



# CodeWarrior™ Build Tools Reference for Freescale™ 56800/E Digital Signal Controllers

Revised: 17 June 2006





Freescale, the Freescale logo, and CodeWarrior are trademarks or registered trademarks of Freescale Corporation in the United States and/or other countries. All other trade names and trademarks are the property of their respective owners.

Copyright © 2006 by Freescale Semiconductor company. All rights reserved.

**No portion of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, without prior written permission from Freescale. Use of this document and related materials is governed by the license agreement that accompanied the product to which this manual pertains. This document may be printed for non-commercial personal use only in accordance with the aforementioned license agreement. If you do not have a copy of the license agreement, contact your Freescale representative or call 1-800-377-5416 (if outside the U.S., call +1-512-996-5300).**

Freescale reserves the right to make changes to any product described or referred to in this document without further notice. Freescale makes no warranty, representation or guarantee regarding the merchantability or fitness of its products for any particular purpose, nor does Freescale assume any liability arising out of the application or use of any product described herein and specifically disclaims any and all liability. **Freescale software is not authorized for and has not been designed, tested, manufactured, or intended for use in developing applications where the failure, malfunction, or any inaccuracy of the application carries a risk of death, serious bodily injury, or damage to tangible property, including, but not limited to, use in factory control systems, medical devices or facilities, nuclear facilities, aircraft navigation or communication, emergency systems, or other applications with a similar degree of potential hazard.**

## How to Contact Freescale

<b>Corporate Headquarters</b>	Freescale Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
<b>World Wide Web</b>	<a href="http://www.freescale.com/codewarrior">http://www.freescale.com/codewarrior</a>
<b>Technical Support</b>	<a href="http://www.freescale.com/support">http://www.freescale.com/support</a>

# Table of Contents

---

<b>1</b>	<b>Introduction</b>	<b>13</b>
	Compiler Architecture . . . . .	13
	Linker Architecture . . . . .	14
<b>2</b>	<b>Using Build Tools with the CodeWarrior IDE</b>	<b>15</b>
	Invoking CodeWarrior Compilers and Linkers . . . . .	15
	Specifying File Locations . . . . .	15
	IDE Options and Pragmas . . . . .	16
	IDE Settings Panels . . . . .	16
	C/C++ Language (C only) Settings Panel . . . . .	16
	C/C++ Preprocessor Panel . . . . .	20
	C/C++ Warnings Panel . . . . .	22
<b>3</b>	<b>Using Build Tools on the Command Line</b>	<b>27</b>
	Naming Conventions . . . . .	28
	Configuring Command-Line Tools . . . . .	28
	CWFold Environment Variable . . . . .	28
	Setting the PATH Environment Variable . . . . .	29
	Invoking Command-Line Tools . . . . .	29
	Getting Help . . . . .	30
	Help Guidelines . . . . .	30
	File Name Extensions . . . . .	31
	Specifying Source File Locations . . . . .	32
	Environmental Variables . . . . .	32
	Standard C and C++ Conformance Options . . . . .	33
	-ansi . . . . .	33
	-stdkeywords . . . . .	34
	-strict . . . . .	34
	Language Translation and Extensions Options . . . . .	34
	-char . . . . .	35
	-defaults . . . . .	36
	-encoding . . . . .	36



## Table of Contents

---

-flag	.37
-gccext	.37
-gcc_extensions	.38
-M	.38
-make	.38
-mapcr	.38
-MM	.39
-MD	.39
-MMD	.39
-multibyteaware	.40
-once	.40
-pragma	.40
-relax_pointers	.41
-requireprotos	.41
-search	.41
-trigraphs	.41
Errors, Warnings, and Diagnostic Options	.42
-disassemble	.42
-help	.43
-maxerrors	.44
-maxwarnings	.44
-msgstyle	.45
-nofail	.45
-progress	.45
-S	.46
-stderr	.46
-verbose	.46
-version	.47
-timing	.47
-warnings	.47
-wraplines	.50
Preprocessing and Precompilation Options	.50
-convertpaths	.51
-cwd	.52
-D+	.52

---

---

-define . . . . .	53
-E . . . . .	53
-EP . . . . .	53
-gccincludes . . . . .	54
-I . . . . .	54
-I+ . . . . .	55
-include . . . . .	55
-ir . . . . .	55
-noprecompile . . . . .	56
-nosyspath . . . . .	56
-P . . . . .	56
-precompile . . . . .	56
-preprocess . . . . .	57
-ppopt . . . . .	57
-prefix . . . . .	58
-stdinc . . . . .	58
-U+ . . . . .	59
-undefine . . . . .	59
Library and Linking Options . . . . .	59
-keepobjects . . . . .	59
-map showbyte . . . . .	60
-nolink . . . . .	60
-o . . . . .	60
Object Code Organization and Generation Options . . . . .	61
-c . . . . .	61
-codegen . . . . .	61
-enum . . . . .	62
-ext . . . . .	62
-strings . . . . .	63
Optimization Options . . . . .	63
-factor1 . . . . .	64
-factor2 . . . . .	64
-factor3 . . . . .	64
-inline . . . . .	64
-ipa . . . . .	65

---

## Table of Contents

---

-nofactor1 . . . . .	.66
-nofactor2 . . . . .	.66
-nofactor3 . . . . .	.66
-O . . . . .	.66
-O+ . . . . .	.67
-opt . . . . .	.68
<b>4 Linker</b>	<b>71</b>
Introduction . . . . .	.71
<b>5 C</b>	<b>73</b>
Extensions to Standard C . . . . .	.73
Unnamed Arguments in Function Definitions . . . . .	.74
C++ Comments . . . . .	.74
A # Not Followed by a Macro Argument . . . . .	.74
Using an Identifier After #endif . . . . .	.75
Using Typecasted Pointers as Ivalues . . . . .	.75
Inline Functions . . . . .	.76
Pascal Calling Conventions . . . . .	.76
Character Constants as Integer Values . . . . .	.76
Converting Pointers to Types of the Same Size . . . . .	.76
Getting Alignment and Type Information at Compile Time . . . . .	.77
Arrays of Zero Length in Structures . . . . .	.77
The “D” Constant Suffix . . . . .	.77
The __typeof__() and typeof() operators . . . . .	.77
Implementation-Defined Behavior . . . . .	.78
Diagnostic Messages . . . . .	.78
Identifiers . . . . .	.78
<b>6 Tool Performance</b>	<b>79</b>
Precompiled Header Files . . . . .	.79
When to Use Precompiled Files . . . . .	.79
What Can be Precompiled . . . . .	.80
Precompiling C++ Source Code . . . . .	.80
Using a Precompiled Header File . . . . .	.81

---

---

Preprocessing and Precompiling . . . . .	82
Pragma Scope in Precompiled Files . . . . .	82
Precompiling a File in the CodeWarrior IDE . . . . .	83
Updating a Precompiled File Automatically . . . . .	83
<b>7 Optimization</b>	<b>85</b>
Optimization Considerations . . . . .	85
Inlining . . . . .	85
Profiling . . . . .	86
String Literals . . . . .	86
Pooling Strings . . . . .	86
Reusing Strings . . . . .	86
Optimizations . . . . .	87
Dead Code Elimination . . . . .	88
Expression Simplification . . . . .	88
Common Subexpression Elimination . . . . .	89
Copy Propagation . . . . .	89
Dead Store Elimination . . . . .	90
Live Range Splitting . . . . .	91
Loop-Invariant Code Motion . . . . .	91
Strength Reduction . . . . .	92
Loop Unrolling . . . . .	93
M56800E Specific Optimizations . . . . .	93
<b>8 Inline Assembly Language and Intrinsics</b>	<b>113</b>
Introduction . . . . .	113
<b>9 Predefined Symbols</b>	<b>115</b>
Using Predefined Symbols . . . . .	115
Version Symbol . . . . .	115
__MWERKS__ . . . . .	115
Date and Time Symbol . . . . .	116
__DATE__ . . . . .	116
__TIME__ . . . . .	116
IDE Symbol . . . . .	116

---

## Table of Contents

---

__ide_target("target_name")	116
Name Symbols	117
__FILE__	117
__LINE__	117
Object Code Organization and Generation Symbol	117
__profile__	117
C Symbols	118
__STDC__	118
<b>10 Pragmas</b>	<b>119</b>
Using Pragmas	119
Checking Pragma Settings	119
Saving and Restoring Pragma Settings	124
Determining Which Settings Are Saved and Restored	126
Illegal Pragmas	126
Pragma Scope	127
Standard C and C++ Conformance Pragmas	127
ANSI_strict	127
only_std_keywords	129
Language Translation and Extensions Pragmas	129
gcc_extensions	129
mpwc_newline	130
mpwc_relax	131
Errors, Warnings, and Diagnostic Control Pragmas	132
check_c_src_pipeline	133
check_inline_asm_pipeline	133
check_inline_sp_effects	134
extended_errorcheck	134
require_prototypes	135
suppress_init_code	135
suppress_warnings	136
unsigned_char	136
unused	137
warn_any_ptr_int_conv	138
warn_emptydecl	138

---



---

warn_extracomma . . . . .	139
warn_filenameecaps . . . . .	140
warn_filenameecaps_system . . . . .	140
warn_illpragma . . . . .	141
warn_impl_f2i_conv . . . . .	141
warn_impl_i2f_conv . . . . .	142
warn_impl_s2u_conv . . . . .	143
warn_implicitconv . . . . .	144
warn_largeargs . . . . .	145
warn_missingreturn . . . . .	145
warn_no_side_effect . . . . .	146
warn_notinlined . . . . .	146
warn_padding . . . . .	147
warn_possunwant . . . . .	147
warn_ptr_int_conv . . . . .	148
warn_resultnotused . . . . .	148
warn_undefmacro . . . . .	149
warn_unusedarg . . . . .	150
warn_unusedvar . . . . .	150
warning_errors . . . . .	150
Preprocessing and Precompilation Pragmas . . . . .	151
dollar_identifiers . . . . .	151
fullpath_prepdump . . . . .	152
mark . . . . .	152
notonce . . . . .	153
once . . . . .	153
pop, push . . . . .	153
syspath_once . . . . .	154
Library and Linking Control Pragmas . . . . .	155
define_section . . . . .	155
explicit_zero_data . . . . .	156
initializedzerodata . . . . .	157
section . . . . .	157
use_rodata . . . . .	158
Object Code Organization and Generation Pragmas . . . . .	160

---

## Table of Contents

---

always_inline . . . . .	161
auto_inline . . . . .	161
const_strings . . . . .	161
defer_codegen . . . . .	162
dont_inline . . . . .	163
dont_reuse_strings . . . . .	163
enumsalwaysint . . . . .	164
inline_bottom_up . . . . .	165
interrupt (for the DSP56800) . . . . .	166
interrupt (for the DSP56800E) . . . . .	168
packstruct . . . . .	172
pool_strings . . . . .	172
readonly_strings . . . . .	173
reverse_bitfields . . . . .	173
suppress_init_code . . . . .	174
syspath_once . . . . .	174
Optimization Pragmas . . . . .	175
factor1 . . . . .	176
factor2 . . . . .	176
factor3 . . . . .	176
nofactor1 . . . . .	177
nofactor2 . . . . .	177
nofactor3 . . . . .	178
opt_common_subs . . . . .	178
opt_dead_assignments . . . . .	179
opt_dead_code . . . . .	179
opt_lifetimes . . . . .	179
opt_loop_invariants . . . . .	180
opt_propagation . . . . .	180
opt_strength_reduction . . . . .	181
opt_strength_reduction_strict . . . . .	181
opt_unroll_loops . . . . .	182
optimization_level . . . . .	182
optimize_for_size . . . . .	182
peephole . . . . .	183

---



## Table of Contents

---

Profiler Pragmas .....	183
profile .....	184
<b>Index</b>	<b>185</b>



## Table of Contents

---

# Introduction

---

This reference describes how to use the CodeWarrior compiler and linker tools to build software.

CodeWarrior build tools are programs that 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. The host platform is the machine on which CodeWarrior build tools run. The target platform is the machine on which the software generated by the build tools runs.

This section introduces how CodeWarrior build tools are organized:

- [Compiler Architecture on page 13](#)
- [Linker Architecture on page 14](#)

## Compiler Architecture

From your perspective, a CodeWarrior compiler is a single program. Internally, however, a CodeWarrior compiler has two parts:

- the front-end, shared by all CodeWarrior compilers, translates human-readable source code into a platform-independent intermediate representation of the program being compiled
- the back-end, customized to generate software for a target platform, converts the intermediate representation into object code containing data and native instructions for the target processor

A CodeWarrior compiler coordinates its front-end and back-end to translate source code into object code in several steps:

- configure settings requested from the compiler to the CodeWarrior IDE or passed to the linker from the command-line
- translate human-readable source code into an intermediate representation
- optionally output symbolic debugging information
- optimize the intermediate representation
- convert the intermediate representation to native object code
- optimize the native object code
- output the native, optimized object code

# Linker Architecture

A linker combines and arranges the object code in libraries and object code generated by compilers and assemblers into a single file or image, ready to execute on the target platform. The CodeWarrior linker builds an executable image in several steps:

- configure settings requested from the linker to the CodeWarrior IDE or passed to the linker from the command-line
- read settings from a linker control file
- read object code
- search for and ignore unused objects (“deadstripping”)
- build and output the executable file
- optionally output a map file

# 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 on page 15](#)
- [Specifying File Locations on page 15](#)
- [IDE Options and Pragmas on page 16](#)
- [IDE Settings Panels on page 16](#)

## Invoking CodeWarrior Compilers and Linkers

The IDE uses settings in the Target Settings panel of the Target Settings window to determine which compilers and linkers to use for a project's build target. 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 Target Settings window to determine which types of files may be added to a project's build target and which compiler plugin 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 directives and updates their settings as pragmas are encountered.

## IDE Settings Panels

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

- [C/C++ Language \(C only\) Settings Panel on page 16](#)
- [C/C++ Preprocessor Panel on page 20](#)
- [C/C++ Warnings Panel on page 22](#)

### C/C++ Language (C only) Settings Panel

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

- [Inline Depth on page 17](#)
- [Auto-Inline on page 17](#)
- [Interprocedural Analysis Support on page 17](#)
- [Bottom-up Inlining on page 18](#)
- [ANSI Strict on page 18](#)
- [ANSI Keywords Only on page 19](#)
- [Expand Trigraphs on page 19](#)



- [Legacy for-scoping on page 19](#)
- [Require Function Prototypes on page 19](#)
- [Enums Always Int on page 20](#)
- [Enums Always Int on page 20](#)
- [Use Unsigned Chars on page 20](#)
- [Pool Strings on page 20](#)
- [Reuse Strings on page 20](#)

## 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 on page 17](#)

**Table 2.1 Settings for the Inline Depth Pop-up Menu**

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

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`.

## Interprocedural Analysis Support

Interprocedural Analysis (IPA) allows the compiler to generate better and smaller code by inspecting more than just one function or data object at the same time. The compiler supports three different interprocedural analysis modes:

- [Function Mode \(aka "off"\) \(default\) on page 18](#)
- [File Mode on page 18](#)

### Function Mode (aka "off") (default)

Functions are optimized and code is generated when the function has been parsed. This mode allows no interprocedural analysis. This mode is enabled by selecting **off** in the IPA popup menu in the **C/C++ Language (C only)** preference panel or by specifying `-ipa function` or `-ipa off` on the command line.

### File Mode

A translation unit is completely parsed before any code or data is generated. This allows optimizations and inlining on a per-file basis, it replaces the **deferred inlining/codegen** mode. This mode is enabled by selecting `file` in the IPA popup menu in the **C/C++ Language (C only)** preference panel or by specifying `-ipa file` on the command line. This mode will require more memory and it can be slightly slower than **Function** mode. The compiler will also do an early dead code/data analysis in this mode, so objects with internal linkage that are not referenced will be dead-stripped in the compiler rather than in the linker.

### 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`.

### ANSI Strict

Only recognizes source code that conforms to the ISO/ANSI standards. 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 `enumsalwaysint` 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 C, §6.4.1) The CodeWarrior compiler can recognize several additional reserved keywords. If you enable this setting, the compiler generates an error 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: `inline`, `__inline__`, `__inline`, and `pascal`.

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

## Expand Trigraphs

(ISO 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 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_conform` and the command-line option `-for_scoping`.

## Require Function Prototypes

Enforce the requirement of function prototypes. If you enable the **Require Function Prototypes** setting, the compiler generates an error message if you define a previously

## Using Build Tools with the CodeWarrior IDE

### IDE Settings Panels

---

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`.

## Enums Always Int

Uses signed integers to represent enumerated constants. This option corresponds to the `enumsalwaysint` pragma and the command-line option `-enum`.

## 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 object in the object code it generates. If you disable this setting, the compiler creates a unique data object 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.

- [Source encoding on page 21](#)
- [Use prefix text in precompiled header on page 21](#)
- [Emit file changes on page 21](#)
- [Emit #pragmas on page 21](#)

- [Show full paths on page 21](#)
- [Keep comments on page 21](#)
- [Use #line on page 22](#)
- [Keep whitespace on page 22](#)

## 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 preprocess request appear in this panel.

## Use prefix text in precompiled header

Controls whether a \*.pch or \*.pch++ file incorporates the prefix text into itself.

This option defaults to “off” 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 set to “on”.

## Emit file changes

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

## Emit #pragmas

Controls whether #pragmas 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.

## 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 controls which warning messages the CodeWarrior C/C++ compiler issues as it translates source code:

- [Illegal Pragmas on page 23](#)
- [Possible Errors on page 23](#)
- [Extended Error Checking on page 23](#)
- [Implicit Arithmetic Conversions on page 23](#)
- [Float To Integer on page 24](#)
- [Signed/Unsigned on page 24](#)
- [Integer To Float on page 24](#)
- [Pointer/Integral Conversions on page 24](#)
- [Unused Variables on page 24](#)
- [Unused Arguments on page 24](#)
- [Missing 'return' Statements on page 25](#)
- [Expression Has No Side Effect on page 25](#)
- [Enable All on page 25](#)
- [Disable All on page 25](#)
- [Extra Commas on page 25](#)
- [Inconsistent 'class'/'struct' Usage on page 25](#)
- [Empty Declarations on page 25](#)
- [Include File Capitalization on page 26](#)
- [Check System Includes on page 26](#)
- [Pad Bytes Added on page 26](#)
- [Undefined Macro in #if on page 26](#)
- [Non-Inlined Functions on page 26](#)

- [Treat All Warnings As Errors on page 26](#)

## 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, 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`.

## 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
- 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, anyptrintconv`.

## 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 `warn_unusedarg` pragma and the command-line option `-warnings unusedarg`.



## Missing 'return' Statements

Issues a warning message if a function that is defined to return a value has no return statement.

This setting corresponds to the `warn_missingreturn` pragma and the command-line option `-warnings 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 `warn_no_side_effect` pragma and the command-line option `-warnings 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.

This setting corresponds to the `warn_extracomma` pragma and the command-line option `-warnings 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 `warn_structclass` pragma and the command-line option `-warnings 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 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 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 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 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`.

## 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 on the Command Line

---

The CodeWarrior command line compilers and assemblers translate source code (for example, C and C++) into object code, storing this object 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 Freescale 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:

- [Naming Conventions on page 28](#)
- [Configuring Command-Line Tools on page 28](#)
- [Invoking Command-Line Tools on page 29](#)
- [Getting Help on page 30](#)
- [File Name Extensions on page 31](#)
- [Specifying Source File Locations on page 32](#)
- [on page 32Environmental Variables on page 32](#)
- [Standard C and C++ Conformance Options on page 33](#)
- [Language Translation and Extensions Options on page 34](#)
- [Errors, Warnings, and Diagnostic Options on page 42](#)
- [Preprocessing and Precompilation Options on page 50](#)
- [Library and Linking Options on page 59](#)
- [Library and Linking Options on page 59](#)
- [Object Code Organization and Generation Options on page 61](#)
- [Optimization Options on page 63](#)

## Naming Conventions

The names of the CodeWarrior command-line tools follow a convention:

`mw tool platform`

where *tool* is `cc` for the C/C++ compiler, `ld` for the linker, and `asm` for the assembler.

*Platform* is usually the target platform that the tool generates software for, except where there are multiple versions of tools for a target platform.

For example, the command-line compiler, assembler, and linker for the dsp56800 are named `mwcc56800`, `mwas56800`, and `mwld56800`, respectively; and for the dsp56800e are named `mwcc56800e`, `mwas56800e`, and `mwld56800e`, respectively;.

## 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 or in the **Environment** tab under the **System** control panel in Windows NT/2000/XP.

The CodeWarrior command-line tools refer to environment variables for configuration information:

- [CWFoldEr Environment Variable on page 28](#)
- [Setting the PATH Environment Variable on page 29](#)

### CWFoldEr Environment Variable

In this example, `%CWFoldEr%` refers to the path where CodeWarrior for 56800 was installed. Note that it is not necessary to include quote marks when defining environment variables that include spaces. Windows does not strip out the quotes and this leads to unknown directory warnings. Use the following syntax if defining variables in batch files or at the command line ([Listing 3.1 on page 28](#)).

#### Listing 3.1 Example of setting CWFoldEr.

---

```
set CWFoldEr=C:\Program Files\Freescale\CodeWarrior
```

---

---

## Setting the PATH Environment Variable

The PATH variable should include the paths for the 56800 tools (other tools will be different), shown in [Listing 3.2 on page 29](#).

### Listing 3.2 Example of PATH Settings

---

```
%CWFold%Bin
%CWFold%\DSP56800x_EABI_Tools\Command_Line_Tools
```

---

The first path in [Listing 3.2 on page 29](#) 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:

```
%CWFold%\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

```
tool options files
```

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

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

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

## Using Build Tools on the Command Line

### Getting Help

---

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
```

## Parameter Formats

Parameters in an option are formatted as follows:

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

## 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 OS.
- 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 but also emits a warning. By default, the compiler assumes that a file

## Using Build Tools on the Command Line

### Specifying Source File Locations

---

with any extensions besides `.c`, `.h`, `.pch` is C++ source. The linker ignores all files that it can not 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, for 56800, see [Listing 3.3 on page 32](#).

#### Listing 3.3 Example of using linker command files

---

```
mwld56800e file.o "MSL C 56800E.lib" "Runtime 56800E.Lib" linker.cmd
```

---

For more information on linker command files, refer to the *Targeting* manual for your platform.

## Specifying Source File Locations

Several environment variables are used at build time to search for system include paths and libraries which can shorten command lines for many tasks. All of the variables mentioned here are lists which are separated by semicolons (“;”) in Windows and colons (“:”) in Solaris.

For example, in 56800, unless `-nodefaults` is passed to on the command line, the compiler searches for an environment variable called `MWC56800Includes` for the DSP56800 and `MWC56800EIncludes` for the DSP56800E. This variable contains a list of system access paths to be searched after the system access paths specified by the user. The assembler also does this, using the variable `MWAsm56800Includes` for the DSP56800 and `MWAsm56800EIncludes` for the DSP56800E.

Analogously, unless `-nodefaults` or `-disassemble` is given, the linker will search the environment for a list of system access paths and system library files to be added to the end of the search and link orders. For example, with 56800, the variable `MW56800Libraries` and `MW56800ELibraries` contains a list of system library paths to search for files, libraries, and command files.

Associated with this list is the variable `MW56800LibraryFiles` and `MW56800ELibraryFiles` which contains a list of libraries (or object files or command files) to add to the end of the link order. These files may be located in any of the cumulative access paths at runtime.

## Environmental Variables

There are environmental variable for the DSP56800 and DSP56800E.

The environmental variables for the DSP56800 are:

- `MW56800Libraries`: a semicolon sperated list of paths to the libraries



- MW56800LibraryFiles: a semicolon separated list of libraries to be linked against
- MWAsm56800Includes: a semicolon separated list of paths to files needed by the assembler
- MWC56800Includes: a semicolon separated list of paths to files needed by the assembler

The environmental variables for the DSP56800E are:

- MW56800ELibraries: a semicolon sperated list of paths to the libraries
- MW56800ELibraryFiles: a semicolon separated list of libraries to be linked against
- MWAsm56800EIncludes: a semicolon separated list of paths to files needed by the assembler
- MWC56800EIncludes: a semicolon separated list of paths to files needed by the assembler

## Standard C and C++ Conformance Options

The Standard C and C++ Conformance options are:

- [-ansi on page 33](#)
- [-stdkeywords on page 34](#)
- [-strict on page 34](#)

---

### **-ansi**

Controls the ANSI conformance options, overriding the given settings.

#### **Syntax**

`-ansi keyword`

The arguments for `keyword` are:

`off`

Turn ANSI conformance off. Same as `-stdkeywords off`, `-enum min`, and `-strict off`.

`on` | `relaxed`

Turn ANSI conformance on in relaxed mode. Same as `-stdkeywords on`, `-enum min`, and `-strict on`.

## Using Build Tools on the Command Line

### Language Translation and Extensions Options

---

`strict`

Turn ANSI conformance on in strict mode. Same as `-stdkeywords on`, `-enum int`, and `-strict on`.

---

### **-stdkeywords**

Controls the requirement for the use of ANSI standard keywords.

#### **Syntax**

`-stdkeywords on | off`

#### **Remarks**

Default setting is `off`.

---

### **-strict**

Controls the use of non-standard ANSI language features.

#### **Syntax**

`-strict on | off`

#### **Remarks**

Default setting is `off`.

---

## Language Translation and Extensions Options

The Language Translation and Extensions options are:

- [-char on page 35](#)
  - [-defaults on page 36](#)
  - [-encoding on page 36](#)
  - [-flag on page 37](#)
  - [-gccext on page 37](#)
  - [-gcc\\_extensions on page 38](#)
-

- [-M on page 38](#)
  - [-make on page 38](#)
  - [-mapcr on page 38](#)
  - [-MM on page 39](#)
  - [-MD on page 39](#)
  - [-MMD on page 39](#)
  - [-multibyteaware on page 40](#)
  - [-once on page 40](#)
  - [-pragma on page 40](#)
  - [-relax\\_pointers on page 41](#)
  - [-relax\\_pointers on page 41](#)
  - [-requireprotos on page 41](#)
  - [-search on page 41](#)
  - [-trigraphs on page 41](#)
- 

### **-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`.

## Using Build Tools on the Command Line

### Language Translation and Extensions Options

---

#### **-defaults**

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

#### **Syntax**

```
-defaults
-[no]defaults
```

#### **Remarks**

This command is global. To enable 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`

Use local system format.

`UTF[8` | `-8]`

Unicode Transformation Format (UTF).

`SJIS` | `Shift-JIS` | `ShiftJIS`

Shift Japanese Industrial Standard (Shift-JIS) format.

---

EUC[JP | -JP]

Japanese Extended UNIX Code (EUCJP) format.

ISO[2022JP | -2022-JP]

International Organization of Standards (ISO) Japanese format.

#### Remarks

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

---

## -flag

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

#### Syntax

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

#### Examples

`-flag foo`

is equivalent to `#pragma foo on`.

`-flag no-foo`

is the same as `#pragma foo off`.

---

## -gccext

Enable GCC (Gnu Compiler Collection) C language extensions.

#### Syntax

`-gcc[ext] on | off`

#### Remarks

The default setting is `off`.

---

## Using Build Tools on the Command Line

### *Language Translation and Extensions Options*

---

#### **-gcc\_extensions**

Equivalent to the `-gccext` option.

##### **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.

---

### **-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.

---

## Using Build Tools on the Command Line

### Language Translation and Extensions Options

---

#### Syntax

-MMD

#### Remarks

This command is global and case-sensitive.

---

## -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[requireprotos]`

---

## **-search**

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

### **Syntax**

`-search`

---

## **-trigraphs**

Controls the use of ISO trigraph sequences.

---

## Using Build Tools on the Command Line

### Errors, Warnings, and Diagnostic Options

---

#### Syntax

`-trigraphs on | off`

#### Remarks

Default setting is `off`.

## Errors, Warnings, and Diagnostic Options

The Errors, Warnings, and Diagnostic options are:

- [-disassemble on page 42](#)
- [-help on page 43](#)
- [-maxerrors on page 44](#)
- [-maxwarnings on page 44](#)
- [-nofail on page 45](#)
- [-progress on page 45](#)
- [-S on page 46](#)
- [-stderr on page 46](#)
- [-verbose on page 46](#)
- [-version on page 47](#)
- [-timing on page 47](#)
- [-warnings on page 47](#)
- [-wraplines on page 50](#)

---

### **-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); for 'keyword', maximum length 63 chars

`[no] compatible`

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

`[no] deprecated`

Show deprecated options

`[no] ignored`

Show ignored options

`[no] meaningless`

Show options meaningless for this target

`[no] normal`

Show only standard options

`[no] obsolete`

Show obsolete options

`[no] spaces`

Insert blank lines between options in printout.

`opt [ion]=name`

Show help for a given option; for 'name', maximum length 63 chars

`search=keyword`

Show help for an option whose name or help contains 'keyword' (case-sensitive); for 'keyword', maximum length 63 chars

## Using Build Tools on the Command Line

### Errors, Warnings, and Diagnostic Options

---

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

Categorize 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 to show.

#### Syntax

`-maxerrors max`

`max`

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

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

### -maxwarnings

Specify the maximum number of warnings to show.

#### Syntax

`-maxerrors max`

`max`

Use `max` to specify the number of warnings. 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 `gcc` message style.

`ide`

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

`mpw`

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

`parseable`

Uses context-free machine parseable message style.

`std`

Uses standard message style. This is the default.

---

### **-nofail**

Continue processing after getting errors in earlier files.

#### **Syntax**

`-nofail`

---

### **-progress**

Show progress and version information.

#### **Syntax**

`-progress`

## Using Build Tools on the Command Line

### Errors, Warnings, and Diagnostic Options

---

#### **-S**

Disassemble 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  
-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 verbose, cumulative information in messages.

#### **Syntax**

-v [erbose]

#### **Remarks**

Use of this argument implies the use of the [-progress on page 45](#) argument.

---

#### -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`

---

#### -warnings

Specify which warnings the command-line tool issues. This command is global.

##### Syntax

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

The options for keyword are:

`off`

Turn off all warnings. Passed to all tools. Prefix file setting: `#pragma warning off`.

`on`

Turn on most warnings. Passed to all tools. Prefix file setting: `#pragma warning on`.

`[no] cmdline`

passed to all tools; # command-line driver/parser warnings

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

Treat warnings as errors. Passed to all tools. Prefix file setting: `#pragma warning_errors`.

---

## Using Build Tools on the Command Line

### Errors, Warnings, and Diagnostic Options

---

all

Turn on all warnings and require prototypes.

[no]pragmas | [no]illpragmas

Issue warnings on illegal #pragmas. Prefix file setting: #pragma warn\_illpragma.

[no]empty[decl]

Issue warnings on empty declarations. Prefix file setting: #pragma warn\_emptydecl.

[no]possible | [no]unwanted

Issue warnings on possible unwanted effects. Prefix file setting: #pragma warn\_possunwanted.

[no]unusedarg

Issue warnings on unused arguments. Prefix file setting: #pragma warn\_unusedarg.

[no]unusedvar

Issue warnings on unused variables. Prefix file setting: #pragma warn\_unusedvar.

[no]unused

Same as -w [no]unusedarg, [no]unusedvar.

[no]extracomma | [no]comma

Issue warnings on extra commas in enumerations. Prefix file setting: #pragma warn\_extracomma.

[no]pedantic | [no]extended

pedantic error checking

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

Issue warnings on hidden virtual functions. Prefix file setting: #pragma warn\_hidevirtual.

[no]implicit[conv]

Issue warnings on implicit arithmetic conversions. Implies -warn impl\_float2int, impl\_signedunsigned.

[no]impl\_int2float

Issue warnings on implicit integral to floating conversions. Prefix file setting: #pragma warn\_impl\_i2f\_conv.



- [no]impl\_float2int  
Issue warnings on implicit floating to integral conversions. Prefix file setting: #pragma warn\_impl\_f2i\_conv.
- [no]impl\_signedunsigned  
Issue warnings on implicit signed/unsigned conversions.
- [no]notinlined  
Issue warning when inline functions are not inlined. Prefix file setting: #pragma warn\_notinlined.
- [no]largeargs  
Issue warning when passing large arguments to unprototyped functions. Prefix file setting: #pragma warn\_largeargs.
- [no]structclass  
Issue warning on inconsistent use of class and struct. Prefix file setting: #pragma warn\_structclass.
- [no]padding  
Issue warning when padding is added between struct members. Prefix file setting: #pragma warn\_padding
- [no]notused  
Issue warning when the result of non-void-returning functions are not used. Prefix file setting: #pragma warn\_resultnotused.
- [no]missingreturn  
Issue warning when a return without a value in non-void-returning function occurs. Prefix file setting: #pragma warn\_missingreturn.
- [no]unusedexpr  
Issue warning when encountering the use of expressions as statements without side effects. Prefix file setting: #pragma warn\_no\_side\_effect.
- [no]ptrintconv  
Issue warning when lossy conversions occur from pointers to integers.
- [no]anyptrintconv  
Issue warning on any conversion of pointers to integers. Prefix file setting: #pragma warn\_ptr\_int\_conv.
- [no]undef [macro]  
Issue warning on the use of undefined macros in #if/#elif conditionals. Prefix file setting: #pragma warn\_undefmacro.

## Using Build Tools on the Command Line

### Preprocessing and Precompilation Options

---

[no]filecaps

Issue warning when #include "... " statements use incorrect capitalization.  
Prefix file setting: #pragma warn\_filenameecaps.

[no]sysfilecaps

Issue warning when #include <...> statements use incorrect capitalization.  
Prefix file setting: #pragma warn\_filenameecaps\_system.

[no]tokenpasting

Issue warning when token is not formed by ## operator. Prefix file setting:  
#pragma warn\_illtokenpasting.

display | dump

Display list of active warnings.

#### Description

Choose **Edit > targetname Settings** from the CodeWarrior IDE's menu bar, then select the **C/C++ Warnings** settings panel. Enable or disable specific warnings by clicking the appropriate checkboxes.

---

## -wraplines

Controls the word wrapping of messages.

#### Syntax

-wraplines

-nowraplines

# Preprocessing and Precompilation Options

The Preprocessing and Precompilation options are:

- [-convertpaths on page 51](#)
- [-cwd on page 52](#)
- [-D+ on page 52](#)
- [-define on page 53](#)
- [-E on page 53](#)
- [-EP on page 53](#)
- [-gccincludes on page 54](#)

- [-I- on page 54](#)
- [-I+ on page 55](#)
- [-include on page 55](#)
- [-ir on page 55](#)
- [-noprecompile on page 56](#)
- [-nosyspath on page 56](#)
- [-P on page 56](#)
- [-precompile on page 56](#)
- [-preprocess on page 57](#)
- [-ppopt on page 57](#)
- [-prefix on page 58](#)
- [-stdinc on page 58](#)
- [-U+ on page 59](#)
- [-undefine on page 59](#)

---

### **-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® – 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.

## Using Build Tools on the Command Line

### Preprocessing and Precompilation Options

---

**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>`.

---

### **-cwd**

Controls where a search begins for `#include` files. The path represented by *keyword* is searched before searching access paths defined for the build target.

#### **Syntax**

`-cwd keyword`

The options for `keyword` are:

`explicit`

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

`include`

Begin search in directory of referencing file.

`proj`

Begin search in current working directory (default).

`source`

Begin search in directory that contains the source file.

---

### **-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.

### **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. This command is global and case-sensitive.

### **Syntax**

```
-E
```

---

## **-EP**

Tells the command-line tool to preprocess source files that are stripped of `#line` directives. This command is global and case-sensitive.

### **Syntax**

```
-EP
```

### **Remarks**

Output is generated using the `#pragma simple_predump` on setting and sent to a new unsaved editor window.

## Using Build Tools on the Command Line

### Preprocessing and Precompilation Options

---

#### **-gccincludes**

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

#### **Syntax**

```
-gccinc[ludes]
```

#### **Remarks**

Use `-gccinclude` to control the CodeWarrior compiler understanding of 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. This command is global.

#### **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

### **-I+**

Appends a non-recursive access path to the current `#include` list. This command is global and case-sensitive.

#### **Syntax**

`-I+path`

`-i path`

The parameters are:

`path`

The non-recursive access path to append.

---

### **-include**

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

#### **Syntax**

`-include file`

`file`

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

#### **Remarks**

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

---

### **-ir**

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

#### **Syntax**

`-ir path`

The parameters are:

---

## Using Build Tools on the Command Line

### Preprocessing and Precompilation Options

---

path

The recursive access path to append.

---

### **-noprocompile**

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

#### **Syntax**

-noprocompile

---

### **-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.

---

### **-P**

Preprocess the source files without generating object code, and send output to file. This command is global and case-sensitive.

#### **Syntax**

-P

---

### **-precompile**

Precompile a header file from selected source files.

---



#### Syntax

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

The parameters are:

*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 statement `-precompile filesource` is equivalent to `-c -o filesource`.

---

## -preprocess

Preprocess the source files. This command is global .

#### Syntax

```
-preprocess
```

---

## -ppopt

Specify options affecting the preprocessed output. The default settings is `break`.

#### Syntax

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

The arguments for *keyword* are:

`[no]break`

Emit file and line breaks. This is the default.

## Using Build Tools on the Command Line

### Preprocessing and Precompilation Options

---

[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`.

---

## **-prefix**

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

### **Syntax**

`-prefix file`

---

## **-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.

### **Syntax**

`-U+name`

---

## **-undefine**

Undefine the specified symbol name. This command is case-sensitive.

### **Syntax**

`-u [undefine] name`

`-U+name`

The parameters are:

`name`

The symbol name to undefine.

# Library and Linking Options

The Library and Linking options are:

- [-keepobjects on page 59](#)
  - [-map showbyte on page 60](#)
  - [-nolink on page 60](#)
  - [-o on page 60](#)
- 

## **-keepobjects**

Retains or deletes object files after invoking the linker.

## Using Build Tools on the Command Line

### Library and Linking Options

---

#### Syntax

`-keepobj [ects]`

`-nokeepobj [ects]`

#### Remarks

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

---

**NOTE** Object files are always kept when compiling.

---

---

## **-map showbyte**

This option activates the IDE feature with the name "Annotate Byte Symbols".

---

## **-nolink**

Compile the source files, without linking.

#### Syntax

`-nolink`

#### Remarks

This command is global.

---

## **-o**

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

#### Syntax

`-o file | dir`

The parameters are:

`file`

The output file name.

---

`dir`

The directory to store object files or text output.

#### Remarks

Choose **Edit > *targetname* Settings** from the CodeWarrior IDE's menu bar, then select the **Access Paths** settings panel. Enable the **Always Search User Paths** option.

## Object Code Organization and Generation Options

The Object Code Organization and Generation options are:

- [-c on page 61](#)
- [-codegen on page 61](#)
- [-enum on page 62](#)
- [-ext on page 62](#)
- [-strings on page 63](#)

---

### -c

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

#### Syntax

`-c`

#### Remarks

This option is global.

---

### -codegen

Controls the generation of object code.

#### Syntax

`-codegen`

## Using Build Tools on the Command Line

### Object Code Organization and Generation Options

---

`-nocodegen`

#### Remarks

This option is global.

---

## **-enum**

Specify the default size for enumeration types. Default setting is `min`.

#### Syntax

`-enum keyword`

The arguments for `keyword` are:

`int`

Use `int` size for enumerated types.

`min`

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

---

## **-ext**

Tells the command-line tool the extension to apply to object files.

#### Syntax

`-ext extension`

The value of `extension` is:

`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 == o`, then `source.cpp` becomes `source.o`.
  - Extensions specified with a leading period (`.extension`) are appended to the object files name. For example, if `extension == .o`, 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**

`-str[ings] 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.

## Optimization Options

The Optimization options are:

- [-factor1 on page 64](#)
- [-factor2 on page 64](#)
- [-factor3 on page 64](#)
- [-inline on page 64](#)
- [-ipa on page 65](#)
- [-nofactor1 on page 66](#)
- [-nofactor2 on page 66](#)
- [-nofactor3 on page 66](#)
- [-O on page 66](#)
- [-O+ on page 67](#)
- [-opt on page 68](#)

## Using Build Tools on the Command Line

### Optimization Options

---

#### **-factor1**

Turns on factorization step 1.

#### **Syntax**

`-factor1`

#### **Remarks**

To turn off factorization step 1, see [-nofactor1 on page 66](#).

---

#### **-factor2**

Turns on factorization step 2.

#### **Syntax**

`-factor2`

#### **Remarks**

To turn off factorization step 2, see [-nofactor2 on page 66](#).

---

#### **-factor3**

Turns on factorization step 3.

#### **Syntax**

`-factor3`

#### **Remarks**

To turn off factorization step 3, see [-nofactor3 on page 66](#).

---

#### **-inline**

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

---



#### Syntax

`-inline keyword`

The options for `keyword` are:

`off` | `none`

Turn off inlining.

`on` | `smart`

Turn on inlining for `inline` functions. This is the default.

`auto`

If `inline` not explicitly specified, auto-inline small functions.

`noauto`

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

`deferred`

Defer inlining until end of compilation unit. This allows inlining of functions in both directions.

`level=n`

Inline 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`

Turn on aggressive inlining. This option is the same as `-inline on`, `-inline auto`.

---

## -ipa

Specify Interprocedural Analysis Support (IPA) options.

#### Syntax

`-ipa keyword[,...]`

Select the interprocedural analysis level.

The `keyword` arguments are:

`function` | `off`

traditional mode (per function optimization)

`file`

per file optimization (same as `-deferred codegen`)

## Using Build Tools on the Command Line

### Optimization Options

---

#### **-nofactor1**

Turns off factorization step 1.

##### **Syntax**

`-nofactor1`

##### **Remarks**

To turn on factorization step 1, see [-factor1 on page 64](#).

---

#### **-nofactor2**

Turns off factorization step 2.

##### **Syntax**

`-nofactor2`

##### **Remarks**

To turn on factorization step 2, see [-factor2 on page 64](#).

---

#### **-nofactor3**

Turns off factorization step 3.

##### **Syntax**

`-nofactor3`

##### **Remarks**

To turn on factorization step 3, see [-factor3 on page 64](#).

---

#### **-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, *intrinsic*s.

p

Equivalent to -opt speed.

s

Equivalent to -opt space.

**Remarks**

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

## Using Build Tools on the Command Line

### Optimization Options

---

#### -opt

Specify code optimization options to apply to object code.

#### Remarks

`-optkeyword [ , ... ]`

The keyword arguments are:

`off` | `none`

Suppress 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`

Set a specific optimization level. The options for `num` are:

- 0 – Global register allocation only for temporary values. Prefix file equivalent: `#pragma optimization_level 0`.
- 1 – Adds dead code elimination, branch and arithmetic optimizations, expression simplification, and peephole optimization. Prefix file equivalent: `#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. Prefix file equivalent: `#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. Prefix file pragma equivalent: `optimization_level 3`.
- 4 – Like level 3, but with more comprehensive optimizations from levels 1 and 2. Prefix file equivalent: `#pragma optimization_level 4`.

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

`[no] space`

Optimize object code for size. Prefix file equivalent: `#pragma optimize_for_size on`.

`[no] speed`

Optimize object code for speed. Prefix file equivalent: `#pragma optimize_for_size off`.

`[no] cse` | `[no] commonsubs`

Common subexpression elimination. You can also add `#pragma opt_common_sub` to a prefix file.

`[no] deadcode`

Removal of dead code. Prefix file equivalent: `#pragma opt_dead_code`.

`[no] deadstore`

Removes dead assignments. Prefix file equivalent: `#pragma opt_dead_assignments`

`[no] lifetimes`

Computes variable lifetimes. Prefix file equivalent: `#pragma opt_lifetimes`

`[no] loop[invariants]`

Removes loop invariants. Prefix file equivalent: `#pragma opt_loop_invariants`

`[no] prop[agation]`

Propagation of constant and copy assignments. Prefix file equivalent: `#pragma opt_propagation`.

`[no] strength`

Strength reduction. Reducing multiplication by an array index variable to addition. Prefix file equivalent: `#pragma opt_strength_reduction`.

`[no] dead`

Same as `-opt [no] deadcode` and `[no] deadstore`. Prefix file equivalent: `#pragma opt_dead_code on|off` and `#pragma opt_dead_assignments`

`[no] peep[hole]`

Peephole optimization. Prefix file equivalent: `#pragma peephole`.

`[no] color[ing]`

Register coloring. Prefix file equivalent: `#pragma register_coloring`.

`[no] intrinsics`

Inlining of intrinsic functions.

`[no] schedule`

Perform instruction scheduling.

`display` | `dump`



## Using Build Tools on the Command Line

### *Optimization Options*

---

Display complete list of active optimizations.

# Linker

---

## Introduction

For 56800/E Target specific information about the ELF Linker and Command Language, see the “Elf Linker and Command Language” Chapter in either: *Code Warrior Development Studio for Freescale 56800/E Digital Signal Controllers: DSP56F80x/DSP56F82x Family Targeting Manual* or *Code Warrior Development Studio for Freescale 56800/E Digital Signal Controllers: MC56F83xx/DSP5685x Family Targeting Manual*.



## **Linker**

### *Introduction*

---



# C

---

The CodeWarrior C programming language closely follows the ISO C Standard (ISO/IEC 9899:1990). CodeWarrior C also has extensions to work more effectively with the target platform it generates object code for and to be compatible with other compilers.

This chapter describes these extensions to the ISO C Standard and implementation-defined behaviors:

- [Extensions to Standard C on page 73](#)
- [Implementation-Defined Behavior on page 78](#)

---

**NOTE** For 56800/E Target specific information about C, see the “C for DSP56800” Chapter or “C for DSP56800E” Chapter in either the: *Code Warrior Development Studio for Freescale 56800/E Digital Signal Controllers: DSP56F80x/DSP56F82x Family Targeting Manual* or *Code Warrior Development Studio for Freescale 56800/E Digital Signal Controllers: MC56F83xx/DSP5685x Family Targeting Manual*

---

## Extensions to Standard C

- [Unnamed Arguments in Function Definitions on page 74](#)
- [C++ Comments](#)
- [A # Not Followed by a Macro Argument](#)
- [Using an Identifier After #endif](#)
- [Using Typecasted Pointers as lvalues](#)
- [Inline Functions](#)
- [Pascal Calling Conventions](#)
- [Character Constants as Integer Values](#)
- [Converting Pointers to Types of the Same Size](#)
- [Getting Alignment and Type Information at Compile Time](#)
- [Arrays of Zero Length in Structures](#)
- [The “D” Constant Suffix](#)
- [The `\_typeof` \(\) and `typeof`\(\) operators](#)

## Unnamed Arguments in Function Definitions

(ISO C, §6.9.1) The C compiler can accept unnamed arguments in a function definition.

### Listing 5.1 Unnamed Function Arguments

---

```
void f(int ) {} /* OK if ANSI strict checking is disabled */
void f(int i) {} /* ALWAYS OK */
```

---

The compiler allows this extension if ANSI strict checking is disabled:

- in the IDE, use the **C/C++ Language Settings** panel's **ANSI Strict** setting
- on the command line, use the compiler's `-ansi strict` option
- in source code, use `#pragma ANSI_strict`

## C++ Comments

(ISO C, §6.4.9) The C compiler can accept C++ comments (`//`) in source code. C++ comments consist of anything that follows `//` on a line.

### Listing 5.2 Example of a C++ Comment

---

```
a = b; // This is a C++ comment
```

---

To use this feature, disable the ANSI Strict setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#).

## A # Not Followed by a Macro Argument

(ISO C, §6.10.3) The C compiler can accept `#` tokens that do not appear before arguments in macro definitions.

### Listing 5.3 Preprocessor Macros Using # Without an Argument

---

```
#define add1(x) #x #1 // OK, but probably not what you wanted:
                        // add1(abc) creates "abc"#1
#define add2(x) #x "2" // OK: add2(abc) creates "abc2"
```

---

To use this feature, disable the ANSI Strict setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#).

---

## Using an Identifier After #endif

(ISO C, §6.10.1) The C compiler can accept identifier tokens after #endif and #else. This extension helps you match an #endif statement with its corresponding #if, #ifdef, or #ifndef statement, as shown here:

---

```
#ifdef __MWERKS__
#   ifndef __cplusplus
#       /*
#         * . . .
#         */
#   endif __cplusplus
#endif __MWERKS__
```

---

To use this feature, disable the ANSI Strict setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#).

---

**TIP** If you enable the ANSI Strict setting (thereby disabling this extension), you can still match your #ifdef and #endif directives. Simply put the identifiers into comments, as shown in following example:

---

```
#ifdef __MWERKS__
#   ifndef __cplusplus
#       /*
#         * . . .
#         */
#   endif /* __cplusplus */
#endif /* __MWERKS__ */
```

---

## Using Typecasted Pointers as lvalues

The C compiler can accept pointers that are typecasted to other pointer types as lvalues.

### Listing 5.4 Example of a Typecasted Pointer as an lvalue

---

```
char *cp;
((long *) cp)++; /* OK if ANSI Strict is disabled. */
```

---

To use this feature, disable the ANSI Strict setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#).

## Inline Functions

As in C++, the CodeWarrior C compiler allows the `inline`, `__inline__`, or `__inline` keyword to appear before a function declaration and definition. An inline keyword specifies to the compiler that it should attempt to replace calls to the function with the function's body.

## Pascal Calling Conventions

The CodeWarrior C compiler allows the `pascal` keyword to precede a function declaration and definition. This keyword specifies to the compiler that it should use Pascal calling conventions to call this function.

## Character Constants as Integer Values

(ISO C, §6.4.4.4) The C compiler lets you use string literals containing 2 to 8 characters to denote 32-bit integer values. [Table 5.1 on page 76](#) shows examples.

**Table 5.1 Integer Values as Character String Constants**

Character constant	Equivalent hexadecimal integer value
'ABCD'	0x41424344 (32-bit value)
'ABC'	0x00414243 (32-bit value)
'AB'	0x4142 (16-bit value)

You cannot disable this extension, and it has no corresponding pragma or setting in any panel.

---

**NOTE** This feature differs from using multibyte character sets, where a single character requires a data type larger than 1 byte.

---

## Converting Pointers to Types of the Same Size

The C compiler allows the conversion of pointer types to integral data types of the same size in global initializations. Since this type of conversion does not conform to the ANSI C standard, it is only available if the ANSI Strict setting is disabled in the [C/C++ Language \(C only\) Settings Panel on page 16](#).

---

**Listing 5.5 Converting a Pointer to a Same-sized Integral Type**

---

```
char c;  
long arr = (long)&c; // accepted (not ISO C)
```

---

## Getting Alignment and Type Information at Compile Time

The C compiler has two built-in functions that return information about a data type's byte alignment and its data type.

The function call `__builtin_align(typeID)` returns the byte alignment used for the data type *typeID*. This value depends on the target platform for which the compiler is generating object code.

The function call `__builtin_type(typeID)` returns an integral value that describes the data type *typeID*. This value depends on the target platform for which the compiler is generating object code.

## Arrays of Zero Length in Structures

If you disable the ANSI Strict setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#), the compiler lets you specify an array of no length as the last item in a structure. [Listing 5.6 on page 77](#) shows an example. You can define arrays with zero as the index value or with no index value.

---

**Listing 5.6 Using Zero-length Arrays**

---

```
struct listOfLongs {  
    long listCount;  
    long list[0]; // OK if ANSI Strict is disabled, [] is OK, too.  
}
```

---

## The “D” Constant Suffix

When the compiler finds a “D” immediately after a floating point constant value, it treats that value as data of type double.

## The `__typeof__()` and `typeof()` operators

With the `__typeof__()` operator, the compiler lets you specify the data type of an expression. [Listing 5.7 on page 78](#) shows an example.

## C

## Implementation-Defined Behavior

---

```
__typeof__ (expression)
```

where *expression* is any valid C expression or data type. Because the compiler translates a `__typeof__()` expression into a data type, you can use this expression wherever a normal type would be specified.

Like the `sizeof()` operator, `__typeof__()` is only evaluated at compile time, not at runtime.

---

**Listing 5.7 Example of `__typeof__()` and `typeof()` Operators**


---

```
char *cp;
int *ip;
long *lp;

__typeof__(*ip) i; /* equivalent to "int i;" */
__typeof__(*lp) l; /* equivalent to "long l;" */

#pragma gcc_extensions on
typeof(*cp) c;    /* equivalent to "char c;" */
```

---

## Implementation-Defined Behavior

The ISO C Standard cannot practically define every possible aspect of a compiler implementation. It does, however, list issues that must be defined by the implementation of the compiler. This section describes aspects of the CodeWarrior C compiler that the ISO C standard refers that are not covered in the rest of this manual:

- [Diagnostic Messages on page 78](#)
- [Identifiers on page 78](#)

### Diagnostic Messages

(ISO C, §6.3.1) In the CodeWarrior IDE, the CodeWarrior C compiler reports error and warning messages in the **Errors and Warnings** window. See the *IDE User's Guide* for more information on viewing and navigating messages in this window. On the command-line, the CodeWarrior C compiler reports error and warning messages to the standard error file.

### Identifiers

(ISO C, §6.4.2) The CodeWarrior C language allows identifiers to have unlimited length. However, only the first 255 characters are significant for internal and external linkage.

# Tool Performance

---

CodeWarrior compilers can “precompile” a header file to speed up translation of source code. Precompiling a header file that is included often in other source files will reduce the time the compiler uses to translate source code.

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

## Precompiled Header Files

- [When to Use Precompiled Files on page 79](#)
- [What Can be Precompiled on page 80](#)
- [Precompiling C++ Source Code on page 80](#)
- [Using a Precompiled Header File on page 81](#)
- [Preprocessing and Precompiling on page 82](#)
- [Pragma Scope in Precompiled Files on page 82](#)
- [Precompiling a File in the CodeWarrior IDE on page 83](#)
- [Updating a Precompiled File Automatically on page 83](#)

## When to Use Precompiled Files

Source code files in a project typically 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 same header files repeatedly during compilation. To shorten the time spent compiling and recompiling the same header files, CodeWarrior compilers can precompile a header file, allowing it to be subsequently preprocessed much faster than a regular text source code file.

For example, 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 the project, saving the programmer some time and effort while writing source code.

## Tool Performance

### Precompiled Header Files

---

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 multiple duplications, 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).

## Precompiling C++ Source Code

The CodeWarrior compiler has these requirements for precompiling C++ 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 and constant variable declarations (const)
- A C++ source code file that will be automatically precompiled must have a .pch++ file name extension.



---

## 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 **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 6.1 on page 81](#) and [Listing 6.2 on page 81](#) show an example.

### Listing 6.1 Header File that Creates a Precompiled Header File for C

---

```
// sock_header.pch
// When compiled or precompiled, this file will generate a
// precompiled file named "sock_precomp.mch"

#pragma precompile_target "sock_precomp.mch"

#define SOCK_VERSION "SockSorter 2.0"
#include "sock_std.h"
#include "sock_string.h"
#include "sock_sorter.h"
```

---

### Listing 6.2 Using a Precompiled File

---

```
// sock_main.c
// Instead of including all the files included in
// sock_header.pch, we use sock_precomp.h instead.
//
// A precompiled file must be included before anything else.

#include "sock_precomp.mch"

int main(void)
{
    // ...
}
```

---

## Tool Performance

### Precompiled Header Files

---

```
    return 0;
}
```

---

## 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 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 FOO_H
#define FOO_H
    // file contents
#endif
```

The compiler will not load the file twice, saving some small amount of time in the process.

## 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 6.3 on page 82](#) specifies that the variable `xxx` is a far variable.

### Listing 6.3 Pragma Settings in a Precompiled Header

---

```
// my_pch.pch

// Generate a precompiled header named pch.mch.
#pragma precompile_target "my_pch.mch"

#pragma far_data on
extern int xxx;
```

---

The source code in [Listing 6.4 on page 82](#) includes the precompiled version of [Listing 6.3 on page 82](#).

### Listing 6.4 Pragma Settings in an Included Precompiled File

---

```
// test.c
#pragma far_data off // far data is disabled
```

---

```
#include "my_pch.mch" // this precompiled file sets far_data on  
  
// 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 80](#) 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. [on page 80](#)
  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.



## Tool Performance

### *Precompiled Header Files*

---

- 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.

# Optimization

---

CodeWarrior build tools offer features to reduce the size of object code, improve a program's execution speed, and often do both at the same time. Compiler optimizations rearrange, add, or remove instructions to reduce size or improve performance.

This chapter describes how to take advantage of these optimizations:

- [Optimization Considerations on page 85](#)
- [Inlining on page 85](#)
- [Profiling on page 86](#)
- [String Literals on page 86](#)
- [Optimizations on page 87](#)

## Optimization Considerations

There are several issues to take into consideration when selecting optimizations. Code can be optimized for size or for speed, and there are optimizations that could effect the size and the performance of the compiler. It is important to understand the full effects of the optimizations. For example, inlining will decrease the overhead of making function calls. However, if too many functions are called the resulting executable could be too large to run on the target platform.

Inlining also effects the ability to debug a program. Programs are optimally debugged at optimization level 0, and with no additional optimization options enabled. Users should keep in mind that optimization could result in incorrect data being displayed while debugging, and stepping through functions could also seem incorrect.

Finally, the performance of the compiler could also be negatively effected by enabling optimizations. If there are many optimizations enabled, the compile time could increase because of the extra time needed to process the optimizations.

All of these issues should be considered when selecting optimizations.

## Inlining

When inlining is enabled certain function calls are replaced with the function code. Inlining function optimizes for speed, as there is no call. However, overall code may be larger if function code is repeated in several places.

The inlining of a function is based on the complexity of the function and the settings of several compiler options: IPA, Inline Depth, Auto Inline and Bottom up inline. These options are discussed in [“IDE Settings Panels” on page 16](#).

## Profiling

For more details about profiling see the *CodeWarrior Development Studio IDE 5.5 User's Guide Profiler Supplement* and the “Profiler” Chapter in your target specific Targeting Manual.

## String Literals

The compiler and linker manage character strings so that they occupy less space in the object code and executable file.

String literals are:

- [Pooling Strings on page 86](#)
- [Reusing Strings on page 86](#)

## Pooling Strings

The **Pool Strings** setting in the C/C++ Language Panel controls how the compiler stores string constants.

If you enable this setting, the compiler collects all string constants into a single data object so that your program needs only one TOC (table of content) entry for all of them. While this decreases the number of TOC entries in your program, it also increases your program size because it uses a less efficient method to store the address of the string.

If you disable this setting, the compiler creates a unique data object and TOC entry for each string constant.

Enable this setting if your program is large and has many string constants.

The **Pool Strings** setting corresponds to the `pragma pool_strings`. To check this setting, use `__option(pool_strings)`. By default, this setting is disabled. See also [“pool\\_strings” on page 172](#) and [“Checking Pragma Settings” on page 119](#).

## Reusing Strings

The **Reuse Strings** setting in the C/C++ Language Panel controls how the compiler stores string literals.

If you enable this setting, the compiler stores each string literal separately. Otherwise, the compiler stores only one copy of identical string literals. This means if you change one of the strings, you change them all. For example, look at this code:

```
char *str1="Hello";  
char *str2="Hello"; // two identical strings  
*str2 = 'Y';
```

This setting helps you save memory if your program contains identical string literals which you do not modify. If you enable the Reuse Strings setting, the strings are stored separately. After changing the first character, str1 is still Hello, but str2 is Yello.

If you disable the Reuse Strings setting, the two strings are stored in one memory location because they are identical. After changing the first character, both str1 and str2 are Yello, which is counterintuitive and can create bugs that are difficult to locate. The Reuse Strings setting corresponds to the pragma dont\_reuse\_strings. To check this setting, use \_\_option(dont\_reuse\_strings). By default, this setting is enabled, so strings are not reused. See also [“dont\\_reuse\\_strings” on page 163](#) and [“Checking Pragma Settings” on page 119](#).

## Optimizations

The following is a collection of optimization types and examples of how the resulting generated code is affected:

- [Dead Code Elimination on page 88](#)
- [Expression Simplification on page 88](#)
- [Common Subexpression Elimination on page 89](#)
- [Copy Propagation on page 89](#)
- [Dead Store Elimination on page 90](#)
- [Live Range Splitting on page 91](#)
- [Loop-Invariant Code Motion on page 91](#)
- [Strength Reduction on page 92](#)
- [Loop Unrolling on page 93](#)
- [M56800E Specific Optimizations on page 93](#)

## Dead Code Elimination

**Listing 7.1** Dead code elimination, before optimization

---

```
void func(void)
{
    if (0)
    {
        otherfunc1();
    }

    otherfunc2();
}
```

---

**Listing 7.2** Dead code elimination, after optimization

---

```
void func_optimized(void)
{
    otherfunc2();
}
```

---

## Expression Simplification

**Listing 7.3** Expression simplification, before optimization

---

```
#define MY_OFFSET 4

void func(int* result1, int* result2, int* result3, int* result4, int
x)
{
    *result1 = x + 0;
    *result2 = x * 2;
    *result3 = x - x;
    *result4 = 1 + x + MY_OFFSET;
}
```

---

**Listing 7.4** Expression simplification, after optimization

---

```
#define MY_OFFSET 4

void func_optimized(int* result1, int* result2, int* result3, int*
result4, int x)
{
```

---



---

```
*result1 = x;  
*result2 = x << 2;  
*result3 = 0;  
*result4 = 5 + x;  
}
```

---

## Common Subexpression Elimination

### Listing 7.5 Common subexpression elimination, before optimization

---

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

---

### Listing 7.6 Common subexpression elimination, after optimization

---

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

---

## Copy Propagation

### Listing 7.7 Copy propagation, before optimization

---

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

---

## Optimization

### Optimizations

---

```
    }  
}
```

---

#### Listing 7.8 Copy propagation, after optimization

---

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

---

## Dead Store Elimination

#### Listing 7.9 Dead store elimination, before optimization

---

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

---

#### Listing 7.10 Dead store elimination, after optimization

---

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

---

---

## Live Range Splitting

**Listing 7.11** Live range splitting, before optimization

---

```
void func(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 7.12** Live range splitting, after optimization

---

```
void func_optimized(int x, int y)
{
    int temp;

    temp = x * y;
    otherfunc(temp);

    temp = x + y;
    otherfunc(temp);

    temp = x - y;
    otherfunc(temp);
}
```

---

## Loop-Invariant Code Motion

**Listing 7.13** Loop-invariant code motion, before optimization

---

```
void func(float* vec, int max, float val)
{
    float circ;
    int i;
```

---

## Optimization

### Optimizations

---

```

    for (i = 0; i < max; ++i)
    {
        circ = val * 2 * PI;
        vec[i] = circ;
    }

```

---

#### Listing 7.14 Loop-invariant code motion, after optimization

---

```

void func_optimized(float* , int max, float val)
{
    float circ;
    int i;
    circ = val * 2 * PI;
    for (i = 0; i < max; ++i)
    {
        vec[i] = circ;
    }
}

```

---

## Strength Reduction

#### Listing 7.15 Strength reduction, before optimization

---

```

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

```

---

#### Listing 7.16 Strength reduction, after optimization

---

```

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

```

---

```
}  
}
```

---

## Loop Unrolling

### Listing 7.17 Loop unrolling, before optimization

---

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

---

### Listing 7.18 Loop unrolling, after optimization

---

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

---

## M56800E Specific Optimizations

This section provides techniques, programming style suggestions, and information to maximize the efficiency of the Freescale C compiler for the 56800/E Digital Signal controllers.

## Overview of the 56800E Architecture

The 56800/E processors are member of the 56800x family of digital signal micro-controllers. The 56800x instruction set is targeted for efficient micro-controller code

generation and DSP (Digital Signal Processing). The 56800/E are digital signal processors, because they both have a micro-controller and DSP.

Micro-controller instructions include:

- bit manipulation instructions
- flexible branching instructions
- absolute (global) addressing modes to maximize control code density.

DSP features include:

- single cycle MAC (Multiply-Accumulate)
- separate address register file
- separate data/program memory spaces,
- multiple addressing modes, including pointer post-update addressing modes.

The C compiler attempts to target the post-update addressing modes in loops. In this chapter, we describe the programming style that promotes the selection of the post-update addressing modes.

The 56800x family is a native 16-bit machine--data and addresses are 16 bits wide. The 56800/E extends the address bus width to 24-bits (called the large data model), allowing a wider range of data addresses, but at a cost of performance and code density. In this chapter, we discuss the techniques used to minimize the cost of enabling the large data model.

---

**NOTE** Although ANSI-C data types are fully supported, in this chapter, we show that the best code is generated when the programmer favors the native data type size (16-bits).

---

## Working with the 56800E Memory Models

The Freescale 56800E C Compiler supports large and small program and data memory models as shown in [Table 7.1 on page 94](#). The small data model is more code efficient. However, sometimes the application requires a larger data address space.

**Table 7.1 Code and Data Memory Ranges**

Section	Small Data Model		Large Data Model	
	Size (KB)	Range (Word Address)	Size (MB)	Range (Word Address)

**Table 7.1 Code and Data Memory Ranges (continued)**

CODE (P:memory)	128	0 - 0xFFFF	1	0 - 0x7FFFF
DATA (X:memory)	128	0 - 0xFFFF	32	0 - 0x7FFFF
DATA (X:memory) character data	64	0 - 0xFFFF	16	0 - 0x7FFFF

The large data memory model allows data to be placed in memory at addresses greater than the 16-bit address limitation of the small data model. The large data memory model is selected via a preference panel selection in the CodeWarrior IDE. This selection informs the compiler that global and static data should be addressed with the 24-bit variants of the absolute addressing modes of the device. Also in the large memory model, pointers are treated as 24-bit quantities when moved from register to register, memory to register, or register to memory. For information on how the large memory model is selected, see the *Freescale 56800/E Hybrid Controllers: MC56F83xx/DSP5685x Family Targeting Manual*.

One likely scenario in an embedded programming environment is that the total static and global data size, that is, the total size of data objects that the compiler accesses with absolute addressing modes (X:xxxx or X:xxxxxx addressing modes) will comfortably reside within the 16-bit data addressing range. However, the heap (dynamically allocated data memory) or the stack (local, automatic data memory) may require extended addressing as this data may extend beyond the 16-bit address range.

To optimize the program size, use the CodeWarrior IDE targets settings panel **M56800E Processor:Large Data Model: Globals live in lower memory** panel option in conjunction with the large data memory model. The **Globals live in lower memory** panel option reverts the absolute addressing modes to the small data model for static and global variables, while using the large memory model for any address pointers or local variables. Thus, for static and global variables, the efficiency of the small data model is retained even for programs where the total data size may exceed the 16-bit addressing range.

[Listing 7.19 on page 96](#) shows the code generation differences between the large and small data model. In this example, the code performs a bubble sort on an array of integers. At maximum optimization, the code runs in 579 cycles in the small data memory model. The code takes 760 cycles using the large data memory model. When the the large data memory model and **Globals live in lower memory** option is selected, the code runs in 729 cycles. The difference in the cycle count of the two large data model runs is due to the way

## Optimization

### Optimizations

---

global variables are addressed. The **Globals live in lower memory** option forces the access of the global variable “next” to be there as it would be for the small data model.

#### Listing 7.19 Example 1: Memory Model Comparison Code

---

```
int vector[] = { 3,7,6,1,2,5 };
int next;

int main()
{
int i=0, j=0;
int sz = sizeof(vector)/sizeof(int);

    for (i=0; i<sz; i++){
        for(j=0; j<sz-i; j++){
            if (vector[j]>vector[j+1]) {
                next=vector[j];
                vector[j]=vector[j+1];
                vector[j+1]=next;
            }
        }
    }
}
```

---

**Table 7.2 Example 1 at Maximum Optimization**

Small Data Model	Large Data Model	Large Data Model and Global Live in Lower Memory
579 cycles	760 cycles	729 cycles

If the **Globals live in lower memory** option is selected, be sure to locate the `.data` and `.bss` sections in lower memory. Dynamically allocated memory and the stack may be located in either lower or upper memory for the large data model.

## Targeting Post-Update Addressing Modes in Loops

Post-update addressing modes are available for many 56800E instructions. At optimization level 2 and above, the compiler attempts to locate register-based address expressions which change by a linear amount for each iteration through a loop. If such an expression is located and certain conditions are met, the compiler may replace the address



update expression with a post-update addressing mode that is performed concurrently with the move or arithmetic operation. Such a transformation is called ‘strength reduction’ in compiler terminology and means replacing an instruction operation with a cheaper (fewer cycles or words) instruction. Address expressions are normally either address registers that have been loaded directly with the addresses of objects (variables) or address registers holding the calculated address of array elements. Array indices which vary by a regular, linear amount for each iteration through a loop are called ‘induction variables.’ Many times induction variables are completely eliminated when their function is replaced by a post-update addressing mode.

**Listing 7.20 Example 2: Post-Update Addressing Modes**

X: (Rn) +	Address is incremented by 1 (2 for move.l)
X: (Rn) -	Address is decremented by 1 (2 for move.l)
X: (Rn) +N	Address is incremented by value in N register

Some programming guidelines which promote the successful targeting of the post-update addressing mode are:

- The address expression must be within a loop.
- The address expressions must be register based, therefore, global pointer variables are usually not targeted for strength reduction since they may be accessed with absolute addressing modes. Sometimes, it is useful to load the address of a global array into a local pointer variable to make the address expression more obvious to the compiler.
- The address expression should be executed each iteration of the loop. Address expressions embedded in ‘if-then-else’ blocks will not be targeted for post-update addressing.
- Induction variables must be defined at one point in the loop and must vary linearly from its previous value.

In [Listing 7.21 on page 97](#), a simple loop that calculates the sum of elements in a local array is shown. For this example, the induction variable ‘i’ is completely eliminated because:

- a DO loop instruction has been generated, eliminating the need for a test on ‘i’ to determine if the loop has ended
- the use of ‘i’ in the calculation of the array addresses has been eliminated, in favor of a post-update addressing mode (see line 11 in [Listing 7.21 on page 97](#))

**Listing 7.21 Example 3: Successful Strength Reduction**

```
int i;
int sum=0;
```

## Optimization

### Optimizations

```
int arr[] = { 13,14,18,3,7,0,1,4,11,20 };
int sz = sizeof(arr)/sizeof(int);
for (i=0; i < sz; i++)
    sum += arr[i];

printf ( "Sum is %d\n",sum );
```

Assembly output:

```
(1) adda          #<10,SP          ;allocate stack
(2) move.w       #<0,B           ;sum = 0
(3) adda         #-9,SP,R1       ;&arr[0]->R1
(4) moveu.w     #F47,R0         ;temp F47->R0
(5) do          #<10,>_L8_0      ;compiler generated init loop
(6) move.w      X:(R0)+,A       ;initialize arr[]
(7) move.w      A1,X:(R1)+
(8) _L8_0:
(9) adda         #-9,SP,R0       ;&arr[0]->R0
(10) do         #<10,>_L8_1      ;for loop
(11) move.w     X:(R0)+,A       ;arr[i]->A
(12) add        A,B            ;sum = arr[i]+sum
(13) _L8_1:
(14) adda       #<2,SP          ;printf call setup
(15) moveu.w    #@1b(F54),N     ;string temp to stack
(16) move.w     N,X:(SP)
(17) move.w     B1,X:(SP-1)    ;sum to stack
(18) jsr        >Fprintf       ;call printf
(19) suba      #<2,SP          ;restore stack
```

[Listing 7.22 on page 98](#) shows a case where strength reduction of the address expression was not possible, mainly because the access to the array is conditionally executed in the loop. Also, the induction variable ‘i’ is used in the ‘if’ test, but this would not normally prevent a post-update transformation from occurring.

#### Listing 7.22 Example 4: Array Update In Conditional Block

```
for (i=0; i < sz; i++)
    if ( i & 1 )
        sum += arr[i];
```

Assembly output:

```
(1) do          #<10,>_L8_1      ;for loop
(2) brclr      #1,Y0,<_L8_2    ;if ( i & 1 )
(3) move.w     X:(R0),A       ;arr[i]->Av
```

```

(4) add          A,B                ;sum = arr[i]+sum
(5) _L8_2:
(6) adda        #<1,R0             ; &arr = &arr + 1;
(7) add.w       #<1,Y0             ; i = i + 1
(8) nop
(9) _L8_1:

```

In [Listing 7.23 on page 99](#) another situation is shown where strength reduction will fail to find a post-update opportunity. This is when the loop or induction variable is multiply defined in a loop.

---

**NOTE** This also kills the hardware do loop as the compiler cannot determine the static loop count.

---

### Listing 7.23 Example 5: Induction Variable is Multiply Defined

```

for (i=0; i < sz; i++)
    sum += arr[i++];

```

Assembly output:

```

(1) move.w      #<0,A              ; i=0
(2) _L8_1:
(3) move.w      A1,B                ; i -> temp
(4) add.w       #<1,B                ; temp++
(5) move.w      A1,N                ; temp++ -> N
(6) adda        #-9,SP,R0           ; &arr[0] -> R0
(7) move.w      X:(R0+N),A          ; arr[temp++] -> A
(8) add         A,Y0                ; sum = arr[i++] + sum
(9) move.w      B1,A                ; temp++ -> i
(10) add.w      #<1,A                ; i = i + 1
(11) cmp.w     #<10,A              ;
(12) blt        <_L8_1              ; i < 10 ?

```

[Listing 7.24 on page 100](#) demonstrates a simple delay line loop that is structured so post-update addressing is impossible. The final store to memory in the loop is a memory plus displacement addressing mode, `move.w A1, X: (R0+1)`, which doesn't allow post-update addressing. The loop written as is takes approximately 29 cycles and 9 words for NTAPS=6.

## Optimization

### Optimizations

---

#### Listing 7.24 Example 6: Loop Structure Doesn't Allow Post-Update Addressing

---

```

    for (ii = NTAPS - 2; ii >= 0; ii--) {
        z[ii + 1] = z[ii];
    }
Assembly output:
(1) do          #<5,>_L12_1                ; for ()
(2) move.w     Y0,R0                      ; ii -> R0
(3) adda       R3,R0                      ; &z[0] + i
(4) move.w     X:(R0),A                   ; z[ii] -> A
(5) move.w     A1,X:(R0+1)               ; z[ii] -> z[ii +
1]
(6) sub.w      #<1,Y0                    ; ii--
(7) _L12_1:

```

---

The loop in [Listing 7.24 on page 100](#) may be re-written slightly as shown in [Listing 7.25 on page 100](#) to allow for much more efficient processing. The idea is to try to get an instruction that has a post-update variant as the final load or store in the loop. This loop executes in 17 cycles and 8 words.

---

#### Listing 7.25 Example 7: Loop Re-written to Allow Post-Update Addressing

---

```

int *p1 = &z[NTAPS-1];
for (ii = NTAPS - 2; ii >= 0; ii--) {
    *p1-- = z[ii];
}
Assembly output:
R3
(1) tfra       R1,R3                      ;&z[NTAPS-1] ->
R0
(2) adda       #-5,SP,R0                  ;&z[NTAPS-2] ->
(3) tfra       R0,R2                      ;R0 -> R2
(4) do         #<5,>_L9_1                  ;for ()
(5) move.w     X:(R2)-,B                  ;z[ii] -> B
(6) move.w     B1,X:(R3)-                 ;B -> z[ii+1]
(7) _L9_1:

```

---

## The Effects of Casting on Code Quality

The 56800x family is a native 16-bit architecture. Type casting to and from 16-bit data types requires extra instruction words and cycles. Use 16-bit types (int, short, unsigned int, unsigned short) whenever possible to minimize to program memory required for the application. Also be aware that ANSI-C requires implicit promotion of integral types for

arithmetic operations and this may cause implicit type casting. Of course, favoring 16-bit data types may cause an increase in the total data size of an application. The trade off between program and data memory will have to be judged for each application. In general, if program memory is the limiting resource, favor 16-bit types. If data memory is the limiting resource, then using 8-bit data types where possible may be preferred.

Casting ints to char or long types are usually the least costly in terms of words and cycles. Since accumulators (A,B,C,D registers in the 56800E) are the only registers capable of holding 32-bit quantities, they must be used for long operations. Accumulators are composed of two individually addressable 16-bit parts, the MSP or most significant portion and the LSP or least significant portion. The MSP is often treated as a 16-bit register containing an int or short sized quantity (16-bits). An int to long cast requires an `asr16` instruction to move the MSP to the LSP of the accumulator.

---

**Listing 7.26 Example 8: Casting an integer to a long data type**

---

```
int ls;
long ll;

ll = (long)ls;
move.w   X:(SP-2),A;
asr16    A,A
move.l   A10,X:(SP-4)
```

---

Bytes or char variables are stored as portions of integer sized registers. The 56800E does not contain 8-bit registers. An int to char cast requires an explicit sign extension (`sxt.b`) of the integer to properly format the register so that the sign bit of the char is extended into the entire word. This is required for proper arithmetic operations on the char since arithmetic in C occurs on integers by definition. Also, the 56800E only performs 16-bit and 32-bit arithmetic.

---

**Listing 7.27 Example 9: Casting an int to a char data type**

---

```
char lc;
int ls;

lc = (char)ls;

Assembly output:
move.w   X:(SP-2),A
sxt.b    A,A
```

## Optimization

### Optimizations

---

```
move.b    A1, X: (SP)
```

---

Chars that are converted to int or long first require a sign extension of the byte into an integer value. If the char is converted to a long, an addition `asr16` is required to convert to a 32-bit value.

#### Listing 7.28 Example 10: Casting a char to long

---

```
long ll;
char lc;

ll = (long)lc;
```

Assembly output:

```
moveu.b   X: (SP), A
sxt.b     A, A
asr16     A, A
move.l    A10, X: (SP-4)
```

---

It should be clear now that casting causes runtime penalties in terms of code size and cycles. Sometimes the perceived benefit of using shorter data types to save data memory results in runtime costs.

The 56800E has a unique model for handling pointers to character data. Although the data memory is organized by words, that is, each address points to a word (two bytes) of data, individual bytes within a word can still be addressed. The compiler handles this addressing invisibly, but the programmer should be aware of the costs of converting from byte pointers to word pointers and vice versa.

A byte address is generated by the compiler when the programmer chooses to use character data to represent an object. Strings are character data by default in the 56800E compiler and are addressed with byte pointers. Special instructions in the 56800E instruction set expect to see and operate on byte pointer values. A word pointer may be converted to a byte pointer by multiplying the word address by two. Similarly, a byte address is converted to a word address by dividing the byte address by two. When a byte pointer is cast to a word pointer, an explicit, runtime conversion of the pointer quantity is performed. The cost is a one word, one cycle penalty to bit shift the address value to the left, that is, multiply by two, to convert to a byte pointer. The cost is the same to convert to a word pointer, except the shift is to the right, effectively dividing by two. The void pointer is a byte pointer since the void pointer should be able to represent any data type, including chars. Since there is a runtime penalty for converting pointer types, casts back and forth should be limited for efficient C programs. This may be a factor when the void

pointer is used to point to generic data and cast to the proper type at runtime. [Listing 7.29 on page 103](#) shows the effect of casting byte and word pointers.

**Listing 7.29 Example 11: Casting Byte and Word Pointers**

```

void * pvoid;
int vint;
int * pint;
char *pchar;

    pint = (int *)&vint;
adda    #-5,SP,R0
move.w  R0,X:(SP-6)

    pvoid = (void *)pint;
moveu.w X:(SP-6),R0
asla    R0,R0
move.w  R0,X:(SP-4)

    pchar = (char *)pint;
move.w  X:(SP-6),R0
asla    R0,R0
move.w  R0,X:(SP-7)

    pint = (int *)pvoid;
moveu.w X:(SP-4),R0
lsra    R0
move.w  R0,X:(SP-6)

```

## Miscellaneous Techniques

There are other several minor techniques to be aware of when writing the most efficient C code for the compiler.

Initialize local arrays and structures at declaration time, if possible. Local arrays and structures are initialized optimally by the compiler.

Functions with a large number of parameters will probably have to pass some parameters on the stack causing costly memory accesses. Make sure that frequently called functions pass their parameters in registers. For information on the parameter passing rules for the 56800E C Compiler see the *Freescale 56800/E Hybrid Controllers: MC56F83xx/DSP5685x Family Targeting Manual*.

Forcing enums as integers (C/C++ Language Panel, “Enums Always Ints”) may yield better code since integers are usually handled more efficiently.

## Optimization

### Optimizations

---

Loading frequently used global variables into local temporary variables sometimes has a positive effect on code size and performance, since accessing variables through registers is more efficient than absolute addressing modes.

As an illustration of the final point in the list above, the code in [Listing 7.30 on page 104](#) executes in 98 cycles and 20 program memory words. The same function is performed by the code in [Listing 7.31 on page 104](#), but it executes in 57 cycles and 13 program memory words. A temporary local variable is used in processing instead of the global variable. Fewer absolute addressing instructions account for the difference.

#### Listing 7.30 Example 12: Global Structure Example

---

```
#define ARRAY_SIZE 5

static struct s1
{
    unsigned char value_a;
    unsigned char value_b;
    unsigned char value_c;
} s_s1[ARRAY_SIZE];

unsigned int r1;

int main()
{
    int i;
    for (i = 0; i < ARRAY_SIZE; i++)
    {
        r1 += s_s1[i].value_a;
        r1 += s_s1[i].value_b;
        r1 += s_s1[i].value_c;
    }
    return (r1);
}
```

---

#### Listing 7.31 Example 13: Modified Global Structure Example

---

```
int main()
{
    int i;
    unsigned int local_var;

    local_var = r1;
    for (i = 0; i < ARRAY_SIZE; i++)
```

---



```

    {
        local_var += s_sl[i].value_a;
        local_var += s_sl[i].value_b;
        local_var += s_sl[i].value_c;
    }

    r1 = local_var;

    return (r1);
}

```

## Software Pipelining

Software pipelining is a loop transformation that changes the initial loop so that parts of different iterations execute at the same time. This scheduling technique exploits architectural instruction level parallelism.

It may also produce better loop schedules when stalls, hazards or latencies exist between instructions in the initial loop, if they can be avoided in the transformed loop.

Note that the DSP56800e architecture provides limited parallelism by means of parallel move instructions. These limitations narrow down the applicability of this transformation.

An example of software pipelining transformation:

```

#include "intrinsics_56800e.h"
int x[100], y[100], i;
long res;
void main()
{
    long t=0;
    for (i=0; i<100; i++)
    {
        t = L_mac(t, x[i], y[i]);
    }
    res = t;
}

```

This code will compile the loop-body into one cycle:

rep	R1		
mac	Y0, X0, A	X: (R0)+, Y0	X: (R3)+, X0

where mac instruction from first iteration of the loop executes in parallel with load instructions from the second iteration of the initial loop.

This transformation applies to the inner most loops of a program, and currently is enabled only for DO loops.

It is controlled by the `-[no] swp` command line switch, and it is by default enabled for optimization levels higher than 2. Otherwise `#pragma swplevel on/off` may be used to control the transformation. When optimizing for size, software pipelining is disabled, as it usually increases program size.

## Stack Sequence Optimization

This transformation replaces several accesses to adjacent stack locations with a post-increment/-decrement addressing mode by using an available address register.

For `dsp56800e`, this transformation may bring performance gain both in execution speed and code size. Speed is improved as instructions using post-increment access usually take only one cycle as opposed to instructions with immediate offsets that can take 2 or 3 cycles. Code size is reduced when large immediates are present.

An example of stack sequence optimization where the following low-level intermediate piece of code:

```

move.w    X: (SP-2) , A
move.w    X: (SP-1) , Y1
move.w    X: (SP-2) , A
move.w    X: (SP-3) , B
add.w     X: (SP-4) , B

```

will become:

```

adda      #-2 , SP , R0
move.w    X: (R0) + , A
move.w    X: (R0) - , Y1
move.w    X: (R0) - , A
move.w    X: (R0) + , B
add.w     X: (R0) , B

```

which brings an improvement of 3 cycles ( $2+2+2+2+3$  as opposed to  $2+1+1+1+1+2$ ).

In the example above, the transformation actually increases the code size, and that is why it will not be performed on this example when `-Os` optimization is required.

Note that this transformation makes use of both post increment and post decrement update modes, and it can also exploit all instructions accessing the stack, not only loads and stores.

Transformation is controlled by the `-[no] stackseq` command line switch, and it is enabled by default for an optimization level higher than one. Also, `#pragma stackseq on/off` may be used to control the transformation.

## Constant to array reallocation

Constants/large constants encoded in instructions are stored into an array in data memory and immediate operands are changed into data memory access using register-indirect, post-increment operands.

Main target of this optimization is speed, but as a side effect size improvements can also be obtained, although these cases should be rather rare.

Each transformed instruction reduces the execution time of an instruction with 1-2 cycles and reduces program memory size with 1-2 words, but also causes an increase on data memory with 1-2 words, depending on the size of immediates.

Besides the operand mode transformation, more benefit can be obtained from further grouping of the transformed instructions leading to total program memory size decrease.

The following instructions take between 2-3 words of program memory and 2-3 cycles to execute:

```
MOVE.W#xxxxx, HHHHH
MOVE.L#xxxxxxx, HHHHH
```

and they are transformed to:

```
MOVE.W(Rx)+, HHHHH
MOVE.L(Rx)+, HHHHH
```

so that the resulting instruction will take 1 word of program memory and 1 cycle to execute, but it will add an extra 1-2 words into data memory (the immediate values). It will also add an overhead of one instruction per sequence for computing the address of the first element.

If no instruction grouping happens with instructions transformed to post-increment indirect addressing, the total memory size used will slightly increase, due to the computation of stack offset for the first element in a sequence.

An example of how this optimization works on the following piece of low level intermediate code:

```
.code
    move.w  X:(R3)+, X0
    move.w  #<number_1>, Y0
    mac     Y0, X0, A
    move.w  X:(R3)+, X0
```

## Optimization

### Optimizations

---

```

    move.w  #<number_2>, Y0
    mac     Y0, X0, A

```

The code above can be optimized to:

```

.code
    move.w  #<array_starting_address>, R0
    move.w  X:(R3)+, X0
    move.w  X:(R0)+, Y0
    mac     Y0, X0, A
    move.w  X:(R3)+, X0
    move.w  X:(R0)+, Y0
    mac     Y0, X0, A

.data
    array_starting_address:
        <number_1>
        <number_2>

```

This optimization is disabled for `-Os` and is automatically enabled on speed optimization level `>= 2`. Constant to array reallocation can be enabled/disabled at any optimization level using `- [no] constarray` options in the command line. At function level, you should use `#pragma constarray on/off`.

## CRC linker feature

CRC linker feature is designed to allow execution of memory integrity checks at runtime over user defined portions of memory, which can identify unexpected memory writes (caused by coding errors, unexpected writes, etc.).

This is done in 2 stages. In the first stage, a record is created in memory at an initial memory state, either at link time, using the `CRC16_lcf` directive, or at runtime, using a library routine - `CRC16_create`, declared in `crc.h` and implemented in `crc.c`. Both files are placed in the runtime library.

The second stage calls the `CRC16_check` function from the runtime library to verify that the memory area has not changed since the original state.

The `lcf` language has a new directive that allows the user to generate CRC records for specified portions of memory and assign its address to a user-defined symbol.

The `CRC16` directive is used only inside a section input, since it allocates the CRC structure at the current location as defined by its position in `lcf`.

Any 16-bit polynomial can be used to generate a CRC using the same algorithm. Several 16-bit polynomials are used at a large scale and detect almost all 1 or 2 bit errors, a large set of burst errors, etc.

The `CRC16 lcf` directive allocates space for the CRC record and places it in memory according to the current directive location in `lcf`. The `Lcf` directive `CRC16` will return the address of the generated record and assign it to the left-hand of the assignment.

The `CRC16` directive has at least 2 parameters, the first being an `lcf` symbol that gives the start address of the memory area you want to protect and the second symbol gives the end address of the memory zone. The start/end address symbols will need to be defined like: "symbol = ." in the `lcf` file.

The third argument is optional and used when you want to specify a custom polynomial to be used in the computation of CRC. The polynomial is specified as a hexadecimal number, where `0x1021` represents  $X^{16}+X^{12}+X^5+1$  (`0x1021` is the simplified notation for `0x11021`, since the most significant bit, corresponding to  $X^{16}$  needs to be always 1 in order to have a 16-bit CRC).

Examples of `CRC16` directives:

```
addr_sym = CRC16(start_address ,end_address ,0x1021);
addr_sym2 = CRC16(start_address, end_address) ;
addr_sym3 = CRC16(start_address,end_address,0x1005);
```

If the third parameter of the `CRC16` directive is missing, its default value will be `0x1005` (abbreviated from `0x11005`), corresponding to "CRC-16" polynomial  $X^{16} + x^{15} + x^2 + 1$ . In the example above the last 2 `CRC16` calls have the same effect.

Example of `CRC16 lcf` directive usage:

```
.application_code : {
    start_addr = . ;
    WRITEW(0x11223344);
    *(.text)
    end_addr = . ;
} > p_RAM
.data : {
    data_start = . ;
    *(.data)
    data_end = . ;
} > x_RAM
    .crc : {
        crc1 = CRC16(start_addr,end_addr);
```

## Optimization

### Optimizations

---

```

        crc2 = CRC16(data_start,data_end);
    } > x_RAM2

```

Following is an example of creating a CRC record at runtime, creating a CRC record at linktime and runtime checking of a CRC-protected memory area:

LCF:

```

.application_code :
{
    start_text = . ;
    * (.text)
    end_text = . ;
    p_Sym = CRC16(start_text,end_text);

} > .p_RAM
.application_data :
{
    * (.data)
    x_Sym = CRC16(start_text,end_text);
} > .x_RAM

```

Source code:

```

#include <crc.h>
extern __pmem CRC16_record p_Sym;
extern CRC16_record x_Sym;

CRC16_runtime_defs runtime_info1;
CRC16_runtime_defs runtime_info2;

void func()
{
    CRC16_check (p_Sym,&runtime_info1);
    CRC16_check (x_Sym,&runtime_info2);
}

```

```
    p_Sym.crc16 = 0; //clear CRC to test runtime CRC_create
function
    x_Sym.crc16 = 0; //clear CRC to test runtime CRC_create
function

    CRC16_create (&p_Sym, P_RecType);
    CRC16_create (&x_Sym, X_RecType);

    CRC16_check (p_Sym,&runtime_info1);
    CRC16_check (x_Sym,&runtime_info2);
}
```

## Interprocedural Analysis support

Interprocedural Analysis (IPA) allows the compiler to generate better and/or smaller code by inspecting more than just one function or data object at the same time. This technology is currently used by the inliner.

The compiler supports three different interprocedural analysis modes: off (default), file, and program.

With the function mode `-ipa off`, functions are optimized and code is generated when the function has been parsed. This mode allows no interprocedural analysis.

With the mode `-ipa file`, a translation unit is completely parsed before any code or data is generated. This allows optimizations and inlining on a per-file basis. This mode will require more memory and it can be slightly slower than the `-ipa off` mode. The compiler will also do an early dead code/data analysis in this mode, so objects with internal linkage that are not referenced will be dead-stripped in the compiler rather than in the linker.

With the mode `-ipa program` all translation units are completely parsed. Optimizations and code generation are done in a final stage enabling true "whole program" optimizations. For example, auto-inlining of functions that are defined in another translation unit.

"Program IPA" can require a lot of memory and will also be slower, especially in the change/build/debug cycle because all code generation and optimizations will have to be redone whenever a program has to be relinked.

Using this mode from command-line tools is more complicated. If you specify all source files on the command-line you can use `-ipa program`:

```
mwcc56800e -ipa program test1.c test2.c [all sources and
libraries]...
```

## Optimization

### Optimizations

---

This will compile, optimize, codegen, and link binary in "program" ipa mode.

If you want to separate compilation from linking you can either use:

```
mwcc56800e -ipa program -c test1.c
```

This generates test1.o file (empty) and a test1.iobj file.

```
mwcc56800e -ipa program -c test2.c
```

This generates test2.o file (empty) and a test2.iobj file.

```
mwcc56800e -ipa program test1.o test2.o [all *.o and
libraries]...
```

This will optimize, codegen, and link binary in "program" ipa mode.

If you want to invoke the linker separately you will have to use:

```
mwcc56800e -ipa program -c test1.c
```

This generates test1.o file (empty) and a test1.iobj file.

```
mwcc56800e -ipa program -c test2.c
```

This generates test2.o file (empty) and a test2.iobj file.

```
mwcc56800e -ipa program-final test1.iobj test2.iobj
[all *.iobjs]...
```

This will optimize and codegen in "program" ipa mode and update the .o files.

```
mwld56800e -o test.exe test1.obj test2.obj [all *.objs
and libraries]...
```

This will link binaries.

The .iobj files contain an intermediate program representation. Thus the build step corresponding to "make clean" should remove these when the matching .o file is deleted.



# Inline Assembly Language and Intrinsic

---

## Introduction

For 56800/E Target specific information about the Inline Assembly Language and Intrinsic, see the “Inline Assembly Language and Intrinsic” Chapter in either: *Code Warrior Development Studio for Freescale 56800/E Digital Signal Controllers: DSP56F80x/DSP56F82x Family Targeting Manual* or *Code Warrior Development Studio for Freescale 56800/E Digital Signal Controllers: MC56F83xx/DSP5685x Family Targeting Manual*.



## **Inline Assembly Language and Intrinsic**

*Introduction*

---

# Predefined Symbols

---

The compiler preprocessor has predefined macros that describe the compile-time environment and properties of the target processor.

This chapter describes how to use these predefined symbols and lists them:

- [Using Predefined Symbols on page 115](#)
- [Version Symbol on page 115](#)
- [Date and Time Symbol on page 116](#)
- [IDE Symbol on page 116](#)
- [Name Symbols on page 117](#)
- [Object Code Organization and Generation Symbol on page 117](#)
- [C Symbols on page 118](#)

## Using Predefined Symbols

Predefined symbols are in the preprocessor, available at compile-time only.

## Version Symbol

Version symbols:

- [\\_\\_MWERKS\\_\\_ on page 115](#)

---

### **\_\_MWERKS\_\_**

Defined with the version of the CodeWarrior compiler.

CodeWarrior compilers issued after 1995 define this macro with the compiler's version. For example, if the compiler version is 3.2, the value of `__MWERKS__` is `0x3200`.

This macro is defined as `1` if the compiler was issued before the CodeWarrior CW7 that was released in 1995.

## Predefined Symbols

*Date and Time Symbol*

---

# Date and Time Symbol

Date and time symbol:

- [\\_\\_DATE\\_\\_ on page 116](#)
- [\\_\\_TIME\\_\\_ on page 116](#)

---

## **\_\_DATE\_\_**

Defined as the date during compilation.

During compilation, the compiler defines this macro with a character string representation of the current date.

---

## **\_\_TIME\_\_**

Defined as the time of day during compilation.

During compilation, the compiler defines this macro with a character string representation of the current time.

# IDE Symbol

IDE symbol:

- [\\_\\_ide\\_target\("target\\_name"\) on page 116](#)

---

## **\_\_ide\_target("target\_name")**

Returns 1 if *target\_name* is the same as the active build target in the CodeWarrior IDE's active project. Returns 0 otherwise.

## Name Symbols

Name symbols:

- [\\_FILE\\_ on page 117](#)
- [\\_LINE\\_ on page 117](#)

---

### **\_FILE\_**

The name of the source code file being compiled.

During compilation, the compiler defines this macro with a character string representation of the name of the file being compiled.

---

### **\_LINE\_**

The number of the line of source code being compiled.

During compilation, this macro is defined as an integer value representing the number of line of source code being compiled.

## Object Code Organization and Generation Symbol

Object code organization and generation symbol:

- [\\_profile\\_ on page 117](#)

---

### **\_profile\_**

Defined as 1 when generating object code that works with a profiler. Undefined otherwise.

## Predefined Symbols

*C Symbols*

---

# C Symbols

C symbol:

- [\\_\\_STDC\\_\\_ on page 118](#)

---

## **`__STDC__`**

Defined as 1 when compiling ISO Standard C source code, undefined otherwise.

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 C Standard. The compiler does not define this macro otherwise.

# Pragmas

---

The `#pragma` preprocessor directive specifies option settings to the compiler.

This chapter describes how to use pragmas and lists the pragmas that the compiler recognizes:

- [Using Pragmas on page 119](#)
- [Pragma Scope on page 127](#)
- [Standard C and C++ Conformance Pragmas on page 127](#)
- [Language Translation and Extensions Pragmas on page 129](#)
- [Errors, Warnings, and Diagnostic Control Pragmas on page 132](#)
- [Preprocessing and Precompilation Pragmas on page 151](#)
- [Library and Linking Control Pragmas on page 155](#)
- [Object Code Organization and Generation Pragmas on page 160](#)
- [Optimization Pragmas on page 175](#)
- [Profiler Pragmas on page 183](#)

## Using Pragmas

Pragma settings may be manipulated to control the compiler's code generation. The compiler has additional capabilities to manage pragma settings themselves:

- [Checking Pragma Settings on page 119](#)
- [Saving and Restoring Pragma Settings on page 124](#)
- [Determining Which Settings Are Saved and Restored on page 126](#)
- [Illegal Pragmas on page 126](#)

## Checking Pragma Settings

The preprocessor function `__option()` returns the state of pragma settings at compile-time. The syntax is

```
__option(setting-name)
```

where *setting-name* is the name of a pragma that accepts the `on`, `off`, and `reset` options.

## Pragmas

### Using Pragmas

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 options, the compiler issues a warning message.

[Listing 10.1 on page 120](#) shows an example.

#### Listing 10.1 Using the `__option()` preprocessor function

```
#if __option(ANSI_strict)
#include "portable.h" /* Use the portable declarations. */
#else
#include "custome.h" /* Use the specialized declarations. */
#endif
```

**Table 10.1 Preprocessor Setting Names for `__option()`**

This argument...	Corresponds to the...
<code>always_inline</code>	Pragma <code>always_inline</code> .
<code>ANSI_strict</code>	<b>ANSI Strict</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma <code>ANSI_strict</code> .
<code>auto_inline</code>	<b>Auto-Inline</b> setting of the <b>Inlining</b> menu in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma <code>auto_inline</code> .
<code>check_inline_sp_effects</code>	Pragma <code>check_inline_sp_effects</code> .
<code>const_strings</code>	Pragma <code>const_strings</code> .
<code>defer_codegen</code>	Pragma <code>defer_codegen</code> .
<code>dollar_identifiers</code>	Pragma <code>dollar_identifiers</code> .
<code>dont_inline</code>	<b>Don't Inline</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma <code>dont_inline</code> .
<code>dont_reuse_strings</code>	<b>Reuse Strings</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma <code>dont_reuse_strings</code> .



**Table 10.1 Preprocessor Setting Names for \_\_option() (continued)**

This argument...	Corresponds to the...
enumsalwaysint	<b>Enums Always Int</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma enumsalwaysint.
explicit_zero_data	Pragma explicit_zero_data.
factor1	Pragma factor1.
factor2	Pragma factor2.
factor3	Pragma factor3.
extended_errorcheck	<b>Extended Error Checking</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma extended_errorcheck.
fullpath_prepdump	Pragma fullpath_prepdump.
initializedzerodata	Pragma initializedzerodata.
inline_bottom_up	Pragma inline_bottom_up.
interrupt	Pragma interrupt.
line_prepdump	Pragma line_prepdump.
mpwc_newline	<b>Map newlines to CR</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma mpwc_newline.
mpwc_relax	<b>Relaxed Pointer Type Rules</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma mpwc_relax.
nofactor1	Pragma nofactor1.
nofactor2	Pragma nofactor2.
nofactor3	Pragma nofactor3.
only_std_keywords	<b>ANSI Keywords Only</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma only_std_keywords.
opt_common_subs	Pragma opt_common_subs.

## Pragmas

### Using Pragmas

**Table 10.1 Preprocessor Setting Names for \_\_option() (continued)**

This argument...	Corresponds to the...
opt_dead_assignments	Pragma opt_dead_assignments.
opt_dead_code	Pragma opt_dead_code.
opt_lifetimes	Pragma opt_lifetimes.
opt_loop_invariants	Pragma opt_loop_invariants.
opt_propagation	Pragma opt_propagation.
opt_strength_reduction	Pragma opt_strength_reduction.
opt_strength_reduction_strict	Pragma opt_strength_reduction_strict.
opt_unroll_loops	Pragma opt_unroll_loops.
optimize_for_size	Pragma optimize_for_size.
packstruct	Pragma pactstruct.
peephole	Pragma peephole.
pool_strings	<b>Pool Strings</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma pool_strings.
profile	Pragma profile.
readonly_strings	<b>Make String Read Only</b> setting in the M56800 Processor settings panel and pragma readonly_strings.
require_prototypes	<b>Require Function Prototypes</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma require_prototypes.
reverse_bitfields	Pragma reverse_bitfields.
simple_prepdump	Pragma simple_prepdump.
suppress_init_code	Pragma suppress_init_code.
suppress_warnings	Pragma suppress_warnings.
syspath_once	Pragma syspath_once.

**Table 10.1 Preprocessor Setting Names for \_\_option() (continued)**

This argument...	Corresponds to the...
unsigned_char	<b>Use Unsigned Chars</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma unsigned_char.
warn_any_ptr_int_conv	<b>Pragma</b> warn_any_ptr_int_conv.
warn_emptydecl	<b>Empty Declarations</b> setting in the <a href="#">C/C++ Language (C only) Settings Panel on page 16</a> and pragma warn_emptydecl.
warn_extracomma	<b>Extra Commas</b> setting in the <a href="#">C/C++ Preprocessor Panel on page 20</a> and pragma warn_extracomma.
warn_filenameecaps	<b>Pragma</b> warn_filenameecaps.
warn_filenameecaps_system	<b>Pragma</b> warn_filenameecaps_system.
warn_illegal_instructions	<b>Pragma</b> warn_illegal_instructions.
warn_illpragma	<b>Illegal Pragmas</b> setting in the panel and pragma warn_illpragma.
warn_impl_f2i_conv	<b>Pragma</b> warn_impl_f2i_conv.
warn_impl_i2f_conv	<b>Pragma</b> warn_impl_i2f_conv.
warn_impl_s2u_conv	<b>Pragma</b> warn_impl_s2u_conv.
warn_implicitconv	<b>Implicit Arithmetic Conversions</b> setting in the <a href="#">C/C++ Preprocessor Panel on page 20</a> and pragma warn_implicitconv.
warn_largeargs	<b>Pragma</b> warn_largeargs.
warn_missingreturn	<b>Pragma</b> warn_missingreturn
warn_no_side_effect	<b>Pragma</b> warn_no_side_effect.
warn_notinlined	<b>Non-Inlined Functions</b> setting in the <a href="#">C/C++ Preprocessor Panel on page 20</a> and pragma warn_notinlined.
warn_padding	<b>Pragma</b> warn_padding.

**Table 10.1 Preprocessor Setting Names for `__option()` (continued)**

This argument...	Corresponds to the...
<code>warn_possunwant</code>	<b>Possible Errors</b> setting in the <a href="#">C/C++ Preprocessor Panel on page 20</a> and pragma <code>warn_possunwant</code> .
<code>warn_ptr_int_conv</code>	Pragma <code>warn_ptr_int_conv</code>
<code>warn_resultnotused</code>	Pragma <code>warn_resultnotused</code> .
<code>warn_undefmacro</code>	Pragma <code>warn_undefmacro</code> .
<code>warn_unusedarg</code>	<b>Unused Arguments</b> setting in the <a href="#">C/C++ Preprocessor Panel on page 20</a> and pragma <code>warn_unusedarg</code> .
<code>warn_unusedvar</code>	<b>Unused Variables</b> setting in the <a href="#">C/C++ Preprocessor Panel on page 20</a> and pragma <code>warn_unusedvar</code> .
<code>warning_errors</code>	<b>Treat Warnings As Errors</b> setting in the <a href="#">C/C++ Preprocessor Panel on page 20</a> and pragma <code>warning_errors</code> .

## Saving and Restoring Pragma Settings

There are some occasions when you would like to apply pragma settings to a piece of source code independently from the settings in the rest of the source file. For example, a function might require unique optimization settings that should not be used in the rest of the function's source file.

Remembering which pragmas to save and restore is tedious and error-prone. Fortunately, the compiler has mechanisms that save and restore pragma settings at compile time. All pragma settings and some individual pragma 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 10.2 on page 125](#) shows an example.

---

**Listing 10.2 Using push and pop to save and restore pragma settings**

---

```
/* Settings for this file. */
#pragma opt_unroll_loops on
#pragma optimize_for_size off
void fast_func_A(void)
{
/* ... */
}

/* Settings for slow_func(). */
#pragma push /* Save file settings. */
#pragma optimization_size 0
void slow_func(void)
{
/* ... */
}
#pragma pop /* Restore file settings. */

void fast_func_B(void)
{
/* ... */
}
```

---

Pragmas that have a `reset` option perform the same actions as pragmas `push` and `pop`, but apply to a single pragma. A pragma's `on` and `off` settings save the pragma's current setting before changing it to the new setting. A pragma's `reset` option restores the pragma's setting. The `on/off` and `reset` options may be nested to an unlimited depth. [Listing 10.3 on page 125](#) shows an example.

---

**Listing 10.3 Using the reset option to save and restore a pragma setting**

---

```
/* Setting for this file. */
#pragma opt_unroll_loops on

void fast_func_A(void)
{
/* ... */
}

/* Setting for smallslowfunc(). */
#pragma opt_unroll_loops off
void small_func(void)
{
/* ... */
}
/* Restore previous setting. */
```

---

## Pragmas

### Using Pragmas

---

```
#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 10.4 on page 126](#) shows an example that checks if the ANSI\_strict pragma setting is saved and restored by pragmas push and pop.

### Listing 10.4 Testing if pragmas push and pop save and restore a setting

---

```
/* Preprocess this source code. */
#pragma ANSI_strict on
#pragma push
#pragma ANSI_strict off
#pragma pop
#if __option(ANSI_strict)
#error "Saved and restored by push and pop."
#else
#error "Not affected by push and pop."
#endif
```

---

## 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 10.5 on page 126](#) generate warnings with the **Illegal Pragmas** setting enabled.

### Listing 10.5 Illegal Pragmas

---

```
#pragma near_data off // WARNING: near_data is not a pragma.
#pragma ANSI_strict select // WARNING: select is not defined
#pragma ANSI_strict on // OK
```

---

The **Illegal Pragmas** setting corresponds to the pragma `warn_illpragma`, described at [“warn\\_illpragma” on page 141](#). To check this setting, use `__option` (`warn_illpragma`).

See [Checking Pragma Settings on page 119](#) for information on how to use this directive.

## 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 compilers 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

## Standard C and C++ Conformance Pragmas

The 56800x has the following pragmas:

- [on page 127 on page 127](#)[ANSI\\_strict on page 127](#)
- [only\\_std keywords on page 129](#)

---

### 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 if it encounters any of the following common ANSI extensions:

- C++-style comments. [Listing 10.6 on page 128](#) shows an example.

## Pragmas

### Standard C and C++ Conformance Pragmas

---

#### Listing 10.6 C++ Comments

---

```
a = b;    // This is a C++-style comment
```

---

- Unnamed arguments in function definitions. [Listing 10.7 on page 128](#) shows an example.

#### Listing 10.7 Unnamed Arguments

---

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

---

- A # token that does not appear before an argument in a macro definition. [Listing 10.8 on page 128](#) shows an example.

#### Listing 10.8 Using # in Macro Definitions

---

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

#define add2(x) #x "2"
    /* ALWAYS OK: add2(abc) creates "abc2" */
```

---

- An identifier after #endif. [Listing 10.9 on page 128](#) shows an example.

#### Listing 10.9 Identifiers After #endif

---

```
#ifdef __MWERKS__
    /* . . . */
#endif __MWERKS__ /* OK, if ANSI_strict is disabled */

#ifdef __MWERKS__
    /* . . . */
#endif /* __MWERKS__ */ /* ALWAYS OK */
```

---

This pragma corresponds to the **ANSI Strict** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`ANSI_strict`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.



## only\_std\_keywords

Controls the use of ISO keywords.

### Syntax

```
#pragma only_std_keywords on | off | reset
```

### Remarks

The C/C++ compiler recognizes additional reserved keywords. If you are writing code that must follow the ANSI standard strictly, enable the pragma `only_std_keywords`.

This pragma corresponds to the **ANSI Keywords Only** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`only_std_keywords`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

# Language Translation and Extensions Pragmas

The 56800x has the following pragmas:

- [gcc\\_extensions on page 129](#)
- [mpwc\\_newline on page 130](#)
- [mpwc\\_relax on page 131](#)

---

## 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. [Listing 10.10 on page 130](#) provides an example.

## Pragmas

### Language Translation and Extensions Pragmas

---

#### Listing 10.10 Example of Array Initialization with a Non-const Value

---

```
int foo(int arg)
{
    int arr[2] = { arg, arg+1 };
}
```

---

- `sizeof( void ) == 1`
- `sizeof( function-type ) == 1`
- Limited support for GCC statements and declarations within expressions. [Listing 10.11 on page 130](#) provides an example.

#### Listing 10.11 Example of GCC Statements and Declarations Within Expressions

---

```
#pragma gcc_extensions on
#define POW2(n) ({ int i,r; for(r=1,i=n; i>0; --i) r<<=1; r;})

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

---

This feature only works for expressions in function bodies.

- Macro redefinitions without a previous `#undef`.
- The GCC keyword `typeof`.

This pragma does not correspond to any setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check the global optimizer, use `__option` (`gcc_extensions`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## mpwc\_newline

Controls the use of newline character convention used by the Apple MPW C.

### Syntax

```
#pragma mpwc_newline on | off | reset
```

### Remarks

If you enable this pragma, the compiler uses the MPW conventions for the '\n' and '\r' characters. Otherwise, the compiler uses the Freescale C/C++ conventions for these characters.

In MPW, '\n' is a Carriage Return (0x0D) and '\r' is a Line Feed (0x0A). In Freescale C/C++, they are reversed: '\n' is a Line Feed and '\r' is a Carriage Return.

If you enable this pragma, use ANSI C/C++ libraries that were compiled when this pragma was enabled.

If you enable this pragma and use the standard ANSI C/C++ libraries, you cannot read and write '\n' and '\r' properly. For example, printing '\n' brings you to the beginning of the current line instead of inserting a newline.

This pragma corresponds to the **Map newlines to CR** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`mpwc_newline`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

Enabling this setting is not useful for the DSP56800 target.

---

## 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. This setting is especially useful if you are using code written before the ANSI C standard. This old source code frequently used these types interchangeably.

This setting has no effect on C++ source code.

You can use this pragma to relax function pointer checking:

---

```
#pragma mpwc_relax on
extern void f(char *);
extern void(*fp1)(void *) = &f; // error but allowed
extern void(*fp2)(unsigned char *) = &f; // error but allowed
```

---

This pragma corresponds to the **Relaxed Pointer Type Rules** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option`

## Pragmas

### *Errors, Warnings, and Diagnostic Control Pragmas*

---

(`mpwc_relax`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

## Errors, Warnings, and Diagnostic Control Pragmas

The 56800x has the following pragmas:

- [check\\_c\\_src\\_pipeline on page 133](#)
- [check\\_inline\\_asm\\_pipeline on page 133](#)
- [check\\_inline\\_sp\\_effects on page 134](#)
- [extended\\_errorcheck on page 134](#)
- [require\\_prototypes on page 135](#)
- [suppress\\_init\\_code on page 135](#)
- [suppress\\_warnings on page 136](#)
- [unsigned\\_char on page 136](#)
- [unused on page 137](#)
- [warn\\_any\\_ptr\\_int\\_conv on page 138](#)
- [warn\\_emptydecl on page 138](#)
- [warn\\_extracomma on page 139](#)
- [warn\\_filenamecaps on page 140](#)
- [warn\\_filenamecaps\\_system on page 140](#)
- [warn\\_illpragma on page 141](#)
- [warn\\_impl\\_f2i\\_conv on page 141](#)
- [warn\\_impl\\_i2f\\_conv on page 142](#)
- [warn\\_impl\\_s2u\\_conv on page 143](#)
- [warn\\_implicitconv on page 144](#)
- [warn\\_largeargs on page 145](#)
- [warn\\_missingreturn on page 145](#)
- [warn\\_no\\_side\\_effect on page 146](#)
- [warn\\_notinlined on page 146](#)
- [warn\\_padding on page 147](#)
- [warn\\_padding on page 147](#)
- [warn\\_possunwant on page 147](#)

- [warn\\_ptr\\_int\\_conv on page 148](#)
  - [warn\\_resultnotused on page 148](#)
  - [warn\\_undefmacro on page 149](#)
  - [warn\\_unusedarg on page 150](#)
  - [warn\\_unusedvar on page 150](#)
  - [warning\\_errors on page 150](#)
- 

## **check\_c\_src\_pipeline**

This pragma controls detection of a pipeline conflict in the C language code.

### **Compatibility**

This pragma is not compatible with the DSP56800 compiler, but it is compatible with the DSP56800E compiler.

### **Syntax**

```
#pragma check_c_src_pipeline [off|conflict]
```

### **Remarks**

Use this pragma for extra validation of generated C code. The compiler already checks for pipeline conflicts; this pragma tells the compiler to add another check for pipeline conflicts. Should this pragma detect a pipeline conflict, it issues an error message.

---

**NOTE** The pipeline conflicts that this pragma finds are rare. Should this pragma report such a conflict with your code, you should report the matter to Freescale.

---

## **check\_inline\_asm\_pipeline**

This pragma controls detection of a pipeline conflicts and stalls in assembly language source code.

### **Compatibility**

This pragma is not compatible with the DSP56800 compiler, but it is compatible with the DSP56800E compiler.

---

## Pragmas

### Errors, Warnings, and Diagnostic Control Pragmas

---

#### Syntax

```
#pragma check_inline_asm_pipeline
    [off|conflict|conflict_and_stall]
```

#### Remarks

Use this pragma to detect a source-code, assembly language pipeline conflict or stall, then generate an error message. In some cases, the source code can be a mix of assembly language and C language.

The option `conflict` only detects and generates error messages for pipeline conflict.

The option `conflict_and_stall` detects and generates error messages for pipeline conflicts and stalls.

---

## check\_inline\_sp\_effects

Generates a warning if the user specifies an inline assembly instruction which modifies the SP by a run-time dependent amount.

#### Syntax

```
#pragma check_inline_sp_effects on | off | reset
```

#### Remarks

If this pragma is not specified off, instructions which modify the SP by a run-time dependent amount are ignored. In this case, stack-based references may be silently wrong. This pragma is added for compatibility with existing code which may have run-time modifications of the SP already. However, known compile times inconsistencies in SP modifications are always flagged as errors, since the SP must be correct to return from functions.

This pragma does not correspond to any panel setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (check_inline_sp_effects)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## extended\_errorcheck

Controls the issuing of warnings for possible unintended logical errors.

**Syntax**

```
#pragma extended_errorcheck on | off | reset
```

**Remarks**

If you enable this pragma, the C compiler generates a warning (not an error) if it encounters some common programming errors.

This pragma corresponds to the **Extended Error Checking** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option` (`extended_errorcheck`), described in [Checking Pragma Settings on page 119](#). 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 works for non-static functions.

If you enable this pragma, the compiler generates an error if you use a function that does not have a prototype. This pragma helps you prevent errors that happen when you use a function before you define it or refer to it.

This pragma corresponds to the **Require Function Prototypes** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`require_prototypes`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

**suppress\_init\_code**

Controls the suppression of static initialization object code.

**Syntax**

```
#pragma suppress_init_code on | off | reset
```

---

## Pragmas

### Errors, Warnings, and Diagnostic Control Pragmas

---

#### Remarks

If you enable this pragma, the compiler does not generate any code for static data initialization.

---

**WARNING!** Beware when using this pragma because it can produce erratic or unpredictable behavior in your program.

---

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (suppress_init_code)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## suppress\_warnings

Controls the issuing of warnings.

#### Syntax

```
#pragma suppress_warnings on | off | reset
```

#### Remarks

If you enable this pragma, the compiler does not generate warnings, including those that are enabled.

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (suppress_warnings)`, described in [Checking Pragma Settings on page 119](#). 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 ANSI libraries included with CodeWarrior.

---

This pragma corresponds to the **Use Unsigned Chars** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (unsigned_char)`, described in [Checking Pragma Settings on page 119](#). By default, this setting is disabled.

---

## unused

Controls the suppression of warnings for variables and parameters that are not referenced in a function.

### Syntax

```
#pragma unused ( var_name [, var_name ]... )
```

### Remarks

This pragma suppresses the compile time warnings for the unused variables and parameters specified in its argument list. You can use this pragma only within a function body, and the listed variables must be within the scope of the function.

### Listing 10.12 Example of Pragma unused() in C

---

```
#pragma warn_unusedvar on // See pragma on page 150 on page 150
#pragma warn_unusedarg on // See pragma warn\_unusedarg on page 150.

static void ff(int a)
{
    int b;
#pragma unused(a,b) // Compiler does not warn
                    // that a and b are unused
                    // . . .
}
```

---

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). By default, this pragma is disabled.

---

## Pragmas

### Errors, Warnings, and Diagnostic Control Pragmas

---

#### warn\_any\_ptr\_int\_conv

Controls if the compiler generates a warning 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 pointer portability issues. An example is shown in [Listing 10.13 on page 138](#).

#### Listing 10.13 Example of warn\_any\_ptr\_int\_conv

---

```
#pragma warn_ptr_int_conv on
short i, *ip

void foo() {
    i = (short)ip;    // WARNING: integral type is not large
                    // 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
}
```

---

See also [warn\\_ptr\\_int\\_conv on page 148](#).

This pragma corresponds to the **Pointer/Integral Conversions** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option` (`warn_any_ptr_int_conv`), described in [Checking Pragma Settings on page 119](#). 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 when it encounters a declaration with no variables.

**Listing 10.14 Example of Pragma warn\_emptydecl**

---

```
int ;      // WARNING  
int i;    // OK
```

---

This pragma corresponds to the **Empty Declarations** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_emptydecl)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

**warn\_extracomma**

Controls the recognition of superfluous commas.

**Syntax**

```
#pragma warn_extracomma on | off | reset
```

**Remarks**

If you enable this pragma, the compiler issues a warning when it encounters an extra comma.

**Listing 10.15 Example of Pragma warn\_extracomma**

---

```
enum {l,m,n,o,}; // WARNING: When the warning is enabled, it will  
                // generate :
```

---

This pragma corresponds to the **Extra Commas** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_extracomma)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## Pragmas

### Errors, Warnings, and Diagnostic Control Pragmas

---

#### warn\_filenamecaps

Controls the recognition of conflicts involving case-sensitive filenames within user includes.

##### Syntax

```
#pragma warn_filenamecaps on | off | reset
```

##### Remarks

If you enable this pragma, the compiler issues a warning when an `include` directive capitalizes a filename within a user include differently from the way the filename appears on a disk. It also recognizes 8.3 DOS filenames in Windows when a long filename is available. This pragma helps avoid porting problems to operating systems with case-sensitive filenames.

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 on page 140](#).

This pragma does not correspond to any panel setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_filenamecaps)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

#### 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, the compiler issues a warning when an `include` directive capitalizes a filename within a system include differently from the way the filename appears on a disk. It also recognizes 8.3 DOS filenames in Windows when a long filename is available. This pragma helps avoid porting problems to operating systems with case-sensitive filenames.

To check the spelling of system includes such as the following:

```
#include <file>
```

use this pragma along with the [warn\\_filenamecaps on page 140](#) pragma.

This pragma does not correspond to any panel setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option` (`warn_filenamecaps_system`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

### warn\_illpragma

Controls the recognition of illegal pragma directives.

#### Syntax

```
#pragma warn_illpragma on | off | reset
```

#### Remarks

If you enable this pragma, the compiler displays a warning when it encounters a pragma it does not support. For more information about this warning, see [“Illegal Pragmas” on page 126](#).

This pragma corresponds to the **Illegal Pragmas** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option` (`warn_illpragma`), described in [Checking Pragma Settings on page 119](#). By default, this setting is disabled.

---

### warn\_impl\_f2i\_conv

Controls the issuing of warnings 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 for implicitly converting floating-point values to integral values. [Listing 10.16 on page 141](#) provides an example.

#### Listing 10.16 Example of Implicit float-to-int Conversion

---

```
#pragma warn_implicit_conv on
#pragma warn_impl_f2i_conv on
```

---

## Pragmas

### Errors, Warnings, and Diagnostic Control Pragmas

---

```
float f;
signed int si;

int main()
{
    si = f;    // WARNING

#pragma warn_impl_f2i_conv off
    si = f;    // OK
}
```

---

Use this pragma along with the [warn\\_implicitconv on page 144](#) pragma.

This pragma does not correspond to any panel setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_impl_f2i_conv)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is enabled.

---

## warn\_impl\_i2f\_conv

Controls the issuing of warnings for implicit int-to-float conversions.

### Syntax

```
#pragma warn_impl_i2f_conv on | off | reset
```

### Remarks

If you enable this pragma, the compiler issues a warning for implicitly converting integral values to floating-point values. [Listing 10.17 on page 142](#) provides an example.

### Listing 10.17 Example of Implicit int-to-float Conversion

---

```
#pragma warn_implicit_conv on
#pragma warn_impl_i2f_conv on

float f;
signed int si;

int main()
{
    f = si;    // WARNING

#pragma warn_impl_i2f_conv off
    f = si;    // OK
}
```

---

```
}
```

---

Use this pragma along with the [warn\\_implicitconv on page 144](#) pragma.

This pragma does not correspond to any panel setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_impl_i2f_conv)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## warn\_impl\_s2u\_conv

Controls the issuing of warnings 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 for implicitly converting either from `signed int` to `unsigned int` or vice versa. [Listing 10.18 on page 143](#) provides an example.

#### Listing 10.18 Example of Implicit Conversions Between Signed int and unsigned int

---

```
#pragma warn_implicit_conv on
#pragma warn_impl_s2u_conv on

signed int si;
unsigned int ui;

int main()
{
    ui = si;    // WARNING
    si = ui;    // WARNING

#pragma warn_impl_s2u_conv off
    ui = si;    // OK
    si = ui;    // OK
}
```

---

Use this pragma along with the [warn\\_implicitconv on page 144](#) pragma.

---

## Pragmas

### Errors, Warnings, and Diagnostic Control Pragmas

---

This pragma does not correspond to any panel setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_impl_s2u_conv)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is enabled.

---

## warn\_implicitconv

Controls the issuing of warnings for all implicit arithmetic conversions.

### Syntax

```
#pragma warn_implicitconv on | off | reset
```

### Remarks

If you enable this pragma, the compiler issues a warning for all implicit arithmetic conversions when the destination type might not represent the source value. [Listing 10.19 on page 144](#) provides an example.

#### Listing 10.19 Example of Implicit Conversion

---

```
#pragma warn_implicitconv on

float f;
signed int si;
unsigned int ui;

int main()
{
    f = si;    // OK
    si = f;    // WARNING
    ui = si;   // WARNING
    si = ui;   // WARNING
}
```

---

The default setting for `warn_impl_i2fconf` pragma is disabled. Use the `warn_implicitconv` pragma along with the `warn_impl_i2f_conv` pragma to generate the warning for the int-to-float conversion.

This pragma corresponds to the **Implicit Arithmetic Conversions** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_implicitconv)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---



## warn\_largeargs

Controls the issuing of warnings for passing non-integer numeric values to unSyntaxd functions.

### Syntax

```
#pragma warn_largeargs on | off | reset
```

### Remarks

If you enable this pragma, the compiler issues a warning if you attempt to pass a non-integer numeric value, such as a float or long long, to an unSyntaxd function when the [require prototypes on page 135](#) pragma is disabled.

This pragma does not correspond to any panel setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_largeargs)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## warn\_missingreturn

Issues a warning 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 10.20 on page 145](#).

### Listing 10.20 Example of warn\_missingreturn pragma

---

```
#pragma warn_missingreturn on

int foo()
{
    // no return statement in foo()
}
// generates a warning: return value expected
```

---

This pragma corresponds to the **Missing ‘return’ Statements** option in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option`

---

## Pragmas

### Errors, Warnings, and Diagnostic Control Pragmas

---

(`warn_missingreturn`), described in [Checking Pragma Settings on page 119](#).

By default, this pragma is set to the same value as `__option` (`extended_errorcheck`).

---

## warn\_no\_side\_effect

Controls the issuing of warnings for redundant statements.

### Syntax

```
#pragma warn_no_side_effect on | off | reset
```

### Remarks

If you enable this pragma, the compiler issues a warning when it encounters a statement that produces no side effect. To suppress this warning, cast the statement with `(void)`. [Listing 10.21 on page 146](#) provides an example.

#### Listing 10.21 Example of Pragma `warn_no_side_effect`

---

```
#pragma warn_no_side_effect on
void foo(int a,int b)
{
    a+b;           // WARNING: expression has no side effect
    (void)(a+b);  // void cast suppresses warning
}
```

---

This pragma does not correspond to any panel setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option` (`warn_no_side_effect`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## warn\_notinlined

Controls the issuing of warnings for functions the compiler cannot inline.

### Syntax

```
#pragma warn_notinlined on | off | reset
```

---

**Remarks**

The compiler issues a warning for non-inlined inline function calls.

This pragma corresponds to the **Non-Inlined Functions** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_notinlined)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

**warn\_padding**

Controls the issuing of warnings for data structure padding.

**Syntax**

```
#pragma warn_padding on | off | reset
```

**Remarks**

If you enable this pragma, the compiler warns about any bytes that were implicitly added after an ANSI C `struct` member to improve memory alignment.

This pragma corresponds to the **Pad Bytes Added** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_padding)`, described in [Checking Pragma Settings on page 119](#). By default, this setting is disabled.

---

**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 errors that are legal C/C++ but might produce unexpected results, such as putting in unintended semicolons or confusing `=` and `==`.

This pragma corresponds to the **Possible Errors** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_possunwant)`, described in [Checking Pragma Settings on page 119](#). By default, this setting is disabled.

---

## Pragmas

### Errors, Warnings, and Diagnostic Control Pragmas

---

#### 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 if an expression attempts to convert a pointer value to an integral type that is not large enough to hold the pointer value.

#### Listing 10.22 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.”](#)

This pragma corresponds to the **Pointer / Integral Conversions** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_ptr_int_conv)`, described in [Checking Pragma Settings on page 119](#). By default, this setting is disabled.

---

#### warn\_resultnotused

Controls the issuing of warnings when function results are ignored.

##### Syntax

```
#pragma warn_resultnotused on | off | reset
```

##### Remarks

If you enable this pragma, the compiler issues a warning when it encounters a statement that calls a function without using its result. To prevent this, cast the statement with `(void)`. [Listing 10.23 on page 149](#) provides an example.

---

---

**Listing 10.23 Example of Function Calls with Unused Results**

---

```
#pragma warn_resultnotused on

extern int bar();
void foo()
{
    bar();           // WARNING: result of function call is not used
    (void)bar();    // 'void' cast suppresses warning
}
```

---

This pragma does not correspond to any panel setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_resultnotused)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

**warn\_undefmacro**

Controls the detection of undefined macros in `#if / #elif` conditionals.

**Syntax**

```
#pragma warn_undefmacro on | off | reset
```

**Remarks**

[Listing 10.24 on page 149](#) provides an example.

**Listing 10.24 Example of Undefined Macro**

---

```
#if UNDEFINEDMACRO == 4 // WARNING: undefined macro
                        // 'UNDEFINEDMACRO' used in
                        // #if/#elif conditional
```

---

Use this pragma to detect the use of undefined macros (especially expressions) where the default value 0 is used.

**NOTE** A warning is only issued when a macro is evaluated. A short-circuited “&&” or “|” test or unevaluated “?:” will not produce a warning.

---

This pragma corresponds to the **Undefined Macro in #if** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_undefmacro)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is `off`.

---

## Pragmas

### Errors, Warnings, and Diagnostic Control Pragmas

---

#### 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 when it encounters an argument you declare but do not use. To suppress this warning in C++ source code, leave an argument identifier out of the function parameter list.

This pragma corresponds to the **Unused Arguments** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_unusedarg)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

#### 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 when it encounters a variable you declare but do not use.

This pragma corresponds to the **Unused Variables** setting in the [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warn_unusedvar)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

#### 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 warnings 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 [C/C++ Warnings Panel on page 22](#). To check this setting, use `__option (warning_errors)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

## Preprocessing and Precompilation Pragmas

The 56800x has the following pragmas:

- [dollar\\_identifiers on page 151](#)
- [fullpath\\_prepdump on page 152](#)
- [mark on page 152](#)
- [notonce on page 153](#)
- [once on page 153](#)
- [pop\\_push on page 153](#)
- [syspath\\_once on page 154](#)

---

### 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, alphabetic, and numeric characters in an identifier.

This pragma does not correspond to any panel setting. To check this setting, use the `__option (dollar_identifiers)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

## Pragmas

### Preprocessing and Precompilation Pragmas

---

#### 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 does not correspond to any panel setting. To check this setting, use the `__option` (`fullpath_prepdump`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

#### 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 setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). By default, this pragma is disabled.



## 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, `include` statements can be repeatedly included, even if you have enabled `#pragma once on`. For more information, see [“once” on page 153](#).

This pragma does not correspond to any setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). By default, this pragma is disabled.

---

## once

Controls whether or not a header file can be included more than once in the same source file.

### Syntax

```
#pragma once [ on ]
```

### Remarks

Use this pragma to ensure that the compiler includes header files only once in a source file.

There are two versions of this pragma: `#pragma once` and `#pragma once on`. Use `#pragma once` in a header file to ensure that the header file is included only once in a source file. Use `#pragma once on` in a header file or source file to insure that *any* file is included only once in a source file.

This pragma does not correspond to any setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). By default, this pragma is disabled.

---

## pop, push

Save and restore pragma settings.

---

## Pragmas

### Preprocessing and Precompilation Pragmas

---

#### 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 10.25 on page 154](#).

#### Listing 10.25 push and pop Example

---

```
#pragma peephole on
#pragma packstruct on
#pragma push      // push all compiler settings
#pragma peephole off
#pragma packstruct off
                  // pop restores "peephole" and "packstruct"
#pragma pop
```

---

If you are writing new code and need to set a pragma setting to its original value, use the reset argument, described in [“Using Pragmas” on page 119](#).

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#).

---

## syspath\_once

Controls how include files are treated.

#### Syntax

```
#pragma syspath_once on | off | reset
```

#### Remarks

If you enable this pragma, files called in `#include <>` and `#include "` directives are treated as distinct, even if they refer to the same file.

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (syspath_once)`, described in [Checking Pragma Settings on page 119](#). By default, this setting is enabled. For example, the same include file could reside in two distinct directories.

---

---

# Library and Linking Control Pragmas

The 56800x has the following pragmas:

- [define\\_section on page 155](#)
- [explicit\\_zero\\_data on page 156](#)
- [initializedzerodata on page 157](#)
- [section on page 157](#)
- [use\\_rodata on page 158](#)

---

## define\_section

This pragma controls the definition of a custom section.

### Syntax

```
#pragma define_section <sectname> <istring> [ <ustring> ]  
[ <accmode> ]
```

### Remarks

Arguments:

<sectname>

Identifier by which this user-defined section is referenced in the source, that is, via the following instructions:

- #pragma section <sectname> begin
- \_\_declspec(<sectname>)

<istring>

Section name string for initialized data assigned to <section>.

For example:

```
".data"
```

Optional Arguments:

<ustring>

Section name string for uninitialized data assigned to <section>. If ustring is not specified then istring is used.

<accmode>

One of the following indicates the attributes of the section

## Pragmas

### Library and Linking Control Pragmas

---

**Table 10.2 Section Attributes**

R	readable
RW (default)	readable and writable
RX	readable and executable
RWX	readable, writable, and executable

---

**NOTE** For an example of `define_section`, see [Listing 10.26 on page 158](#).

---

### Related Pragma

[section on page 157](#)

---

## explicit\_zero\_data

Controls the section where zero-initialized global variables are emitted.

### Syntax

```
#pragma explicit_zero_data on | off | reset
```

### Remarks

If you enable this pragma, zero-initialized global variables are emitted to the `.data` section (which is normally stored in ROM) instead of the `.BSS` section. This results in a larger ROM image. This pragma should be enabled if customized startup code is used and it does not initialize the `.BSS` section. The `.BSS` section is initialized to zero by the default CodeWarrior startup code.

This pragma does not correspond to any setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option(explicit_zero_data)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

**NOTE** The pragmas `explicit_zero_data` and `initializedzerodata` are the same, however, the preferred syntax is `explicit_zero_data`.

---

## initializedzerodata

Controls the section where zero-initilaized global variables are emitted.

### Syntax

```
#pragma initializedzerodata on | off | reset
```

### Remarks

If you enable this pragma, zero-initilaized global variables are emitted to the .data section (which is normally stored in ROM) instead of the .BSS section. This results in a larger ROM image. This pragma should be enabled if customized startup code is used and it does not initialize the .BSS section. The .BSS section is initialized to zero by the default CodeWarrior startup code.

This pragma does not correspond to any setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option(initializedzerodata)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

**NOTE** The pragmas `initializedzerodata` and `explicit_zero_data` are the same, however, the preferred syntax is `explicit_zero_data`.

---

---

## section

This pragma controls the organization of object code.

### Syntax

```
#pragma section <sectname> begin  
[...data...]  
#pragma section <sectname> end
```

### Remarks

Argument:

<sectname>

Identifier by which this user-defined section is referenced in the source.

## Pragmas

### Library and Linking Control Pragmas

---

#### Listing 10.26 Sample Code - pragma define\_section and pragma section

---

```

/* 1. Define the section */
#pragma define_section mysection ".mysection.data" RW

/* 2. Specify the data to be put into the section. */
#pragma section mysection begin
int a[10] = {'0','1','2','3','4','5','6','7','8','9'};
int b[10];
#pragma section mysection end

int main(void) {
    int i;
    for (i=0;i<10;i++)
        b[i]=a[i];
}

/* 3. In the linker command file, add ".mysection.data" in the ".data"
sections area of the linker command file by inserting the following
line:
    * (.mysection.data)
*/

```

---

#### Related Pragma

[define\\_section on page 155](#)

---

## use\_rodata

Controls the section where constant data is emitted.

### Compatibility

This pragma is compatible with the DSP56800, but it is not compatible with the DSP56800E.

### Syntax

```
#pragma use_rodata [ on | off | reset ]
```

### Remarks

By default, the compiler emits const defined data to the .data section. There are two ways to cause the compiler to emit const defined data to the .rodata section:

1. Setting the “write const data to .rodata section” option in the M56800 Processor Settings panel.

This method is a global change and emits all const-defined data to the .rodata section for the current build target.

2. Using `#pragma use_rodata [on | off | reset]`.

on	Write const data to .rodata section.
off	Write const data to .data section.
reset	Toggle pragma state.

To use this pragma, place the pragma before the const data that you wish the compiler to emit to the .rodata section. This method overrides the target setting and allows a subset of constant data to be emitted to or excluded from the .rodata section.

To see the usage of the pragma `use_rodata` see the code example in [Listing 10.27 on page 159](#).

---

#### **Listing 10.27 Sample Code \_ Pragma use\_rodata**

```
const UInt16 len_l_mult_ls_data = sizeof(l_mult_ls_data) /
sizeof(Frac32) ;
const Int16 g = a+b+c;

#pragma use_rodata on
const Int16 d[]={0xdddd};
const Int16 e[]={0xeeee};
const Int16 f[]={0xffff};
#pragma use_rodata off

main()
{
    // ... code
}
```

---

You must then appropriately locate the .rodata section created by the compiler using the linker command file. For example, see [Listing 10.28 on page 159](#).

---

#### **Listing 10.28 Sample Linker Command File - Pragma use\_rodata**

```
MEMORY {
    .text_segment (RWX) : ORIGIN = 0x2000, LENGTH = 0x00000000
    .data_segment (RW) : ORIGIN = 0x3000, LENGTH = 0x00000000
    .rodata_segment (R) : ORIGIN = 0x5000, LENGTH = 0x00000000
}
SECTIONS {
    .main_application :
```

---

## Pragmas

### Object Code Organization and Generation Pragmas

---

```

{
    # .text sections
} > .text_segment

.main_application_data :
{
    # .data sections
    # .bss sections
} > .data_segment

.main_application_constant_data:
{
    # constant data sections
    * (.rodata)
} > .rodata_segment
}

```

---

## Object Code Organization and Generation Pragmas

The 56800x has the following pragmas:

- [always inline on page 161](#)
- [auto inline on page 161](#)
- [const strings on page 161](#)
- [defer codegen on page 162](#)
- [dont inline on page 163](#)
- [dont reuse strings on page 163](#)
- [enumsalwaysint on page 164](#)
- [inline bottom up on page 165](#)
- [interrupt \(for the DSP56800\) on page 166](#)
- [interrupt \(for the DSP56800E\) on page 168](#)
- [packstruct on page 172](#)
- [pool strings on page 172](#)
- [readonly strings on page 173](#)
- [reverse bitfields on page 173](#)
- [suppress init code on page 174](#)
- [syspath once on page 174](#)



## always\_inline

Controls the use of inlined functions.

### Syntax

```
#pragma always_inline on | off | reset
```

### Remarks

This pragma is strongly deprecated. Use the **Inline Depth** pull-down menu of the [C/C++ Language \(C only\) Settings Panel on page 16](#) instead.

If you enable this pragma, the compiler ignores all inlining limits and attempts to inline all functions where it is legal to do so.

This pragma does not correspond to any panel setting. To check this setting, use `__option` (`always_inline`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## 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.

This pragma corresponds to the **Auto-Inline** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`auto_inline`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## const\_strings

Controls the const-ness of string literals.

---

## Pragmas

### Object Code Organization and Generation Pragmas

---

#### Syntax

```
#pragma const_strings [ on | off | reset ]
```

#### Remarks

If you enable this pragma, the compiler will generate a warning when string literals are not declared as const. [Listing 10.29 on page 162](#) shows an example.

#### Listing 10.29 const\_strings example

---

```
char *string1 = "hello";           /*OK, if const_strings is disabled*/
const char *string2 = "world";    /* Always OK */
```

---

This pragma does not correspond to any setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (const_strings)`, described in [Checking Pragma Settings on page 119](#).

---

## defer\_codegen

Controls the inlining of functions that are not yet compiled.

#### Syntax

```
#pragma defer_codegen on | off | reset
```

#### Remarks

This setting lets you use inline and auto-inline functions that are called before their definition:

#### Listing 10.30 defer\_codegen example

---

```
#pragma defer_codegen on
#pragma auto_inline on

extern void f();
extern void g();

main()
{
    f(); // will be inlined
    g(); // will be inlined
}

inline void f() {}
```

---

---

```
void g() {}
```

---

**NOTE** The compiler requires more memory at compile time if you enable this pragma.

This pragma corresponds to the **Deferred Inlining** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use the `__option` (`defer_codegen`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## 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. However, it will not override those declared with the `inline` keyword. Also, it does not automatically inline functions, regardless of the setting of the `auto_inline` pragma. 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 of the **Inline Depth** pull-down menu of the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`dont_inline`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## dont\_reuse\_strings

Controls whether or not to store each string literal separately in the string pool.

### Syntax

```
#pragma dont_reuse_strings on | off | reset
```

## Pragmas

### Object Code Organization and Generation Pragmas

---

#### Remarks

If you enable this pragma, the compiler stores each string literal separately. Otherwise, the compiler stores only one copy of identical string literals. This pragma helps you save memory if your program contains a lot of identical string literals that you do not modify.

For example, take this code segment:

---

```
char *str1="Hello";
char *str2="Hello";
*str2 = 'Y';
```

---

If you enable this pragma, `str1` is "Hello", and `str2` is "Yello". Otherwise, both `str1` and `str2` are "Yello".

This pragma corresponds to the **Reuse Strings** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`__dont_reuse_strings`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## enumsalwaysint

Specifies the size of enumerated types.

#### Syntax

```
#pragma enumsalwaysint on | off | reset
```

#### Remarks

If you enable this pragma, the 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. 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 int`.

[Listing 10.31 on page 164](#) shows an example.

#### Listing 10.31 Example of Enumerations the Same as Size as `int`

---

```
enum SmallNumber { One = 1, Two = 2 };
/* If you enable enumsalwaysint, this type is
the same size as an int. Otherwise, this type is
short int. */

enum BigNumber
```

---

---

```
{ ThreeThousandMillion = 3000000000 };
```

```
/* If you enable enumsalwaysint, the compiler might
   generate an error. Otherwise, this type is
   the same size as a long int. */
```

---

This pragma corresponds to the **Enums Always Int** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (enumsalwaysint)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

**NOTE** The size of a char on the DSP56800 target is 16 bits, and 8 bits on the DSP56800E.

---



---

## 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 10.32 on page 165](#) and [Listing 10.33 on page 165](#).

#### Listing 10.32 Maximum Complexity of an Inlined Function

---

```
// maximum complexity of an inlined function
#pragma inline_max_size( max )           // default max == 256
```

---

#### Listing 10.33 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

## Pragmas

### Object Code Organization and Generation Pragmas

---

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 Inlining** setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`inline_bottom_up`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## interrupt (for the DSP56800)

Controls the compilation of object code for interrupt service routines (ISR).

### Compatibility

This pragma is compatible with the DSP56800, but it is not compatible with the DSP56800E. For the DSP56800E, see [interrupt \(for the DSP56800E\) on page 168](#).

### Syntax

```
#pragma interrupt [called|warn|saveall[warn]]
```

### Remarks

The compiler generates a special prologue and epilogue for functions so that they may be used to handle interrupts. The contents of the epilogue and prologue vary depending on the mode selected.

The compiler also emits an RTI or RTS for the return statement depending upon the mode selected. The SA, R, and CC bits of the OMR register are set to system default.

There are several ways to use this pragma as described below:

- `pragma interrupt [warn]`

The compiler performs the following using the `pragma interrupt [warn]` argument:

- Sets M01 to -1 if M01 is used by ISR
  - Sets OMR to system default (see OMR settings)
  - Saves/restores only registers used by ISR
  - Generates an RTI to return from interrupt.
-

- If [warn] is present, then emits warnings if this ISR makes a function call that is not defined with # pragma interrupt called

Important considerations of usage:

- This type of usage is required within the ISR function body as follows:

```
void ISR(void)
{
#pragma interrupt
... code here
```

- pragma interrupt [called]

The compiler performs the following using the pragma interrupt [called] argument:

- Saves/restores only registers used by routine
- Generates an RTS to return from function

Important considerations of usage:

- You must use this argument before the interrupt body is compiled
- You can use this argument on the function Syntax or within the function body as described below.

On the function Syntax:

```
#pragma interrupt called
void function_called_from_interrupt (void);
```

Within the function body:

```
void function_called_from_interrupt (void)
{
#pragma interrupt called
asm (nop);
}
```

- You should use this pragma for all functions called from #pragma interrupt enabled ISRs. This is optional for #pragma interrupt saveall enabled ISRs, since for this case, the entire context is saved.

- pragma interrupt saveall [warn]

The compiler performs the following using the pragma interrupt saveall [warn] argument:

- Always sets M01 to -1
- Sets OMR to system default (see OMR settings)

## Pragmas

### Object Code Organization and Generation Pragmas

---

- Saves/restores entire hardware stack via runtime call
- Generates an RTI to return from interrupt
- If [warn] is present then emits a warning if the ISR makes a function call that is not defined with `#pragma interrupt called`

Important considerations of usage:

- This type of usage is required within the ISR function body as follows:

```
void interrupt_function(void)
{
    #pragma interrupt saveall
    ... code here
}
```

- This pragma should be used if the runtime library is called by the interrupt routine

In [Table 10.3 on page 168 on page 168](#), the advantages and disadvantages of the `interrupt` and `interrupt saveall` pragmas are listed.

**Table 10.3 Comparison of Usage**

Pragma	Advantages	Disadvantages
<code>interrupt saveall</code>	<ul style="list-style-type: none"> <li>• entire context save</li> <li>• no need for <code>#pragma interrupt called</code> for called functions</li> </ul>	<ul style="list-style-type: none"> <li>• larger initial performance hit due to entire context save, but becomes advantageous for ISRs with several function calls</li> </ul>
<code>interrupt</code>	<ul style="list-style-type: none"> <li>• smaller context save, less performance hit</li> <li>• generally good for ISRs with a small number of function calls</li> </ul>	<ul style="list-style-type: none"> <li>• <code>#pragma interrupt called</code> required for all called functions</li> </ul>

---

## **interrupt (for the DSP56800E)**

This pragma controls the compilation of object code for interrupt routines.

### **Compatibility**

This pragma is not compatible with the DSP56800, but it is compatible with the DSP56800E. For the DSP56800, see [interrupt \(for the DSP56800\) on page 166](#).



#### Syntax

```
#pragma interrupt [<options>] [<mode>] [on|off|reset]
```

#### Remarks

An Interrupt Service Routine (ISR) is a routine that is executed when an interrupt occurs. Setting C routines as ISRs is done using pragmas (`pragma interrupt`). To make a routine service an interrupt, you must:

- Write the routine.
- Set up the routine so that it is called when some interrupt occurs.

The `pragma interrupt` option can be used to:

- Instruct the compiler to push register values on the software stack at entry to a C function and restore them upon exit.
- Preserve the register values for the function that was interrupted.
- Emit an RTI for the return statement depending upon the mode selected. If the interrupt routine has a return value, the return register is not saved.

There are several ways to use this pragma, with an `on|off|reset` arguments, or with no arguments.

**Table 10.4 Arguments**

<code>&lt;options&gt;</code>	<code>alignsp</code>	Aligns the stack pointer register correctly to allow long values to be pushed on to the stack. Use this option when your project mixes C code and assembly code. Use this option specifically on ISRs which may interrupt assembly routines that do not maintain the long stack alignment requirements at all times. Restores the stack pointer to its original value before returning from the subroutine.
	<code>comr</code>	The Operating Mode Register (OMR) is set for the following to ensure correct execution of C code in the ISR: 36-bit values used for condition codes. (CM bit cleared) Convergent Rounding. (R bit cleared) No Saturation mode. (SA bit cleared) Instructions fetched from P memory. (XP bit cleared)
<code>&lt;mode&gt;</code>	<code>saveall</code>	Preserves register values by saving and restoring all registers by calling the <code>INTERRUPT_SAVEALL</code> and <code>INTERRUPT_RESTOREALL</code> routines in the Runtime Library.

## Pragmas

### Object Code Organization and Generation Pragmas

**Table 10.4 Arguments (continued)**

	called	Preserves register values by saving and restoring registers used by the routine. The routine returns with an RTS. Routines with pragma interrupt enabled in this mode are safe to be called by ISRs.
	default	This is the mode when no mode is specified. In this mode, the routine preserves register values by saving and restoring the registers that are used by the routine. The routine returns with an RTI.
on off reset	on	Enables the option to compile all C routines as interrupt routines.
	off	Disables the option to compile all C routines as interrupt routines.
	reset	Restores the option to its previous setting.

---

**NOTE** Use on or off to change the pragma setting, and then use reset to restore the previous pragma setting.

---

To disable the pragma, use #pragma interrupt off after #pragma interrupt ([Listing 10.35 on page 171](#))

#### Listing 10.34 Sample Code - #pragma interrupt on | off | reset

```
#pragma interrupt off // To be used as default value
// Non ISR code
#pragma interrupt on
void ISR_1(void) {
    // ISR_1 code goes here.
}

void ISR_2(void) {
    // ISR_2 code goes here.
}
#pragma interrupt reset
```

---

If the pragma is inside a function block, compile the current routine as an interrupt routine. If the pragma is not inside a function block, compile the next routine as an interrupt routine. This concept is developed in [Listing 10.35 on page 171](#).

**Listing 10.35 Sample Code - #pragma interrupt and function block**

```
// Non ISR code
void ISR_1(void) {
#pragma interrupt
    // ISR_1 code goes here.
}
#pragma interrupt
void ISR_2(void) {
    // ISR_2 code goes here.
}
#pragma interrupt off
```

See [Listing 10.36 on page 171](#) for an example of using the 'called' option in the interrupt pragma.

**Listing 10.36 Sample Code - using the 'called' option in # pragma interrupt**

```
extern long Data1, Data2, Datain;

void ISR1_inc_Data1_by_Data2(void)
{
/* This is a routine called by the interrupt service routine ISR1(). */
#pragma interrupt called
Data1+=Data2;
return;
}

void ISR1(void)
{
/* This is an interrupt service routine. */
#pragma interrupt
Data2 = Datain+2;
ISR_inc_Data1_by_Data2();
}
```

## Avoiding Possible Hitches with enabled Pragma Interrupt

Pragma interrupt with the *called* or *default* mode for a C routine saves only the volatile registers for that C routine. Register values are not preserved if the ISR makes one or more function calls. You might want to avoid the situations described below:

## Pragmas

### Object Code Organization and Generation Pragmas

---

If a routine that has pragma interrupt enabled (caller) calls another C function/routine (callee), it is possible that the callee can change some registers that are not saved by the caller. To avoid this, use either of the following options:

Call only pragma interrupt enabled routines from routines that are pragma interrupt enabled using the *called* mode, or

Use the pragma interrupt *saveall* mode for the caller.

The first option may be more efficient because only the registers that are used are preserved. The second option is easier to implement, but is likely to have a large overhead.

The situation described above also holds true for library functions because library functions do not have pragma interrupt enabled. These calls include: C Standard Library calls and Runtime Library calls (such as multiplication, division and floating point math).

---

## packstruct

Controls the alignment of long words in structures.

### Compatibility

This pragma is compatible with the DSP56800, but it is not compatible with the DSP56800E.

### Syntax

```
#pragma packstruct on | off | reset
```

### Remarks

If you enable this pragma, integer longs within structures are aligned on four byte boundaries. When this pragma is disabled there is no alignment within structures. This pragma does not correspond to any setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option(packstruct)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is enabled.

---

## pool\_strings

Controls how the compiler stores string constants.

### Compatibility

This pragma is not compatible with the DSP56800, but it is compatible with the DSP56800E.

---

### Syntax

```
#pragma pool_strings on | off | reset
```

### Remarks

If you enable this setting, the compiler collects all string constants into a single data object so that your program needs only one TOC entry for all of them. While this decreases the number of TOC entries in your program, it also increases your program size because it uses a less efficient method to store the address of the string.

If you disable this setting, the compiler creates a unique data object and TOC entry for each string constant.

Enable this setting if your program is large and has many string constants.

The **Pool Strings** setting corresponds to the pragma poolstring. To check this setting, use `__option` (`pool_strings`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## readonly\_strings

Controls the output of C strings to the 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 (.rodata) instead of the global data section (.data). In effect, these strings act like `const char *`, even though their type is really `char *`.

For the DSP56800, this pragma corresponds to the "Make Strings Read Only" panel setting in the **M56800 Processor** settings panel. To check this setting, use `__option` (`readonly_strings`), described in [Checking Pragma Settings on page 119](#).

For the DSP56800E, there is no "Make Strings Read Only" panel setting in the **M56800E Processor** settings panel.

---

## reverse\_bitfields

Controls whether or not the compiler reverses the bitfield allocation.

---

## Pragmas

### Object Code Organization and Generation Pragmas

---

#### Syntax

```
#pragma reverse_bitfields on | off | reset
```

#### Remarks

This pragma reverses the bitfield allocation.

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`reverse_bitfields`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## 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.

---

**WARNING!** Beware when using this pragma because it can produce erratic or unpredictable behavior in your program.

---

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`suppress_init_code`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## syspath\_once

Controls how include files are treated.

#### Syntax

```
#pragma syspath_once on | off | reset
```

---

### Remarks

If you enable this pragma, files called in `#include <>` and `#include "` directives are treated as distinct, even if they refer to the same file.

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (syspath_once)`, described in [Checking Pragma Settings on page 119](#). By default, this setting is enabled. For example, the same include file could reside in two distinct directories.

C Standard Library and Runtime Library (CW libraries) functions require the AGU (Address Generation Unit) to be in linear addressing mode, that is, the M01 registers are set to -1. If a function is interrupted and was using modulo address arithmetic, any calls to CW libraries from the ISR do not work unless the M01 is set to -1 in the ISR. Also, the M01 register would need to be restored before exiting the ISR so that the interrupted function can resume as before, with the same modulo address arithmetic mode settings.

## Optimization Pragmas

The 56800x has the following pragmas:

- [factor1 on page 176](#)
- [factor2 on page 176](#)
- [factor3 on page 176](#)
- [nofactor1 on page 177](#)
- [nofactor2 on page 177](#)
- [nofactor3 on page 178](#)
- [opt\\_common\\_subs on page 178](#)
- [opt\\_dead\\_assignments on page 179](#)
- [opt\\_dead\\_code on page 179](#)
- [opt\\_lifetimes on page 179](#)
- [opt\\_loop\\_invariants on page 180](#)
- [opt\\_propagation on page 180](#)
- [opt\\_strength\\_reduction on page 181](#)
- [opt\\_strength\\_reduction\\_strict on page 181](#)
- [opt\\_unroll\\_loops on page 182](#)
- [optimization\\_level on page 182](#)
- [optimize\\_for\\_size on page 182](#)
- [peephole on page 183](#)

## Pragmas

### Optimization Pragmas

---

#### factor1

Turns on factorization step 1.

##### Syntax

```
#pragma factor1
```

##### Remarks

Compiler performs the factorization step 1. To turn off factor1, use [nofactor1 on page 177](#). This optimization is performed on global variables before register allocation, takes into account register pressure, and replaces absolute addressing with indirect addressing.

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). By default, this pragma is enabled at global optimization level 2 and above.

---

#### factor2

Turns on factorization step 2.

##### Syntax

```
#pragma factor2
```

##### Remarks

Compiler performs the factorization step 2. To turn off factor2, use [nofactor2 on page 177](#). This optimization is performed on global variables after register allocation, replaces absolute addressing with indirect addressing, and detects a physical address register that is available to do the factorization. Register allocation spilling decreases pressure so new webs, that could not be created before register allocation, can be created.

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). By default, this pragma is enabled at global optimization level 2 and above.

---

#### factor3

Turns on factorization step 3.

---



**Syntax**

```
#pragma factor3
```

**Remarks**

Compiler performs the factorization step 3. To turn off factor3, use [nofactor3 on page 178](#). This optimization is performed on local variables after register allocation. (SP-offset) addressing is transformed in register indirect addressing. This optimization is performed after register allocation because only at this point are the local variables accessed by stack location.

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). By default, this pragma is enabled at global optimization level 2 and above.

---

**nofactor1**

Turns off factorization step 1.

**Syntax**

```
#pragma nofactor1
```

**Remarks**

Compiler does not perform the factorization step 1. To turn on factorization step 1, use [factor1 on page 176](#).

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#).

---

**nofactor2**

Turns off factorization step 2.

**Syntax**

```
#pragma nofactor2
```

**Remarks**

Compiler does not perform the factorization step 2. To turn on factorization step 2, use [factor2 on page 176](#).

---

## Pragmas

### Optimization Pragmas

---

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#).

---

## nofactor3

Turns off factorization step 3.

### Syntax

```
#pragma nofactor3
```

### Remarks

Compiler does not perform the factorization step 3. To turn on factorization step 3, use [factor3 on page 176](#).

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#).

---

## 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 in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (opt_common_subs)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## 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 in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (opt_dead_assignments)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## 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 in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (opt_dead_code)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## opt\_lifetimes

Controls the use of lifetime analysis optimization.

---

## Pragmas

### Optimization Pragmas

---

#### 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 in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (opt_lifetimes)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## opt\_loop\_invariants

Controls the use of loop invariant optimization.

#### Syntax

```
#pragma opt_loop_invariants on | off | reset
```

#### Remarks

If you enable this pragma, the compiler moves all computations that do not change inside a loop outside the loop, which then runs faster.

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option (opt_loop_invariants)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## 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 in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`opt_propagation`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## 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 in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`opt_strength_reduction`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## opt\_strength\_reduction\_strict

Uses a safer variation of strength reduction optimization.

### Syntax

```
#pragma opt_strength_reduction_strict on | off | reset
```

### Remarks

Like the [opt\\_strength\\_reduction on page 181](#) 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 in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`opt_strength_reduction_strict`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## Pragmas

### Optimization Pragmas

---

#### opt\_unroll\_loops

Controls the use of loop unrolling optimization.

##### Syntax

```
#pragma opt_unroll_loops on | off | reset
```

##### Remarks

If you enable this pragma, the compiler places multiple copies of a loop's statements inside a loop to improve its speed.

This pragma does not correspond to any panel setting in the [C/C++ Language \(C only\) Settings Panel on page 16](#). To check this setting, use `__option` (`opt_unroll_loops`), described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

#### 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 *Code Warrior IDE User's Guide*.

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.

---

### Syntax

```
#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. This pragma does not effect the inline directive or the inlining of explicitly inlined functions. This pragma can be used in conjunction with the `dont_inline` pragma to decrease the code size. 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. To check this setting, use `__option (optimize_for_size)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.

---

## peephole

Controls the use peephole optimization.

### Syntax

```
#pragma peephole on | off | reset
```

### Remarks

If you enable this pragma, the compiler performs *peephole optimizations*, which are small, local optimizations that eliminate some compare instructions and improve branch sequences.

For the DSP56800, this pragma corresponds to the **Peephole Optimization** setting in the M56800 Processor settings panel. Yet for the DSP56800E, there is no corresponding setting for the the M56800 Processor settings panel. To check this setting, use `__option (peephole)`, described in [Checking Pragma Settings on page 119](#).

## Profiler Pragmas

The 56800x has just one profiler pragma:

- [profile on page 184](#)

## Pragmas

### Profiler Pragmas

---

#### profile

Controls code to enable or disable the profiler.

#### Syntax

```
#pragma profile on | off | reset
```

#### Remarks

This setting lets you choose whether the compiler adds code to a function to call profiler library functions. If you enable this pragma, the compiler calls profiling functions at the beginning and end of the current function. If you disable this pragma, the compiler adds no additional code. For further information on the profiler, see the Chapter “Profiler” in either of the Targeting Manuals.

The pragma corresponds to the **Generate code for profiling** setting on the **M56800E Processor** settings panel. To check this setting, use `__option (profile)`, described in [Checking Pragma Settings on page 119](#). By default, this pragma is disabled.



# Index

## Symbols

- #, and macros 74
- #else 75
- #endif 75
- #include directive
  - getting path 152
- #pragma statement
  - illegal 126
- .lcf 32
- \_\_builtin\_align() 77
- \_\_builtin\_type() 77
- \_\_ide\_target() 116
- \_\_INTEL\_\_ 117
- \_\_typeof\_\_() 77
- \_\_VEC\_\_ 118

## A

- always\_inline pragma 161
- ANSI Keywords Only option 19
- ANSI\_strict pragma 127
- arguments
  - unnamed 74
- auto\_inline pragma 17, 161
- auto-inlining
  - See inlining.

## C

- C/C++ Language panel
  - Don't Inline option 17
- C/C++ Warnings panel 22
- char type 20
- character strings
  - See strings.
- characters
  - as integer values 76
- check\_inline\_asm\_pipeline pragma 133
- command files 32
- comments, C++-styles 74
- const\_strings pragma 158, 162
- CWFolder 28

## D

- D constant suffix 77
- defer\_codegen pragma 162
- Deferred Inlining 163
- define\_section 155
- disassemble 32
- dollar sign 151
- dollar\_identifiers pragma 151
- Don't Inline option 17, 120
- dont\_inline pragma 17, 163
- dont\_reuse\_strings pragma 20, 163

## E

- #else 75
- #endif 75
- enumerated types 164
- enumsalwaysint pragma 164
- Environment tab 28
- export pragma 156, 157
- extended\_errorchecking pragma 135

## F

- FlexLM 29
- fullpath\_prepdump pragma 152
- function
  - interrupt 166
  - result, warning 148

## G

- gcc\_extensions pragma 129
- GNU C
  - pragma 129

## H

- header files
  - getting path 152

## I

- identifier
  - \$ 151
  - dollar signs in 151

- 
- significant length 78
    - size 78
  - Illegal Pragmas option 126
  - inline\_intrinsics pragma 165
  - inlining
    - before definition 162
    - stopping 163
  - integer
    - specified as character literal 76
  - \_\_INTEL\_\_ 117
  - interrupt
    - interrupt pragma 166
  - interrupt pragma 175
  - interrupt pragma 166
- K**
- keywords
    - additional 19
    - standard 129
- L**
- license 29
  - linker command files 32
  - LM\_LICENSE\_FILE 29
- M**
- macros
    - and # 74
  - Microsoft Windows 28
  - mpwc\_newline pragma 130
  - mpwc\_relax pragma 131
  - multi-byte characters 76
  - MWAsmIncludes 32
  - MWCIncludes 32
  - MWLibraries 32
  - MWLibraryFiles 32
- N**
- nodefaults 32
  - notonce pragma 153
- O**
- once pragma 153
- only\_std\_keywords pragma 129
  - opt\_common\_subs pragma 178
  - opt\_dead\_assignments pragma 176, 177, 178, 179
  - opt\_dead\_code pragma 179
  - opt\_lifetimes pragma 180
  - opt\_loop\_invariants pragma 180
  - opt\_propagation pragma 180
  - opt\_strength\_reduction pragma 181
  - opt\_strength\_reduction\_strict pragma 181
  - opt\_unroll\_loops pragma 182
  - optimization
    - global 182
    - level of 182
    - loops 182
    - opt\_unroll\_loops pragma 182
    - optimization\_level pragma 182
    - optimize\_for\_size pragma 183
    - size 183
  - optimization\_level pragma 182
  - optimize\_for\_size pragma 183
- P**
- PATH 29
  - peephole pragma 172, 173, 183
  - pop pragma 154
  - pragma
    - define\_section 155
    - illegal 126
    - scope 127
    - section 157
  - pragmas
    - check\_inline\_asm\_pipeline 133
    - interrupt 175
  - Precompile command 83
  - preprocessor
    - and # 74
    - header files 152
  - prototypes
    - requiring 19
  - push pragma 154
-

## R

readonly\_strings pragma 173  
 Require Function Prototypes option 19  
 require\_prototypes pragma 135  
 reverse\_bitfields pragma 174

## S

sample code  
     pragma define\_section and pragma  
         section 158  
 section 157  
 settings panel  
     C/C++ Warnings 22  
 side effects  
     warning 146  
 statements  
     #pragma 126  
 strings  
     pooling 163  
     reusing 20  
     storage 163  
 suffix, constant 77  
 suppress\_init\_code pragma 135, 174  
 suppress\_warnings pragma 136  
 syspath\_once pragma 154, 174  
 System control panel 28

## T

Target Settings window 27  
 trigraph characters 19  
 types  
     char 20  
     unsigned char 20

## U

unnamed arguments 74  
 unsigned char type 20  
 unsigned\_char pragma 136  
 unused pragma 137

## V

\_\_VEC\_\_ 118

## W

warn\_any\_ptr\_int\_conv pragma 138  
 warn\_emptydecl pragma 139  
 warn\_extracomma pragma 139  
 warn\_filenameecaps pragma 140  
 warn\_filenameecaps\_system pragma 140  
 warn\_illpragma pragma 127, 141  
 warn\_impl\_f2i\_conv pragma 141  
 warn\_impl\_i2f\_conv pragma 142  
 warn\_impl\_s2u\_conv pragma 143  
 warn\_implicitconv pragma 144  
 warn\_largeargs pragma 145  
 warn\_missingreturn pragma 145  
 warn\_no\_side\_effect pragma 146  
 warn\_notinlined pragma 134, 146  
 warn\_padding pragma 147  
 warn\_possunwant pragma 147  
 warn\_ptr\_int\_conv pragma 148  
 warn\_resultnotused pragma 148  
 warn\_undefmacro pragma 149  
 warn\_unusedarg pragma 150  
 warn\_unusedvar pragma 150  
 warning pragma 47, 48, 49, 50  
 warning\_errors pragma 151  
 warnings  
     illegal pragmas 126  
     setting in the IDE 22  
 Windows operating system 28

