table 2.1.

<table>
<thead>
<tr>
<th>This setting</th>
<th>Does this...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Don’t Inline</td>
<td>Inlines no functions, not even C or C++ functions declared <code>inline</code>.</td>
</tr>
<tr>
<td>Smart</td>
<td>Inlines small functions to a depth of 2 to 4 inline functions deep.</td>
</tr>
<tr>
<td>1 to 8</td>
<td>Inlines to the depth specified by the numerical selection.</td>
</tr>
</tbody>
</table>

The Smart and 1 to 8 items correspond to the pragma `inline_depth` and the command-line option `-inline level=n`, where `n` is 1 to 8. The Don’t Inline item corresponds to the pragma `dont_inline` and the command-line option `-inline off`.

Auto-Inline

Lets the compiler choose which functions to inline. Also inlines C++ functions declared `inline` and member functions defined within a class declaration. This setting corresponds to the pragma `auto_inline` and the command-line option `-inline auto`.

Bottom-up Inlining

Inline functions starting at the last function to the first function in a chain of function calls. This setting corresponds to the pragma `inline_bottom_up` and the command-line option `-inline bottomup`. 
Using Build Tools with the CodeWarrior IDE
IDE Settings Panels

ANSI Strict

Only recognizes source code that conforms to the ISO/IEC 9899-1990 standard for C. The compiler does not recognize several CodeWarrior extensions to the C language:

- C++-style comments
- unnamed arguments in function definitions
- a # not followed by a macro directive
- using an identifier after a #endif directive
- using typecasted pointers as lvalues
- converting points to type of the same size
- arrays of zero length in structures
- the D constant suffix
- enumeration constant definitions that cannot be represented as signed integers when the Enums Always Int option is on in the IDE’s C/C++ Language settings panel or the enumalwaysint pragma is on
- a C++ main() function that does not return an integer value

You cannot enable individual extensions that are controlled by the ANSI Strict setting.

This setting corresponds to the pragma ANSI_strict and the command-line option -ansi strict.

ANSI Keywords Only

Controls whether the compiler recognizes non-standard keywords.

(ISO/IEC 9899-1990 C, §6.4.1) The CodeWarrior compiler can recognize several additional reserved keywords. If you enable this setting, the compiler generates an error message if it encounters any of the additional keywords that it recognizes. If you must write source code that strictly adheres to the ISO standard, enable the ANSI Strict setting.

If you disable this setting, the compiler recognizes the following non-standard keywords:

far, inline, __inline__, __inline, and pascal.

This setting corresponds to the pragma only_std_keywords and the command-line option -stdkeywords.

Expand Trigraphs

(ISO/IEC 9899-1990 C, §5.2.1.1) The compiler normally ignores trigraph characters. Many common character constants look like trigraph sequences, and this extension lets you use them without including escape characters.
This setting corresponds to the pragma `trigraphs` and the command-line option `-trigraphs`.

### Legacy for-scoping
Generates an error message when the compiler encounters a variable scope usage that the ISO/IEC 14882-1998 C++ standard disallows, but is allowed in the C++ language specified in *The Annotated C++ Reference Manual* ("ARM").

This setting corresponds to the pragma `ARM_scoping` and the command-line option `-for_scoping`.

### Require Function Prototypes
Enforces the requirement of function prototypes. If you enable the **Require Function Prototypes** setting, 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, then enabling the **Require Function Prototypes** setting causes the compiler to issue a warning message.

This setting corresponds to the pragma `require_prototypes` and the command-line option `-requireprotos`.

### Enable C99 Extensions
Recognizes ISO/IEC 9899-1999 ("C99") language features that are supported by the CodeWarrior compiler.

This setting corresponds to the pragma `c99` and the command-line option `-dialect c99`.

### Enable GCC Extensions
Lets you use language features of the GCC (Gnu Compiler Collection) C compiler that are supported by CodeWarrior.

This setting corresponds to the pragma `gcc_extensions` and the command-line option `-gcc_extensions`.

### Enums Always Int
Uses signed integers to represent enumerated constants. This option corresponds to the `enumsalwaysint` pragma and the command-line option `-enum`.
Using Build Tools with the CodeWarrior IDE
IDE Settings Panels

Use Unsigned Chars
Treats char declarations as unsigned char declarations. This setting corresponds to the pragma unsigned_char and the command-line option -char unsigned.

Pool Strings
Controls where the compiler stores character string literals.
If you enable this setting, the compiler collects all string constants into a single data section in the object code it generates. If you disable this setting, the compiler creates a unique section for each string constant.
This option corresponds to the pragma pool_strings and the command-line option -strings pool.

Reuse Strings
When on, the compiler stores only one copy of identical string literals. When off, the compiler stores each string literal separately.
The Reuse Strings setting corresponds to opposite of the pragma dont_reuse_strings and the command-line option -string reuse.

C/C++ Preprocessor Panel
The C/C++ Preprocessor settings panel controls the operation of the CodeWarrior compiler’s preprocessor.
- Prefix Text
- Source encoding
- Use prefix text in precompiled header
- Emit file changes
- Emit #pragmas
- Show full paths
- Keep comments
- Use #line
- Keep whitespace

Prefix Text
Contains source code that the compiler inserts at the beginning of each translation unit. A translation unit is the combination of a source code file and all the files that it includes.
Using Build Tools with the CodeWarrior IDE

IDE Settings Panels

Source encoding

Allows you to specify the default encoding of source files. The compiler recognizes Multibyte and Unicode source text. To replicate the obsolete option Multi-Byte Aware, set this option to System or Autodetect. Additionally, options that affect the preprocessor request appear in this panel.

Use prefix text in precompiled header

Controls whether the compiler inserts the source code in the Prefix Text field at the beginning of a precompiled header file.

This option defaults to disabled to correspond with previous versions of the compiler that ignore the prefix file when building precompiled headers. If any pragmas are imported from old C/C++ Language Panel settings, this option is enabled.

Emit file changes

Controls whether notification of file changes (or #line changes) appear in the output.

Emit #pragmas

Controls whether pragmas directives encountered in the source text appear in the preprocessor output.

NOTE

This option is essential for producing reproducible test cases for bug reports.

Show full paths

Controls whether file changes show the full path or the base filename of the file.

Keep comments

Controls whether comments are emitted in the output.

Use #line

Controls whether file changes appear in comments (as before) or in #line directives.
Using Build Tools with the CodeWarrior IDE
IDE Settings Panels

Keep whitespace

Controls whether whitespace is stripped out or copied into the output. This is useful for keeping the starting column aligned with the original source, though the compiler attempts to preserve space within the line. This doesn’t apply when macros are expanded.

C/C++ Warnings Panel

The C/C++ Warnings settings panel contains options that control which warning messages the CodeWarrior C/C++ compiler issues as it translates source code:

- Illegal Pragmas
- Possible Errors
- Extended Error Checking
- Hidden Virtual Functions
- Implicit Arithmetic Conversions
- Float To Integer
- Signed/Unsigned
- Integer To Float
- Pointer/Integral Conversions
- Unused Variables
- Unused Arguments
- Missing ‘return’ Statements
- Expression Has No Side Effect
- Enable All
- Disable All
- Extra Commas
- Inconsistent ‘class’/’struct’ Usage
- Empty Declarations
- Include File Capitalization
- Check System Includes
- Pad Bytes Added
- Undefined Macro in #if
- Non-Inlined Functions
- Treat All Warnings As Errors
Illegal Pragmas

Issues a warning message if the compiler encounters an unrecognized pragma. This setting corresponds to the `warn_illpragma` pragma and the command-line option `-warnings illpragmas`.

Possible Errors

Issues warning messages for common, usually-unintended logical errors:

- in conditional statements, using the assignment (`=`) operator instead of the equality comparison (`==`) operator
- in expression statements, using the `==` operator instead of the `=` operator
- placing a semicolon (`;`) immediately after a `do`, `while`, `if`, or `for` statement

This setting corresponds to pragma `warn_possunwant` and the command-line option `-warnings possible`.

Extended Error Checking

Issues warning messages for common programming errors:

- mis-matched return type in a function’s definition and the return statement in the function’s body
- mismatched assignments to variables of enumerated types

This setting corresponds to pragma `extended_errorcheck` and the command-line option `-warnings extended`.

Hidden Virtual Functions

Generates 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.

This setting corresponds to pragma `warn_hidevirtual` and the command-line option `-warnings hidevirtual`.

Implicit Arithmetic Conversions

Issues a warning message when the compiler applies implicit conversions that may not give results you intend:

- assignments where the destination is not large enough to hold the result of the conversion
- a signed value converted to an unsigned value
Using Build Tools with the CodeWarrior IDE
IDE Settings Panels

- an integer or floating-point value is converted to a floating-point or integer value, respectively

This setting corresponds to the warn_implicitconv pragma and the command-line option -warnings implicitconv.

**Float To Integer**

Issues a warning message for implicit conversions from floating point values to integer values.

This setting corresponds to the warn_impl_f2i_conv pragma and the command-line option -warnings impl_float2int.

**Signed/Unsigned**

Issues a warning message for implicit conversions from a signed or unsigned integer value to an unsigned or signed value, respectively.

This setting corresponds to the warn_impl_s2u_conv pragma and the command-line option -warnings signedunsigned.

**Integer To Float**

Issues a warning message for implicit conversions from integer to floating-point values.

This setting corresponds to the warn_impl_i2f_conv pragma and the command-line option -warnings impl_int2float.

**Pointer/Integral Conversions**

Issues a warning message for implicit conversions from pointer values to integer values and from integer values to pointer values.

This setting corresponds to the warn_any_ptr_int_conv and warn_ptr_int_conv pragmas and the command-line option -warnings ptrintconv, anyptrinvconv.

**Unused Variables**

Issues a warning message for local variables that are not referred to in a function.

This setting corresponds to the warn_unusedvar pragma and the command-line option -warnings unusedvar.
Unused Arguments
Issues a warning message for function arguments that are not referred to in a function.
This setting corresponds to the \texttt{warn\_unusedarg} pragma and the command-line option \\
\texttt{-warns unusedarg}.

Missing ‘return’ Statements
Issues a warning message if a function that is defined to return a value has no \texttt{return} statement.
This setting corresponds to the \texttt{warn\_missingreturn} pragma and the command-line option \\
\texttt{-warns missingreturn}.

Expression Has No Side Effect
Issues a warning message if a statement does not change the program’s state.
This setting corresponds to the \texttt{warn\_no\_side\_effect} pragma and the command-line option \\
\texttt{-warns unusedexpr}.

Enable All
Turns on all warning options.

Disable All
Turns off all warning options.

Extra Commas
Issues 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.
This setting corresponds to the \texttt{warn\_extracomma} pragma and the command-line option \\
\texttt{-warns extracomma}.

Inconsistent ‘class’/’struct’ Usage
Issues a warning message if the class and struct keywords are used interchangeably in the definition and declaration of the same identifier in C++ source code.
This setting corresponds to the \texttt{warn\_structclass} pragma and the command-line option \\
\texttt{-warns structclass}.
Empty Declarations

Issues a warning message if a declaration has no variable name.
This setting corresponds to the pragma `warn_emptydecl` and the command-line option `-warnings emptydecl`.

Include File Capitalization

Issues a warning message if the name of the file specified in a `#include "file"` directive uses different letter case from a file on disk.
This setting corresponds to the `warn_filenamecaps` pragma and the command-line option `-warnings filecaps`.

Check System Includes

Issues a warning message if the name of the file specified in a `#include <file>` directive uses different letter case from a file on disk.
This setting corresponds to the `warn_filenamecaps_system` pragma and the command-line option `-warnings sysfilecaps`.

Pad Bytes Added

Issues a warning message when the compiler adjusts the alignment of components in a data structure.
This setting corresponds to the `warn_padding` pragma and the command-line option `-warnings padding`.

Undefined Macro in #if

Issues a warning message if an undefined macro appears in `#if` and `#elif` directives.
This setting corresponds to the `warn_undefmacro` pragma and the command-line option `-warnings undefmacro`.

Non-Inlined Functions

Issues 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.
This setting corresponds to the `warn_notinlined` pragma and the command-line option `-warnings notinlined`. 

Using Build Tools with the CodeWarrior IDE
IDE Settings Panels

---

CodeWarrior Build Tools Reference ColdFire™ Architectures Edition
Treat All Warnings As Errors

Issues warning messages as error messages.
This setting corresponds to the `warning_errors` pragma and the command-line option `-warnings error`. 
Using Build Tools with the CodeWarrior IDE

IDE Settings Panels
Using Build Tools on the Command Line

The CodeWarrior command line compilers and assemblers translate source code into object code, storing this object code in files. CodeWarrior command-line linkers then combine one or more of these object code files to produce an executable image ready to load and execute on the target platform.

Each command-line tool has options that you configure when you invoke the tool. The CodeWarrior IDE (Integrated Development Environment) uses these same compilers and linkers, however CodeWarrior provides versions of these tools that you can directly invoke on the command line. Many command-line options correspond to settings in the IDE’s Target Settings window.

This chapter contains these topics:
- Configuring Command-Line Tools
- Invoking Command-Line Tools
- Getting Help
- File Name Extensions

Configuring Command-Line Tools

To use the command-line tools, several environment variables must be changed or defined.

If you are using CodeWarrior command-line tools with Microsoft Windows, environment variables may be assigned in the autoexec.bat file in Windows 95/98 operating systems or in the Environment tab under the System control panel in Windows NT/2000/XP operating systems.

The CodeWarrior command-line tools refer to environment variables for configuration information:
- CWFolder Environment Variable
- Setting the PATH Environment Variable
Using Build Tools on the Command Line

**Invoking Command-Line Tools**

**CWFolder Environment Variable**

In this example, CWFolder refers to the path where you installed your CodeWarrior software. Note that you must not include quote marks when defining environment variables that include spaces. The Windows operating system will not remove the quotes, which leads to warning messages for unknown directories. Use the following syntax if defining variables in batch files or at the command line (Listing 3.1).

Listing 3.1 Example of setting CWFolder.

```
set CWFolder=C:\Program Files\CodeWarrior
```

**Setting the PATH Environment Variable**

The PATH variable should include the paths for your CodeWarrior tools, shown in Listing 3.2. Toolset represents the name of the folder that contains the command line tools for your build target.

Listing 3.2 Example of setting PATH

```
%CWFolder%\Bin
%CWFolder%\toolset\Command_Line_Tools
```

The first path in Listing 3.2 contains the FlexLM license manager DLL, and the second path contains the tools.

In order for FlexLM to work properly, you can simply copy the following file into the directory from which you will be using the command line tools:

```
..\CodeWarrior\license.dat
```

Alternately, you can define the variable LM_LICENSE_FILE as:

```
%CWFolder%\license.dat
```

This variable points to license information. It may point to alternate versions of this file, as needed.

**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
Using Build Tools on the Command Line

Getting Help

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.

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.

Help Guidelines

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.

Parameter Formats

Parameters in an option are formatted as follows:

- A parameter included in brackets “[]” is optional.
Using Build Tools on the Command Line

Getting Help

- Use of the ellipsis “...” character indicates that the previous type of parameter may be repeated as a list.

Option Formats

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).

Common Terms

These common terms appear in many option descriptions:

- A “cased” option is considered case-sensitive. By default, no options are case-sensitive.

- “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.

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 cannot identify as object code, libraries, or command files. Linker command files must end in .lcf. They may be simply added to the link line, for example (Listing 3.3).

Listing 3.3 Example of using linker command files

mwldtarget file.o lib.a commandfile.lcf

For more information on linker command files, refer to the Targeting manual for your platform.
Command-Line Standard C Conformance

-ansi

Controls the ISO/IEC 9899-1990 ("C89") 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

-stdkeywords

Controls the use of ISO/IEC 9899-1990 ("C89") keywords.

**Syntax**

-stdkeywords on | off
Command-Line Standard C Conformance

Remarks
Default setting is off.

-strict

Controls the use of non-standard ISO/IEC 9899-1990 (“C89”) language features.

Syntax
-strict on | off

Remarks
If this option is on, the compiler generates an error message if it encounters some CodeWarrior extensions to the C language defined by the ISO/IEC 9899-1990 (“C89”) standard:
• C++-style comments
• unnamed arguments in function definitions
• non-standard keywords
The default setting is off.
Command-Line Standard C++ Conformance

-ARM

Deprecated. Use -for_scoping instead.

-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 recognize the keywords, forcing the source code to provide definitions for these names. The default is on.

-Cpp_exceptions

Controls the use of C++ exceptions.

**Syntax**

```
-cpp_exceptions on | off
```
Command-Line Standard C++ Conformance

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. The default is on.

-dialect
Specify the source language.

Syntax
-dialect keyword
-lang keyword
The arguments for keyword are:
c
c99
c++ | cplus
Always treat source as the C++ language.
ecc++
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.

-for_scoping
Controls legacy scope behavior in for loops.

Syntax
-for_scoping
Command-Line Standard C++ Conformance

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`.

-instmgr

Controls whether the instance manager for templates is active.

Syntax

```
-instmgr [,...]
```

The options for `keyword` are:

- `off`
  - Turn off the C++ instance manager. This is the default.
- `on`
  - Turn 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`.

-iso_templates

Controls whether the ISO/IEC 14882-1998 standard C++ template parser is active.

Syntax

```
-iso_templates on | off
```

Remarks
Default setting is `off`. 
Command-Line Standard C++ Conformance

-RTTI

Controls the availability of runtime type information (RTTI).

Syntax

-RTTI on | off

Remarks

Default setting is on.

-som

Obsolete. This option is no longer available.

-som_env_check

Obsolete. This option is no longer available.

-wchar_t

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

Syntax

-wchar_t on | off

Remarks

The wchar_t 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.
6

Command-Line Language
Translation

-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 signed.

-defaults

Controls whether the compiler uses additional environment variables to provide default settings.

Syntax
-defaults
-nodefaults
Command-Line Language Translation

Remarks

This option is global. To tell the command-line compiler to use the same set of
default settings as the CodeWarrior IDE, use -defaults. For example, in the
IDE, all access paths and libraries are explicit. defaults is the default setting.
Use -nodefaults to disable the use of additional environment variables.

-encoding

Specify 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 multibyte encoding.
system
Uses local system format.
UTF8 | -8
Unicode Transformation Format (UTF).
SJIS | Shift-JIS | ShiftJIS
Shift Japanese Industrial Standard (Shift-JIS) format.
EUCJP | -JP
Japanese Extended UNIX Code (EUCJP) format.
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.
Command-Line Language Translation

- **-flag**

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

  **Syntax**
  
  `-f[lag] [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`

- **-gccext**

  Enables GCC (Gnu Compiler Collection) C language extensions.

  **Syntax**
  
  `-gcc[ext] on | off`

  **Remarks**
  
  See “GCC Extensions” on page 142 for a list of language extensions that the compiler recognizes when this option is on.

  The default setting is off.

- **-gcc_extensions**

  Equivalent to the `-gccext` option.
Command-Line Language Translation

Syntax
```
-gcc[._extensions] on | off
```

-M
Scan source files for dependencies and emit a Makefile, without generating object code.

Syntax
```
-M
```

Remarks
This command is global and case-sensitive.

-make
Scan source files for dependencies and emit a Makefile, without generating object code.

Syntax
```
-make
```

Remarks
This command is global.

-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.
Command-Line Language Translation

-**MM**

Scan source files for dependencies and emit a Makefile, without generating object code or listing system `#include` files.

**Syntax**

```
-MM
```

**Remarks**

This command is global and case-sensitive.

-**MD**

Scan source files for dependencies and emit a Makefile, generate object code, and write a dependency map.

**Syntax**

```
-MD
```

**Remarks**

This command is global and case-sensitive.

-**MMD**

Scan source files for dependencies and emit a Makefile, generate object code, write a dependency map, without listing system `#include` files.

**Syntax**

```
-MMD
```

**Remarks**

This command is global and case-sensitive.
Command-Line Language Translation

-msext

Allows Microsoft Visual C++ extensions.

Syntax

-msext on | off

Remarks

Turn on this option to allow Microsoft Visual C++ extensions:

- Redefinition of macros
- Allows XXX : :yyy syntax when declaring method yyy of class XXX
- Allows extra commas
- Ignores casts to the same type
- Treats function types with equivalent parameter lists but different return types as equal
- Allows pointer-to-integer conversions, and various syntactical differences

-multibyteaware

Allows multi-byte characters encodings in source text.

Syntax

-multibyte[aware]
.nomultibyte[aware]

-once

Prevents header files from being processed more than once.

Syntax

-once

Remarks

You can also add #pragma once on in a prefix file.
-pragma

Defines a pragma for the compiler.

**Syntax**

-pragma ‘name ["setting"]’

The arguments are:

- **name**

  Name of the new pragma enclosed in single-quotes.

- **setting**

  Setting for the new pragma. When adding a setting, setting must be enclosed in double-quotes.

-relax_pointers

Relaxes the pointer type-checking rules in C.

**Syntax**

-relaxpointers

**Remarks**

This option is equivalent to

#pragma mpwc_relax on

-requireprotos

Controls whether or not the compiler should expect function prototypes.

**Syntax**

-r[quireprotos]
Command-Line Language Translation

-search

Globally searches across paths for source files, object code, and libraries specified in the command line.

Syntax
-search

-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

Command-Line Diagnostic Messages

-disassemble

Tells the command-line tool to disassemble files and send result to stdout.

Syntax
-dis[assemble]

Remarks
This option is global.

-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).

[nocompatible]

Use compatible to show options compatible with this compiler. Use nocompatible to show options that do not work with this compiler.
Command-Line Diagnostic Messages

[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); for 'keyword', 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.

-maxerrors

Specify the maximum number of errors messages to show.

Syntax
   -maxerrors max
Command-Line Diagnostic Messages

**max**

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

- 0 (zero) – disable maximum count, show all error messages.
- 100 – Default setting.

**-maxwarnings**

Specify the maximum number of warning messages to show.

**Syntax**

```
-maxerrors max
```

`max` specifies the number of warning messages. Common values are:

- 0 (zero) – Disable maximum count (default).
- `n` – Maximum number of warnings to show.

**-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.
Command-Line Diagnostic Messages

**std**

Uses standard message style. This is the default.

**-nofail**

Continues processing after getting error messages in earlier files.

**Syntax**

-nofail

**-progress**

Shows progress and version information.

**Syntax**

-progress

**-S**

Disassembles all files and send output to a file. This command is global and case-sensitive.

**Syntax**

-S

**-stderr**

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

**Syntax**

-stderr
Command-Line Diagnostic Messages

-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.

-verbose

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

Syntax
-v[erbose]

Remarks
This option also gives progress and version information.

-version

Displays version, configuration, and build data.

Syntax
-v[ersion]

-timing

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

Syntax
-timing
Command-Line Diagnostic Messages

-warnings

Specify 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 illegal 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_posunwanted

[no]unusedarg
   Issues warning messages on unused arguments. Equivalent to
   #pragma warn_unusedarg
Command-Line Diagnostic Messages

[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]notinlined

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

#pragma warn_notinlined
Command-Line Diagnostic Messages

[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
Command-Line Diagnostic Messages

[no]filecaps
   Issues warning messages when #include " " directives use incorrect capitalization. Equivalent to
   #pragma warn_filenamecaps

[no]sysfilecaps
   Issue warning messages when #include <> statements use incorrect capitalization. Equivalent to
   #pragma warn_filenamecaps_system

[no]tokenpasting
   Issue warning messages when token is not formed by the ## preprocessor operator.
   Equivalent to
   #pragma warn_illtokenpasting

display | dump
   Display list of active warnings.

-wrapping

Controls the word wrapping of messages.

Syntax
-wrapping
-nowrapping
Command-Line Diagnostic Messages
Command-Line Preprocessing and Precompilation

-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 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>. 
Command-Line Preprocessing and Precompilation

-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.

-D+

Same as the -define option.

**Syntax**
-D+name

The parameters are:

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

-define

Defines a preprocessor symbol.
Command-Line Preprocessing and Precompilation

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.

-E
Tells the command-line tool to preprocess source files.

Syntax
- E

Remarks
This option is global and case sensitive.

-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.

-gccincludes
Controls the compilers use of GCC #include semantics.
Command-Line Preprocessing and Precompilation

Syntax

-gccinc[cludes]

Remarks

Use -gccinclude 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.

-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. 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.

-I+

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

Syntax

- I+path
Command-Line Preprocessing and Precompilation

-\texttt{i\ path}

The parameters are:
\begin{verbatim}
path
\end{verbatim}

The non-recursive access path to append.

\textbf{Remarks}

This command is global and case-sensitive.

-\texttt{-include}

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

\textbf{Syntax}

\begin{verbatim}
-include file
\end{verbatim}

file

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

\textbf{Remarks}

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

-\texttt{-ir}

Appends a recursive access path to the current \texttt{#include} list. This command is global.

\textbf{Syntax}

\begin{verbatim}
-ir path
\end{verbatim}

path

The recursive access path to append.

-\texttt{-P}

Preprocess the source files without generating object code, and send output to file.
Command-Line Preprocessing and Precompilation

**Syntax**

- `P`

**Remarks**

This option is global and case-sensitive.

---

**-precompile**

Precompile 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 filesourc
```

is equivalent to

```
-c -o filesourc
```

---

**-preprocess**

Preprocess the source files. This command is global.

**Syntax**

```
-preprocess
```
Command-Line Preprocessing and Precompilation

-ppopt

Specify 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.

-prefix

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

Syntax

-prefix file
Command-Line Preprocessing and Precompilation

-noprecompile

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

Syntax
-noprecompile

-nosyspath

Perform searches 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.

-stdinc

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

Syntax
-stdinc
-nostdinc

Remarks
Add this option after all system -I paths.

-U+

Same as the -undefine option.
Command-Line Preprocessing and Precompilation

Syntax

-undefine

Syntax

-u[name]

The symbol name to undefine.

Remarks

This option is case-sensitive.
Command-Line Preprocessing and Precompilation
Command-Line Library and Linking

-keepobjects

Retains or deletes object files after invoking the linker.

Syntax
-keepobjects
-nokeepobjects

Remarks
Use -keepobjects to retain object files after invoking the linker. Use -nokeepobjects to delete object files after linking. This option is global.

NOTE Object files are always kept when compiling.

-nolink

Compile the source files, without linking.

Syntax
-nolink

Remarks
This command is global.
Command-Line Library and Linking

-\texttt{O}

Specify the output filename or directory for storing object files or text output during compilation, or the output file if calling the linker.

\textbf{Syntax}

\begin{verbatim}
-o file | dir
file
The output file name.
dir
The directory to store object files or text output.
\end{verbatim}
Command-Line Object Code

- **c**

Instructs the compiler to compile but not invoke the linker to link the object code.

**Syntax**

```
-c
```

**Remarks**

This option is global.

- **codegen**

Instructs the compiler to compile without generating object code.

**Syntax**

```
-codegen
+nocodegen
```

**Remarks**

This option is global.

- **enum**

Specify the default size for enumeration types.

**Syntax**

```
-enum keyword
```

The arguments for `keyword` are:
Command-Line Object Code

```
int
   Uses int size for enumerated types.
min
   Uses minimum size for enumerated types. This is the default.
```

-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.

-ext

Tells the command-line tool the 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 (extension) 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 setting is no extension.
-strings

Controls how string literals are stored and used.

Remarks

-strings keyword[, ...]

The keyword arguments are:

[no]pool

All string constants are stored as a single data object so your program needs one data section for all of them.

[no]reuse

All equivalent string constants are stored as a single data object so your program can reuse them. This is the default.

[no]readonly

Make all string constants read-only. This is the default.
Command-Line Object Code
Command-Line for Optimization

-inline

Specify 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 -inline on, -inline auto.
Command-Line for Optimization

-O

Sets optimization settings to -opt level=2.

Syntax
-O

Remarks
Provided for backwards compatibility.

-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.
4
   Equivalent to -opt level=4,intrinsics.
p
   Equivalent to -opt speed.
s
   Equivalent to -opt space.
Command-Line for Optimization

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

-opt

Specify code optimization options to apply to object code.

Remarks
-opt keyword [, ...]
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=4,intrinsics,noframe
l[level]=num
    Sets a specific optimization level. The options for num are:
    • 0 – Global register allocation only for temporary values. Equivalent to
      #pragma optimization_level 0.
    • 1 – Adds dead code elimination, branch and arithmetic optimizations,
      expression simplification, and peephole optimization. Equivalent to #pragma
      optimization_level 1.
    • 2 – Adds common subexpression elimination, copy and expression propagation,
      stack frame compression, stack alignment, and fast floating-point to integer
      conversions. Equivalent to: #pragma optimization_level 2.
    • 3 – Adds dead store elimination, live range splitting, loop-invariant code
      motion, strength reduction, loop transformations, loop unrolling (with -opt
      speed only), loop vectorization, lifetime-based register allocation, and
      instruction scheduling. Equivalent to optimization_level 3.
    • 4 – Like level 3, but with more comprehensive optimizations from levels 1 and
      2. Equivalent to #pragma optimization_level 4.

For num options 0 through 4 inclusive, the default is 0.

[no]space
Command-Line for Optimization

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 of dead code. Equivalent to #pragma opt_dead_code.

[no] deadstore
Removes of dead assignments. Equivalent to #pragma opt_dead_assignments.

[no] lifetimes
Computation of variable lifetimes. Equivalent to #pragma opt_lifetimes.

[no] loop[inv]ariables]
Removes of 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.

[no] peep[hol]e
Peephole optimization. Equivalent to #pragma peephole.

[no] color[ing]
Register coloring. Equivalent to #pragma register_coloring.

[no] intrinsics
Inlines intrinsic functions.

[no] schedule
Command-Line for Optimization

Performs instruction scheduling.

display | dump

Displays complete list of active optimizations.
Command-Line for Optimization
Linker

The compiler organizes its object code into sections that the linker arranges when it creates its output file.

To generate an output file, the linker reads from input ELF (Executable and Linkable Format) files generated by compiler and other tools. The linker also reads a linker command file to determine how to build its output file. The linker then writes to its output file, an ELF file. This output file is the executable image, ready to load and run on the target platform.

This chapter describes the sections in the object code of and how to arrange them in the linker’s output file:

- Specifying Link Order in the IDE
- Defining Sections in Source Code
- Using a Linker Command File
- Linker Command File Syntax

Specifying Link Order in the IDE

To specify link order, use the Link Order page of the CodeWarrior IDE’s Project window. (For certain targets, the name of this page is Segments.)

Regardless of the order that the Link Order page specifies, the linker always processes source code files before it processes relocatable (.o) files or archive (.a) files. This policy means that the linker prefers using a symbol definition from a source file rather than a library file definition for the same symbol.

There is an exception, however: if the source file defines a weak symbol, the linker uses a global-symbol definition from a library. Use #pragma overload to create weak symbols.

Well-constructed projects usually do not have strong link-order dependencies.

The linker ignores executable files of the project. You may find it convenient to keep the executable files in the project folder so that you can disassemble it. If a build is successful, a check mark appears in the touch column on the left side of the project window. This indicates that the new file in the project is out of date. If a build is unsuccessful, the IDE is not able to find the executable file and it stops the build with an appropriate message.
Defining Sections in Source Code

The compiler defines its own sections to organize the data and executable code it generates. You may also define your own sections directly in your program’s source code.

The `section` pragma specifies to the compiler where to place proceeding definitions in source code. Use the `push`, `section`, and `pop` pragmas to enclose source code definitions. Listing 12.1 shows an example that places variables named `red` and `sky` in a section named `.myData`.

---

**Listing 12.1 Using pragma section to specify where to place definitions**

```
#pragma push /* Save the compiler’s state. */
#pragma section data_type "".myData" "".myData" data_mode=far_abs
int red;
int sky;
#pragma pop /* Restore the compiler’s state. */
```

An alternative to using the `section` pragma is to use `__declspec` to specify where to place a single definition in object code. Listing 12.2 shows an example.

---

**Listing 12.2 Using __declspec to specify where to place definitions**

```
__declspec (section "".myData"") int red;
__declspec (section "".myData"") int sky;
__declspec (section "".myISRSection"") ISRType
InterruptVectorTable[256];
```

Using a Linker Command File

A linker command file (.lcf file) is a text file that the linker reads to determine how to arrange object code from input files to produce an output file.

Use a linker command file to control dead-stripping, describe the target platform’s memory map, define and arrange sections, and control addresses and alignment:

- Dead-Stripping
- Defining the Target’s Memory Map
- Defining Sections in the Output File
- Associating Input Sections With Output Sections
- Controlling Alignment
- Specifying Memory Area Locations and Sizes
Dead-Stripping

Normally, the CodeWarrior linker ignores object code that is not referred to by other object code. If the linker detects that an object is not referred to by the rest of the program being linked, the linker will not place that object in its output file. In other words, the linker “dead-strips” objects that are not used.

Dead-stripping ensures the smallest possible output file. Also, dead-stripping relieves you from having to manually exclude unused source code from the compiler and unused object code from the linker.

There are some objects, however, that need to be in the linker’s output file even if these objects are not explicitly referred to by other parts of your program. For example, an executable image might contain an interrupt table that the target platform needs, but this interrupt table is not referred to by the rest of the image.

Use the FORCEACTIVE and FORCEFILES directives in a linker command file to specify to the linker which objects and files must not be dead-stripped.

Listing 12.3 shows an example from a linker command file that tells the linker not to dead-strip an object named InterruptVectorTable and all the objects in an input file named segfault.o.

Listing 12.3  FORCEACTIVE and FORCEFILES example

```
FORCEACTIVE { InterruptVectorTable }
FORCEFILES { segfault.o }
```

Defining the Target’s Memory Map

Use the linker command file’s MEMORY directive to delineate areas in the target platform’s memory map and associate a name for each of these areas. Names defined in a MEMORY directive may be used later in the linker command file to specify where object code should be stored. Listing 12.4 shows an example.

Listing 12.4  MEMORY directive example

```
MEMORY
{  
  ISR_table : org = 0x00000000, len = 0x400  
  data : org = 0x00000400, len = 0x10000  
  flash: org = 0x10000000, len = 0x10000  
  text : org = 0x80000000  
}
```
This example defines 3 memory areas named ISR_table, data, and text. The org argument specifies the beginning byte address of a memory area. The len argument is optional. It specifies how many bytes of data or executable code the linker may store in an area. The linker issues a warning message if an attempt to store object code in an area exceeds its length.

### Defining Sections in the Output File

Use the linker command file’s `SECTIONS` directive to

- define sections in the linker’s output file
- to specify in which memory area on the target platform a section in the output file should be loaded at runtime

Use `GROUP` directives in a `SECTIONS` directive to organize objects.

The linker will only create a section in the output file if the section is not empty, even if the section is defined in a `SECTION` or `GROUP` directive.

Listing 12.5 shows an example.

**Listing 12.5 SECTIONS and GROUP example**

```plaintext
SECTIONS
{
    GROUP :
    {
        .text : {}
        .rodata : {}
    } > text

    GROUP
    {
        .sdata : {}
        .sbss : {}
    } > data

    GROUP
    {
        .sdata2 : {}
        .sbss2 : {}
    } > data
}
```

This example defines the `.text` and `.rodata` sections in the output file and specifies that they should be loaded in the memory area named `text` on the target platform at runtime. The example then defines sections named `.sdata` and `.sbss`. These sections will be loaded in the memory named `data`. The last `GROUP` directive in the example...
defines sections named .sdata2, and .sbss2. These sections will also be loaded in the memory area named data, after the sections .sdata and .sbss.

**Associating Input Sections With Output Sections**

Normally the linker stores sections from input object code in the sections of the linker’s output file that have the same name. The linker command file’s `SECTIONS` and `GROUP` directives allow you to specify other ways to associate input object code with sections in linker output. Listing 12.6 shows an example.

**Listing 12.6  Associating object code with sections in linker output**

```
SECTIONS
{
    GROUP :
    {
        .myText : { main.o (.text) }
        .text : ( *(.text) )
    } > text
}
```

This example defines a section in the output file named .myText. This section will contain the objects that are in the .text section in the object code taken from the input file named main.o. The example also defines a section in the output file named .text. This section will contain all objects in the .text sections of all input files containing object code. Both these sections in the output file, .myText and .text, will be loaded in the memory area named text on the target platform.

The `SECTIONS` and `GROUP` directives also allow you to filter what kinds of object code from input files will be stored in a section in the output file. Table 12.1 shows the kinds of data that may be filtered.

**Table 12.1  Filter types for object code in input files**

<table>
<thead>
<tr>
<th>This filter</th>
<th>allows input objects that have these permissions</th>
<th>and contain this kind of object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>TEXT</td>
<td>readable, executable</td>
<td>initialized</td>
</tr>
<tr>
<td>CODE</td>
<td>readable, executable</td>
<td>initialized</td>
</tr>
<tr>
<td>DATA</td>
<td>readable, writable</td>
<td>initialized</td>
</tr>
<tr>
<td>BSS</td>
<td>readable, writable</td>
<td>uninitialized</td>
</tr>
</tbody>
</table>
Linker

Using a Linker Command File

Table 12.1 Filter types for object code in input files

<table>
<thead>
<tr>
<th>This filter</th>
<th>allows input objects that have these permissions</th>
<th>and contain this kind of object code</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONST</td>
<td>readable</td>
<td>initialized</td>
</tr>
<tr>
<td>MIXED</td>
<td>readable, writable, executable</td>
<td>initialized</td>
</tr>
</tbody>
</table>

Listing 12.7 shows an example.

Listing 12.7 Filtering objects from input files

```c
SECTIONS
{
  .text (TEXT) : { } > text
  .bss (BSS) : { } > data
}
```

This example defines a section in the output file named .text. The linker will only store objects from input object code that are readable, executable, and initialized. This example also defines a section in the output file named .bss. This section will only contain objects from the linker’s input files that are readable, writable, and uninitialized.

Controlling Alignment

Use the ALIGN argument in a SECTIONS or GROUP directive to specify a byte boundary on which to align a section in the output file.

Listing 12.8 shows an example.

Listing 12.8 Example of the ALIGN directive

```c
SECTIONS
{
  GROUP:
  {
    .init ALIGN(0x1000) : {}
    .text ALIGN(0x1000) : {}
  } > text
}
```

This example defines two sections named .init and .text. At runtime, each section will be loaded at the next available address that is evenly divisible by 0x1000 in the memory area named text on the target platform.
Specifying Memory Area Locations and Sizes

Normally, the linker stores sections in the output file in sequential order. Each object from the linker’s output is stored after the last object in the output file. Use the BIND, ADDR, and SIZEOF keywords in SECTIONS and GROUP directives to precisely specify where sections in the output file will be loaded.

Listing 12.9 shows an example.

**Listing 12.9  BIND, ADDR, and SIZEOF example**

```plaintext
SECTIONS
{
   .text BIND(0x00010000) : ()
   .rodata : {}
   .data BIND(ADDR(.rodata + SIZEOF(.rodata)) ALIGN(0x010) : {})
}
```

This example defines a section in the output file named `.text`. This section will be loaded at address 0x00010000 on the target platform at runtime. The next section, `.rodata`, will be loaded at the address immediately proceeding the last byte in the `.text` section. The last section, `.data`, will be loaded at the address that is the sum of the beginning of the `.rodata` section’s address and the size of the `.rodata` section. This last section will be aligned at the next address that is evenly divisible by 0x10.

The dot keyword (".") is a convenient way to set the linker’s place in the current output section.

Listing 12.10 shows an example.

**Listing 12.10  Skipping areas of memory**

```plaintext
SECTIONS
{
   GROUP :
   {
      .ISR_Table : {}.
      . = 0x2000
   } > flash

   GROUP :
   {
      .paramsection : {}
   } > flash
}
```
Linker

**Linker Command File Syntax**

This example defines two sections. The first section, `.ISRTable`, will be loaded at
beginning of the memory area named `flash` on the target platform at runtime. The
second section, `.paramsection`, will be loaded at the address that is 0x2000 bytes
past the beginning of the memory area named `flash`.

**Linker Command File Syntax**

*Listing 12.11* shows the syntax for linker command files.

**Listing 12.11**  Linker Command File Syntax

```plaintext
<linker command file> =
  <commands>* <memory>? <sections>? <commands>*

<commands> =
  <exclude files> | <force active> | <force files> | <include dwarf> |
  <shorten names for tornado 101> | <cats bss mod> | <cats header mod> |
  <data type converts> | <entry> | <init> | <term> |
  <external symbol> | <internal symbol> | <memory gaps>

<exclude files> =
  "EXCLUDEFILES" "(" <file name> ")"

<force active> =
  "FORCEACTIVE" "(" <identifier> + ")"

<letter> =
  'a'|'b'|'c'|'d'|'e'|'f'|'g'|'h'|'i'|'j'|'k'|'l'|'m' |
  'n'|'o'|'p'|'q'|'r'|'s'|'t'|'u'|'v'|'w'|'x'|'y'|'z' |
  'A'|'B'|'C'|'D'|'E'|'F'|'G'|'H'|'I'|'J'|'K'|'L'|'M' |
  'N'|'O'|'P'|'Q'|'R'|'S'|'T'|'U'|'V'|'W'|'X'|'Y'|'Z'

<file name> =
  (<letter> |"_*") (<letter> |<digit> |"_*")*
  ("_*")?(<letter> |<digit> |"_*")*
```
Linker Command File Syntax

- `<object file> = (\<letter\> |\"\_\") (\<letter\> |\<digit\> |\"\_\")* (\"\.\") (\"o\" |\"O\")
- `<archive file> = (\<letter\> |\"\_\") (\<letter\> |\<digit\> |\"\_\")* (\"\.\") (\"a\" |\"A\")
- `<force files> = "FORCEFILES" \"(" (\<object file\> | \<archive file\>) \"(\" object file \")\")+ \")"*
- `<include dwarf> = "INCLUDEDWARF" \"\{" <file name> \"\}"
- `<shorten names for tornado 101> = "SHORTEN_NAMES_FOR_TOR_101"
- `<cats bss mod> = "CATS_BSS_MOD"
- `<cats header mod> = "CATS_HEADER_MOD"
- `<data type converts> = "DATA_TYPE CONVERTS"
- `<entry> = "ENTRY" \"\(" <identifier> \")\""
- `<init> = "INIT" \"\(" <identifier> \")\"
- `<term> = "TERM" \"\(" <identifier> \")\"
- `<external symbol> = "EXTERNAL_SYMBOL" \"\{" <identifier> \"\}"
- `<internal symbol> = "INTERNAL_SYMBOL" \"\{" <identifier> \"\}"
- `<group> = "GROUP" <address modifiers> ":\" 
  \"\(" (\<section spec\>) \"\)* \"\{" \"<fill shortnumber> \}
  \"<\" <mem area symbolic name> \"
- `<hexadigit> = \"0\" | \"1\" | \"2\" | \"3\" | \"4\" | \"5\" | \"6\" | \"7\" | \"8\" | \"9\" 
  \"A\" | \"B\" | \"C\" | \"D\" | \"E\" | \"a\" | \"b\" | \"c\" | \"d\" | \"e\""
Linker

Linker Command File Syntax

<digit> =
   '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'

<hexadecimal number> =
   "0"("x"|"X")(<hexadigit> )+

<decimal number> =
   (<digit> )+

<number> =
   <hexadecimal number> | <decimal number>

<binary op> =
   '+'|'-'|'*'|'/'|'&' |'%' |'=' |'==' |'!=' |'>' |'>'= |'<' |'<'= |'<'| '<=' |'
   '&&' | '||' |'>>' |'<<' |'&' |'|' |'' |''

<unary op> =
   '++' |'--' |'~' |'!'  

<postfix unary op> =
   '++' |'--'

<symbol declaration> =
   { <identifier> "=" <address spec> } |
   { "PROVIDE" "("<identifier> "=" <address spec> ")" }

<identifier> =
   (<letter> |"_") (<letter> |"_"|<digit> )*

<operand> =
   <number> |
   (*ADDR" "(<output section spec> | <address expr> ")* ) |
   (*ROMADDR" "(<output section spec> | <address expr> ")* ) |
   (*SIZEOF" "(<output section spec> | <address expr> ")")

<address spec> =
   <number> |
   "=" |
   <operand> |
   (<address spec> <binary op> <operand> ) |
   (<unary op> <address spec> ) |
   (<address spec> <postfix unary op> )

<memory spec> =
   <mem area symbolic name> ":=" "origin" |
   "org" |
   "o" ":=" <number> "," "length"
Linker

Linker Command File Syntax

"len" | "l" "+" <number>

<memory gaps>= "." "=" <address spec>

<memory>= "MEMORY" ":" "{" <memory spec> "+"}"

.sections>= "SECTIONS" "{" <section spec> | <memory gaps> | <symbol declaration> | <group> }" "}"

@section spec = <output section name>
["(" <input type> ")"]
[<address modifiers> ] "+" "("
[['<input section spec> ]"] "+"
[= fill shortnumber] [ > mem area symbolic name ]

<output section name> = <section name>

<input type> = [ "TEXT" | "DATA" | "BSS" | "CONST" | "MIXED" | "ZTEXT" | "ZCODE" ]

<address modifiers> = ["BIND" "("<address spec> ")" ]
["ALIGN" "("<address spec> ")" | "NEXT" "("<address spec> ")"]
["LOAD" | "INTERNAL_LOAD" ] "("<address spec> ")"

<input section spec> = <file name> | <file name> "{<section name> "} | "+"{"<section name> "} | <symbol declaration> | <data write>

<data write> = 
("LONG" | "SHORT" | "BYTE" ) "(" <number> ")"

<fill shortnumber> = <number>


Linker
Commands, Directives, and Keywords

Commands, Directives, and Keywords

The rest of this chapter consists of explanations of all valid LCF functions, keywords, directives, and commands, in alphabetic order.

Table 12.2 LCF Functions, Keywords, Directives, and Commands

<table>
<thead>
<tr>
<th>. (location counter)</th>
<th>ADDR</th>
<th>ALIGN</th>
</tr>
</thead>
<tbody>
<tr>
<td>ALIGNALL</td>
<td>EXCEPTION</td>
<td>EXPORTSTRTAB</td>
</tr>
<tr>
<td>EXPORTSYMTAB</td>
<td>FORCE_ACTIVE</td>
<td>IMPORTSTRTAB</td>
</tr>
<tr>
<td>IMPORTSYMTAB</td>
<td>INCLUDE</td>
<td>KEEP_SECTION</td>
</tr>
<tr>
<td>MEMORY</td>
<td>OBJECT</td>
<td>REF_INCLUDE</td>
</tr>
<tr>
<td>SECTIONS</td>
<td>SIZEOF</td>
<td>SIZEOF_ROM</td>
</tr>
<tr>
<td>WRITEB</td>
<td>WRITEH</td>
<td>WRITEW</td>
</tr>
<tr>
<td>WRITES0COMMENT</td>
<td>ZERO_FILL_UNINITIALIZED</td>
<td></td>
</tr>
</tbody>
</table>

. (location counter)

Denotes the current output location.

Remarks

The period always refers to a location in a sections segment, so is valid only in a sections-section definition. Within such a definition, `. ’ may appear anywhere a symbol is valid.

Assigning a new, greater value to `. ’ causes the location counter to advance. But it is not possible to decrease the location-counter value, so it is not possible to assign a new, lesser value to `. ’ You can use this effect to create empty space in an output section, as the Listing 12.12 example does.

Example

The code of Listing 12.12 moves the location counter to a position 0x10000 bytes past the symbol __start.

Listing 12.12 Moving the Location Counter

```plaintext
..data :
```

102 CodeWarrior Build Tools Reference ColdFire™ Architectures Edition
Linker

Commands, Directives, and Keywords

{ *(data)
 *(bss)
 *(COMMON)
 __start = .;
 . = __start + 0x10000;
 __end = .;
} > DATA

ADDR

Returns the address of the named section or memory segment.

ADDR (sectionName | segmentName)

Parameters

sectionName
Identifier for a file section.

segmentName
Identifier for a memory segment

Example

The code of Listing 12.13 uses the ADDR function to assign the address of ROOT to the symbol __rootbasecode.

Listing 12.13  ADDR() Function

MEMORY{
  ROOT (RWX) : ORIGIN = 0x80000400, LENGTH = 0
}

SECTIONS{
  .code :
  {
    __rootbasecode = ADDR(ROOT);
    *.text;
  }
  > ROOT
}
Linker
Commands, Directives, and Keywords

ALIGN

Returns the location-counter value, aligned on a specified boundary.
ALIGN(alignValue)

Parameter
alignValue
Alignment-boundary specifier; must be a power of two.

Remarks
The ALIGN function does not update the location counter; it only performs arithmetic. Updating the location counter requires an assignment such as:

. = ALIGN(0x10); #update location counter to 16-byte alignment

ALIGNALL

Forces minimum alignment for all objects in the current segment to the specified value.
ALIGNALL(alignValue);

Parameter
alignValue
Alignment-value specifier; must be a power of two.

Remarks
ALIGNALL is the command version of the ALIGN function. It updates the location counter as each object is written to the output.

Example
Listing 12.14 is an example use for ALIGNALL() command.

Listing 12.14 ALIGNALL Example

.code :
{  
   ALIGNALL(16); // Align code on 16-byte boundary
  * (.init)
* (.text)

ALIGNALL(64); //align data on 64-byte boundary
* (.rodata)
* > .text

---

**EXCEPTION**

Creates the exception table index in the output file.

**Remarks**

Only C++ code requires exception tables. To create an exception table, add the EXCEPTION command, with symbols __exception_table_start__ and __exception_table_end__, to the end of your code section segment, just as **Listing 12.15** shows. (At runtime, the system knows the values of the two symbols.)

**Example**

**Listing 12.15** shows the code for creating an exception table.

---

**Listing 12.15 Creating an Exception Table**

__exception_table_start__ = .;
EXCEPTION
__exception_table_end__ = .;

---

**EXPORTSTRTAB**

Creates a string table from the names of exported symbols.

**Remarks**

**Table 12.3** shows the structure of the export string table. As with an ELF string table, the system zero-terminates the library and symbol names.

---
Linker
Commands, Directives, and Keywords

Table 12.3 Export String Table Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>1 byte</td>
</tr>
<tr>
<td>library name</td>
<td>varies</td>
</tr>
<tr>
<td>symbol1 name</td>
<td>varies</td>
</tr>
<tr>
<td>symbol2 name</td>
<td>varies</td>
</tr>
</tbody>
</table>

Example
Listing 12.16 shows the code for creating an export string table.

Listing 12.16 Creating an Export String Table

.expstr:
{ EXPORTSTRTAB
} > EXPSTR

EXPORTSYMTAB

Creates a jump table of the exported symbols.

EXPORTSYMTAB

Remarks
Table 12.4 shows the structure of the export symbol table. The start of the export symbol table must be aligned on at least a four-byte boundary.

Table 12.4 Export Symbol Table Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Size (in bytes)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (in bytes) of export table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to library name in export string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to symbol1 name in export string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Address of symbol1</td>
<td>4 bytes</td>
</tr>
<tr>
<td>A5 value for symbol1</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>
Example
Listing 12.17 shows the code for creating an export symbol table.

Listing 12.17 Creating an Export Symbol Table

```
.expsym:
  ( EXPORTSYMTAB
) > EXPSYM
```

**FORCE_ACTIVE**

Starts an optional LCF closure segment that specifies symbols the linker should *not* deadstrip.

```
FORCE_ACTIVE{ symbol[, symbol] }
```

**Parameter**

- `symbol`
  - Any defined symbol.

**IMPORTSTRTAB**

Creates a string table from the names of imported symbols.

```
IMPORTSTRTAB
```

**Remarks**

Table 12.5 shows the structure of the import string table. As with an ELF string table, the system zero-terminates the library and symbol names.
Linker
Commands, Directives, and Keywords

Table 12.5 Import String Table Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x00</td>
<td>1 byte</td>
</tr>
<tr>
<td>library name</td>
<td>varies</td>
</tr>
<tr>
<td>symbol1 name</td>
<td>varies</td>
</tr>
<tr>
<td>symbol2 name</td>
<td>varies</td>
</tr>
</tbody>
</table>

Example
Listing 12.18 shows the code for creating an import string table.

Listing 12.18 Creating an Import String Table
重要作用
.example
.import
{   IMPORTSTRTAB
} > IMPSTR

IMPORTSYMTAB

Creates a jump table of the imported symbols.

IMPORTSYMTAB

Remarks

Table 12.6 shows the structure of the import symbol table. The start of the import symbol table must be aligned on at least a four-byte boundary.

Table 12.6 Import Symbol Table Structure

<table>
<thead>
<tr>
<th>Field</th>
<th>Size (in bytes) of import table</th>
</tr>
</thead>
<tbody>
<tr>
<td>Size (in bytes) of import table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to library1 name in import string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Number of entries in library1</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to symbol1 name in import string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Address of symbol1 vector in export string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to symbol2 name in import string table</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>
Table 12.6 Import Symbol Table Structure (continued)

<table>
<thead>
<tr>
<th>Field</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Address of symbol2 vector in export string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Index to library2 name in import string table</td>
<td>4 bytes</td>
</tr>
<tr>
<td>Number of entries in library2</td>
<td>4 bytes</td>
</tr>
</tbody>
</table>

Example

Listing 12.19 shows the code for creating an import symbol table.

Listing 12.19 Creating an Import Symbol Table

```c
.include

INCLUDE filename

Parameter

filename

Name of a binary file in the project. The File Mappings target settings panel must specify resource file for all files that have the same extension as this file.

KEEP_SECTION

Starts an optional LCF closure segment that specifies sections the linker should not deadstrip.

KEEP_SECTION( sectionType[, sectionType] )

Parameter

sectionType

Identifier for any user-defined or predefined section.
MEMORY

Starts the LCF memory segment, which defines segments of target memory.

MEMORY { memory_spec[, memory_spec] }

Parameters

memory_spec

segmentName (accessFlags) : ORIGIN = address,
LENGTH = length [> fileName]

segmentName
Name for a new segment of target memory. Consists of alphanumeric characters; can include the underscore character.

accessFlags
ELF-access permission flags — R = read, W = write, or X = execute.

address
A memory address, such as 0x80000400, or an AFTER command. The format of the AFTER command is AFTER (name[, name]); this command specifies placement of the new memory segment at the end of the named segments.

length
Size of the new memory segment: a value greater than zero. Optionally, the value zero for autolength, in which the linker allocates space for all the data and code of the segment. (Autolength cannot increase the amount of target memory, so the feature can lead to overflow.)

fileName
Optional, binary-file destination. The linker writes the segment to this binary file on disk, instead of to an ELF program header. The linker puts this binary file in the same folder as the ELF output file. This option has two variants:

• > fileName: writes the segment to a new binary file.
• >> fileName: appends the segment to an existing binary file.

Remarks

The LCF contains only one MEMORY directive, but this directive can define as many memory segments as you wish.

For each memory segment, the ORIGIN keyword introduces the starting address, and the LENGTH keyword introduces the length value.
There is no overflow checking for the autolength feature. To prevent overflow, you should use the AFTER keyword to specify the segment’s starting address.

If an AFTER keyword has multiple parameter values, the linker uses the highest memory address.

Example

Listing 12.20 is an example use of the MEMORY directive.

Listing 12.20 MEMORY Directive Example

MEMORY {
    TEXT (RX) : ORIGIN = 0x00003000, LENGTH = 0
    DATA (RW) : ORIGIN = AFTER(TEXT), LENGTH = 0
}

OBJECT

Sections-segment keyword that specifies a function. Multiple OBJECT keywords control the order of functions in the output file.

OBJECT (function, sourcefile.c)

Parameters

function
    Name of a function.

sourcefile.c
    Name of the C file that contains the function.

Remarks

If an OBJECT keyword tells the linker to write an object to the output file, the linker does not write the same object again, in response to either the GROUP keyword or the "*" wildcard character.
**Linker**

*Commands, Directives, and Keywords*

---

**REF_INCLUDE**

Starts an optional LCF closure segment that specifies sections the linker should **not**
deadstrip, if program code references the files that contain these sections.

```
REF_INCLUDE( sectionType[, sectionType] )
```

**Parameter**

`sectionType`

Identifier for any user-defined or predefined section.

**Remarks**

Useful if you want to include version information from your source file
components.

---

**SECTIONS**

Starts the LCF sections segment, which defines the contents of target-memory sections.
Also defines global symbols to be used in the output file.

```
SECTIONS ( section_spec[, section_spec] )
```

**Parameters**

`section_spec`

```
sectionName : [AT (loadAddress)] {contents}
> segmentName
```

`sectionName`

Name for the output section, such as `mysection`. Must start with a period.

`AT (loadAddress)`

Optional specifier for the load address of the section. The default value is the
relocation address.

`contents`

Statements that assign a value to a symbol or specify section placement, including
input sections.

`segmentName`

Predefined memory-segment destination for the contents of the section. The two
variants are:
**Linker**

*Commands, Directives, and Keywords*

- > segmentName: puts section contents at the beginning of memory segment segmentName.
- >> segmentName: appends section contents to the end of memory segment segmentName.

**Example**

Listing 12.21 is an example sections-segment definition.

**Listing 12.21 SECTIONS Directive Example**

```c
SECTIONS {
  .text : {
    _textSegmentStart = .;
    alpha.c (.text)
    . = ALIGN (0x10);
    beta.c (.text)
    _textSegmentEnd = .;
  }
  .data : { *(.data) }
  .bss : { *(.bss)
    *(COMMON)
  }
}
```

**SIZEOF**

Returns the size (in bytes) of the specified segment or section.

`SIZEOF(segmentName | sectionName)`

**Parameters**

- `segmentName`
  - Name of a segment; must start with a period.
- `sectionName`
  - Name of a section; must start with a period.
Linker
Commands, Directives, and Keywords

SIZEOF_ROM
Returns the size (in bytes) that a segment occupies in ROM.

\texttt{SIZEOF\_ROM}\ (\texttt{segmentName})

\textbf{Parameter}
\texttt{segmentName}
Name of a ROM segment; must start with a period.

\textbf{Remarks}
Always returns the value 0 until the ROM is built. Accordingly, you should use \texttt{SIZEOF\_ROM} only within an expression inside a \texttt{WRITEB}, \texttt{WRITEH}, \texttt{WRITEW}, or \texttt{AT} function.

Furthermore, you need \texttt{SIZEOF\_ROM} only if you use the \texttt{COMPRESS} option on the memory segment. Without compression, there is no difference between the return values of \texttt{SIZEOF\_ROM} and \texttt{SIZEOF}.

WRITEB
Inserts a byte of data at the current address of a section.

\texttt{WRITEB}\ (\texttt{expression});

\textbf{Parameter}
\texttt{expression}
Any expression that returns a value 0x00 to 0xFF.

WRITEH
Inserts a halfword of data at the current address of a section.

\texttt{WRITEH}\ (\texttt{expression});

\textbf{Parameter}
\texttt{expression}
Any expression that returns a value 0x0000 to 0xFFFF
WRITEW

Inserts a word of data at the current address of a section.

WRITEW (expression);

Parameter

expression

Any expression that returns a value 0x00000000 to 0xFFFFFFFF.

WRITES0COMMENT

Inserts an S0 comment record into an S-record file.

WRITES0COMMENT "comment"

Parameter

comment

Comment text: a string of alphanumerical characters 0-9, A-Z, and a-z, plus space, underscore, and dash characters. Double quotes must enclose the comment string. (If you omit the closing double-quote character, the linker tries to put the entire LCF into the S0 comment.)

Remarks

This command, valid only in an LCF sections segment, creates an S0 record of the form:

S0aa0000bbbbbbbbbbbd
dd

- aa — hexadecimal number of bytes that follow
- bb — ASCII equivalent of comment
- dd — the checksum

This command does not null-terminate the ASCII string.

Within a comment string, do not use these character sequences, which are reserved for LCF comments:

# /* */ //


Linker
Commands, Directives, and Keywords

Example
This example shows that multi-line $0$ comments are valid:

\begin{verbatim}
WRITESOCOMMENT "Line 1 comment
Line 2 comment"
\end{verbatim}

ZERO_FILL_UNINITIALIZED
Forces the linker to put zeroed data into the binary file for uninitialized variables.

Remarks
This directive must be between directives MEMORY and SECTIONS; placing it anywhere else would be a syntax error.

Using linker configuration files and the define_section pragma, you can mix uninitialized and initialized data. As the linker does not normally write uninitialized data to the binary file, forcing explicit zeroing of uninitialized data can help with proper placement.

Example
The code of Listing 12.22 tells the linker to write uninitialized data to the binary files as zeros.

Listing 12.22 ZERO_FILL_UNINITIALIZED Example

\begin{verbatim}
MEMORY {
  TEXT (RX) :ORIGIN = 0x00030000, LENGTH = 0
  DATA (RW) :ORIGIN = AFTER(TEXT), LENGTH = 0
}

ZERO_FILL_UNINITIALIZED

SECTIONS {
  .main_application:
  {
    *(.text)
    .=ALIGN(0x8);
    *(.rodata)
    .=ALIGN(0x8);
  } > TEXT
  ...
\end{verbatim}
Linker

Commands, Directives, and Keywords

)}
Linker

Commands, Directives, and Keywords
ColdFire Linker

This chapter describes how to use the features in the CodeWarrior linker that are specific to ColdFire software development.

You access these functions through commands in the linker command file (LCF). The LCF syntax and structure are similar to those of a programming language; the syntax includes keywords, directives, and expressions.

This chapter consists of these sections:

- Deadstripping
- Executable files in Projects
- S-Record Comments
- Deadstripping
- LCF Syntax

Deadstripping

As the linker combines object files into one executable file, it recognizes portions of executable code that execution cannot possibly reach. Deadstripping is removing such unreachable object code — that is, not including these portions in the executable file. The CodeWarrior linker performs this deadstripping on a per-function basis.

The CodeWarrior linker deadstrips unused code and data from only object files that a CodeWarrior compiler generates. The linker never deadstrips assembler-relocatable files, or object files from a different compiler.

Deadstripping is particularly useful for C++ programs or for linking to large, general-purpose libraries. Libraries (archives) built with the CodeWarrior compiler only contribute the used objects to the linked program. If a library has assembly or other compiler built files, only those files that have at least one referenced object contribute to the linked program. The linker always ignores unreferenced object files.

Well-constructed projects probably do not contain unused data or code. Accordingly, you can reduce the time linking takes by disabling deadstripping:

- To disable deadstripping completely, check the **Disable Deadstripping** checkbox of the **ColdFire Linker** panel.
- To disable deadstripping for particular symbols, enter the symbol names in the **Force Active Symbols** text box of the **ColdFire Linker** Panel.
ColdFire Linker

Executable files in Projects

- To disable deadstripping for individual sections of the linker command file, use the KEEP_SECTION() directive. As code does not directly reference interrupt-vector tables, a common use for this directive is disabling deadstripping for these interrupt-vector tables. The subsection Closure Segments provides additional information about the KEEP_SECTION() directive.

NOTE To deadstrip files from standalone assembler, you must make each assembly functions start its own section (for example, a new .text directive before functions) and using an appropriate directive.

Executable files in Projects

It may be convenient to keep executable files in a project, so that you can disassemble them later. As the linker ignores executable files, the IDE portrays them as out of date — even after a successful build. The IDE out-of-date indicator is a check mark in the touch column, at the left side of the project window.

Dragging/dropping the final elf and disassembling it is a useful way to view the absolute code.

S-Record Comments

You can insert one comment at the beginning of an S-Record file via the linker-command-file directive WRITES0COMMENT.

LCF Structure

Linker command files consist of three kinds of segments, which must be in this order:
- A memory segment, which begins with the MEMORY{} directive
- Optional closure segments, which begin with the FORCE_ACTIVE{}, KEEP_SECTION{}, or REP_INCLUDE{} directives
- A sections segment, which begins with the SECTIONS{} directive

Memory Segment

Use the memory segment to divide available memory into segments. Listing 13.1 shows the pattern.
Listing 13.1 Example Memory Segment

MEMORY {
    segment_1 (RWX): ORIGIN = 0x80001000, LENGTH = 0x19000
    segment_2 (RWX): ORIGIN = AFTER(segment_1), LENGTH = 0
    segment_x (RWX): ORIGIN = memory address, LENGTH = segment size 
    and so on...
}

In this pattern:
- The (RWX) portion consists of ELF-access permission flags: R = read, W = write, or X = execute.
- ORIGIN specifies the start address of the memory segment — either an actual memory address or, via the AFTER keyword, the name of the preceding segment.
- LENGTH specifies the size of the memory segment. The value 0 means unlimited length.

The segment_2 line of Listing 13.1 shows how to use the AFTER and LENGTH commands to specify a memory segment, even though you do not know the starting address or exact length.

Closure Segments

An important feature of the linker is deadstripping unused code and data. At times, however, an output file should keep symbols even if there are no direct references to the symbols. Linking for interrupt handlers, for example, usually is at special addresses, without any explicit, control-transfer jumps.

Closure segments let you make symbols immune from deadstripping. This closure is transitive, so that closing a symbol also forces closure on all other referenced symbols.

For example, suppose that:
- Symbol _abc references symbols _def and _ghi,
- Symbol _def references symbols _jkl and _mno, and
- Symbol _ghi references symbol _pqr

Specifying symbol _abc in a closure segment would force closure on all six of these symbols.

The three closure-segment directives have specific uses:
- FORCE_ACTIVE — Use this directive to make the linker include a symbol that it otherwise would not include.
- KEEP_SECTION — Use this directive to keep a section in the link, particularly a user-defined section.
ColdFire Linker

LCF Structure

- REF_INCLUDE — Use this directive to keep a section in the link, provided that there is a reference to the file that contains the section. This is a useful way to include version numbers.

Listing 13.2 shows an example of each directive.

Listing 13.2  Example Closure Sections

# 1st closure segment keeps 3 symbols in link
FORCE_ACTIVE {break_handler, interrupt_handler, my_function}

# 2nd closure segment keeps 2 sections in link
KEEP_SECTION {.interrupt1, .interrupt2}

# 3rd closure segment keeps file-dependent section in link
REF_INCLUDE {.version}

Sections Segment

Use the sections segment to define the contents of memory sections, and to define any global symbols that you want to use in your output file. Listing 13.3 shows the format of a sections segment.

Listing 13.3  Example Sections Segment

SECTIONS {
  .section_name : #The section name, for your reference,
  { # must begin with a period.
    filename.c (.text) #Put .text section from filename.c,
    filename2.c (.text) #then put .text section from filename2.c,
    filename.c (.data) #then put .data section from filename.c,
    filename2.c (.data) #then put .data section from filename2.c,
    filename.c (.bss) #then put .bss section from filename.c,
    filename2.c (.bss) #then put .bss section from filename2.c.
    . = ALIGN (0x10); #Align next section on 16-byte boundary.
  } > segment_1 #Map these contents to segment_1.
  .next_section_name:
  { #more content descriptions
    } > segment_x #End of .next_section_name definition
  }
} #End of sections segment
LCF Syntax

This section explains LCF commands, including practical ways to use them. Subsections are:

- Variables, Expressions, and Integrals
- Arithmetic, Comment Operators
- Alignment
- Specifying Files and Functions
- Stack and Heap
- Static Initializers
- Exception Tables
- Position-Independent Code and Data
- ROM-RAM Copying
- Writing Data Directly to Memory

Variables, Expressions, and Integrals

In a linker command file, all symbol names must start with the underscore character (_). The other characters can be letters, digits, or underscores. These valid lines for an LCF assign values to two symbols:

```
_decay_num = 99999999;
_hex_num_ = 0x9011276;
```

Use the standard assignment operator to create global symbols and assign their addresses, according to the pattern:

```
_symbolicname = some_expression;
```

**NOTE**

There must be a semicolon at the end of a symbol assignment statement. A symbol assignment is valid only at the start of an expression, so a line such as this is not valid:

```
_sym1 + _sym2 = _sym3;
```

When the system evaluates an expression and assigns it to a variable, the expression receives the type value `absolute` or a `relocatable`:

- Absolute expression — the symbol contains the value that it will have in the output file.
ColdFire Linker

LCF Syntax

- Relocatable expression — the value expression is a fixed offset from the base of a section.

LCF syntax for expressions is very similar to the syntax of the C programming language:
- All integer types are long or unsigned long.
- Octal integers begin with a leading zero; other digits are 0 through 7, as these symbol assignments show:
  _octal_number = 01374522;
  _octal_number2 = 032405;
- Decimal integers begin with any non-zero digit; other digits are 0 through 9, as these symbol assignments show:
  _dec_num = 99999999;
  _decimal_number = 123245;
  _decalfour = 9011276;
- Hexadecimal integers begin with a zero and the letter x; other digits are 0 through f, as these symbol assignments show:
  _hex_number = 0x999999FF;
  _firstfactorspace = 0X123245EE;
  _fifthhexval = 0xFFEE;
- Negative integers begin with a minus sign:
  _decimal_number = -123456;

Arithmetic, Comment Operators

Use standard C arithmetic and logical operations as you define and use symbols in the LCF. All operators are left-associative. Table 13.1 lists these operators in the order of precedence. For additional information about these operators, refer to the C Compiler Reference.

Table 13.1 LCF Arithmetic Operators

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>- ~ !</td>
</tr>
<tr>
<td>2</td>
<td>* / %</td>
</tr>
<tr>
<td>3</td>
<td>+ -</td>
</tr>
<tr>
<td>4</td>
<td>&gt;&gt; &lt;&lt;</td>
</tr>
</tbody>
</table>
To add comments to your file, use the pound character, C-style slash and asterisk characters, or C++-style double-slash characters, in any of these formats:

```
#  This is a one-line comment
/* This is a
   multiline comment */
* (.text) // This is a partial-line comment
```

### Alignment

To align data on a specific byte boundary, use the ALIGN keyword or the ALIGNALL command. **Listing 13.4** and **Listing 13.5** are examples for bumping the location counter to the next 16-byte boundary.

**Listing 13.4  ALIGN Keyword Example**

```c
file.c (.text)
 . = ALIGN (0x10);
file.c (.data)  # aligned on 16-byte boundary.
```

**Listing 13.5  ALIGNALL Command Example**

```c
file.c (.text)
ALIGNALL (0x10);  #everything past this point aligned
    # on 16 byte boundary
file.c (.data)
```

**NOTE**  If one segment entry imposes an alignment requirement, that segment’s starting address must conform to that requirement. Otherwise, there could be

---

Table 13.1  LCF Arithmetic Operators (continued)

<table>
<thead>
<tr>
<th>Precedence</th>
<th>Operators</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>== != &gt; &lt; &lt;= &gt;=</td>
</tr>
<tr>
<td>6</td>
<td>&amp;</td>
</tr>
<tr>
<td>7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>&amp;&amp;</td>
</tr>
<tr>
<td>9</td>
<td></td>
</tr>
</tbody>
</table>

---

ColdFire Linker

LCF Syntax

Empty
conflicting section alignment in the code the linker produces. In general, the instructions for data alignment should be last before the end of the section.

## Specifying Files and Functions

Defining the contents of a sections segment includes specifying the source file of each section. The standard method is merely listing the files, as **Listing 13.6** shows.

### Listing 13.6 Standard Source-File Specification

```
SECTIONS {
  .example_section :
  {
    main.c (.text)
    file2.c (.text)
    file3.c (.text)
    # and so forth
  }
```

For a large project, however, such a list can be very long. To shorten it, you can use the asterisk (`*`) wild-card character, which represents the filenames of every file in your project. The line

```
* (.text)
```

in a section definition tells the system to include the `.text` section from each file. Furthermore the `*` wildcard does not duplicate sections already specified; you need not replace existing lines of the code. In **Listing 13.6**, replacing the `# and so forth` comment line with

```
* (.text)
```

would add the `.text` sections from all other project files, without duplicating the `.text` sections from files `main.c`, `file2.c`, or `file3.c`.

Another possibility as you define a sections segment, is specifying sections from a named group of files. To do so, use the `GROUP` keyword:

```
GROUP(fileGroup1) (.text)
GROUP(fileGroup4) (.data)
```

These two lines would specify including the `.text` sections from all `fileGroup1` files, and the `.data` sections from all `fileGroup4` files.

For precise control over function placement within a section, use the `OBJECT` keyword. For example, to place functions `beta` and `alpha` before anything else in a section, your definition could be like **Listing 13.7**.
Listing 13.7 Function Placement Example

SECTIONS {
  .program_section :
  {
    OBJECT (beta, main.c)  # Function beta is 1st section item
    OBJECT (alpha, main.c) # Function alpha is 2nd section_item
    * (.text)  # Remaining_items are .text sections from all files
  } > ROOT
}

**NOTE** For C++, you must specify functions by their mangled names.

If you use the **OBJECT** keyword to specify a function, subsequently using * wild-card character does **not** specify that function a second time.

**Stack and Heap**

Reserving space for the stack requires some arithmetic operations to set the symbol values used at runtime. **Listing 13.8** is a sections-segment definition code fragment that shows this arithmetic.

**Listing 13.8 Stack Setup Operations**

```plaintext
_stack_address = __END_BSS;
_stack_address = _stack_address & ~7; /*align top of stack by 8*/
__SP_INIT = _stack_address + 0x4000; /*set stack to 16KB*/
```

The heap requires a similar space reservation, which **Listing 13.9** shows. Note that the bottom address of the stack is the top address of the heap.

**Listing 13.9 Heap Setup Operations**

```plaintext
__heap_addr = __SP_INIT; /* heap grows opposite stack */
__heap_size = 0x50000; /* heap size set to 500KB */
```

**Static Initializers**

You must invoke static initializers to initialize static data before the start of `main()`. To do so, use the **STATICINIT** keyword to have the linker generate the static initializer sections.

In your linker command file, use lines similar to these to tell the linker where to put the table of static initializers (relative to the `.` location counter):
ColdFire Linker

LCF Syntax

___sinit__ = .;
STATICINIT

The program knows the symbol ___sinit__ at runtime. So in startup code, you can use corresponding lines such as these:

```c
#ifdef __cplusplus
/* call the c++ static initializers */
__call_static_initializers();
#endif
```

Exception Tables

You need exception tables only for C++ code. To create one, add the EXCEPTION command to the end of your code section — Listing 13.10 is an example.

The program knows the two symbols __exception_table_start__ and __exception_table_end__ at runtime.

Listing 13.10 Creating an Exception Table

```c
__exception_table_start__ = .;
EXCEPTION
__exception_table_end__ = .;
```

Position-Independent Code and Data

For position-independent code (PIC) and position-independent data (PID), your LCF must include .picdynrel and .piddynrel sections. These sections specify where to store the PIC and PID dynamic relocation tables.

In addition, your LCF must define these six symbols:

```c
__START_PICTABLE __END_PICTABLE __PICTABLE_SIZE
__START_PIDTABLE __END_PIDTABLE __PIDTABLE_SIZE
```

Listing 13.11 is an example definition for PIC and PID.

Listing 13.11 PIC, PID Section Definition

```c
.pictables :
{
      = ALIGN(0x8);
__START_PICTABLE = .;
*(.picdynrel)__END_PICTABLE = .;
__PICTABLE_SIZE = __END_PICTABLE - __START_PICTABLE;
```
ROM-RAM Copying

In embedded programming, it is common that data or code of a program residing in ROM gets copied into RAM at runtime.

To indicate such data or code, use the LCF to assign it two addresses:

- The memory segment specifies the intended location in RAM
- The sections segment specifies the resident location in ROM, via its AT (address) parameter

For example, suppose that we want to copy all initialized data into RAM at runtime. At runtime, the system loads the `.main_data` section containing the initialized data to RAM address 0x80000, but until runtime, this section remains in ROM. Listing 13.12 shows part of the corresponding LCF.

**Listing 13.12 Partial LCF for ROM-to-RAM Copy**

```c
# ROM location: address 0x0
# RAM location: address 0x800000
# For clarity, no alignment directives in this listing

MEMORY {
    TEXT (RX) : ORIGIN = 0x0, LENGTH = 0
    DATA (RW) : ORIGIN = 0x800000, LENGTH = 0
}

SECTIONS{
    .main :
        {
            *(.text)
            *(.rodata)
        } > TEXT

    .main_data : AT( ADDR(.main) + SIZEOF(.main) )
        {
            *(.data)
            *(.sdata)
            *(.sbss)
        } > DATA
```

__START_PIDTABLE = .;
*(.piddynrel)__END_PIDTABLE = .;
__PIDTABLE_SIZE = __END_PIDTABLE - __START_PIDTABLE;
} >> DATA
ColdFire Linker

LCF Syntax

```
.uninitialized_data:
{  *(SCOMMON)
  *(.bss)
  *(COMMON)
} >> DATA
```

For program execution to copy the section from ROM to RAM, a copy table such as Listing 13.13 must supply the information that the program needs at runtime. This copy table, which the symbol __S_romp identifies, contains a sequence of three word values per entry:

- ROM start address
- RAM start address
- size

The last entry in this table must be all zeros: this is the reason for the three lines WRITEW(0) before the table closing brace character.

**Listing 13.13 LCF Copy Table for Runtime ROM Copy**

```
# Locate ROM copy table into ROM after initialized data
_romp_at = _main_ROM + SIZEOF(.main_data);

.romp : AT (_romp_at)
{  __S_romp = _romp_at;
     WRITEW(_main_ROM);       #ROM start address
     WRITEW(ADDR(.main_data)); #RAM start address
     WRITEW(SIZEOF(.main_data)); #size
     WRITEW(0);
     WRITEW(0);
     WRITEW(0);
  }
__SP_INIT=.+0 x 4000;   # set stack to 16kb
__heap_addr = __SP_INIT; # heap grows opposite stack direction
__heap_size = 0x10000;   # set heap to 64kb
}
```

Writing Data Directly to Memory

To write data directly to memory, use appropriate WRITEx keywords in your LCF:

- WRITEB writes a byte
ColdFire Linker

LCF Syntax

- WRITEH writes a two-byte halfword
- WRITEW writes a four-byte word.

The system inserts the data at the section’s current address. Listing 13.14 shows an example.

**Listing 13.14 Embedding Data Directly into Output**

```assembly
.example_data_section :
    { WRITEB 0x48; /* 'H' */
      WRITEB 0x69; /* 'i' */
      WRITEB 0x21; /* '!' */
    }
```

To insert a complete binary file, use the INCLUDE keyword, as Listing 13.15 shows.

**Listing 13.15 Embedding a Binary File into Output**

```assembly
_musicStart = .;
INCLUDE music.mid
_musicEnd = .;
}
```  

You must include the file in your IDE project. Additionally, the File Mappings target settings panel must specify resource file for all files that have the same extension as the binary file. Figure 13.1 shows how to make this type designation.
ColdFire Linker

LCF Syntax

Figure 13.1 Marking a Binary File Type as a Resource File
C Compiler

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

- Extensions to Standard C
- C99 Extensions
- GCC Extensions

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 9899-199 C standard (“C89”).

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

Controlling Standard C Conformance

The compiler offers settings that verify how closely your source code conforms to the ISO/IEC 9899-1990 C standard (“C89”). 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.
C Compiler
Extensions to Standard C

Table 14.5 shows how to control the compiler’s features for ISO conformance.

Table 14.1 Controlling conformance to the ISO/IEC 9899-1990 C language

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>ANSI Strict and ANSI Keywords Only in the C/C++ Language Settings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma ANSI_strict</td>
</tr>
<tr>
<td></td>
<td>#pragma only_std_keywords</td>
</tr>
<tr>
<td>command line</td>
<td>-ansi</td>
</tr>
</tbody>
</table>

C++-style Comments
When ANSI strictness is off, the C compiler allows C++-style comments. Listing 14.1 shows an example.

Listing 14.1 C++ Comments

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

Unnamed Arguments
When ANSI strictness is off, the C compiler allows unnamed arguments in function definitions. Listing 14.2 shows an example.

Listing 14.2 Unnamed Arguments

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

Extensions to the 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. Listing 14.3 and Listing 14.4 show examples.
Listing 14.3 Using # in Macro Definitions

```c
#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 14.4 Identifiers After #endif

```c
#ifdef __MWERKS__
   /* . . . */
#endif /*__MWERKS__*/ /* Always OK. */
```

Non-Standard Keywords

When the ANSI keywords setting is off, the C compiler recognizes non-standard keywords that extend the language.

C99 Extensions

The CodeWarrior C compiler accepts most of 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 Compiler
C99 Extensions

- C++-Style Comments
- C++-Style Digraphs
- Empty Arrays in Structures
- Hexadecimal Floating-Point Constants
- Variable-Length Arrays
- Unsuffixed Decimal Literal Values

Controlling C99 Extensions

Table 14.2 shows how to control C99 extensions.

Table 14.2  Controlling C99 extensions to the C language

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Enable C99 Extensions in the C/C++ Language Settings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma c99</td>
</tr>
<tr>
<td>command line</td>
<td>-c99</td>
</tr>
</tbody>
</table>

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. Listing 14.5 shows an example.

Listing 14.5  Trailing comma in enumeration example

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

C99 Extensions

### Compound Literal Values

When the C99 extensions setting is on, the compiler allows literal values of structures and arrays. **Listing 14.6** shows an example.

**Listing 14.6  Example of a Compound Literal**

```c
#pragma c99 on
struct my_struct {
    int i;
    char c[2];
} my_var;

my_var = ((struct my_struct) {x + y, 'a', 0});
```

### Designated Initializers

When the C99 extensions setting is on, the compiler allows an extended syntax for specifying which structure or array members to initialize. **Listing 14.7** shows an example.

**Listing 14.7  Example of Designated Initializers**

```c
#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. */
```

### Predefined Symbol __func__

When the C99 extensions setting is on, the compiler offers the `__func__` predefined variable. **Listing 14.8** shows an example.
C Compiler
C99 Extensions

Listing 14.8 Predefined symbol __func__

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

Implicit Return From main()
When the C99 extensions setting is on, the compiler inserts this statement at the end of a
program’s main() function if the function does not return a value:

```c
return 0;
```

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

Variable Argument Macros
When the C99 extensions setting is on, the compiler allows macros to have a variable
number of arguments. Listing 14.9 shows an example.

Listing 14.9 Variable argument macros example

```c
#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;
}
```
Extra C99 Keywords

When the C99 extensions setting is on, the compiler recognizes extra keywords and the language features they represent. Table 14.3 lists these keywords.

Table 14.3 Extra C99 Keywords

<table>
<thead>
<tr>
<th>This keyword or combination of keywords...</th>
<th>represents this language feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>_Bool</td>
<td>boolean data type</td>
</tr>
<tr>
<td>long long</td>
<td>integer data type</td>
</tr>
<tr>
<td>restrict</td>
<td>type qualifier</td>
</tr>
<tr>
<td>inline</td>
<td>function qualifier</td>
</tr>
<tr>
<td>_Complex</td>
<td>complex number data type</td>
</tr>
<tr>
<td>_Imaginary</td>
<td>imaginary number data type</td>
</tr>
</tbody>
</table>

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 continue until the end of a source code line.

A C-style comment begins with

```
/*
```

ends with

```
*/
```

and may span more than one line.
C++-Style Digraphs

When the C99 extensions setting is on, the compiler recognizes C++-style two-character combinations that represent single-character punctuation. Table 14.4 lists these digraphs.

Table 14.4 C++-Style Digraphs

<table>
<thead>
<tr>
<th>This digraph</th>
<th>is equivalent to this character</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;:</td>
<td>[</td>
</tr>
<tr>
<td>:&gt;</td>
<td>]</td>
</tr>
<tr>
<td>&lt;%</td>
<td>{</td>
</tr>
<tr>
<td>%&gt;</td>
<td>}</td>
</tr>
<tr>
<td>%:</td>
<td>#</td>
</tr>
<tr>
<td>%:%</td>
<td>##</td>
</tr>
</tbody>
</table>

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. Listing 14.10 shows an example.

Listing 14.10 Example of an Empty Array as the Last struct Member

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

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
- 0xEplf
- 0x1.8p0l
The standard library supports printing values of type float in this format using the “%a”
and “%A” specifiers.

Variable-Length Arrays

Variable length arrays are supported within local or function prototype scope, as required

Listing 14.11  Example of C99 Variable Length Array usage

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

While the example shown in Listing 14.12 generates an error message.

Listing 14.12  Bad Example of C99 Variable Length Array usage

```c
#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 Listing 14.13.

Listing 14.13  Bad Example of C99 usage in Function Prototype

```c
#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
```
Unsuffixed Decimal Literal Values

Listing 14.14 shows an example of specifying decimal literal values without a suffix to specify the literal’s type.

Listing 14.14 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)

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 (“C89”) 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
Controlling GCC Extensions

Table 14.5 shows how to turn GCC extensions on or off.

Table 14.5  Controlling GCC extensions to the C language

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Enable GCC Extensions in the C/C++ Language Settings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma gcc_extensions</td>
</tr>
<tr>
<td>command line</td>
<td>-gcc_extensions</td>
</tr>
</tbody>
</table>

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. Listing 14.15 shows an example.

Listing 14.15  Initializing arrays and structures with non-constant values

```c
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 }; 
}
```

The 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 (“C99”) does not specify the size of the `void` type and functions. Listing 14.16 shows an example.
Listing 14.16 Using the sizeof() operator with void and function types

```c
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 */
}
```

**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. **Listing 14.17** shows an example.

Listing 14.17 Using statements and definitions in expressions

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

int main()
{
    return POW2(4);
}
```

**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. **Listing 14.18** shows an example.

Listing 14.18 Redefining a macro without undefining first

```c
#define SOCK_MAXCOLOR 100
#undef SOCK_MAXCOLOR
#define SOCK_MAXCOLOR 200 /* OK: this macro is previously undefined. */
#define SOCK_MAXCOLOR 300
```
The 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. Listing 14.19 shows an example.

Listing 14.19 Using the typeof() operator

```c
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;
```

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.

The __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.

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. Listing 14.20 shows an example.

Listing 14.20 Forward declaration of an empty array

```c
static int a[]; /* Allowed only when GCC extensions are on. */
/* ... */
static int a[10]; /* Complete declaration. */
```
C Compiler

GCC Extensions

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. Listing 14.21 shows an example.

Listing 14.21 Using the shorter form of the conditional expression

```c
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. */
}
```

The __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. Listing 14.22 shows an example.

Listing 14.22 Example for __builtin_expect() operator

```c
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);
        }
    }
}
```
Void Return Statements

When the GCC extensions setting is on, the compiler allows you to place expressions of type `void` in a `return` statement. Listing 14.23 shows an example.

Listing 14.23  Returning void

```c
void f(int a)
{
    /* ... */
    return; /* Always OK. */
}

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

Minimum and Maximum Operators

The compiler recognizes built-in minimum (`<?`) and maximum (`>?>`) operators.

Listing 14.24  Example of minimum and maximum operators

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

GCC Extensions
C++ Compiler

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

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

C++ Compiler Performance

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

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

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.

Using the Instance Manager

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

- template functions
C++ Compiler

Extensions to Standard C++

- 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.

Table 15.1 shows how to control the C++ instance manager.

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Use Instance Manager in the C/C++ Language Settings panel</td>
</tr>
<tr>
<td>source code</td>
<td><code>#pragma instmgr_file</code></td>
</tr>
<tr>
<td>command line</td>
<td><code>-instmgr</code></td>
</tr>
</tbody>
</table>

Extensions to Standard C++

The CodeWarrior C++ compiler has features and capabilities that are not described in the ISO/IEC 14882-1998 C++ standard:

- **`__PRETTY_FUNCTION__`** Identifier
- Standard and Non-Standard Template Parsing

**__PRETTY_FUNCTION__** Identifier

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

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 14882-1998 C++, §14.6). Listing 15.1 shows an example.

### Listing 15.1 Using the `typename` keyword

```cpp
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. Listing 15.2 shows an example.

### Listing 15.2 Using the `template` keyword

```cpp
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. Listing 15.3 shows an example.

### Listing 15.3 Binding non-dependent identifiers

```cpp
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 14882-1998 C++, §14.6.2). See Listing 15.4.


C++ Compiler

Extensions to Standard C++

Listing 15.4 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 14882-1998 C++, §14.6.2.2) and the context of its instantiation (ISO/IEC 14882-1998 C++, §14.6.4.2). Listing 15.5 shows an example.

Listing 15.5 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 Listing 15.6.
C++ Compiler
Implementation-Defined Behavior

Listing 15.6 Function call with type-dependent argument and external names

```cpp
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 Listing 15.7.

Listing 15.7 Assembly statements cannot depend on template arguments

```cpp
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 Listing 15.8.

Listing 15.8 Address of Template-id Supported

```cpp
template <typename T> void funcA(T) {}
template <typename T> void funcB(T) {}
...
funcA( &funcB<int> ); // now accepted
```

Implementation-Defined Behavior

Annex A of the ISO/IEC 14882-1998 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 Table 15.2, based on the ISO/IEC 14882-1998 C++ Standard, Annex A.
### C++ Compiler

*Implementation-Defined Behavior*

**NOTE** The term *unlimited* in Table 15.2 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>
<thead>
<tr>
<th>Behavior</th>
<th>Standard Minimum Guideline</th>
<th>CodeWarrior Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Nesting levels of compound statements, iteration control structures, and selection control structures</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Nesting levels of conditional inclusion</td>
<td>256</td>
<td>32</td>
</tr>
<tr>
<td>Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Nesting levels of parenthesized expressions within a full expression</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Number of initial characters in an internal identifier or macro name</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Number of initial characters in an external identifier</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>External identifiers in one translation unit</td>
<td>65536</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Identifiers with block scope declared in one block</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Macro identifiers simultaneously defined in one translation unit</td>
<td>65536</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Parameters in one function definition</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Arguments in one function call</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Parameters in one macro definition</td>
<td>256</td>
<td>128</td>
</tr>
</tbody>
</table>
## Table 15.2 Implementation Quantities for the C/C++ Compiler (ISO/IEC 14882-1998 C++, §A) (continued)

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Standard Minimum Guideline</th>
<th>CodeWarrior Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arguments in one macro invocation</td>
<td>256</td>
<td>128</td>
</tr>
<tr>
<td>Characters in one logical source line</td>
<td>65536</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Characters in a character string literal or wide string literal (after concatenation)</td>
<td>65536</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Size of an object</td>
<td>262144</td>
<td>2 GB</td>
</tr>
<tr>
<td>Nesting levels for <code>#include</code> files</td>
<td>256</td>
<td>32</td>
</tr>
<tr>
<td><strong>Case labels for a switch statement</strong> (excluding those for any nested switch statements)</td>
<td>16384</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Data members in a single class, structure, or union</td>
<td>16384</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Enumeration constants in a single enumeration</td>
<td>4096</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Levels of nested class, structure, or union definitions in a single struct-declaration-list</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Functions registered by <code>atexit()</code></td>
<td>32</td>
<td>64</td>
</tr>
<tr>
<td>Direct and indirect base classes</td>
<td>16384</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Direct base classes for a single class</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Members declared in a single class</td>
<td>4096</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Final overriding virtual functions in a class, accessible or not</td>
<td>16384</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Direct and indirect virtual bases of a class</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Static members of a class</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Friend declarations in a class</td>
<td>4096</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Access control declarations in a class</td>
<td>4096</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Member initializers in a constructor definition</td>
<td>6144</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>
C++ Compiler

GCC Extensions

The CodeWarrior C++ compiler recognizes some extensions to the ISO/IEC 14882-1998 C++ standard that are also recognized by the GCC (GNU Compiler Collection) C++ compiler.

These extensions are:

- Using the :: Operator in Class Declarations

### Using the :: Operator in Class Declarations

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

**Listing 15.9 Using the :: operator in class declarations**

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

---

Table 15.2 Implementation Quantities for the C/C++ Compiler (ISO/IEC 14882-1998 C++, §A) (continued)

<table>
<thead>
<tr>
<th>Behavior</th>
<th>Standard Minimum Guideline</th>
<th>CodeWarrior Limit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scope qualifications of one identifier</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Nested external specifications</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Template arguments in a template declaration</td>
<td>1024</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Recursively nested template instantiations</td>
<td>17</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Handlers per try block</td>
<td>256</td>
<td>Unlimited</td>
</tr>
<tr>
<td>Throw specifications on a single function declaration</td>
<td>256</td>
<td>Unlimited</td>
</tr>
</tbody>
</table>

---

GCC Extensions

The CodeWarrior C++ compiler recognizes some extensions to the ISO/IEC 14882-1998 C++ standard that are also recognized by the GCC (GNU Compiler Collection) C++ compiler.

These extensions are:

- Using the :: Operator in Class Declarations

**Using the :: Operator in Class Declarations**

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

**Listing 15.9 Using the :: operator in class declarations**

```cpp
class MyClass {
    int MyClass::getval();
};
```
Embedded C++

Embedded C++ (EC++) is a subset of the ISO/IEC 14882-1998 C++ language that is intended to compile into smaller, faster executable code suitable for embedded systems. Embedded C++ source code is upwardly compatible with ISO/IEC C++ source code.

- Activating EC++
- Differences Between ISO C++ and EC++
- EC++ Specifications

Activating EC++

Table 15.3 shows how to control Embedded C++ conformance.

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>EC++ Compatibility Mode in the C/C++ Language Settings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma ecplusplus</td>
</tr>
<tr>
<td>command line</td>
<td>-dialect ec++</td>
</tr>
</tbody>
</table>

To test for EC++ compatibility mode at compile time, use the __embedded_cplusplus predefined symbol.

Differences Between ISO C++ and EC++

The EC++ proposal does not support the following ISO/IEC 14882-1998 C++ features:

- Templates
- Libraries
- File Operations
- Localization
- Exception Handling
- Unsupported Language Features
Templates

ISO/IEC C++ specifies templates. The EC++ proposal does not include template support for class or functions.

Libraries

The EC++ proposal supports the `<string>`, `<complex>`, `<ios>`, `<streambuf>`, `<istream>`, and `<ostream>` classes, but only in a non-template form. The EC++ specifications do not support any other ISO/IEC C++ libraries, including the STL-type algorithm libraries.

File Operations

The EC++ proposal does not support any file operations except simple console input and output file types.

Localization

The EC++ proposal does not contain localization libraries because of the excessive memory requirements.

Exception Handling

The EC++ proposal does not support exception handling.

Unsupported Language Features

The EC++ proposal does not support the following language features:

- mutable specified
- RTTI
- namespace
- multiple inheritance
- virtual inheritance

EC++ Specifications

Topics in this section describe how to design software that adhere to the EC++ proposal:

- Language Related Issues
- Library-Related Issues
Language Related Issues

To make sure your source code complies with both ISO/IEC 14882-1998 C++ and EC++ standards, follow these guidelines:

- Do not use RTTI (Run Time Type Identification).
- Do not use exception handling, namespaces, or other unsupported features.
- Do not use multiple or virtual inheritance.

Library-Related Issues

Do not refer to routines, data structures, and classes in the Metrowerks Standard Library (MSL) for C++.
Tool Performance

Some options for CodeWarrior compilers and linkers affect how much time these tools take. By managing these options so that they are used only when they are needed, you can reduce the time needed to build your software.

Precompiling

Source code files in a project often use many header files. Typically, the same header files are included by each source code file in a project, forcing the compiler to read these header files repeatedly during compilation. To shorten the time spent compiling and recompiling the same header files, CodeWarrior compilers can precompile a header file once instead of preprocessing it several times.

- When to Use Precompiled Files
- What Can be Precompiled
- Using a Precompiled Header File
- Preprocessing and Precompiling
- Pragma Scope in Precompiled Files
- Precompiling a File in the CodeWarrior IDE
- Updating a Precompiled File Automatically

When to Use Precompiled Files

As a convenience, programmers often create a header file that contains commonly-used preprocessor definitions and includes frequently-used header files. This header file is then included by each source code file in a project, saving the programmer some time and effort while writing source code.

This convenience comes at a cost, though. While the programmer saves time typing, the compiler does extra work, preprocessing and compiling this header file each time it compiles a source code file that includes it.

This header file can be precompiled so that, instead of preprocessing files several times, the compiler needs to load just one precompiled header file.
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.
- A C source code file that will be automatically precompiled must have .pch file name extension.
- Precompiled files must have a .mch file name extension.
- The file to be precompiled does not have to be in a CodeWarrior IDE project, although a project must be open to precompile the file.
- The CodeWarrior IDE uses the build target settings to precompile a file.
- 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 build targets are not interchangeable.
- A source file may include only one precompiled file.
- A file may not define any items before including a precompiled file.

Typically, a source code file includes a precompiled header file before anything else (except comments).

Using a Precompiled Header File

Although a precompiled file is not a text file, you use it like you would a regular header file. To include a precompiled header file in a source code file, use the #include directive.

NOTE Unlike regular header files in text format, a source code file may include only one precompiled file.

TIP Instead of explicitly including a precompiled file in each source code file with the #include directive, put the #include directive in the Prefix Text field of the CodeWarrior IDE’s C/C++ Preprocessor settings panel and make sure that the Use prefix in precompiled headers option is on. If the Prefix File field already specifies a file name, include the precompiled file in the prefix file with the #include directive.

Listing 16.1 and Listing 16.2 show an example.
Preprocessing and Precompiling

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. Thus, if a file’s contents are surrounded with:

\#ifndef MYHEADER_H
\#define MYHEADER_H
/* file contents */
\#endif
Pragma Scope in Precompiled Files

Pragma settings inside a precompiled file affect only the source code within that file. The pragma settings for an item declared in a precompiled header file (such as data or a function) are saved then restored when the precompiled header file is included. For example, the source code in Listing 16.3 specifies that the variable xxx is a far variable.

Listing 16.3 Pragma Settings in a Precompiled Header

```c
/* 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 Listing 16.4 includes the precompiled version of Listing 16.3.

Listing 16.4 Pragma Settings in an Included Precompiled File

```c
/* 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.

Precompiling a File in the CodeWarrior IDE

To precompile a file in the CodeWarrior IDE, use the Precompile command in the Project menu:

1. Start the CodeWarrior IDE.
2. Open or create a project.
3. Choose or create a build target in the project.
   The settings in the project’s active build target will be used when preprocessing and
   precompiling the file you want to precompile.

4. Open the source code file to precompile.
   See “What Can be Precompiled” on page 162 for information on what a precompiled
   file may contain.

5. From the Project menu, choose Precompile.
   A save dialog box appears.

6. Choose a location and type a name for the new precompiled file.
   The IDE precompiles the file and saves it.

7. Click Save.
   The save dialog box closes, and the IDE precompiles the file you opened, saving it in
   the folder you specified, giving it the name you specified.

You may now include the new precompiled file in source code files.

**Updating a 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 (for C header files).
- 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.

See the CodeWarrior IDE User Guide for information on how the IDE determines
that a file must be updated.

The IDE uses the build target’s settings to preprocess and precompile files.
Tool Performance
Pragma Scope in Precompiled Files
Intermediate 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 describes these optimizations and how to apply them:

- **Interprocedural Analysis**
- **Intermediate Optimizations**
- **Inlining**

**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 and programs to improve performance and reduce size.

- **Invoking Interprocedural Analysis**
- **File-Level Optimizations**
- **Program-Level Optimizations**
- **Program-Level Requirements**
Intermediate Optimizations

Interprocedural Analysis

Invoking Interprocedural Analysis

Table 17.1 describes how to control interprocedural analysis.

**Table 17.1 Controlling interprocedural analysis**

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose an item in the <strong>IPA</strong> option of the <strong>C</strong>/<strong>C++</strong> Language Settings settings pane.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma ipa off</td>
</tr>
<tr>
<td>command line</td>
<td>-ipa off</td>
</tr>
</tbody>
</table>

File-Level Optimizations

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 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.

Program-Level Optimizations

When interprocedural analysis is set to optimize at the program level, the compiler reads and analyzes all files in a program before generating instructions and data.

At this level of interprocedural analysis, the compiler generates the most efficient instructions and data for inline function calls and C++ exception handling compared to other levels. The compiler is also able to increase character string reuse and pooling, reducing the size of object code.

Program-Level Requirements

Program-level interprocedural analysis imposes some requirements and limitations on the source code files that the compiler translates:

- Dependencies Among Source Files
- Function and Top-level Variable Declarations
- Type Definitions
Intermediate Optimizations
Interprocedural Analysis

- Unnamed Structures and Enumerations in C

Dependencies Among Source Files
A change to even a single source file in a program still requires that the compiler read and analyze all files in the program, even those files that are not dependent on the changed file. This requirement significantly increases compile time.

Function and Top-level Variable Declarations
Because the compiler treats all files that compose a program as if they were a single, large source file. Make sure all non-static declarations for variables or functions with the same name are identical. See Listing 17.1 for an example of declarations that prevent the compiler from applying program-level analysis. Listing 17.2 fixes this problem by renaming the conflicting symbols.

Listing 17.1 Declaration conflicts in program-level interprocedural analysis

/* file1.c */
extern int i;
extern int f();
int main(void)
{
    return i + f();
}

/* file2.c */
short i;       /* Conflict with variable i in file1.c. */
extern void f(); /* Conflict with function f() in file1.c */

Listing 17.2 Fixing declaration conflicts for program-level interprocedural analysis

/* file1.c */
extern int i1;
extern int f1();
int main(void)
{
    return i1 + f1();
}

/* file2.c */
short i2;
extern void f2();
Intermediate Optimizations
Interprocedural Analysis

Type Definitions

Because the compiler examines all source files for a program, make sure all definitions for a type are the same. See Listing 17.3 for an example of conflicting type definitions. Listing 17.4 and Listing 17.5 show suggested solutions.

Listing 17.3 Type definitions conflicts in program-level interprocedural analysis

```c
/* fileA.c */
struct a_rec { int i, j; }
   a_rec a;

/* fileB.c */
struct a_rec { char c; }; /* Conflict with a_rec in fileA.c */
   a_rec b;
```

Listing 17.4 Fixing type definitions conflicts in C

```c
/* fileA.c */
struct a1_rec { int i, j; }
   a1_rec a;

/* fileB.c */
struct a2_rec { char c; }
   a2_rec b;
```

Listing 17.5 Fixing type definitions conflicts in C++

```c
/* fileA.c */
namespace { struct a_rec { int i, j; } }
   a_rec a;

/* fileB.c */
namespace { struct a_rec { char c; } }
   a_rec b;
```

Unnamed Structures and Enumerations in C

The C language allows anonymous struct and enum definitions in type definitions. Using such definitions prevents the compiler from properly applying program-level interprocedural analysis. Make sure to give names to structures and enumerations in type definitions. Listing 17.6 shows an example of unnamed structures and enumerations and Listing 17.7 shows a suggested solution.
Intermediate Optimizations

Listing 17.6 Unnamed structures and enumerations in C

/* In C, the types x_rec and y_enum each represent a structure and an enumeration with no name.

In C++ these same statements define a type x_rec and y_enum, a structure named x_rec and an enumeration named y_enum.
*/
typedef struct { int a, b, c; } x_rec;
typedef enum { Y_FIRST, Y_SECOND, Y_THIRD } y_enum;

Listing 17.7 Naming structures and enumerations in C

typedef struct x_rec { int a, b, c; } x_rec;
typedef enum y_enum { Y_FIRST, Y_SECOND, Y_THIRD } y_enum;

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**

**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.
Intermediate Optimizations

Table 17.2 describes how to control the optimization for dead code elimination.

Table 17.2 Controlling dead code elimination

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose Level 1, Level 2, Level 3, or Level 4 in the Global Optimizations settings pane.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_dead_code on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]deadcode</td>
</tr>
</tbody>
</table>

In Listing 17.8, the call to `func1()` will never execute because the if statement that it is associated with will never be true. Consequently, the compiler can safely eliminate the call to `func1()`, as shown in Listing 17.9.

Listing 17.8 Before dead code elimination

```c
void func_from(void)
{
    if (0)
    {
        func1();
    }
    func2();
}
```

Listing 17.9 After dead code elimination

```c
void func_to(void)
{
    func2();
}
```

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.
Table 17.3 describes how to control the optimization for expression simplification.

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose Level 1, Level 2, Level 3, or Level 4 in the Global Optimizations settings pane.</td>
</tr>
<tr>
<td>source code</td>
<td>There is no pragma to control this optimization.</td>
</tr>
<tr>
<td>command line</td>
<td>-opt level=1,-opt level=2,-opt level=3,-opt level=4</td>
</tr>
</tbody>
</table>

For example, Listing 17.10 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 17.10 Before expression simplification

```c
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;
}
```

Listing 17.11 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 17.11 After expression simplification

```c
void func_to(int* result1, int* result2, int* result3, int* result4, int x) {
    *result1 = x;
    *result2 = x << 1;
    *result3 = 0;
    *result4 = 5;
}
```
Intermediate Optimizations

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. Table 17.4 describes how to control the optimization for common subexpression elimination.

For example, in Listing 17.12, the subexpression \(x \times y\) occurs twice.

Listing 17.12 Before common subexpression elimination

```c
void func_from(int* vec, int size, int x, int y, int value)
{
    if (x * y < size)
    {
        vec[x * y - 1] = value;
    }
}
```

Listing 17.13 shows equivalent source code after the compiler applies common subexpression elimination. The compiler generates instructions to compute \(x \times y\) and store it in a hidden, temporary variable. The compiler then replaces each instance of the subexpression with this variable.
Intermediate Optimizations

Listing 17.13 After common subexpression elimination

```c
void func_to(int* vec, int size, int x, int y, int value)
{
    int temp = x * y;
    if (temp < size)
    {
        vec[temp - 1] = value;
    }
}
```

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. Table 17.5 describes how to control the optimization for copy propagation.

Table 17.5 Controlling copy propagation

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose Level 2, Level 3, or Level 4 in the Global Optimizations settings pane.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_propagation on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]prop[agation]</td>
</tr>
</tbody>
</table>

For example, in Listing 17.14, the variable `j` is assigned the value of `x`. But `j`’s value is never changed, so the compiler replaces later instances of `j` with `x`, as shown in Listing 17.15.

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.

Listing 17.14 Before copy propagation

```c
void func_from(int* a, int x)
{
    int i;
    int j;
    j = x;
}
```
Intermediate Optimizations

Listing 17.15 After copy propagation

```c
void func_to(int* a, int x)
{
    int i;
    int j;
    j = x;
    for (i = 0; i < x; i++)
    {
        a[i] = x;
    }
}
```

Dead Store Elimination

Dead store elimination removes unused assignment statements. This optimization reduces size and improves speed.

Table 17.6 describes how to control the optimization for dead store elimination.

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose Level 3 or Level 4 in the Global Optimizations settings pane.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_dead_assignments on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]deadstore</td>
</tr>
</tbody>
</table>

For example, in Listing 17.16 the variable \( x \) is first assigned the value of \( y \times y \). However, this result is not used before \( x \) is assigned the result returned by a call to `getresult()`.

In Listing 17.17 the compiler can safely remove the first assignment to \( x \) since the result of this assignment is never used.
Intermediate Optimizations

### Listing 17.16 Before dead store elimination

```c
void func_from(int x, int y)
{
    x = y * y;
    otherfunc1(y);
    x = getresult();
    otherfunc2(y);
}
```

### Listing 17.17 After dead store elimination

```c
void func_to(int x, int y)
{
    otherfunc1(y);
    x = getresult();
    otherfunc2(y);
}
```

### 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.

Table 17.7 describes how to control the optimization for live range splitting.

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose Level 3 or Level 4 in the Global Optimizations settings pane.</td>
</tr>
<tr>
<td>source code</td>
<td>There is no pragma to control this optimization.</td>
</tr>
<tr>
<td>command line</td>
<td>-opt level=3, -opt level=4</td>
</tr>
</tbody>
</table>

For example, in Listing 17.18 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.
Intermediate Optimizations

In Listing 17.19, 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 17.18  Before live range splitting

```c
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);
}
```

Listing 17.19  After live range splitting

```c
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);
}
```

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.
Intermediate Optimizations

Table 17.8 describes how to control the optimization for loop-invariant code motion.

Table 17.8 Controlling loop-invariant code motion

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose Level 3 or Level 4 in the Global Optimizations settings pane.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_loop_invariants on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]loop[invariants]</td>
</tr>
</tbody>
</table>

For example, in Listing 17.20, 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 17.20 Before loop-invariant code motion

```c
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;
    }
}
```

Listing 17.21 After loop-invariant code motion

```c
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;
    }
}
```
Intermediate Optimizations

Strength Reduction

Strength reduction attempts to replace slower multiplication operations with faster addition operations. This optimization improves execution speed but increases code size. Table 17.9 describes how to control the optimization for strength reduction.

Table 17.9 Controlling strength reduction

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose Level 3 or Level 4 in the Global Optimizations settings pane.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_strength_reduction on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt [no]strength</td>
</tr>
</tbody>
</table>

For example, in Listing 17.22, the assignment to elements of the vec array use a multiplication operation that refers to the for loop’s counter variable, i.

In Listing 17.23, 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 17.22 Before strength reduction

```c
void func_from(int* vec, int max, int fac)
{
    int i;
    for (i = 0; i < max; ++i)
    {
        vec[i] = fac * i;
    }
}
```

Listing 17.23 After strength reduction

```c
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;
}
```

### 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 time that the processor executes a loop’s completion test and branching instructions compared to the time the processor executes the loop’s body. This optimization improves execution speed but increases code size.

Table 17.10 describes how to control the optimization for loop unrolling.

<table>
<thead>
<tr>
<th>Turn control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Choose Level 3 or Level 4 in the Global Optimizations settings pane.</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma opt_unroll_loops on</td>
</tr>
<tr>
<td>command line</td>
<td>-opt level=3,-opt level=4</td>
</tr>
</tbody>
</table>

For example, in Listing 17.24, 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.

In Listing 17.25, 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 17.24 Before loop unrolling**

```
const int MAX = 100;
void func_from(int* vec)
```
Intermediate Optimizations

Inlining

```c
{  
  int i;  
  for (i = 0; i < MAX; ++i)  
  {  
    otherfunc(vec[i]);  
  } 
}
```

Listing 17.25  After loop unrolling

```c
const int MAX = 100;
void func_to(int* vec)  
{  
  int i;  
  for (i = 0; i < MAX;)  
  {  
    otherfunc(vec[i]);  
    ++i;  
    otherfunc(vec[i]);  
    ++i;  
  } 
}
```

Inlining

Inlining replaces instructions that call a function and return from it with the actual instructions of the function being called. Inlining functions 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 describes how to specify which functions to inline and how the compiler performs the inlining:

- Choosing Which Functions to Inline
- Inlining Techniques

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
Intermediate Optimizations

Inlining

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. Listing 17.26 shows an example.

Listing 17.26 Specifying to the compiler that a function may be inlined

```c
#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))**. Listing 17.27 shows an example.

Listing 17.27 Specifying to the compiler that a function must never be inlined

```c
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. Listing 17.28 shows an example.

Listing 17.28 Specifying that no functions may be inlined

```c
#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. */
```
Intermediate Optimizations

Inlining

inline int also_attempt_to_inline(void)
{
    return 10;
}

Some kinds of functions are never inlined:

- functions with variable argument lists
- functions declared with __attribute__((never_inline))
- functions compiled with #pragma optimize_for_size on or the Optimize For Size setting in the IDE’s Global Optimizations panel
- functions which have their pointers stored in variables

The compiler will not inline these functions, even if they are defined with the inline, __inline__, or __inline keywords.

Inlining Techniques

The depth of inlining describes how many levels of function calls the compiler will inline. The Inline Depth setting in the IDE’s C/C++ Language settings 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 allows deferred inlining, which specifies to the compiler to delay a function’s compilation until any functions that it calls have been compiled. The Deferred Inlining setting in the IDE’s C/C++ Language settings panel and the defer_codegen 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++ Language 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++ Language 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
Intermediate Optimizations

Inlining

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.
Intermediate Optimizations

Inlining
Inline Assembly

This chapter explains support for inline assembly language programming. Inline assembly language are assembly language instructions and directives embedded in C and C++ source code. The standalone assembler, different software component, is not a topic of this chapter. For information on the stand-alone assembler, refer to the Assembler Guide.

- Inline Assembly Syntax
- Inline Assembly Directives

Inline Assembly Syntax

Syntax explanation topics are:

- Statements
- Additional Syntax Rules
- Preprocessor Features
- Local Variables and Arguments
- Returning From a Routine

Statements

All internal assembly statements must follow this syntax:

[LocalLabel:] (instruction | directive) [operands];

Other rules for statements are:

- The assembly instructions are the standard ColdFire instruction mnemonics.
- Each instruction must end with a newline character or a semicolon (;).
- Hexadecimal constants must be in C style: 0xABCDEF is a valid constant, but $ABCDEF is not.
- Assembler directives, instructions, and registers are not case-sensitive. To the inline assembler, these statements are the same:

  move.l b, DO
  MOVE.L b, d0
Inline Assembly

Inline Assembly Syntax

- To specify assembly-language interpretation for a block of code in your file, use the `asm` keyword.

**NOTE** To make sure that the C/C++ compiler recognizes the `asm` keyword, you must clear the **ANSI Keywords Only** checkbox of the **C/C++ Language** panel.

Listing 18.1 and Listing 18.2 are valid examples of inline assembly code:

### Listing 18.1  Function-Level Sample

```c
long int b;
struct mystruct {
    long int a;
};
static asm long f(void) // Legal asm qualifier
{
    move.l struct(mystruct.a)(A0),D0 // Accessing a struct.
    add.l b,D0 // Using a global variable, put return value
                // in D0.
    rts // Return from the function:
         // result = mystruct.a + b
}
```

### Listing 18.2  Statement-Level Sample

```c
long square(short a)
{
    asm {
        move.w a,d0 // fetch function argument ‘a’
        mulu.w d0,d0 // multiply
        return // return from function (result is in D0)
    }
}
```

**NOTE** Regardless of its settings, the compiler never optimizes assembly-language functions. However, to maintain integrity of all registers, the compiler notes which registers inline assembly uses.

### Additional Syntax Rules

These rules pertain to labels, comments, structures, and global variables:
Inline Assembly

Inline Assembly Syntax

- Each label must end with a colon; labels may contain the @ character. For example, x1: and @x2: would be valid labels, but x3 would not — it lacks a colon.
- Comments must use C/ C++ syntax: either starting with double slash characters ( // ) or enclosed by slash and asterisk characters ( /* ... */ ).
- To refer to a field in a structure, use the struct construct:
  
  \texttt{struct (structTypeName.fieldName) structAddress}
  
  For example, suppose that A0 points to structure WindowRecord. This instruction moves the structure's refCon field to D0:

  \texttt{move.l \ struct(WindowRecord.refCon) \ (A0), \ D0}

- To refer to a global variable, merely use its name, as in the statement

  \texttt{move.w \ x, \ d0 \ // \ Move \ x \ into \ d0}

Preprocessor Features

You can use all preprocessor features, such as comments and macros, in the inline assembler. But when you write a macro definition, remember to:

- End each assembly statement with a semicolon (;) — (the preprocessor ignores newline characters).
- Use the \% character, instead of #, to denote immediate data, — the preprocessor uses # as a concatenate operator.

Local Variables and Arguments

Handling of local variables and arguments depends on the level of inline assembly. However, for optimization level 1 or greater, you can force variables to stay in a register by using the symbol \$.

Function-Level

The function-level inline assembler lets you refer to local variables and function arguments yourself, handles such references for you.

For your own references, you must explicitly save and restore processor registers and local variables when entering and leaving your inline assembly function. You cannot refer to the variables by name, but you can refer to function arguments off the stack pointer. For example, this function moves its argument into d0:

\begin{verbatim}
asm void alpha(short n)
{
  move.w 4(sp),d0  //  n
\end{verbatim}
Inline Assembly

Inline Assembly Syntax

```
// . . .
}
```

To let the *inline assembler* handle references, use the directives `fralloc` and `frfree`, according to these steps:

1. Declare your variables as you would in a normal C function.
2. Use the `fralloc` directive. It makes space on the stack for the local stack variables. Additionally, with the statement `link #x,a6`, this directive reserves registers for the local register variables.
3. In your assembly, you can refer to the local variables and variable arguments by name.
4. Finally, use the `frfree` directive to free the stack storage and restore the reserved registers. (It is somewhat easier to use a C wrapper and statement level assembly.)

Listing 18.3 is an example of using local variables and function arguments in function-level inline assembly.

Listing 18.3 Function-level Local Variables, Function Arguments

```c
static asm short f(short n)
{
    register short a; // Declaring a as a register variable
    short b; // and b as a stack variable
    // Note that you need semicolons after these statements.
    fralloc + // Allocate space on stack, reserve registers.
    move.w n,a // Using an argument and local var.
    add.w a,a
    move.w a,D0
    frfree // Free space that fralloc allocated
    rts
}
```

Statement-Level

Statement-level inline assembly allows full access to local variables and function arguments without using the `fralloc` or `frfree` directives.

Listing 18.4 is an example of using local variables and function arguments in statement-level inline assembly. You may place statement-level assembly code anywhere in a C/C++ program.

Listing 18.4 Statement-Level Local Variables, Function Arguments

```c
long square(short a)
{
```

190  CodeWarrior Build Tools Reference ColdFire™ Architectures Edition
long result=0;
asm {
    move.w a,d0 // fetch function argument ‘a’
    mulu.w d0,d0 // multiply
    move.l d0,result // store in local ‘result’ variable
}
return result;

Returning From a Routine

Every inline assembly function (not statement level) should end with a return statement. Use the \texttt{rts} statement for ordinary C functions, as Listing 18.5 shows.

\begin{verbatim}
Listing 18.5 Assembly Function Return
asm void f(void)
{  add.l d4, d5} // Error, no RTS statement
asm void g(void)
{  add.l d4, d5
   rts} // OK
\end{verbatim}

For statement-level returns, see “\texttt{return}” on page 196 and “\texttt{naked}” on page 195.

Inline Assembly Directives

Table 18.1 lists special assembler directives that the ColdFire inline assembler accepts. Explanations follow the table.

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{dc} & \textbf{ds} & \textbf{entry} \\
\hline
fralloc & ffree & machine \\
naked & opword & return \\
\hline
\end{tabular}
\caption{Inline Assembly Directives}
\end{table}

\textbf{NOTE} Except for dc and ds, the inline assembly directives are available only for function/routine level.
**dc**

Defines blocks of constant expressions as initialized bytes, words, or longwords. (Useful for inventing new opcodes to be implemented via a loop.)

\[
dc[.(b|w|l)]
\]

**Parameters**

- **b**
  - Byte specifier, which lets you specify any C (or Pascal) string constant.
- **w**
  - Word specifier (the default), which lets you specify any 16-bit relative offset to a local label.
- **l**
  - Longword specifier.
- **constexpr**
  - Name for block of constant expressions.

**Example**

```c
asm void alpha(void)
{
    x1: dc.b  "Hello world!\n" // Creating a string
    x2: dc.w  1,2,3,4           // Creating an array
    x3: dc.l  3000000000        // Creating a number
}
```

**ds**

Defines a block of bytes, words, or longwords, initialized with null characters. Pushes labels outside the block.

\[
ds[.(b|w|l)]
\]
Parameters

- **b**
  - Byte specifier.
- **w**
  - Word specifier (the default).
- **l**
  - Longword specifier.
- **size**
  - Number of bytes, words, or longwords in the block.

Example

This statement defines a block big enough for the structure `DRVRHeader`:

```
ds.b sizeof(DRVRHeader)
```

entry

Defines an entry point into the current function. Use the **extern** qualifier to declare a global entry point and use the **static** qualifier to declare a local entry point. If you leave out the qualifier, **extern** is assumed (Listing 18.6).

```
entry [extern|static] name
```

Parameters

- **extern**
  - Specifier for a global entry point (the default).
- **static**
  - Specifier for a local entry point.
- **name**
  - Name for the new entry point.

Example

Listing 18.6 defines the new local entry point `MyEntry` for function `MyFunc`.

```
static long MyEntry(void);
static asm long MyFunc(void)
```

Listing 18.6 Entry Directive Example
Inline Assembly

Inline Assembly Directives

```
{  
  move.l a,d0  
  bra.s L1  
  entry static MyEntry  
  move.l b,d0  
  L1: rts  
}
```

fralloc

Lets you declare local variables in an assembly function.

fralloc [+]

Parameter

+  
  Optional ColdFire-register control character.

Remarks

This directive makes space on the stack for your local stack variables. It also reserves registers for your local register variables (with the statement link #r,a6).

Without the + control character, this directive pushes modified registers onto the stack.

With the + control character, this directive pushes all register arguments into their ColdFire registers.

Counterpart to the frfree directive.

frfree

Frees the stack storage area; also restores the registers (with the statement unlk a6) that fralloc reserved.

frfree
Inline Assembly

Inline Assembly Directives

**machine**

Specifies the CPU for which the compiler generates its inline-assembly instructions.

**Parameter**

**processor**

**Remarks**

If you use this directive to specify a target processor, additional inline-assembler instructions become available — instructions that pertain only to that processor. For more information, see the Freescale processor user’s manual.

**naked**

Suppresses the compiler-generated stackframe setup, cleanup, and return code.

**Remarks**

Functions with this directive cannot access local variables by name. They should not contain C code that implicitly or explicitly uses local variables or memory. Counterpart to the **return** directive.

**Example**

Listing 18.7 is an example use of this directive.

**Listing 18.7  Naked Directive Example**

```c
long square(short)
{
    asm{
        naked // no stackframe or compiler-generated rts
        move.w 4(sp),d0 // fetch function argument from stack
        mulu.w d0,d0 // multiply
        rts // return from function: result in D0
    }
}
```
Inline Assembly

Inline Assembly Directives

opword

Writes machine-instruction constants directly into the executable file, without any error checking.

opword constant[, constant]

Parameter

custom

Any appropriate machine-code value.

Example

opword 0x7C0802A6 — which is equivalent to the instruction mflr r0.

return

Inserts a compiler-generated sequence of stackframe cleanup and return instructions. Counterpart to the naked directive.

return instruction[, instruction]

Parameter

instruction

Any appropriate C instruction.
ColdFire Code Generation

This chapter describes the code generation features and specifications that the CodeWarrior offers.

- Code Generation Limits
- Integer Representation
- Calling Conventions
- Variable Allocation
- Register Variables
- Position-Independent Code
- Cryptographic Acceleration Instructions

Code Generation Limits


Integer Representation

The ColdFire compiler lets you specify the number of bytes that the compiler allocates for an int. Table 19.1 shows the size and range of the integer types available for ColdFire targets.

<table>
<thead>
<tr>
<th>Type</th>
<th>Option Setting</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>bool</td>
<td>n/a</td>
<td>8 bits</td>
<td>true or false</td>
</tr>
</tbody>
</table>

Table 19.1 ColdFire Integer Types
## ColdFire Code Generation

### Integer Representation

<table>
<thead>
<tr>
<th>Type</th>
<th>Option Setting</th>
<th>Size</th>
<th>Range</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>Use Unsigned Chars is off in the C/C++ Language panel</td>
<td>8 bits</td>
<td>-128 to 127</td>
</tr>
<tr>
<td></td>
<td>Use Unsigned Chars is on in the C/C++ Language panel</td>
<td>8 bits</td>
<td>0 to 255</td>
</tr>
<tr>
<td>signed char</td>
<td>n/a</td>
<td>8 bits</td>
<td>-128 to 127</td>
</tr>
<tr>
<td>unsigned char</td>
<td>n/a</td>
<td>8 bits</td>
<td>0 to 255</td>
</tr>
<tr>
<td>short</td>
<td>n/a</td>
<td>16 bits</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td>unsigned short</td>
<td>n/a</td>
<td>16 bits</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td>int</td>
<td>4-Byte Integers is off in the ColdFire Processor panel</td>
<td>16 bits</td>
<td>-32,768 to 32,767</td>
</tr>
<tr>
<td></td>
<td>4-Byte Integers is on in the ColdFire Processor panel</td>
<td>32 bits</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned int</td>
<td>4-Byte Integers is off in the ColdFire Processor panel</td>
<td>16 bits</td>
<td>0 to 65,535</td>
</tr>
<tr>
<td></td>
<td>4-Byte Integers is on in the ColdFire Processor panel</td>
<td>32 bits</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>long</td>
<td>n/a</td>
<td>32 bits</td>
<td>-2,147,483,648 to 2,147,483,647</td>
</tr>
<tr>
<td>unsigned long</td>
<td>n/a</td>
<td>32 bits</td>
<td>0 to 4,294,967,295</td>
</tr>
<tr>
<td>long long</td>
<td>n/a</td>
<td>64 bits</td>
<td>-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807</td>
</tr>
<tr>
<td>unsigned long long</td>
<td>n/a</td>
<td>64 bits</td>
<td>0 to 18,446,744,073,709,551,615</td>
</tr>
</tbody>
</table>
Calling Conventions

For ColdFire development, the calling conventions are:

- **Standard** — the compiler uses the default amount of memory, expanding everything to int size.
- **Compact** — the compiler tries to minimize memory consumption.
- **Register** — the compiler tries to use memory registers, instead of the stack.

**NOTE** The corresponding levels for the supported calling conventions are `standard_abi` (the default), `compact_abi`, and `register_abi`.

The compiler passes parameters on the stack in reverse order. It passes the return value in different locations, depending on the nature of the value and compiler settings:

- Integer return value: register D0.
- Pointer return value: register A0.
- Any other return value: temporary storage area. (For any non-integer, non-pointer return type, the calling routine reserves this area in its stack. The calling routine passes a pointer to this area as its last argument. The called function returns its value in this temporary storage area.)

To have the compiler return pointer values in register D0, use the pragma `pointers_in_D0`, which the *C Compiler reference guide* explains.

To reset pointer returns, use the pragma `pointers_in_A0`.

**NOTE** If you use the pragma `pointers_in_A0`, be sure to use correct prototypes. Otherwise, the pragma may not perform reliably.

*Figure 19.1* depicts the stack when you use the ColdFire compiler to call a C function.

![Figure 19.1 Calling a C Function: Stack Depiction](image-url)
ColdFire Code Generation
Variable Allocation

Variable Allocation

For a ColdFire target, the compiler lets you declare structs and arrays to be any size, but imposes a few limits on how you allocate their space:

- Maximum bitfield size is 32 bits.
- There is no limit to local-variable space for a function. However, access is twice as fast for frames that do not exceed 32 kilobytes. To keep within this limit,
  - Dynamically allocate large variables, or
  - Declare large variables to be `static` (provided that this does not exceed the 32-kilobyte limit on global variables).
- Maximum declaration size for a global variable is 32 kilobytes, unless you use `far` data. You must do one of the following:
  - Dynamically allocate the variable.
  - Use the `far` qualifier when declaring the variable.
  - Select the `Far (32 bit)` option from the `Code and Data model` in the ColdFire Processor settings panel.

Listing 19.1 shows how to declare a large `struct` or `array`. Keep in mind that declaring large static arrays works only if the device has enough physical memory.

Listing 19.1 Declaring a large structure

```c
int i[50000]; // Wrong with ColdFire compiler and the Far Data // option in the Processor settings panel is off
far int j[50000]; // ALWAYS OK.
int *k;
k = malloc(50000 * sizeof(int)); // ALWAYS OK.
```

Register Variables

The ColdFire back-end compiler automatically allocates local variables and parameters to registers, according to frequently of use and how many registers are available.

The ColdFire compiler can use these registers for local variables:

- A2 through A5 — for pointers
- D3 through D7 — for integers and pointers.
- FP3 through FP7 — for 64-bit floating-point numbers (provided that you select `Hardware` in the Floating Point list box of the ColdFire Processor panel).

If you optimize for speed, the compiler gives preference to variables in loops.
ColdFire Code Generation

Position-Independent Code

The ColdFire back-end compiler gives preference to variables declared `register`, but does not automatically assign them to registers. For example, if the compiler must choose between an inner-loop variable and a variable declared `register`, the compiler places the inner-loop variable in the register.

Position-Independent Code

If you specify position-independent code, the compiler generates code that is the same regardless of its load address. Different processes of your application can share such code.

Listing 19.2 Position Independent Code

```c
int relocatableAlpha();
int (*alpha)()=relocatableAlpha;
```

Follow these steps to enable the PIC compiler and runtime support:

1. Add a `.picdynrel` section to the linker command file.
2. Enable PIC generation in the processor settings panel.
3. Customize and recompile the runtime to support your loading routine.

Cryptographic Acceleration Instructions

MCU52235 and related ColdFire-family processors have a cryptography acceleration unit (CAU). This instruction-level coprocessor speeds up software-based encryption/decryption. The CAU enhances these actions for the DES, 3DES, AES, MD5, and SHA-1 encryption algorithms.

Table 19.2 contrasts megabyte-per-second performance of regular software and the CAU for several encryption algorithms, noting the CAU improvement.

Table 19.2 CAU Performance Improvement

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Software</th>
<th>CAU</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>DES, 3DES</td>
<td>2</td>
<td>82</td>
<td>41 times</td>
</tr>
<tr>
<td>AES-128</td>
<td>9</td>
<td>99</td>
<td>11 times</td>
</tr>
<tr>
<td>MD5</td>
<td>47</td>
<td>118</td>
<td>2.5 times</td>
</tr>
<tr>
<td>SHA-1</td>
<td>22</td>
<td>55</td>
<td>2.5 times</td>
</tr>
</tbody>
</table>
ColdFire Code Generation
Cryptographic Acceleration Instructions

To access the CAU, you use generic instructions that identify the CAU coprocessor and include appropriate CAU commands.

**NOTE** Syntax (prototypes) of this text is correct if the CAU is coprocessor 0. For an implementation that includes the CAU as coprocessor 1, you would have to substitute 1 for 0 in the instructions.

Table 19.3 lists the CAU instructions.

Table 19.3 ColdFire CAU Commands

<table>
<thead>
<tr>
<th>This instruction...</th>
<th>performs this operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADR</td>
<td>Add to register</td>
</tr>
<tr>
<td>ADRA</td>
<td>Add register to accumulator</td>
</tr>
<tr>
<td>AESC</td>
<td>AES column operation</td>
</tr>
<tr>
<td>AESIC</td>
<td>Inverse AES column operation</td>
</tr>
<tr>
<td>AESIR</td>
<td>Inverse AES shift rows</td>
</tr>
<tr>
<td>AESIS</td>
<td>Inverse AES substitution</td>
</tr>
<tr>
<td>AESR</td>
<td>AES shift rows</td>
</tr>
<tr>
<td>AESS</td>
<td>AES substitution</td>
</tr>
<tr>
<td>CNOP</td>
<td>Coprocessor no operation</td>
</tr>
<tr>
<td>DESK</td>
<td>DES key setup</td>
</tr>
<tr>
<td>DESR</td>
<td>DES round</td>
</tr>
<tr>
<td>HASH</td>
<td>Hash function</td>
</tr>
<tr>
<td>ILL</td>
<td>Illegal command</td>
</tr>
<tr>
<td>LDR</td>
<td>Load register</td>
</tr>
<tr>
<td>MDS</td>
<td>Message digest shift</td>
</tr>
<tr>
<td>MVAR</td>
<td>Move accumulator to register</td>
</tr>
<tr>
<td>MVRA</td>
<td>Move register to accumulator</td>
</tr>
<tr>
<td>RADR</td>
<td>Reverse and add to register</td>
</tr>
<tr>
<td>ROTL</td>
<td>Rotate left</td>
</tr>
</tbody>
</table>
### Table 19.3 ColdFire CAU Commands (continued)

<table>
<thead>
<tr>
<th>This instruction...</th>
<th>performs this operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>SHS</td>
<td>Secure hash shift</td>
</tr>
<tr>
<td>STR</td>
<td>Store register</td>
</tr>
<tr>
<td>XOR</td>
<td>Exclusive or</td>
</tr>
</tbody>
</table>
Cryptographic Acceleration Instructions
ColdFire Runtime Libraries

The CodeWarrior tool chain includes libraries conforming to ISO/IEC-standards for C and C++, runtime libraries, and other code. CodeWarrior tools come with prebuilt configurations of these libraries with variants for:

- integer size
- hardware floating-point operations
- different applications binary interfaces (ABIs)
- UART control
- console input/output support

This chapter explains how to use prebuilt libraries, and how to create reduced working-set libraries. This chapter consists of these sections:

- MSL for ColdFire Development
- Runtime Libraries

NOTE With respect to the Main Standard Libraries (MSL) for C and C++, this chapter is an extension of the MSL C Reference and the MSL C++ Reference. Consult those manuals for general information.

MSL for ColdFire Development

The Main Standard Library provides the libraries described in the ISO/IEC standards for C and C++. MSL also provides some extensions to the standard libraries.

- Customizing MSL Libraries
- Using MSL for ColdFire
- Serial I/O and UART Libraries
- Reduced Working Set Libraries
- Memory, Heaps, and Other Libraries
Customizing MSL Libraries

Full compliance with the ISO/IEC standards can increase code size — a problem if you must run an application in a small memory, or if you require more efficient memory usage. In such a case, you can discard library files whose functionality you do not need.

In addition to compiled binaries, the CodeWarrior development tools include source code and project files for MSL so that you can customize the libraries.

NOTE The MCF52235 and related processors have smaller memories than many other members of the ColdFire family. Accordingly, C and C++ libraries include special, small library files, appropriate for such limited-memory devices. The names of these small library files include the designation $SZ$.

Using MSL for ColdFire

Your CodeWarrior installation includes the Main Standard Libraries (MSL), a complete C and C++ library that you can use in your embedded projects. The installation includes all the source files necessary to build MSL as well as project files for different MSL configurations.

NOTE If an MSL version already is on your computer, the CodeWarrior installer installs only the additional MSL files necessary for ColdFire projects.

The names of library files follow this pattern, which Table 20.1 explains:

Table 20.1 MSL Library Name Parameters

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>Language</td>
<td>C_</td>
<td>C language</td>
</tr>
<tr>
<td></td>
<td>C++_</td>
<td>C++ language</td>
</tr>
<tr>
<td>IO</td>
<td>TRK_</td>
<td>Console IO</td>
</tr>
<tr>
<td>Int_size</td>
<td>2i_</td>
<td>Code generation with 2-byte integers</td>
</tr>
<tr>
<td></td>
<td>4i_</td>
<td>Code generation with 4-byte integers</td>
</tr>
<tr>
<td>CF_</td>
<td>CF_</td>
<td>Code generation for a ColdFire target processor</td>
</tr>
<tr>
<td>FPU</td>
<td>FPU_</td>
<td>Floating-point support</td>
</tr>
</tbody>
</table>
ColdFire Runtime Libraries

MSL for ColdFire Development

Table 20.1  MSL Library Name Parameters (continued)

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Specifies</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABI</td>
<td>(nothing)</td>
<td>Code generation with the compact ABI</td>
</tr>
<tr>
<td>RegABI_</td>
<td></td>
<td>Code generation with the register ABI</td>
</tr>
<tr>
<td>StdABI_</td>
<td></td>
<td>Code generation with the standard ABI</td>
</tr>
<tr>
<td>Position</td>
<td>PI_</td>
<td>Code generation with position-independent code and data</td>
</tr>
<tr>
<td>Size</td>
<td>SZ_</td>
<td>Small libraries working set</td>
</tr>
<tr>
<td>MSL.a</td>
<td>MSL.a</td>
<td>Library name constant (do not change)</td>
</tr>
</tbody>
</table>

For example, the name C_4i_CF_MSL.a is the fully compliant, standard C library using 4-byte integers and the compact ABI.

Another example is C++_4i_CF_RegABI_PI_SZ_MSL.a — the reduced working-set C++ library using 4-byte integers, the register ABI, and position-independent code and data.

NOTE
1. As C++ libraries are built over C libraries, a C++ application almost always requires a C library for linking.
2. As C++ relies on low-level C functionality for IO, TRK_ is not part of any C++-library file names.

The factory configuration for all libraries uses:
- far code and data models
- no .sdata section
- no PC-relative strings
- no A6 frames (except for C++ exception handling)
- full optimization with emphasis on reducing code size

Fully compliant, non-FPU libraries use ISA_A instructions; FPU libraries use ISA_B instructions. Library code does not depend on the MAC or EMAC. The startup code (E68k_startup.c) sets the initial values of the SR, A7, and A5 registers. Otherwise, the libraries do not manipulate system registers.

The C and C++ libraries include special, small library files, appropriate for use with MCF52235 and related processors, which have smaller memories than many other members of the ColdFire family. These files, which use ISA_A instructions, include the designation SZ_ in their names.
ColdFire Runtime Libraries

MSL for ColdFire Development

Table 20.2 lists the MSL C libraries — which are in subdirectory \E68K_Support\msl\MSL_C\MSL_E68k\Lib of your CodeWarrior installation directory.

Table 20.3 lists the MSL C++ libraries — which are in subdirectory \E68K_Support\msl\MSL_C++\MSL_E68k\Lib of your CodeWarrior installation directory.

Table 20.4 lists MSL EC++ libraries — which are in subdirectory \E68K_Support\msl\(MSL_EC++\)\MSL_E68k\Lib of your CodeWarrior installation directory.

Table 20.2  C Libraries

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully Compliant UART IO</td>
<td>C_2i_CF_MSL.a</td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_PI_MSL.a</td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_RegABI_MSL.a</td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_RegABI_PI_MSL.a</td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_StdABI_MSL.a</td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_StdABI_PI_MSL.a</td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_MSL.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_PI_MSL.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_RegABI_MSL.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_RegABI_PI_MSL.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_StdABI_MSL.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_StdABI_PI_MSL.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>

BuildToolsRefColdFire.book  Page 208  Tuesday, July 11, 2006  12:54 PM


## Table 20.2 C Libraries (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully Compliant</td>
<td>C_TRK_2i_CF_MSL.a</td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td>Console IO</td>
<td>C_TRK_2i_CF_PI_MSL.a</td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_TRK_2i_CF_RegABI_MSL.a</td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_TRK_2i_CF_RegABI.PI_MSL.a</td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_TRK_2i_CF_StdABI_MSL.a</td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_TRK_2i_CF_StdABI.PI_MSL.a</td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_MSL.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_PI_MSL.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_RegABI_MSL.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_RegABI.PI_MSL.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_StdABI_MSL.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_StdABI.PI_MSL.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
### Table 20.2 C Libraries (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small UART IO</td>
<td>C_2i_CF_SZ_MSL.a</td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_PI_SZ_MSL.a</td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_RegABI_SZ_MSL.a</td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_RegABI_PI_SZ_MSL.a</td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_StdABI_SZ_MSL.a</td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_StdABI_PI_SZ_MSL.a</td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_SZ_MSL.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_PI_SZ_MSL.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_RegABI_SZ_MSL.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_RegABI_PI_SZ_MSL.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_StdABI_SZ_MSL.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_StdABI_PI_SZ_MSL.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
### Table 20.2 C Libraries (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Small Console IO</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_2i_CF_SZ_MSL.a</code></td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_2i_CF_PI_SZ_MSL.a</code></td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_2i_CF_RegABI_SZ_MSL.a</code></td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_2i_CF_RegABI_PI_SZ_MSL.a</code></td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_2i_CF_StdABI_SZ_MSL.a</code></td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_2i_CF_StdABI_PI_SZ_MSL.a</code></td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_4i_CF_SZ_MSL.a</code></td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_4i_CF_PI_SZ_MSL.a</code></td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_4i_CF_RegABI_SZ_MSL.a</code></td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_4i_CF_RegABI_PI_SZ_MSL.a</code></td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_4i_CF_StdABI_SZ_MSL.a</code></td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td><code>C_TRK_4i_CF_StdABI_PI_SZ_MSL.a</code></td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td><strong>HW Floating-Point UART IO</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td><code>C_4i_CF_FPU_MSL.a</code></td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td><code>C_4i_CF_FPU_PI_MSL.a</code></td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C_4i_CF_FPU_RegABI_MSL.a</code></td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td><code>C_4i_CF_FPU_RegABI_PI_MSL.a</code></td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C_4i_CF_FPU_StdABI_MSL.a</code></td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td><code>C_4i_CF_FPU_StdABI_PI_MSL.a</code></td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
# ColdFire Runtime Libraries

*MSL for ColdFire Development*

## Table 20.2 C Libraries (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW Floating-Point Console IO</td>
<td>C_TRK_4i_CF_FPU_MSL.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_FPU_PI_MSL.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_FPU_RegABI_MSL.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_FPU_RegABI_PI_MSL.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_FPU_StdABI_MSL.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_TRK_4i_CF_FPU_StdABI_PI_MSL.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
## Table 20.3 C++ Libraries

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fully Compliant</td>
<td><code>C++_2i_CF_MSL.a</code></td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td><code>C++_2i_CF_PI_MSL.a</code></td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C++_2i_CF_RegABI_MSL.a</code></td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td><code>C++_2i_CF_RegABI_PI_MSL.a</code></td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C++_2i_CF_StdABI_MSL.a</code></td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td><code>C++_2i_CF_StdABI_PI_MSL.a</code></td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C++_4i_CF_MSL.a</code></td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td><code>C++_4i_CF_PI_MSL.a</code></td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C++_4i_CF_RegABI_MSL.a</code></td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td><code>C++_4i_CF_RegABI_PI_MSL.a</code></td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td><code>C++_4i_CF_StdABI_MSL.a</code></td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td><code>C++_4i_CF_StdABI_PI_MSL.a</code></td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
ColdFire Runtime Libraries

MSL for ColdFire Development

Table 20.3 C++ Libraries (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small Working Set</td>
<td>C++_2i_CF_SZ_MSL.a</td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C++_2i_CF_PI_SZ_MSL.a</td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C++_2i_CF_RegABI_SZ_MSL.a</td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C++_2i_CF_RegABI_PI_SZ_MSL.a</td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C++_2i_CF_StdABI_SZ_MSL.a</td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C++_2i_CF_StdABI_PI_SZ_MSL.a</td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_SZ_MSL.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_PI_SZ_MSL.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_RegABI_SZ_MSL.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_RegABI_PI_SZ_MSL.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_StdABI_SZ_MSL.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_StdABI_PI_SZ_MSL.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td>HW Floating-Point UART IO</td>
<td>C_4i_CF_FPU_SZ_MSL.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_PI_SZ_MSL.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_RegABI_SZ_MSL.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_RegABI_PI_SZ_MSL.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_StdABI_SZ_MSL.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_StdABI_PI_SZ_MSL.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
## ColdFire Runtime Libraries

### MSL for ColdFire Development

### Table 20.3 C++ Libraries (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW Floating-Point</td>
<td>C++_4i_CF_FPU_MSL.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_FPU_PI_MSL.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_FPU_RegABI_MSL.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_FPU_RegABI_PI_MSL.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_FPU_StdABI_MSL.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C++_4i_CF_FPU_StdABI_PI_MSL.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
## ColdFire Runtime Libraries

*MSL for ColdFire Development*

### Table 20.4 EC++ Libraries

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Fully Compliant</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>EC++-2i_CF_MSL.a</td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>EC++-2i_CF_PI_MSL.a</td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>EC++-2i_CF_RegABI_MSL.a</td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>EC++-2i_CF_RegABI_PI_MSL.a</td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>EC++-2i_CF_StdABI_MSL.a</td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>EC++-2i_CF_StdABI_PI_MSL.a</td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>EC++-4i_CF_MSL.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>EC++-4i_CF_PI_MSL.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>EC++-4i_CF_RegABI_MSL.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>EC++-4i_CF_RegABI_PI_MSL.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>EC++-4i_CF_StdABI_MSL.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>EC++-4i_CF_StdABI_PI_MSL.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td><strong>Hardware floating point, UART input/output</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_MSL.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_PI_MSL.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_RegABI_MSL.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_RegABI_PI_MSL.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_StdABI_MSL.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_StdABI_PI_MSL.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
The ColdFire Metrowerks Standard Libraries support console I/O through the serial port. This support includes:

- Standard C-library I/O.
- All functions that do not require disk I/O.
- Memory functions malloc() and free().

To use C or C++ libraries for console I/O, you must include a special serial UART driver library in your project. These driver library files are in folder E68K_Tools\MetroTRK\Transport\m68k\.

Table 20.5 lists target boards and corresponding UART library files.

### Table 20.5  Serial I/O UART Libraries

<table>
<thead>
<tr>
<th>Board</th>
<th>Filename</th>
</tr>
</thead>
<tbody>
<tr>
<td>CF5206e SBC</td>
<td>mot_sbc_5206e_serial\Bin\UART_SBC_5206e_Aux.a</td>
</tr>
<tr>
<td>CF5206e LITE</td>
<td>mot_5206e_lite_serial\Bin\UART_5206e_lite_Aux.a</td>
</tr>
<tr>
<td>CF5307 SBC</td>
<td>mot_sbc_5307_serial\Bin\UART_SBC_5307_Aux.a</td>
</tr>
<tr>
<td>CF5407 SBC</td>
<td>mot_sbc_5407_serial\Bin\UART_SBC_5407_Aux.a</td>
</tr>
<tr>
<td>CF5249 SBC</td>
<td>mot_sbc_5249_serial\Bin\UART_SBC_5249_Aux.a</td>
</tr>
</tbody>
</table>
ColdFire Runtime Libraries
MSL for ColdFire Development

**Reduced Working Set Libraries**

Before the current (6.3) software release, specialists defined a reduced-functionality set of files, to reduce library size. This idea became particularly appropriate for MCF52235 and related processors, which have smaller memories than other ColdFire-family devices. When you specify such a target processor — either by selecting it in a settings panel or using it as a -proc command-line option — CodeWarrior software automatically specifies this library-file working set as the default.

However, you can control this library specification at the topmost declaration level, such as the preprocessor settings panel or a prefix file. To do so, define __CF_USE_FULL_LIBS or __CF_USE_SMALL_LIBS. This specification affects available declarations, so you will see its effects at compilation time.

As the reduced working set is a proper subset of the fully compliant library, using the full working set in declarations, but specifying small library files causes:

- Link errors for completely removed functions, or
- Reduced functionality, such as printf inability to display floating-point values.

Using the reduced working set in declarations, but specifying fully compliant library files bloats your code.

Table 20.7 summarizes guidance for specifying libraries.

<table>
<thead>
<tr>
<th>Base</th>
<th>Processor/Define</th>
<th>Library Set</th>
</tr>
</thead>
<tbody>
<tr>
<td>Processor</td>
<td>MCF5213, MCF5223x, MCF5222x</td>
<td>Reduced working set (SZ_ in name)</td>
</tr>
<tr>
<td></td>
<td>Other ColdFire</td>
<td>Full compliance set (no SZ_ in name)</td>
</tr>
<tr>
<td>Macro</td>
<td>#define __CF_USE_SMALL_LIBS</td>
<td>Reduced working set (SZ_ in name)</td>
</tr>
<tr>
<td></td>
<td>#define __CF_USE_FULL_LIBS</td>
<td>Full compliance set (no SZ_ in name)</td>
</tr>
</tbody>
</table>

Beyond affecting code size, your library-set specification sets or clears certain configuration flags. In turn, this affects certain functionality. Table 20.7 explains these effects:

- For fully compliant libraries
- For reduced functionality libraries before the 6.3 release
- For reduced functionality libraries beginning with the 6.3 release
Memory, Heaps, and Other Libraries

The heap you create in your linker command file becomes the default heap, so it does not need initialization. Additional memory and heap points are:

- To have the system link memory-management code into your code, call malloc() or new().
- Initialize multiple memory pools to form a large heap.
- To create each memory pool, call init_alloc(). (You do not need to initialize the memory pool for the default heap.)

You may be able to use another standard C library with CodeWarrior projects. You should check the stdarg.h file in this other standard library and in your runtime libraries. Additional points are:

- The CodeWarrior ColdFire C/C++ compiler generates correct variable-argument functions only with the header file that the MSL include.

Table 20.7 Configuration-Flag Functionality

<table>
<thead>
<tr>
<th>Flag</th>
<th>Full</th>
<th>&lt;6.3</th>
<th>6.3+</th>
<th>Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>_MSL_THREADSAFE</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>no threads on bareboard</td>
</tr>
<tr>
<td>_MSL_C_LOCALE_ONLY</td>
<td>on</td>
<td>on</td>
<td>on</td>
<td>default, smallest size</td>
</tr>
<tr>
<td>_MSL_CURATE_BUT_LARGE_ANSI_FP</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>default</td>
</tr>
<tr>
<td>_MSL_STRERROR_KNOWS_ERROR_NAMES</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>default</td>
</tr>
<tr>
<td>_MSL_ASSERT_DISPLAYS_FUNC</td>
<td>off</td>
<td>off</td>
<td>off</td>
<td>default</td>
</tr>
<tr>
<td>_MSL_C99</td>
<td>on</td>
<td>on</td>
<td>off</td>
<td>C99 standard compliance</td>
</tr>
<tr>
<td>_MSL_LONGLONG</td>
<td>on</td>
<td>on</td>
<td>off</td>
<td>int longlong support</td>
</tr>
<tr>
<td>_MSL_WIDE_CHAR</td>
<td>on</td>
<td>on</td>
<td>off</td>
<td>multi-byte char support</td>
</tr>
<tr>
<td>_MSL_FLOATING_POINT</td>
<td>on</td>
<td>on</td>
<td>off</td>
<td>floating point operations</td>
</tr>
<tr>
<td>_MSL_FLOATING_POINT_IO</td>
<td>on</td>
<td>off</td>
<td>off</td>
<td>printf knows floating point</td>
</tr>
<tr>
<td>_MSL_NO_WCHART_C_SUPPORT</td>
<td>off</td>
<td>off</td>
<td>on</td>
<td>C multi-byte char support</td>
</tr>
<tr>
<td>_MSL_NO_WCHART_CPP_SUPPORT</td>
<td>off</td>
<td>off</td>
<td>on</td>
<td>C++ multi-byte char support</td>
</tr>
<tr>
<td>_MSL_NO_MATH_LIB</td>
<td>off</td>
<td>off</td>
<td>on</td>
<td>floating point operations</td>
</tr>
<tr>
<td>_MSL_NO_CONDITION</td>
<td>off</td>
<td>off</td>
<td>on</td>
<td>C++ threading</td>
</tr>
</tbody>
</table>
ColdFire Runtime Libraries

Runtime Libraries

- You may find that other implementations are also compatible.
- You may also need to modify the runtime to support a different standard C library; you must include __va_arg.c.
- Other C++ libraries are not compatible.

**NOTE** If you are working with any kind of embedded OS, you may need to customize MSL to work properly with that OS.

Runtime Libraries

Every ColdFire project must include a runtime library, which provides basic runtime support, basic initialization, system startup, and the jump to the main routine. RAM-based debug is the primary reason behind runtime-library development for ColdFire boards, so you probably must modify a library for your application.

Find your setup in Table 20.8, then include the appropriate runtime library file:

- For a C project, use the file that starts with C_.
- For a C++ project, use the file that starts with Cpp_.
- All these files are in folder \E68K_Support\Runtime\{Sources\}. 
<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>C_2i_CF_Runtime.a</td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_PI_Runtime.a</td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_RegABI_Runtime.a</td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_RegABI_PI_Runtime.a</td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_StdABI_Runtime.a</td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_StdABI_PI_Runtime.a</td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_Runtime.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_PI_Runtime.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_RegABI_Runtime.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_RegABI_PI_Runtime.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_StdABI_Runtime.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_StdABI_PI_Runtime.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
### ColdFire Runtime Libraries

#### Runtime Libraries

**Table 20.8  Runtime Libraries (continued)**

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C Floating Point</td>
<td>C_2i_CF_FPU_SZ_Runtime.a</td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_FPU_PI_SZ_Runtime.a</td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_FPU_RegABI_SZ_Runtime.a</td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_FPU_RegABI_PI_SZ_Runtime.a</td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_FPU_StdABI_SZ_Runtime.a</td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_2i_CF_FPU_StdABI_PI_SZ_Runtime.a</td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_SZ_Runtime.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_PI_SZ_Runtime.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_RegABI_SZ_Runtime.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_RegABI_PI_SZ_Runtime.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_StdABI_SZ_Runtime.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>C_4i_CF_FPU_StdABI_PI_SZ_Runtime.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
Table 20.8  Runtime Libraries (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++, EC++</td>
<td>Cpp_2i_CF_Runtime.a</td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_PI_Runtime.a</td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_RegABI_Runtime.a</td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_RegABI_PI_Runtime.a</td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_StdABI_Runtime.a</td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_StdABI_PI_Runtime.a</td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_Runtime.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_PI_Runtime.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_RegABI_Runtime.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_RegABI_PI_Runtime.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_StdABI_Runtime.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_StdABI_PI_Runtime.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>
ColdFire Runtime Libraries

Runtime Libraries

Table 20.8 Runtime Libraries (continued)

<table>
<thead>
<tr>
<th>Category</th>
<th>Library File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>C++, EC++ Floating-Point</td>
<td>Cpp_2i_CF_FPU_SZ_Runtime.a</td>
<td>2-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_FPU_PI_SZ_Runtime.a</td>
<td>2-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_FPU_RegABI_SZ_Runtime.a</td>
<td>2-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_FPU_RegABI_PI_SZ_Runtime.a</td>
<td>2-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_FPU_StdABI_SZ_Runtime.a</td>
<td>2-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_2i_CF_FPU_StdABI_PI_SZ_Runtime.a</td>
<td>2-byte integers, standard ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_FPU_SZ_Runtime.a</td>
<td>4-byte integers, compact ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_FPU_PI_SZ_Runtime.a</td>
<td>4-byte integers, compact ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_FPU_RegABI_SZ_Runtime.a</td>
<td>4-byte integers, register ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_FPU_RegABI_PI_SZ_Runtime.a</td>
<td>4-byte integers, register ABI, position-independent</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_FPU_StdABI_SZ_Runtime.a</td>
<td>4-byte integers, standard ABI</td>
</tr>
<tr>
<td></td>
<td>Cpp_4i_CF_FPU_StdABI_PI_SZ_Runtime.a</td>
<td>4-byte integers, standard ABI, position-independent</td>
</tr>
</tbody>
</table>

NOTE ABI corresponds directly to the parameter-passing setting of the ColdFire Processor Settings panel (Standard, Compact or Register).
If your target supports floating points, you should use an FPU-enabled runtime library file.

Position-Independent Code

To use position-independent code or position-independent data in your program, you must customize the runtime library. Follow these steps:
1. Load project file MSL_RuntimeCF.mcp, from the folder \E68K_Support\runtime.

2. Modify runtime functions.
   a. Open file E68K_startup.c.
   b. As appropriate for your application, change or remove runtime function __block_copy_section. (This function relocates the PIC/PID sections in the absence of an operating system.)
   c. As appropriate for your application, change or remove runtime function __fix_addr_references. (This function creates the relocation tables.)

3. Change the prefix file.
   a. Open the C/C++ preference panel for your target.
   b. Make sure this panel specifies prefix file PICPIDRuntimePrefix.h.

4. Recompile the runtime library for your target.

Once you complete this procedure, you are ready to use the modified runtime library in your PIC/PID project. Source-file comments and runtime-library release notes may provide additional information.

Board Initialization Code

Your CodeWarrior development tools come with several basic, assembly-language hardware initialization routines, which may be useful in your programs.

You need not include this code when you are debugging, as the debugger or debug kernel already performs the same board initialization.

You should have your program do as much initialization as possible, minimizing the initializations that the configuration file performs. This facilitates the transition from RAM-based debugging to Flash/ROM.

Custom Modifications

As text above shows, specific library files support specific functionality. If target-device memory is particularly small, you may need to delete library files for functionality that your application does not use. Follow this guidance:

- **Configuration settings** — Change them in projects or makefiles. Generally, modifying flags from configuration header ansi_prefix.CF.size.h is sufficient to modify the working set. Sometimes, however, you also must modify header ansi_prefix.e68k.h.

- **Projects** — The easiest way to create a new project is starting from a copy of a full-compliance project. Turning off such flags as floating point forces you to remove some files from the project file list. But this is appropriate, as your project will not
need those files. Although changing the basic configuration can require editing all
targets of all project files, usually modifying the single targets your application uses
is sufficient.

- **Makefiles** — Makefile targets already are set up to build any library; the CFLAGS
  macro defines the basic configuration. Target all does not include all targets, but a
  commented variation of all these targets is present in every makefile.
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 lists the predefined symbols that all CodeWarrior compilers make available.

__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.

__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 a three-letter abbreviation of the month, dd is the day of the month, and yyyy is the year.
Predefined Symbols

__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.

__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.

__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.
Predefined Symbols

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.

__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.

__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. Expands to 0 otherwise. The ISO standards do not specify this symbol.
Predefined Symbols

____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.

____MWERKS____

Preprocessor macro defined as the version of the CodeWarrior compiler.

Syntax

____MWERKS____

Remarks

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.

____PRETTY_FUNCTION____

Predefined variable containing a character string of the “unmangled” name of the C++ function being compiled.
Predefined Symbols

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 14882-1998 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.

__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. Undefined otherwise. The ISO standards does not specify this symbol.

__STDC__

Defined as 1 when compiling ISO/IEC Standard C source code, undefined otherwise.

Syntax

__STDC__
Predefined Symbols

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. The compiler does not define this macro
otherwise.

__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.
ColdFire 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 lists the predefined symbols made available by the CodeWarrior compiler for ColdFire processors.

__BACKENDVERSION__

Preprocessor macro defined to describe the version of CodeWarrior compiler’s back-end.

**Syntax**

```c
#define __BACKENDVERSION__ version
```

**Remarks**

The compiler defines this macro to be a character string literal containing a numeric value.

__COLDFIRST__

Preprocessor macro defined to describe the target ColdFire processor.

**Syntax**

```c
#define __COLDFIRST__ processor_code
```

**Remarks**

The compiler defines this macro to describe the ColdFire processor that the compiler is generating object code for. Table 22.1 lists the ColdFire processors that each value of `processor_code` represents.
ColdFire Predefined Symbols

Table 22.1 ColdFire processor models and compiler codes

<table>
<thead>
<tr>
<th>ColdFire processor...</th>
<th>then the compiler defines <strong>COLDFIRE</strong> to this value</th>
</tr>
</thead>
<tbody>
<tr>
<td>MCF5206E</td>
<td>0x206e</td>
</tr>
<tr>
<td>MCF5208</td>
<td>0x2008</td>
</tr>
<tr>
<td>MCF521X</td>
<td>0x2013</td>
</tr>
<tr>
<td>MCF5222X</td>
<td>0x2022</td>
</tr>
<tr>
<td>MCF5223X</td>
<td>0x2023</td>
</tr>
<tr>
<td>MCF5249</td>
<td>0x2049</td>
</tr>
<tr>
<td>MCF5270, MCF5271, MCF5274, MCF5275</td>
<td>0x2008</td>
</tr>
<tr>
<td>MCF5272</td>
<td>0x2072</td>
</tr>
<tr>
<td>MCF5280, MCF52801, MCF52802</td>
<td>0x2082</td>
</tr>
<tr>
<td>MCF5307</td>
<td>0x3070</td>
</tr>
<tr>
<td>MCF532X</td>
<td>0x3020</td>
</tr>
<tr>
<td>MCF5407</td>
<td>0x4070</td>
</tr>
<tr>
<td>MCF547X</td>
<td>0x4080</td>
</tr>
<tr>
<td>MCF548X</td>
<td>0x4080</td>
</tr>
</tbody>
</table>

__STDABI__

Preprocessor macro defined to describe the compiler’s parameter-passing setting.

Syntax

```c
#define __STDABI__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the compiler is set to use standard parameter-passing code generation, 0 otherwise.
ColdFire Predefined Symbols

__REGABI__

Preprocessor macro defined to describe the compiler’s parameter-passing setting.

Syntax

#define __REGABI__ 0 | 1

Remarks

The compiler defines this macro to be 1 if the compiler is set to use register-based parameter-passing code generation, 0 otherwise.
ColdFire Predefined Symbols
Using Pragmas

The #pragma preprocessor directive specifies option settings to the compiler to control the compiler and linker’s code generation.

- CheckingPragmaSettings
- Saving and RestoringPragmaSettings
- Determining WhichSettings AreSaved and Restored
- IllegalPragmas

CheckingPragmaSettings

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.

Listing 23.1 shows an example.

Listing 23.1 Using the __option() preprocessor function

```c
#if __option(ANSI_strict)
#include "portable.h" /* Use the portable declarations. */
#else
#include "custom.h" /* Use the specialized declarations. */
#endif
```

BuildToolsRefColdFire.book Page 237 Tuesday, July 11, 2006 12:54 PM
Saving and RestoringPragma 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. Listing 23.2 shows an example.

Listing 23.2  Using `push` and `pop` to save and restore pragma settings

```c
/* 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)
{
    /* ... */
}
```
Using Pragmas

Determining Which Settings Are Saved and Restored

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. Listing 23.3 shows an example.

Listing 23.3 Using the reset option to save and restore a pragma setting

```c
/* 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)
{
    /* ... */
}
```

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.

Listing 23.4 shows an example that checks if the ANSI_strict pragma setting is saved and restored by pragmas push and pop.

Listing 23.4 Testing if pragmas push and pop save and restore a setting

```c
/* Preprocess this source code. */
#pragma ANSI_strict on
```
Using Pragmas

Illegal Pragmas

```c
#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
```

Illegal Pragmas

If you enable the illegal pragmas setting, the compiler issues a warning when it encounters a pragma it does not recognize. For example, the pragma statements in Listing 23.5 generate warnings with the illegal pragmas setting enabled.

Listing 23.5  Illegal Pragmas

```c
#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
```

Table 23.1 shows how to control the recognition of illegal pragmas.

Table 23.1  Controlling illegal pragmas

<table>
<thead>
<tr>
<th>To control this option from here...</th>
<th>use this setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>CodeWarrior IDE</td>
<td>Illegal Pragmas in the C/C++ Warnings panel</td>
</tr>
<tr>
<td>source code</td>
<td>#pragma warn_illpragma</td>
</tr>
<tr>
<td>command line</td>
<td>-warnings illpragmas</td>
</tr>
</tbody>
</table>

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
Using Pragmas
Pragma Scope
Pragmas for Standard C Conformance

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 (“C89”) standard:

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

This pragma corresponds to the **ANSI Strict** setting in the CodeWarrior IDE’s CodeWarrior IDE’s **C/C++ Language** settings panel. By default, this pragma is **off**.

---

c99

Controls the use of a subset of ISO/IEC 9899-1999 (“C99”) language features.

**Syntax**

```
#pragma c99 on | off | reset
```
Pragmas for Standard C Conformance

Remarks

If you enable this pragma, the compiler accepts many of the language features described by the ISO/IEC 9899-1999 standard:

- 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 and _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

This pragma corresponds to the Enable C99 Extensions setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is disabled.
Pragmas for Standard C Conformance

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 Listing 24.1 defines a prototype for a function with an old-style definition.

**Listing 24.1  Mixing Old-style and Prototype Function Declarations**

```c
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.

only_std_keywords

Controls the use of ISO/IEC keywords.

Syntax

```
#pragma only_std_keywords on | off | reset
```
Pragmas for Standard C Conformance

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.
This pragma corresponds to the ANSI Keywords Only setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is disabled.

require_prototypes
Controls whether or not the compiler should expect function prototypes.

Syntax
#pragma require_prototypes on | off | reset

Remarks
This pragma only affects non-static functions.
If you enable this pragma, the compiler generates an error message if 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 Listing 24.2, function main() calls PrintNum() with an integer argument even though PrintNum() takes an argument of type float.

Listing 24.2 Unnoticed Type-mismatch

```c
#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);
}
```

CodeWarrior Build Tools Reference ColdFire™ Architectures Edition
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 intended. Since `PrintNum()` does not have a prototype, the compiler does not know to generate 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 Listing 24.3, 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 24.3 Using a Prototype to Avoid Type-mismatch**

```c
#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.

This pragma corresponds to the **Require Function Prototypes** setting in the CodeWarrior IDE’s **C/C++ Language** settings panel.
Pragmas for Standard C Conformance
Pragmas for C++

access_errors

Controls whether or not to change illegal 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 illegal access to protected or private class members.

This pragma does not correspond to any IDE panel setting. By default, this pragma is on.

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.

arg_dep_lookup

Controls C++ argument-dependent name lookup.
Pragmas for C++

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.

ARM_conform

This pragma is no longer available. Use ARM_scoping instead.

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. Listing 25.1 shows an example.
This pragma corresponds to the Legacy for-scoping setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is off.

Listing 25.1  Example of Using Variables Declared in for Statement

for(int i=1; i<1000; i++) { /* . . . */ }  // OK if ARM_scoping is on, error if ARM_scoping is off.
Pragmas for C++

---

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.

---

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 either the Don’t Inline setting or the `dont_inline` pragma, the compiler ignores the setting of the auto_inline pragma and does not inline any functions.

This pragma corresponds to the Auto-Inline setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is disabled.

---

bool

Determines whether or not `bool`, `true`, and `false` are treated as keywords in C++ source code.

**Syntax**

```
#pragma bool on | off | reset
```
Pragmas for C++

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 14882-1998 C++ Standard. Source code that does not treat bool as a distinct type might not compile properly.

This pragma corresponds to the Enable bool Support setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this setting is on.

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.

This pragma corresponds to the Force C++ Compilation setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is disabled.

cpp_extensions
Controls language extensions to ISO/IEC 14882-1998 C++.
Pragmas for C++

Syntax

#pragma cpp_extensions on | off | reset

Remarks

If you enable this pragma, you can use the following extensions to the ISO/IEC 14882-1998 C++ standard that would otherwise be illegal:

- Anonymous struct & union objects. Listing 25.2 shows an example.

Listing 25.2 Example of Anonymous struct & union Objects

```cpp
#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. Listing 25.3 shows an example.

Listing 25.3 Example of an Unqualified Pointer to a Member Function

```cpp
#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.
  
  This pragma does not correspond to any setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is disabled.

debuginline

Controls whether the compiler emits debugging information for expanded inline function calls.
Pragmas for C++

Syntax

```cpp
#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.

def_inherited

Controls the use of `inherited`.

Syntax

```cpp
#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

```cpp
typedef base inherited;
```

as the first member in classes with a single base class.

**NOTE** The ISO/IEC 14882-1998 C++ standard does not support the `inherited` symbol. Only the CodeWarrior C++ language implements the `inherited` symbol for single inheritance.

This pragma does not correspond to any setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is off.
**defers_codegen**

Obsolete pragma. Replaced by interprocedural analysis options. See "Interprocedural Analysis" on page 167.

**defers_defarg_parsing**

Defers the parsing of default arguments in member functions.

**Syntax**

```cpp
#pragma defers_defarg_parsing on | off
```

**Remarks**

To be accepted as valid, some default expressions with template arguments will require additional parenthesis. For example, Listing 25.4 results in an error message.

**Listing 25.4 Deferring parsing of default arguments**

```cpp
template<typename T,typename U> struct X { T t; U u; };

struct Y {
    void f(X<int,int> = X<int,int>();
};
```

Listing 25.5 does not generate an error message.

**Listing 25.5 Correct default argument deferral**

```cpp
template<typename T,typename U> struct X { T t; U u; };

struct Y {
    void f(X<int,int> = (X<int,int>());
};
```
Pragmas for C++

This pragma does not correspond to any panel setting. By default, this pragma is on.

direct_destruction

This pragma is obsolete. It is no longer available.

direct_to_som

This pragma is obsolete. It is no longer available.

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 "auto_inline" on page 251. If you disable this pragma, the compiler expands all inline function calls, within the limits you set through other inlining-related pragmas.

This pragma corresponds to the Don’t Inline setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is off.

ecplusplus

Controls the use of embedded C++ features.

Syntax

```
#pragma ecplusplus on | off | reset
```
Pragmas for C++

Remarks
If you enable this pragma, the C++ compiler disables the non-EC++ features of ISO/IEC 14882-1998 C++ such as templates, multiple inheritance, and so on.

This pragma corresponds to the **EC++ Compatibility Mode** setting in the CodeWarrior IDE’s **C/C++ Language** settings panel. By default, this pragma is **off**.

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 does not correspond to an option in any IDE settings panel. By default, this pragma is **on**.

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 described in “extended_errorcheck” on page 283.
Pragmas for C++

It also issues a warning message when it encounters a delete operator for a class or structure that has not been defined yet. Listing 25.6 shows an example.

Listing 25.6 Attempting to delete an undefined structure

```c
#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, Listing 25.7 results in a warning message.

Listing 25.7 A non-void function with an empty return statement

```c
int MyInit(void)
{
    int err = GetMyResources();
    if (err != -1)
    {
        err = GetMoreResources();
    }
    return; /* WARNING: empty return statement */
}
```

Listing 25.8 shows how to prevent this warning message.

Listing 25.8 A non-void function with a proper return statement

```c
int MyInit(void)
{
    int err = GetMyResources();
    if (err != -1)
    {
        err = GetMoreResources();
    }
    return err; /* OK */
}
```

This pragma corresponds to the Extended Error Checking setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this setting is off.

Listing 25.8 shows how to prevent this warning message.
Pragmas for C++

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 Listing 25.9 and Listing 25.10.

Listing 25.9 Maximum Complexity of an Inlined Function

// Maximum complexity of an inlined function
#pragma inline_max_size(max)          // default max == 256

Listing 25.10 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.

This pragma corresponds to the Bottom-up setting of the Inline Depth menu in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is disabled.
Pragmas for C++

**inline_bottom_up_once**

Performs a single bottom-up function inlining operation.

**Syntax**

```c
#pragma inline_bottom_up_once on | off | reset
```

**Remarks**

By default, this pragma is **off**.

**inline_depth**

Controls how many passes are used to expand inline function calls.

**Syntax**

```c
#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:

  ```c
  #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.
Pragmas for C++

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.

Remarks

The pragmas dont_inline and always_inline override this pragma. This pragma corresponds to the Inline Depth setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is disabled.

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.

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 )
Pragmas for C++

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.

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.

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.
Pragmas for C++

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.

Listing 25.11 shows an example:

Listing 25.11  Example of an Internalized List

```c
extern int f(), g;
#pragma internal list f, g
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

new_mangler

Controls the inclusion or exclusion of a template instance’s function return type to the mangled name of the instance.

Syntax

```c
#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.

no_conststringconv

Disables the deprecated implicit const string literal conversion (ISO/IEC 14882-1998 C++, §4.2).

Syntax

```c
#pragma no_conststringconv on | off | reset
```
Pragmas for C++

Remarks
When enabled, the compiler generates an error message when it encounters an implicit const string conversion.

Listing 25.12 Example of const string conversion

```c
#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.

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.

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.
Pragmas for C++

old_pods

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

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.

old_vtable

This pragma is no longer available.

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

Listing 25.13 shows an example.

Listing 25.13  Example #pragma opt_classresults

#pragma opt_classresults on
Pragmas for C++

```cpp
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.

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.

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
```
Pragmas for C++

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.

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. The other RTTI expressions are available even if you disable the Enable RTTI setting. Note that *type_info::before(const type_info&) is not implemented.
This pragma corresponds to the Enable RTTI setting in the CodeWarrior IDE’s C/C++ Language settings panel.

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 because it can produce erratic or unpredictable behavior in your program.
Pragmas for C++

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

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.

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 25.14  Static local initialization example

```c
int func(void) {
    // There may be synchronization problems if this function is
    // called by multiple threads.
    static int countdown = 20;
}```
### Pragmas for C++

```cpp
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.

**Listing 25.15** shows another example.

**Listing 25.15  Example thread_safe_init**

```cpp
#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.

---

### warn_hidevirtual

Controls the recognition of a non-virtual member function that hides a virtual function in a superclass.

**Syntax**

```cpp
#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. **Listing 25.16** shows an example.
### Pragmas for C++

#### Listing 25.16  Hidden Virtual Functions

```cpp
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 14882-1998 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.

This pragma corresponds to the **Hidden Virtual Functions** setting in the CodeWarrior IDE’s C/C++ Warnings settings panel.

---

### 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**

Listing 25.17 shows an example.

**Listing 25.17  Example of warn_no_explicit_virtual pragma**

```cpp
#pragma warn_no_explicit_virtual on

struct A {
    virtual void f();
};

struct B {
```
Pragmas for C++

```cpp
void f();
// WARNING: override B::f() is declared without virtual keyword
```

**TIP**
This warning message is not required by the ISO/IEC 14882-1998 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.

---

**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 14882-1998 C++ template parser.

---

**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 inline (i.e., on those indicated by the **inline** keyword or in line in a class declaration) function calls.

This pragma corresponds to the **Non-Inlined Functions** setting in the CodeWarrior IDE’s **C/C++ Warnings** settings panel. By default, this pragma is disabled.
Pragmas for C++

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 25.18 Inconsistent use of class and struct

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.”

This pragma corresponds to the Inconsistent ‘class’ / ‘struct’ Usage setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is disabled.

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.
Pragmas for C++
Pragmas for C++
Pragmas for Language Translation

asmpoundcomment

Controls whether the “#” symbol is treated as a comment character in inline assembly.

**Syntax**

```
#pragma asmpoundcomment on | off | reset
```

**Remarks**

Some targets may have additional comment characters, and may treat these characters as comments even when
```
#pragma asmpoundcomment off
```

is used.

Using this pragma may interfere with the function-level inline assembly language.

This pragma does not correspond to any panel setting. By default, this pragma is on.

asmsemicolcomment

Controls whether the “;” symbol is treated as a comment character in inline assembly.

**Syntax**

```
#pragma asmsemicolcomment on | off | reset
```
Pragmas for Language Translation

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.

const_strings
Controls the const-ness of character string literals.

Syntax
#pragma const_strings [ on | off | reset ]

Remarks
If you enable this pragma, the type of string literals is an array const char[\(n\)],
or const wchar_t[\(n\)] for wide strings, where \(n\) is the length of the string literal plus 1 for a terminating NUL character. Otherwise, the type char[\(n\)] or wchar_t[\(n\)] is used.
This pragma does not correspond to any setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is on when compiling C++ source code and off when compiling C source code.

dollar_identifiers
Controls use of dollar signs ($) in identifiers.

Syntax
#pragma dollar_identifiers on | off | reset

Remarks
If you enable this pragma, the compiler accepts dollar signs ($) in identifiers.
Otherwise, the compiler issues an error if it encounters anything but underscores,
Pragmas for Language Translation

alphabet, numeric character, and universal characters (\uxxxx, \Uxxxxxxxxx) in an identifier.

This pragma does not correspond to any panel setting. By default, this pragma is off.

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

This pragma corresponds to the Enable GCC Extensions setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is disabled.
Pragmas for Language Translation

mark

Adds an item to the Function pop-up menu in the IDE editor.

Syntax
#pragma mark itemName

Remarks
This pragma adds itemName to the source file’s Function pop-up menu. If you open the file in the CodeWarrior Editor and select the item from the Function pop-up menu, the editor brings you to the pragma. Note that if the pragma is inside a function definition, the item does not appear in the Function pop-up menu.

If itemName begins with “--”, a menu separator appears in the IDE’s Function pop-up menu:

#pragma mark --

This pragma does not correspond to any panel setting.

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.
Pragmas for Language Translation

**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.

Listing 26.1 shows how to use this pragma to relax function pointer checking.

**Listing 26.1  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.

**multibyteaware**

Controls how the **Source encoding** option in the IDE is treated.

**Syntax**

```
#pragma multibyteaware on | off | reset
```
Pragmas for Language Translation

Remarks

This pragma is deprecated. See `#pragma text_encoding` for more details.

This pragma does not correspond to any panel setting, but the replacement option `Source encoding` appears in the CodeWarrior IDE’s C/C++ Preprocessor settings panel. By default, this pragma is `off`.

multibyteaware_preserve_literals

Controls the treatment of multibyte character sequences in narrow character string literals.

Syntax

```c
#pragma multibyteaware_preserve_literals on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

text_encoding

Identifies the character encoding of source files.

Syntax

```c
#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
Pragmas for Language Translation

**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, `#pragma text_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.

This pragma corresponds to the **Source Encoding** option in the CodeWarrior IDE’s **C/C++ Preprocessor** settings panel. By default, this setting is **ASCII**.

**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**

<table>
<thead>
<tr>
<th>Trigraph</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>???=</td>
<td>#</td>
</tr>
<tr>
<td>???/</td>
<td>\</td>
</tr>
<tr>
<td>??^</td>
<td>`</td>
</tr>
<tr>
<td>??(</td>
<td>[</td>
</tr>
<tr>
<td>??)</td>
<td>]</td>
</tr>
<tr>
<td>??!</td>
<td>`</td>
</tr>
<tr>
<td>??&lt;</td>
<td>`</td>
</tr>
</tbody>
</table>
Pragmas for Language Translation

### Table 26.1 Trigraph table

<table>
<thead>
<tr>
<th>Trigraph</th>
<th>Character</th>
</tr>
</thead>
<tbody>
<tr>
<td>??&gt;</td>
<td>}</td>
</tr>
<tr>
<td>???</td>
<td>~</td>
</tr>
</tbody>
</table>

**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 26.2 Example of Pragma trigraphs**

```c
char c = '????'; /* ERROR: Trigraph sequence expands to '??^ */
char d = '\\??\\???'; /* OK */
```

This pragma corresponds to the Expand Trigraphs setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is disabled.

### 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.

This pragma corresponds to the Use unsigned chars setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this setting is disabled.
Pragmas for Diagnostic Messages

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. Listing 27.1 shows an example.

Listing 27.1 Assigning to an Enumerated Type

```c
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, Listing 27.2 results in a warning message.

Listing 27.2 A non-void function with an empty return statement

```c
int MyInit(void)
{
    int err = GetMyResources();
    if (err != -1)
```
Pragmas for Diagnostic Messages

```c
{   err = GetMoreResources();
} return; /* WARNING: empty return statement */
```

Listing 27.3 shows how to prevent this warning message.

**Listing 27.3 A non-void function with a proper return statement**

```c
int MyInit(void)
{
    int err = GetMyResources();
    if (err != -1)
    {
        err = GetMoreResources();
    }
    return err; /* OK */
}
```

This pragma corresponds to the **Extended Error Checking** setting in the CodeWarrior IDE’s **C/C++ Warnings** settings panel. By default, this setting is **off**.

**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
Pragmas for Diagnostic Messages

This pragma does not correspond to any panel setting. By default, this pragma is off.

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

In the CodeWarrior IDE, the message appears in the Errors & Warnings window. On the command line, the message is sent to the standard error stream.

This pragma does not correspond to any panel setting.

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.
Pragmas for Diagnostic Messages

**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.

**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.

**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
Pragmas for Diagnostic Messages

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.

**unused**

Controls the suppression of warning messages for variables and parameters that are not referenced in a function.

**Syntax**

```c
#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 27.4 Example of Pragma unused() in C**

```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. */
}
```

---

*CodeWarrior Build Tools Reference ColdFire™ Architectures Edition*
Pragmas for Diagnostic Messages

Listing 27.5 Example of Pragma unused() in C++

```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.

**warning**

Controls which warning numbers are displayed during compiling.

**Syntax**

```c
#pragma warning on | off | reset (num [, ...])
```

This alternate syntax is allowed but ignored (message numbers do not match):

```c
#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.
Pragmas for Diagnostic Messages

Remarks

Use the pragma `showmessagenumber` to display warning messages with their warning numbers.

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.

---

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.

This pragma corresponds to the **Treat All Warnings as Errors** setting in the CodeWarrior IDE’s **C/C++ Warnings** panel.

---

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 27.6  Example of warn_any_ptr_int_conv

```
#pragma warn_ptr_int_conv on
```

---
Pragmas for Diagnostic Messages

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. */
}

Remarks
This pragma corresponds to the Pointer/Integral Conversions setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is off.

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 27.7  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 27.8  Example of empty declaration in C++

```c
#pragma warn_emptydecl on
extern "C" {
}; /* WARNING */
```

This pragma corresponds to the Empty Declarations setting in the CodeWarrior IDE’s C/C++ Warnings panel. By default, this pragma is disabled.

**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, Listing 27.9 is acceptable source code but generates a warning message when you enable this setting.

Listing 27.9  Warning about extra commas

```c
#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.

**warn_filenamecaps**

Controls the recognition of conflicts involving case-sensitive filenames within user includes.
Pragmas for Diagnostic Messages

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 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, see
warn_filenamecaps_system.

This pragma corresponds to the Include File Capitalization setting in the
CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is
off.

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 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.
Pragmas for Diagnostic Messages

This pragma corresponds to the Check System Includes setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is off.

**NOTE** Some SDKs (Software Developer Kits) use “colorful” capitalization, so this pragma may issue a lot of unwanted messages.

**warn_hiddenlocals**

Controls the recognition of a local variable that hides another local variable.

**Syntax**

```c
#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 Listing 27.10.

**Listing 27.10  Example of hidden local variables warning**

```c
#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.

**warn_illpragma**

Controls the recognition of illegal pragma directives.

**Syntax**

```c
#pragma warn_illpragma on | off | reset
```

*CodeWarrior Build Tools Reference ColdFire™ Architectures Edition* 293
Pragmas for Diagnostic Messages

Remarks
If you enable this pragma, the compiler displays a warning message when it encounters a pragma it does not recognize.
This pragma corresponds to the Illegal Pragmas setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this setting is off.

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.

warn_illunionmembers
Controls whether or not to issue a warning message when illegal union members are made, 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.
Pragmas for Diagnostic Messages

**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. **Listing 27.11** provides an example.

**Listing 27.11  Example of Implicit float-to-int 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 */
}
```

This pragma corresponds to the **Float to Integer** setting in the CodeWarrior IDE’s **C/C++ Warnings** settings panel. By default, this pragma is on.

**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
```
Pragmas for Diagnostic Messages

Remarks
If you enable this pragma, the compiler issues a warning message for implicitly converting integral values to floating-point values. Listing 27.12 shows an example.

Listing 27.12  Example of implicit int-to-float conversion

```c
#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 */
}
```

This pragma corresponds to the Integer to Float setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is off.

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. Listing 27.13 provides an example.

Listing 27.13  Example of implicit conversions between signed int and unsigned int

```c
#pragma warn_impl_s2u_conv on

signed int si;
```
Pragmas for Diagnostic Messages

unsigned int ui;

int main()
{
    ui = si; /* WARNING */
    si = ui; /* WARNING */

    #pragma warn_impl_s2u_conv off
    ui = si; /* OK */
    si = ui; /* OK */
}

This pragma corresponds to the Signed / Unsigned setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is enabled.

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. Listing 27.14 provides an example.

Listing 27.14 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 */
}

CodeWarrior Build Tools Reference ColdFire™ Architectures Edition
Pragmas for Diagnostic Messages

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.

This pragma corresponds to the Implicit Arithmetic Conversions setting in the
CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is
off.

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.

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 Listing 27.15.

Listing 27.15  Example of warn_missingreturn pragma

#pragma warn_missingreturn on
Pragmas for Diagnostic Messages

```c
int func()
{
    /* WARNING: no return statement. */
}
```

This pragma corresponds to the Missing 'return' Statements option in the CodeWarrior IDE's C/C++ Warnings settings panel.

#### 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)`. **Listing 27.16** provides an example.

**Listing 27.16  Example of Pragma warn_no_side_effect**

```c
#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. */
}
```

This pragma corresponds to the Expression Has No Side Effect panel setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is off.

#### warn_padding

Controls the issuing of warning messages for data structure padding.

**Syntax**

```
#pragma warn_padding on | off | reset
```
Pragmas for Diagnostic Messages

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.

warn_pch_portability
Controls whether or not to issue a warning message when `#pragma once` 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` 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.

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:
Pragmas for Diagnostic Messages

- 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 ==. Listing 27.17 shows an example.

Listing 27.17  Confusing = and == in Comparisons

```c
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 =. Listing 27.18 shows an example.

Listing 27.18  Confusing = and == Operators in Assignments

```c
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, Listing 27.19 generates a warning message.

Listing 27.19  Empty statement

```c
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 Listing 27.20 suppress the above error or warning messages.

Listing 27.20  Intentional empty statements

```c
while (i++) ; /* OK: White space separation. */
while (i++) /* OK: Comment separation */ ;
```

This pragma corresponds to the Possible Errors setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is off.
Pragmas for Diagnostic Messages

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 27.21  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” on page 289.

This pragma corresponds to the Pointer / Integral Conversions setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is off.

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). Listing 27.22 provides an example.
Pragmas for Diagnostic Messages

Listing 27.22  Example of Function Calls with Unused Results

```c
#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.

warn_undefmacro

Controls the detection of undefined macros in `#if` and `#elif` directives.

**Syntax**

```c
#pragma warn_undefmacro on | off | reset
```

**Remarks**

Listing 27.23 provides an example.

Listing 27.23  Example of Undefined Macro

```c
#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 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.

This pragma corresponds to the **Undefined Macro in #if** setting in the CodeWarrior IDE’s **C/C++ Warnings** settings panel. By default, this pragma is off.
Pragmas for Diagnostic Messages

warn_uninitializedvar

Controls the compiler to perform some dataflow analysis and emits warning messages whenever local variables are initialized before being used.

Syntax

#pragma warn_uninitializedvar on | off | reset

Remarks

This pragma has no corresponding setting in the CodeWarrior IDE. By default, this pragma is on.

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. Listing 27.24 shows an example.

Listing 27.24  Warning about unused function arguments

```c
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 Listing 27.25.
Listing 27.25 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. Listing 27.26 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 27.26 Unused, Unnamed Arguments

void func(int /* temp */, int error)
{
    /* Compiler does not warn that "temp" is not used. */
    error=do_something();
}

This pragma corresponds to the Unused Arguments setting in the C/C++ Warnings Panel. By default, this pragma is off.

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. Listing 27.27 shows an example.
Pragmas for Diagnostic Messages

Listing 27.27 Unused Local Variables Example

```c
int error;
void func(void)
{
    int temp, errer; /* NOTE: errer is missspelled. */
    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 Listing 27.28.

Listing 27.28 Suppressing Unused Variable Warnings

```c
void func(void)
{
    int i, temp, error;

    #pragma unused (i, temp) /* Do not warn that i and temp */
    error = do_something(); /* are not used */
}
```

This pragma corresponds to the Unused Variables setting in the CodeWarrior IDE’s C/C++ Warnings settings panel. By default, this pragma is off.
Pragmas for Preprocessing and Precompilation

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.

faster_pch_gen

Controls the performance of precompiled header generation.

Syntax

#pragma faster_pch_gen on | off | reset
Pragmas for Preprocessing and Precompilation

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.

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.

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.
Pragmas for Preprocessing and Precompilation

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.

This pragma corresponds to the Show full paths option in the CodeWarrior IDE’s C/C++ Preprocessor settings panel. By default, this pragma is off.

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.

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.
Pragmas for Preprocessing and Precompilation

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.

**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.

**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.

**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
```

310 CodeWarrior Build Tools Reference ColdFire™ Architectures Edition
Pragmas for Preprocessing and Precompilation

Remarks
This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is on.

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, see “once” on page 311. This pragma does not correspond to any CodeWarrior IDE panel setting.

oldPragma_once

This pragma is no longer available.

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
Pragmas for Preprocessing and Precompilation

and

#pragma once

Use #pragma once in a header file to ensure that the header file is included only once in a source file. Use #pragma once in a header file or source file to insure 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 oldPragmaOnce on pragma, the once pragma completely ignores path names.

This pragma does not correspond to any panel setting. By default, this pragma is off.

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 Listing 28.1.

Listing 28.1 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.

312 CodeWarrior Build Tools Reference ColdFire™ Architectures Edition
Pragmas for Preprocessing and Precompilation

**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.

---

**pragma_prepdump**

Controls whether pragma directives in the source text appear in the preprocessing output.

**Syntax**

```
#pragma pragma_prepdump on | off | reset
```

**Remarks**

This pragma corresponds to the `Emit #pragmas` option in the CodeWarrior IDE’s C/C++ Preprocessor settings panel. By default, this pragma is `off`.

**TIP** When submitting bug reports with a preprocessor dump, be sure this option is enabled.

---

**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.
Pragmas for Preprocessing and Precompilation

Listing 28.2 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 28.2 Using #pragma precompile_target

```c
#ifdef __cplusplus
#pragma precompile_target "MyCPPHeaders"
#else
#pragma precompile_target "MyCHeaders"
#endif
```

This pragma does not correspond to any panel setting.

**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.

This pragma corresponds to the *Emit file changes* option in the CodeWarrior IDE’s C/C++ Preprocessor settings panel. By default, this pragma is *off*.

**space_prepdump**

Controls whether or not the compiler removes or preserves whitespace in the preprocessor’s output.

**Syntax**

```
#pragma space_prepdump on | off | reset
```
Pragmas for Preprocessing and Precompilation

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.

This pragma corresponds to the Keep whitespace option in the CodeWarrior IDE’s C/C++ Preprocessor settings panel. By default, this pragma is off.

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.

This pragma corresponds to the Source-relative includes option in the Access Paths panel. By default, this pragma is off.

syspath_once
Controls how included files are treated when #pragma once 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>
Pragmas for Preprocessing and Precompilation

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.

shows an example.

This pragma does not correspond to any panel setting. By default, this setting is on.

Listing 28.3 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. */
Pragmas for Library and Linking

always_import

Controls whether or not #include directives are treated as #pragma import directives.

Syntax

#pragma always_import on | off | reset

Remarks

If you enable this pragma, the compiler treats all #include statements as #pragma import statements.
This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is off.

export

Controls the exporting of data and functions to be accessible from outside a program or library.

Syntax

#pragma export on | off | reset
#pragma export list name1 [, name2, ...]

name1, name2

Names of functions or global variables to export.
Pragmas for Library and Linking

Remarks
When using the \#pragma export on format, all functions in the source file being compiled will be accessible from outside the program or library that the compiler and linker are building.

Use the \#pragma export list format to specify global variables and functions for exporting. In C++, this form of the pragma applies to all variants of an overloaded function. You cannot use this pragma for C++ member functions or static class members. Listing 29.1 shows an example:

Listing 29.1 Example of an Exported List
extern int f(), g;
\#pragma export list f, g

force_active

Controls how “dead” functions and global variables are linked.

Syntax
\#pragma force_active on | off | reset

Remarks
If you enable this pragma, the linker leaves functions and global in the finished application, even if the functions are never called in the program.

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is off.

import

Controls the importing of global data or functions.

Syntax
\#pragma import on | off | reset
\#pragma import list name1 [, name2, ...]
name1, name2

Names of functions or global variables to import.
Pragmas for Library and Linking

Remarks

When using the `#pragma import on` format, all functions are automatically imported.

Use the `#pragma import list` format to specify data or functions for importing. In C++, this form of the pragma applies to all variants of an overloaded function. You cannot use this pragma for C++ member functions or static class members.

Listing 29.2 shows an example:

Listing 29.2  Example of an Imported List

```c
extern int f(), g;
#pragma import list f, g
```

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `off`.

lib_export

Controls the exporting of data or functions.

Syntax

```
#pragma lib_export on | off | reset
#pragma lib_export list name1 [, name2 ]*
```

Remarks

When using the `#pragma lib_export on` format, the linker marks all data and functions that are within the pragma’s scope for export.

Use the `#pragma lib_export list` format to tag specific data or functions for exporting. In C++, this form of the pragma applies to all variants of an overloaded function. You cannot use this pragma for C++ member functions or static class members.

Listing 29.3 shows an example:

Listing 29.3  Example of a lib_export List

```c
extern int f(), g;
#pragma lib_export list f, g
```
Pragmas for Library and Linking

This pragma does not correspond to any panel setting. By default, this pragma is disabled.
30

Pragmas for Code Generation

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. Listing 30.1 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.

This pragma corresponds to the Reuse Strings setting in the CodeWarrior IDE’s C/C++ Language settings panel. By default, this pragma is off.
Pragmas for Code Generation

Listing 30.1 Reusing string literals

```c
#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. */
}
```

enumsalwaysint

Specifies the size of enumerated types.

**Syntax**

```c
#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.

Listing 30.2 shows an example.

Listing 30.2 Example of Enumerations the Same as Size as int

```c
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
```
Pragmas for Code Generation

`generate an error message. Otherwise, this type is the same size as a long long. */`

This pragma corresponds to the **Enums Always Int** setting in the CodeWarrior IDE’s **C/C++ Language** settings panel. By default, this pragma is **off**.

**errno_name**

Tells the optimizer how to find the `errno` identifier.

**Syntax**

```c
#pragma errno_name id | ...
```

**Remarks**

When this pragma is used, the optimizer can use the identifier `errno` (either a macro or a function call) to optimize standard C library functions better. If not used, the optimizer makes worst-case assumptions about the effects of calls to the standard C library.

**NOTE**  The MSL C library already includes a use of this pragma, so you would only need to use it for third-party C libraries.

If `errno` resolves to a variable name, specify it like this:

```c
#pragma errno_name _Errno
```

If `errno` is a function call accessing ordinarily inaccessible global variables, use this form:

```c
#pragma errno_name ...
```

Otherwise, do not use this pragma to prevent incorrect optimizations.

This pragma does not correspond to any panel setting. By default, this pragma is unspecified (worst case assumption).

**explicit_zero_data**

Controls the placement of zero-initialized data.
Pragmas for Code Generation

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.

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.

instmgr_file

Controls where the instance manager database is written, to the target data directory or to a separate file.

Syntax

#pragma instmgr_file "name"

Remarks

When the Use Instance Manager option is on, the IDE writes the instance manager database to the project’s data directory. If the #pragma instmgr_file is used, the database is written to a separate file.
Pragmas for Code Generation

Also, a separate instance file is always written when the command-line tools are used.

NOTE  Should you need to report a bug, you can use this option to create a separate instance manager database, which can then be sent to technical support with your bug report.

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 C89 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.

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 “enumsalwaysint pragma” on page 322).

This pragma does not correspond to any panel setting. By default, this setting is enabled.
Pragmas for Code Generation

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.

options

Specifies how to align structure and class data.

Syntax

#pragma options align= alignment

Parameter

alignment

Specifies the boundary on which structure and class data is aligned in memory.
Values for alignment range from 1 to 16, or use one of the following preset values:

Table 30.1 Structs and Classes Alignment

<table>
<thead>
<tr>
<th>If alignment is …</th>
<th>The compiler …</th>
</tr>
</thead>
<tbody>
<tr>
<td>mac68k</td>
<td>Aligns every field on a 2-byte boundaries, unless a field is only 1 byte long. This is the standard alignment for 68K Mac OS.</td>
</tr>
<tr>
<td>mac68k4byte</td>
<td>Aligns every field on 4-byte boundaries.</td>
</tr>
</tbody>
</table>
### Pragmas for Code Generation

#### pool_strings

Controls how string literals are stored.

**Syntax**

```c
#pragma pool_strings on | off | reset
```

**Remarks**

If you enable this pragma, the compiler collects all string constants into a single data object so your program needs one data section for all of them. If you disable this pragma, the compiler creates a unique data object for each string constant. While this decreases the number of data sections in your program, on some processors it also makes your program bigger because it uses a less efficient method to store the address of the string.

---

**Table 30.1 Structs and Classes Alignment**

<table>
<thead>
<tr>
<th>If alignment is ...</th>
<th>The compiler ...</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>power</code></td>
<td>Aligns every field on its natural boundary. This is the standard alignment for PowerPC Mac OS. For example, it aligns a character on a 1-byte boundary and a 16-bit integer on a 2-byte boundary. The compiler applies this alignment recursively to structured data and arrays containing structured data. So, for example, it aligns an array of structured types containing an 4-byte floating point member on an 4-byte boundary.</td>
</tr>
<tr>
<td><code>native</code></td>
<td>Aligns every field using the standard alignment. It is equivalent to using <code>mac68k</code> for 68K Mac OS and <code>power</code> for PowerPC Mac OS.</td>
</tr>
<tr>
<td><code>packed</code></td>
<td>Aligns every field on a 1-byte boundary. It is not available in any panel. This alignment causes your code to crash or run slowly on many platforms. <em>Use it with caution.</em></td>
</tr>
<tr>
<td><code>reset</code></td>
<td>Resets to the value in the previous <code>#pragma options align</code> statement.</td>
</tr>
</tbody>
</table>

**NOTE** There is a space between `options` and `align`. 

---
Pragmas for Code Generation

This pragma is especially useful if your program is large and has many string constants or uses the CodeWarrior Profiler.

**NOTE** If you enable this pragma, the compiler ignores the setting of the `pcrelstrings` pragma.

This pragma corresponds to the Pool Strings setting in the CodeWarrior IDE’s C/C++ Language settings panel.

**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, C strings used in your source code (for example, "hello") are output to the read-only data section instead of the global data section. In effect, these strings act like `const char *`, even though their type is really `char *`.

This pragma does not correspond to any IDE panel setting.

**reverse_bitfields**

Controls whether or not the compiler reverses the bitfield allocation.

**Syntax**

```
#pragma reverse_bitfields on | off | reset
```

**Remarks**

This pragma reverses the bitfield allocation, so that bitfields are arranged from the opposite side of the storage unit from that ordinarily used on the target. The compiler still orders the bits within a single bitfield such that the lowest-valued bit is in the right-most position.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.
Pragmas for Code Generation

store_object_files

Controls the storage location of object data, either in the target data directory or as a separate file.

Syntax

#pragma store_object_files on | off | reset

Remarks

By default, the IDE writes object data to the project’s target data directory. When this pragma is on, the object data is written to a separate object file.

NOTE

For some targets, the object file emitted may not be recognized as actual object data.

This pragma does not correspond to any panel setting. By default, this pragma is off.
Pragmas for Code Generation
Pragmas for Optimization

### 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 `#pragma optimization_level`. The Global Optimizer is invoked even at `optimization_level 0` if `#pragma global_optimizer` is enabled.

This pragma corresponds to the settings in the *Global Optimizations* panel. By default, this setting is on.

### ipa

Specifies how to apply interprocedural analysis optimizations.

**Syntax**

```
#pragma ipa program | file | on | function | off
```

**Remarks**

See “Interprocedural Analysis” on page 167.

Place this pragma at the beginning of a source file, before any functions or data have been defined. There are three levels of interprocedural analysis:
Pragmas for Optimization

- program-level: the compiler translates all source files in a program then optimizes object code for the entire program
- file-level: the compiler translates each file and applies this optimization to the file
- function-level: the compiler does not apply interprocedural optimization

The options file and on are equivalent. The options function and off are equivalent.

**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 setting is related to the `global_optimizer` pragma.

**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” on page 331 level.

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” on page 331 level.

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” on page 331 level.

opt_loop_invariants
Controls the use of loop invariant optimization.
Pragmas for 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 outside the loop, which then runs faster.
This pragma does not correspond to any panel setting.

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" on page 331 level.

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" on page 331 level.
Pragmas for Optimization

---

**opt_strength_reduction_strict**

Uses a safer variation of strength reduction optimization.

**Syntax**

```c
#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.

---

**opt_unroll_loops**

Controls the use of loop unrolling optimization.

**Syntax**

```c
#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 setting is related to the “global_optimizer” on page 331 level.

---

**opt_vectorize_loops**

Controls the use of loop vectorizing optimization.

**Syntax**

```c
#pragma opt_vectorize_loops on | off | reset
```
Pragmas for Optimization

Remarks

If you enable this pragma, the compiler improves loop performance.

NOTE
Do not confuse loop vectorizing with the vector instructions available in some processors. Loop vectorizing is the rearrangement of instructions in loops to improve performance. This optimization does not optimize a processor’s vector data types.

By default, this pragma is off.

optimization_level

Controls global optimization.

Syntax

#pragma optimization_level 0 | 1 | 2 | 3 | 4

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 4. The higher the argument, the more optimizations performed by the global optimizer.

For more information on the optimization the compiler performs for each optimization level, refer to the Targeting manual for your target platform.

These pragmas correspond to the settings in the Global Optimizations panel. By default, this pragma is disabled.

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.
Pragmas for Optimization

...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.

The pragma corresponds to the Optimize for Size setting on the Global Optimizations panel.

**optimizewithasm**

Controls optimization of assembly language.

**Syntax**

`#pragma optimizewithasm on | off | reset`

**Remarks**

If you enable this pragma, the compiler also optimizes assembly language statements in C/C++ source code.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

**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.
Pragmas for Optimization
Pragmas for ColdFire

This chapter is a placeholder for a chapter containing target-specific material.

ColdFire Diagnostic Pragmas

SDS_debug_support

Tries to make the DWARF output file compatible with the Software Development System (SDS) debugger. The default value is OFF.

#pragma SDS_debug_support [ on | off | reset ]

ColdFire Library and Linking Pragmas

define_section

Specifies a predefined section or defines a new section for compiled object code.

#pragma define_section sname "istr" [.ustr] [addrmode] [accmode]

Parameters

sname

Identifier for source references to this user-defined section.

istr

Section-name string for initialized data assigned to this section. Double quotes must surround this parameter value, which must begin with a period. (Also applies to uninitialized data if there is no ustr value.)

ustr

Optional: ELF section name for uninitialized data assigned to this section. Must begin with a period. Default value is the istr value.
Pragmas for ColdFire
ColdFire Library and Linking Pragmas

addrmode
Optional: any of these address-mode values:
- standard — 32-bit absolute address (default)
- near_absolute — 16-bit absolute address
- far_absolute — 32-bit absolute address
- near_code — 16-bit offset from the PC address
- far_code — 32-bit offset from the PC address
- near_data — 16-bit offset from the A5 register address
- far_data — 32-bit offset from the A5 register address

accmode
Optional: any of these letter combinations:
- R — readable
- RW — readable and writable
- RX — readable and executable
- RWX — readable, writable, and executable (default)
(No other letter orders are valid: WR, XR, or XRW would be an error.)

Remarks
The compiler predefines the common ColdFire sections that Table 32.1 lists.

Table 32.1 ColdFire Predefined Sections

<table>
<thead>
<tr>
<th>Applicability</th>
<th>Definition Pragmas</th>
</tr>
</thead>
<tbody>
<tr>
<td>Absolute Addressing Mode</td>
<td>#pragma define_section text &quot;.text&quot; far_absolute RX</td>
</tr>
<tr>
<td></td>
<td>#pragma define_section data &quot;.data&quot; &quot;.bss&quot; far_absolute RW</td>
</tr>
<tr>
<td></td>
<td>#pragma define_section sdata &quot;.sdata&quot; &quot;.sbss&quot; near_data RW</td>
</tr>
<tr>
<td></td>
<td>#pragma define_section const &quot;.rodata&quot; far_absolute R</td>
</tr>
<tr>
<td>C++, Regardless of Addressing Mode</td>
<td>#pragma define_section exception &quot;.exception&quot; far_absolute R</td>
</tr>
<tr>
<td></td>
<td>#pragma define_section exceptlist &quot;.exceptlist&quot; far_absolute R</td>
</tr>
</tbody>
</table>
Another use for `#pragma define_section` is redefining the attributes of predefined sections:

- To force 16-bit absolute addressing for all data, use
  `#pragma define_section data ".data" ".bss" near_absolute RW`

- To force 32-bit TP-relative addressing for exception tables, use:
  `#pragma define_section exceptlist ".exceptlist" far_code`
  `#pragma define_section exception ".exception" far_code`

You should put any such attribute-redefinition pragmas a prefix file or other header that all your program’s source files will include.

**NOTE** The ELF linker’s Section Mappings settings panel must map any user-defined compiler section to an appropriate segment.

### ColdFire Code Generation Pragmas

#### codeColdFire

Controls organization and generation of ColdFire object code.

`#pragma codeColdFire processor`
Pragmas for ColdFire

ColdFire Code Generation Pragmas

Parameter

processor

Any of these specifier values: MCF521x, MCF5206e, MCF5249, MCF5272, MCF5282, MCF5307, MCF5407, MCF547x, MCF548x — or reset, which specifies the default processor.

const_multiply

Enables support for constant multiplies, using shifts and add/subtracts.

#pragma const_multiply [ on | off | reset ]

Remarks

The default value is on.

emac

Enables EMAC assembly instructions in inline assembly.

#pragma emac [ on | off | reset ]

Remarks

Enables inline-assembly instructionsmac, msac, macl, msacl, move, and movclr for the ColdFire EMAC unit.

The default value is OFF.

explicit_zero_data

Specifies storage area for zero-initialized data.

#pragma explicit_zero_data [ on | off | reset ]

Remarks

The default value OFF specifies storage in the .sbss or .bss section. The value ON specifies storage in the .data section. The value reset specifies storage in the most-recent previously specified section.
Pragmas for ColdFire
ColdFire Code Generation Pragmas

Example

#pragma explicit_zero_data on
int in_data_section = 0;

#pragma explicit_zero_data off
int in_bss_section = 0;

inline_intrinsics
Controls support for inline intrinsic optimizations `strncpy` and `strlen`.
#pragma inline_intrinsics [ on | off | reset ]

Remarks
In the `strncpy` optimization, the system copies the string via a set of move-immediate commands to the source address. The system applies this optimization if the source is a string constant of fewer than 64 characters, and optimizing is set for speed.

In the `strlen` optimization, a move immediate of the length of the string to the result replaces the function call. The system applies this optimization if the source is a string constant.

The default value is ON.

interrupt
Controls compilation for interrupt-routine object code.
#pragma interrupt [ on | off | reset ]

Remarks
For the value ON, the compiler generates special prologues and epilogues for the functions this pragma encapsulates. The compiler saves or restores all modified registers (both nonvolatile and scratch). Functions return via RTE instead of RTS.
Pragmas for ColdFire
ColdFire Code Generation Pragmas

You also can also use __declspec(interrupt) to mark functions as interrupt routines, for example:
__declspec(interrupt) void alpha()
{
    //enter code here
}

readonly_strings

Enables the compiler to place strings in the .rodata section.
#pragma readonly_strings [ on | off | reset ]

Remarks
The default value is ON.
For the OFF value, the compiler puts strings in initialized data sections .data or .sdata, according to the string size.

section

Activates or deactivates a user-defined or predefined section.
#pragma section sname begin | end

Parameters
sname
Identifier for a user-defined or predefined section.
begin
Activates the specified section from this point in program execution.
end
Deactivates the specified section from this point in program execution; the section returns to its default state.

Remarks
Each call to this pragma must include a begin parameter or an end parameter, but not both.
Pragmas for ColdFire

ColdFire Optimization Pragmas

You may use this pragma with `#pragma push` and `#pragma pop` to ease complex or frequent changes to section settings.

**NOTE**
A simpler alternative to `#pragma section` is the `__declspec()` declaration specifier.

---

**ColdFire Optimization Pragmas**

---

**opt_unroll_count**

Limits the number of times a loop can be unrolled; fine-tunes the loop-unrolling optimization.

```
#pragma opt_unroll_count [0..127 | reset]
```

**Remarks**

The default value is 8.

---

**opt_unroll_instr_count**

Limits the number of pseudo-instructions; fine-tunes the loop-unrolling optimization.

```
#pragma opt_unroll_instr_count [0..127 | reset]
```

**Remarks**

There is not always a one-to-one mapping between pseudo-instructions and actual ColdFire instructions.

The default value is 100.

---

**profile**

Organizes object code for the profiler library and enables simple profiling.

```
#pragma profile [on | off | reset]
```

---

*CodeWarrior Build Tools Reference ColdFire™ Architectures Edition* 345
Pragmas for ColdFire

ColdFire Optimization Pragmas

Remarks

Corresponds to the Generate code for profiling checkbox of the ColdFire Processor settings panel.
Index

Symbols
#include
    diagnosing error messages 286
    GCC policy 71
    IDE 30
    importing linker symbols 317
    including once 311
    letter case 36, 67, 291, 292, 36
    levels 155
    other operating systems 69
    paths 308, 309
    precompiled files 149, 162
    reducing compiler time 161
    searching 70, 315
#include 309
$ 276
.lcf 43
= See also assignment, equals.
== See also equals, assignment.
__embedded_cplusplus 157, 228
__ide_target() 229
__INTEL__ 231
__PRETTY_FUNCTION__ 150

A
access_errors 249
addition 180
ADDR linker command 103
ALIGN linker command 104
ALIGNALL linker command 104
alignment, LCF 125
allocation, variable 200
always_import 317
always_inline 249
-ansi 45
ANSI Keywords Only option 28
ANSI_strict 243
arg_dep_lookup 249
arguments
    list 245
    arguments, inline assembly 189–191
    arithmetic operators, LCF 124, 125
-ARM 47
ARM_scoping 250
array_new_delete 251
asm_poundcomment 275
asm_semicoloncomment 275
assignment
    accidental 301
    unused 176
auto_inline 251
auto_inline pragma 27

B
bitfield 328
board-independent code 225
-bool 47
bool 251

C
C
    GNU Compiler Collection extensions 142
-cc 81
C fully compliant console IO MSL files 209
C fully compliant UART IO MSL files 208
C hardware floating point console IO MSL files 212
C hardware floating point UART IO MSL files 211
C small console IO MSL files 211
C small UART IO MSL files 210
C++
    embedded 157
    precompiling 149
C++ fully compliant MSL files 213
C++ hardware floating point MSL files 215
C++ hardware floating point UART IO MSL files 214
C++ small working set MSL files 214
### C/C++ Warnings panel 32

- **c99 243**
- **calling conventions 199**
- **catch statement 26, 257**
- **-char 51**
- **char type 30**

**character strings**
- See strings.
- **check_header_flags 307**
- **closure segments, LCF 121, 122**

**code**
- **board-independent 225**
- **position-independent 224, 225**

**code motion 178**
- **codeColdFire pragma 341**
- **-codegen 81**

**command files 43**

**command-line options**
- **-ansi 45**
- **-ARM 47**
- **-bool 47**
- **-c 81**
- **-char 51**
- **-codegen 81**
- **-convertpaths 69**
- **-Cpp Exceptions 47**
- **-cwd 70**
- **-D 70**
- **-defaults 51**
- **-define 70**
- **-dialect 48**
- **-disassemble 59**
- **-E 71**
- **-encoding 52**
- **-enum 81**
- **-EP 71**
- **-ext 82**
- **-flag 53**
- **-for_scoping 48**
- **-gcc_Extensions 53**
- **-gccext 53**
- **-gcincludes 71**
- **-help 59**
- **-I 72**
- **-include 73**
- **-inline 85**
- **-instmgr 49**
- **-ir 73**
- **-iso_templates 49**
- **-keepobjects 79**
- **-M 54**
- **-make 54**
- **-mapcr 54**
- **-maxerrors 60**
- **-maxwarnings 61**
- **-MD 55**
- **-min_enum_size 82**
- **-MM 55**
- **-MMD 55**
- **-msext 56**
- **-msgstyle 61**
- **-multibyteaware 56**
- **-nofail 62**
- **-nolink 79**
- **-noprecompile 76**
- **-nosyspath 76**
- **-O 86**
- **-o 80**
- **-O+ 86**
- **-once 56**
- **-opt 87**
- **-P 73**
- **-ppopt 75**
- **-pragma 57**
- **-precompile 74**
- **-prefix 75**
- **-preprocess 74**
- **-progress 62**
- **-relax_pointers 57**
- **-requireprotos 57**
- **-RTTI 50**
- **-S 62**
- **-search 58**
- **-som 50**
- **-som_env_check 50**
- **-stderr 62**
- **-stderr 62**
- **-stdinc 76**
-stdkeywords 45
-strict 46
-strings 83
-timing 63
-trigraphs 58
-U+ 76
-undefined 77
-verbose 63
-version 63
-warnings 64
-wchar_t 50
-wraplines 67
commands
  linker 119
  comment operators, LCF 124, 125
  common subexpression elimination 174
  compilers ??-203
  compound literal 137
  configuration flag functionality 219
  const_multiply pragma 342
  const_strings 276
  conventions, calling 199
  -convertpaths 69
  copy propagation 175
  cplusplus 252
  -Cpp_exceptions 47
  cpp_extensions 252
  cryptographic acceleration instructions 201-??
  CSE. See optimization.
  custom modifications 225, 226
  -cwd 70
  CWFolder 40
  cwinst.db 49
dead code 171
dead store elimination 176
deadstripping, linker 119
def_inherited 254
def_defarg_parsing 255
defaults 51
defer_defarg_parsing 255
defered
  code generation 255
  inlining 255
  -define 70
  define_section pragma 339-341
  -dialect 48
  directives
    #line 309
    inline assembly 191-196
    linker 119
  -disassemble 59
  DLL
    See libraries.
  dollar_sign 276
  dollar_identifiers 276
  Don’t Inline option 27
  dont_inline 256
  dont_inline pragma 27
  dont_reuse_strings 321
  dont_reuse_strings pragma 30
  double_type 324
  ds_inline assembly directive 192, 193
dynamic libraries
  See libraries.
  dynamic_cast keyword 26, 267
e-encoding 52
-enum 81
-enumerated types 283
-Environment tab 39
-E 71
-E option 310
EC++ fully compliant MSL files 216
EC++ hardware floating point MSL files 217
EC++ hardware floating point UART IO MSL files 216
ecplusplus 256
ecplusplus pragma 157
emac pragma 342
embedded C++ 157
Enable Exception Handling option 26
-encoding 52
-entry inline assembly directive 193, 194
-enum 81
tenumalwaysint 322
tenumerated types 283
-Environment tab 39
equals instead of assignment 301
erno_name 323
error messages
diagnosing #include directives 286
exception handling 26
EXCEPTION linker command 105
exception tables, LCF 128
exceptions 257
executable files, linker 120
explicit_zero_data pragma 342
export 317
EXPORTSTRTAB linker command 105
EXPOITSYMTAB linker command 106
expression
common elimination 174
simplification 172
expressions, LCF 123, 124
-extend 82
extended_errorcheck 257, 283
GNU C 277
GNU Compiler Collection 142
function declarations 245
prototypes 245
function specification, LCF 126
functionality, configuration flag 219
GCC. See GNU Compiler Collection.
-gcc_extensions 53
gcc_extensions 277
-gccext 53
-gccincludes 71
global_optimizer 331
GNU C extensions 277
GNU Compiler Collection
extension to C 142
file include policy 71
header files
including once 311
searching 315
heap, LCF 127
-help 59
identifier
$ 276
dollar signs in 276
if statement 301
ignore_oldstyle 245
import 318
IMPORTSTRTAB linker command 107
IMPORTSYMTAB linker command 108
inaccessible code 171
-includ 73
INCLUDE linker command 109
infinite loop, creating 301
-inl 85
inline assembly
directives 191–196
dc 192
ds 192, 193
eentry 193, 194
fralloc 194
ffree 194
machine 195
naked 195
opword 196
return 196
local variables, arguments 189–191
returning from routine 191
syntax 187–189
inline_bottom_up 259
inline_bottom_up_once 260
inline_depth 260
inline_intrinsics pragma 343
inline_max_auto_size 261
inline_max_size 261
inline_max_total_size 262
inlining
C++ 150
choosing functions 182
introduction 182
policies 184
reducing object code size 149
instance manager 149
-instmgr 150
-Instmgr 49
Instmgr_file 324
Instmgr_file pragma 150
integer formats 198
integrals, LCF 123, 124
__INTEL__ 231
internal 262
interprocedural analysis
declarations 169
enumerations 170
file level 168
introduction 167
invoking 168
program level 168
requirements 168
source dependencies 169
structures 170
type definitions 170
interrupt pragma 343
ipa 331
-ir 73
-iso_templates 49
K
KEEP_SECTION linker command 109
keepcomments 309
-keepobjects 79
keywords
  additional 28
dynamic_cast 26, 267
  standard 134
typeid 26, 267
keywords, linker 119
L
LCF
alignment 125
arithmetic, comment operators 124, 125
closure segments 121, 122
exception tables 128
expressions 123, 124
file specification 126
function specification 126
heap, stack 127
integrals 123, 124
memory segment 120
position-independent code, data 128
ROM-RAM copying 129–130
sections segment 122
specifying files, functions 126
stack, heap 127
static initializers 127
structure 120–122
syntax 123–132
variables 123, 124
writing to memory 130, 132
lib_export 319
libraries 205–226
dynamic 272
heaps 219
memory 219
Metrowerks standard 205–220
reduced working set 218, 219
runtime 220–226
serial I/O 217
static 272
UART 217
license 40
line_prepdump 309
link order 91
linker
and executable files 91
commands
ADDR 103
ALIGN 104
ALIGNALL 104
EXCEPTION 105
EXPORTSTRTAB 105
EXPORTSYMTAB 106
FORCE_ACTIVE 107
IMPORTSTRTAB 107
IMPORTSYMTAB 108
INCLUDE 109
KEEP_SECTION 109
location counter 102
MEMORY 110–111
OBJECT 111
REF_INCLUDE 112
SECTIONS 112, 113
SIZEOF 113
SIZEOF_ROM 114
WRITE0COMMENT 115
WRITEB 114
WRITEH 114
WRITEW 115
ZERO_FILLED_UNINITIALIZED 11
6
commands, directives, keywords 119
deadstripping 119
executable files 120
importing with #include 317
S-record comments 120
linker command files 43
live range splitting 177
LM_LICENSE_FILE 40
local variables, inline assembly 189–191
location counter linker command 102
longlong 325
longlong.Enums 325
loop
infinite 301
invariant code motion 178
unrolling 181
M
-M 54
machine inline assembly directive 195
macro_prepdump 310
~make 54
Makefile 54, 55
mangled names 150, 272
~mapcr 54
mark 278
maxerrorcount 284
~maxerrors 60
~maxwarnings 61
~MD 55
MEMORY linker command 110–111
memory segment, LCF 120
message 285
Metrowerks standard libraries 205–220
Microsoft Windows™ 39
~min_enum_size 82
min_enum_size 326
~NM 55
~MMD 55
modifications, custom 225, 226
mpwc_newline 278
mpwc_relax 279
~msext 56
msg_show_lineref 310
msg_show_realref 310
~msgstyle 61
MSL 205–220
MSL filename parameters 206
MSL files
fully compliant C++ 213
fully compliant console IO C 209
fully compliant EC++ 216
fully compliant UART IO C 208  
hardware floating point C++ 215  
hardware floating point console IO C 212  
hardware floating point EC++ 217  
hardware floating point UART IO C 211  
hardware floating point UART IO C++ 214  
hardware floating point UART IO EC++ 216  
small console IO C 211  
small UART IO C 210  
small working set C++ 214  
-multipbyteaware 56  
multipbyteaware 279  
multipbyteaware_preserve_literals 280  
multiplication 180

N
naked inline assembly directive 195  
new_mangler 263  
no_conststringconv 263  
no_static_dtors 264  
-nofail 62  
-nolink 79  
-noprecompile 76  
nosyminline 264  
-nosyspath 76  
notonce 311

O
-o 86  
-o 80  
-o+ 86  
object code  
	reducing size 149  
OBJECT linker command 111  
old_pods 265  
-once 56  
once 311  
-only_std_keywords 245  
only_std_keywords pragma 134  
operators, LCF 124, 125  
-opt 87  
opt_classresults 265  
opt_common_subs 332  
opt_common_subs pragma 174  
opt_dead_assignments 332  
opt_dead_assignments pragma 176  
opt_dead_code 333  
opt_dead_code pragma 172  
opt_lifetimes 333  
opt_loop_invariants pragma 179  
opt_propagation 334  
opt_propagation pragma 175  
opt_strength_reduction 334  
opt_strength_reduction pragma 180  
opt_strength_reduction_strict 335  
opt_unroll_count pragma 345  
opt_unroll_instr_count pragma 345  
opt_unroll_loops 335  
opt_unroll_loops pragma 181  
opt_vectorize_loops pragma 335  
optimization  
	common subexpression elimination 174  

copy propagation 175  
dead code 171  
dead store elimination 176  
expression simplification 172  
inlining 184  
intermediate 171  
interprocedural analysis 167  
live range splitting 177  
loop unrolling 181  
loop-invariant code motion 178  
loops 181  
stack size 175, 177  
strength reduction 180  
optimization_level 336  
optimize_for_size 336  
optimizewithasm 337  
options 326  
opword inline assembly directive 196

P
-p 73  
parse_func_templ 266  
parse_mfunc_templ 266  
PATH 40
PIC 201
  LCF 128
PID, LCF 128
pointer
  unqualified 253
pool_strings 327
pop 312
position-independent code 201, 224, 225
-ppopt 75
-pragmas 57
pragma_prepdump 313
pragmas ??–345
  access_errors 249
  always_import 317
  always_inline 249
  ANSI_strict 243
  arg_dep_lookup 249
  ARM_scoping 250
  array_new_delete 251
  asm_poundcomment 275
  asmsemicolcomment 275
  auto_inline 251
  bool 251
  c99 243
  check_header_flags 307
  codeColdFire 341
  const_multiply 342
  const_strings 276
  cplusplus 252
  cpp_extensions 252
  debuginline 253
  def_inherited 254
  defer_defarg_parsing 255
  define_section 339–341
  dollar_identifiers 276
  dont_inline 256
  dont_reuse_strings 321
  eplusplus 256
  emac 342
  enumalwayssint 322
  errno_name 323
  exceptions 257
  explicit_zero_data 342
  explicit_zero_data 323
  export 317
  extended_errorcheck 257, 283
  faster_pch_gen 307
  flat_include 308
  float_constants 324
  force_active 318
  fullpath_file 308
  fullpath_prepdump 309
  gcc_extensions 277
  global_optimizer 331
  ignore_oldstyle 245
  import 318
  inline_bottom_up 259
  inline_bottom_up_once 260
  inline_depth 260
  inline_intrinsics 343
  inline_max_auto_size 261
  inline_max_size 261
  inline_max_total_size 262
  instmgr_file 324
  internal 262
  interrupt 343
  ipa 331
  keepcomments 309
  lib_export 319
  line_prepdump 309
  longlong 325
  longlong_enums 325
  macro_prepdump 310
  mark 278
  maxerrorcount 284
  message 285
  min_enum_size 326
  mpwc_newline 278
  mpwc_relax 279
  msg_show_lineref 310
  msg_show_realref 310
  multibyteaware 279
  multibyteaware_preserve_litera1s 280
  new_mangler 263
  no_conststringconv 263
  no_static_dtors 264
  nosyminline 264
notonce 311
old_pods 265
once 311
only_std_keywords 245
opt_classresults 265
opt_common_subs 332
opt_dead_assignments 332
opt_dead_code 333
opt_lifetimes 333
opt_loop_invariants 333
opt_loop_invariants 333
opt_propagation 334
opt_strength_reduction 334
opt_strength_reduction_strict 335
opt_unroll_count 345
opt_unroll_count 345
optimization_level 336
optimize_for_size 336
optimizewithasm 337
options 326
parse_func_templ 266
parse_mfunc_templ 266
pool_strings 327
pop 312
pragma_prepdump 313
precompile_target 313
profile 345
push 312
readonly_strings 344
readonly_strings 328
require_prototypes 246
reverse_bitfields 328
RTTI 267
scope of 240
SDS_debug_support 339
section 344
show_error_filestack 286
showmessagenumber 285
simple_prepdump 314
space_prepdump 314
srcrelincludes 315
store_object_files 329
strictheaderchecking 337
suppress_init_code 267
suppress_warnings 286
sym 286
syspath_once 315
template_depth 268
text_encoding 280
thread_safe_init 268
trigraphs 281
unsigned_char 282
unused 287
warn_any_ptr_int_conv 289
warn_emptydecl 290
warn_extracomma 291
warn_filenamecaps 291
warn_filenamecaps_system 292
warn_hiddenlocals 293
warn_hidevirtual 269
warn_illpragma 293
warn_illtokenpasting 294
warn_illunionmembers 294
warn_impl_f2i_conv 295
warn_impl_i2f_conv 295
warn_impl_s2u_conv 296
warnImplicitConv 297
warn_largeargs 298
warn_missingreturn 298
warn_no_explicit_virtual 270
warn_no_side_effect 299
warn_no_typename 271
warn_notinlined 271
warn_padding 299
warn_pch_portability 300
warn_possumwant 300
warn_ptr_int_conv 302
warn_resultnotused 302
warn_structclass 272
warn_undefmacro 303
warn_uninitializedvar 304
warn_unusedarg 304
warn_unusedvar 305
warning 288
warning_errors 289
wchar_type 272
pragmas, deprecated
always_inline 249
def_inherited 254
multibyteaware 279
-precompile 74
Precompile command 164
precompile_target 313
precompiling
C++ 149
including files 149
-prefix 75
-preprocess 74
__PRETTY_FUNCTION__ 150
profile pragma 345
-progress 62
prototypes
and old-style declarations 245
not requiring 245
requiring 29
push 312

R
readonly_strings 328
readonly_strings pragma 344
reduced working set libraries 218, 219
REF_INCLUDE linker command 112
register variables 200
-relax_pointers 57
Require Function Prototypes option 29
require_prototypes 246
-requireprotos 57
return inline assembly directive 196
return statement
empty 258, 283
returning from a routine, inline assembly 191
reverse_bitfields 328
ROM-RAM copying, LCF 129–130
-RTTI 50
RTTI 267
runtime code 205–226
runtime libraries 220–226
C files 221
C floating point files 222
C++, E++ files 223
C++, E++ floating point files 224
run-time type information (RTTI) 26, 50, 267

S
-S 62
SDS_debug_support pragma 339
-search 58
section pragma 344
SECTIONS linker command 112, 113
sections segment, LCF 122
semicolon
accidental 301
serial I/O libraries 217
settings panel
C/C++ Warnings 32
show_error_filestack 286
showmessagenumber 285
simple_prepdump 314
SIZEOF linker command 113
SIZEOF_ROM linker command 114
-som 50
-som_env_check 50
space_prepdump 314
srcrelinccludes 315
S-record comments, linker 120
stack size 175, 177
stack, LCF 127
statements
catch 26, 257
for 301
if 301
return 258, 283
throw 26
try 26, 257
while 301
static initializers, LCF 127
static libraries
See libraries.
-stderr 62
-stdinc 76
-stdkeywords 45
store_object_files 329
strength reduction 180
-strict 46
strictheaderchecking 337
-strings 83
strings
    reusing 30
struct keyword
    anonymous 253
    unnamed 253
structure, LCF 120–??
suppress_init_code 267
suppress_warnings 286
sym 286
syntax
    inline assembly 187–189
    LCF 123–132
syspath_once 315
System control panel 39

T
Target Settings window 39
template 151
template_depth 268
templates
    reducing object code size 149
terminate() 257
text_encoding 280
thread_safe_init 268
try statement 26
    -timing 63
trigraph characters 28
    -trigraphs 58
trigraphs 281
try statement 26, 257
type
    char 30
    double 324
    float 324
    unsigned char 30
typeid keyword 26, 267
typename 151
typeof 277

U
    -U+ 76
UART libraries 217
    -undefine 77
unsigned char type 30
unsigned_char 282
unused 287

V
variable
    reducing 177
variables
    allocation 200
    LCF 123, 124
    register 200
    -verbose 63
    -version 63

W
    warn_any_ptr_int_conv 289
    warn_emptydeclearm 290
    warn_extracomma 291
    warn_filename 291
    warn_filenamecaps 291
    warn_filenamecaps_system 292
    warn_hiddenlocals 293
    warn_hidervirtual 269
    warn_illpragma 293
    warn_illtokenpasting 294
    warn_illunionmembers 294
    warn_impl_f2i_conv 295
    warn_impl_i2f_conv 295
    warn_impl_s2u_conv 296
    warn_impliteralconv 297
    warn_largeargs 298
    warn_missingreturn 298
    warn_no_explicit_virtual 270
    warn_no_typename 271
    warn_notinlined 271
    warn_padding 299
    warn_pch_portability 300
    warn_possunwant 300
    warn_ptr_int_conv 302
    warn_resultnotused 302
warn_structclass 272
warn_undefmacro 303
warn_uninitializedvar 304
warn_unusedarg 304
warn_unusedvar 305
warning 288
warning pragma 64, 65, 66, 67
warning_errors 289
-warnings 64
warnings
    setting in the IDE 32
-wchar_t 50
wchar_type 272
while statement 301
Windows™ operating system 39
-wraplines 67
WRITE0COMMENT linker command 115
WRITEB linker command 114
WRITEH linker command 114
WRITEW linker command 115
writing to memory, LCF 130, 132

Z
ZERO_FILLED_UNINITIALIZED linker
    command 116