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

Document Number: CWMCUCFCMPREF
Rev 10.6, 08/2014

freescale™

Contents

Section number **Title** **Page**

Chapter 1
Overview

Chapter 2
Introduction

Chapter 3
Creating Project

Chapter 4
Using Build Tools on Command Line

Chapter 5
Command-Line Options for Standard C Conformance

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

Chapter 9
Command-Line Options for Preprocessing

Chapter 10
Command-Line Options for Library and Linking

Chapter 11
Command-Line Options for Object Code

Chapter 12
Command-Line Options for Optimization

Chapter 13
ColdFire Command-Line Options

Chapter 14
ELF Linker and Command Language

Chapter 15
C Compiler

Chapter 16
C++ Compiler

Chapter 17
Precompiling

Chapter 18
Addressing

Chapter 19
Intermediate Optimizations

Chapter 20
ColdFire Runtime Libraries

Chapter 21
Declaration Specifications

Chapter 22
Predefined Macros

Chapter 23
ColdFire Predefined Symbols

Chapter 24
Using Pragmas

Chapter 25
Pragmas for Standard C Conformance

Chapter 26
Pragmas for C++

Chapter 27
Pragmas for Language Translation

Chapter 28
Pragmas for Diagnostic Messages

Chapter 29
Pragmas for Preprocessing

Chapter 30
Pragmas for Code Generation

Chapter 31
Pragmas for Optimization

Chapter 32
ColdFire Pragmas

Chapter 33
Appendices

Chapter 34
Calling Assembly Functions

Chapter 1
Overview

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

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

`<CWInstallDir>\MCU\Help\PDF`

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

1.1 Accompanying Documentation

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

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

NOTE

To view the online help for the CodeWarrior tools, first select **Help > Help Contents** from the IDE's menu bar. Next, select required manual from the **Contents** list. For general information about the CodeWarrior IDE and

debugger, refer to the *CodeWarrior Common Features Guide* in this folder: `<CWInstallDir>\MCU\Help\PDF`

1.2 Additional Information Resources

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

  `http://www.freescale.com`

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

  `http://www.eg3.com`

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

  `http://www.embedded.com`

- For late-breaking information about new features, bug fixes, known problems, and incompatibilities, read the release notes in this folder:

  `CWInstallDir\MCU`

  where `CWInstallDir` is the directory in which CodeWarrior is installed, and `MCU` is the CodeWarrior Microcontrollers folder.

- To view the online help for the CodeWarrior tools, select **Help > Help Contents** from the IDE's menu bar.

1.3 Miscellaneous

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

- *American National Standard for Programming Languages - C*, ANSI/ISO 9899-1990 (see ANSI X3.159-1989, X3J11)
- *The C Programming Language*, second edition, Prentice-Hall 1988
- *C: A Reference Manual*, second edition, Prentice-Hall 1987, Harbison and Steele
- *C Traps and Pitfalls*, Andrew Koenig, AT&T Bell Laboratories, Addison-Wesley Publishing Company, Nov. 1988, ISBN 0-201-17928-8
- *Data Structures and C Programs*, Van Wyk, Addison-Wesley 1988

- *How to Write Portable Programs in C*, Horton, Prentice-Hall 1989
- *The UNIX Programming Environment*, Kernighan and Pike, Prentice-Hall 1984
- *The C Puzzle Book*, Feuer, Prentice-Hall 1982
- *C Programming Guidelines*, Thomas Plum, Plum Hall Inc., Second Edition for Standard C, 1989, ISBN 0-911537-07-4
- *DWARF Debugging Information Format*, UNIX International, Programming Languages SIG, Revision 1.1.0 (October 6, 1992), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054
- *DWARF Debugging Information Format*, UNIX International, Programming Languages SIG, Revision 2.0.0 (July 27, 1993), UNIX International, Waterview Corporate Center, 20 Waterview Boulevard, Parsippany, NJ 07054
- *System V Application Binary Interface*, UNIX System V, 1992, 1991 UNIX Systems Laboratories, ISBN 0-13-880410-9
- *Programming Microcontroller in C*, Ted Van Sickle, ISBN 1878707140
- *C Programming for Embedded Systems*, Kirk Zurell, ISBN 1929629044
- *Programming Embedded Systems in C and C ++*, Michael Barr, ISBN 1565923545
- *Embedded C*, Michael J. Pont, ISBN 020179523X

Chapter 2
Introduction

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

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

This chapter consists of the following topics:

2.1 Compiler Architecture

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

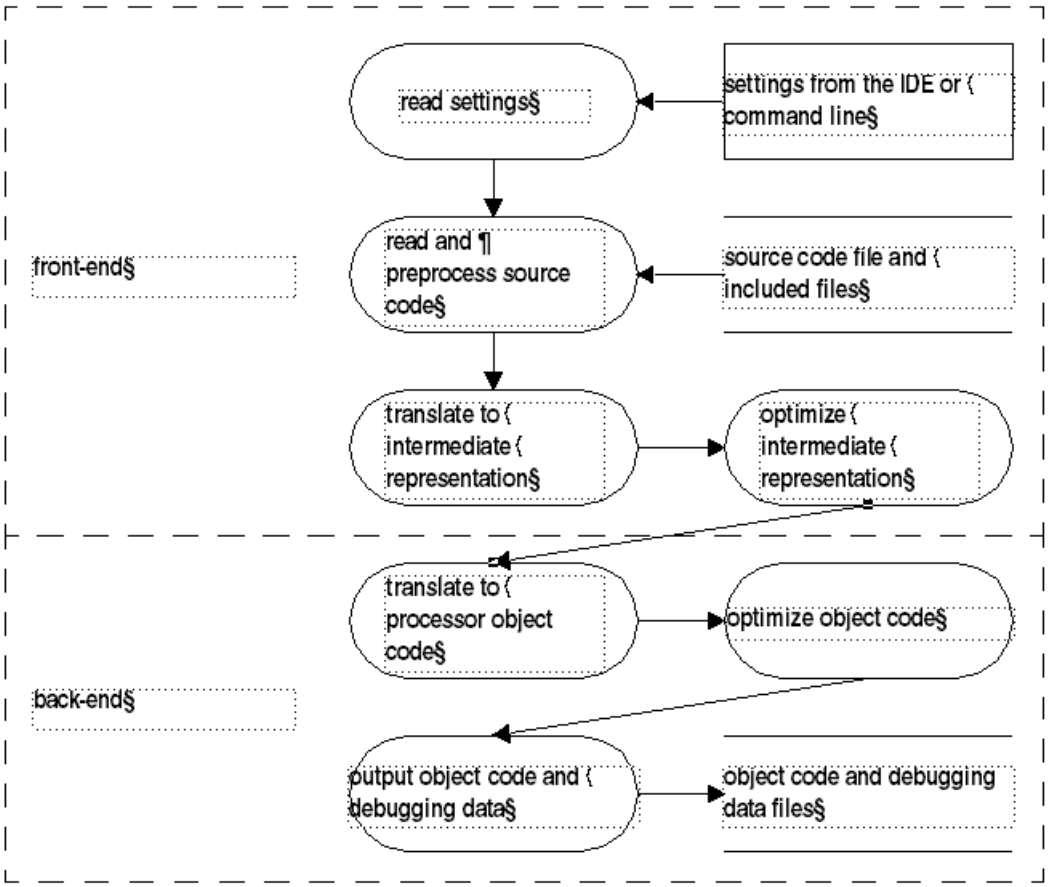

Figure 2-1. CodeWarrior Compiler Steps

Front-end steps:

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

Back-end steps:

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

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

2.2 Linker Architecture

A linker combines and arranges data and instructions from one or more object code files into a single file, or *image*. This image is ready to execute on the target platform. The CodeWarrior linker uses settings from the host's integrated development environment (IDE) or command line to determine how to generate the image file.

The linker also optionally reads a linker command file. A linker command file allows you to specify precise details of how data and instructions should be arranged in the image file.

The following figure shows the steps the CodeWarrior linker takes to build an executable image.

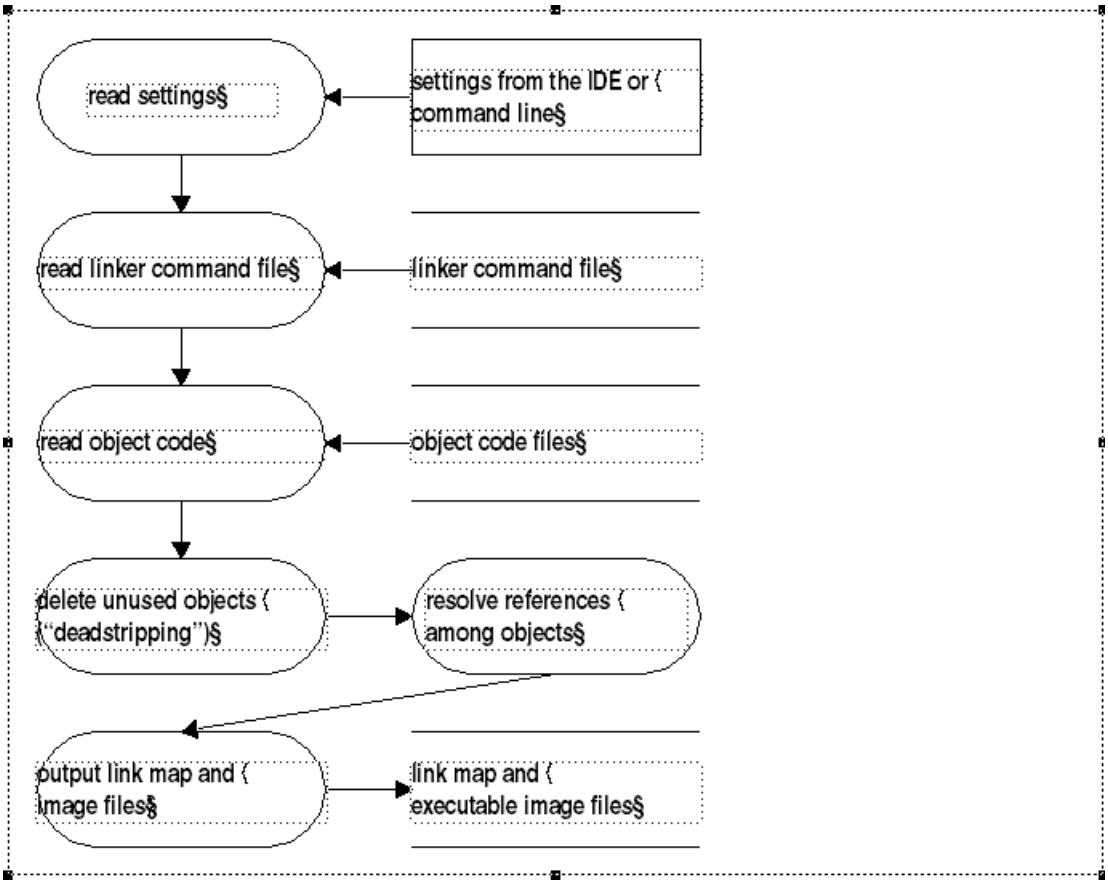

Figure 2-2. CodeWarrior Linker Steps

- **read settings** : retrieves your settings from the IDE or the command line to determine how to perform subsequent steps
- **read linker command file** : retrieves commands to determine how to arrange object code in the final image
- **read object code** : retrieves data and executable objects that are the result of compilation or assembly
- **delete unused objects ("deadstripping")** : deletes objects that are not referred to by the rest of the program
- **resolve references among objects** : arranges objects to compose the image then computes the addresses of the objects
- **output link map and image files** : writes files on the host system, ready to load onto the target system

Chapter 3
Creating Project

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

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

NOTE

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

3.1 Creating and Managing Project Using CodeWarrior IDE

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

3.1.1 Creating Project Using New Bareboard Project Wizard

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

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

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

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

   The CodeWarrior IDE launches.

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

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

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

   The **Devices** page appears.

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

   The **Connections** page appears.

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

   The **ColdFire Build Options** page appears.

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

    The **Rapid Application Development** page appears.

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

NOTE

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

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

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

NOTE

To configure the IDE, so that it automatically builds the project when a project is created, select **Window > Preferences** to open the **Preferences** window. Expand the **General** node and select **Workspace**. In the **Workspace** panel, check the **Build automatically** checkbox and click **OK**.

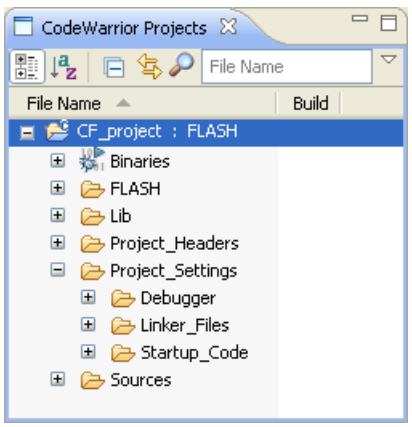

Figure 3-1. *CF_project Project in CodeWarrior Projects View*

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

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

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

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

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

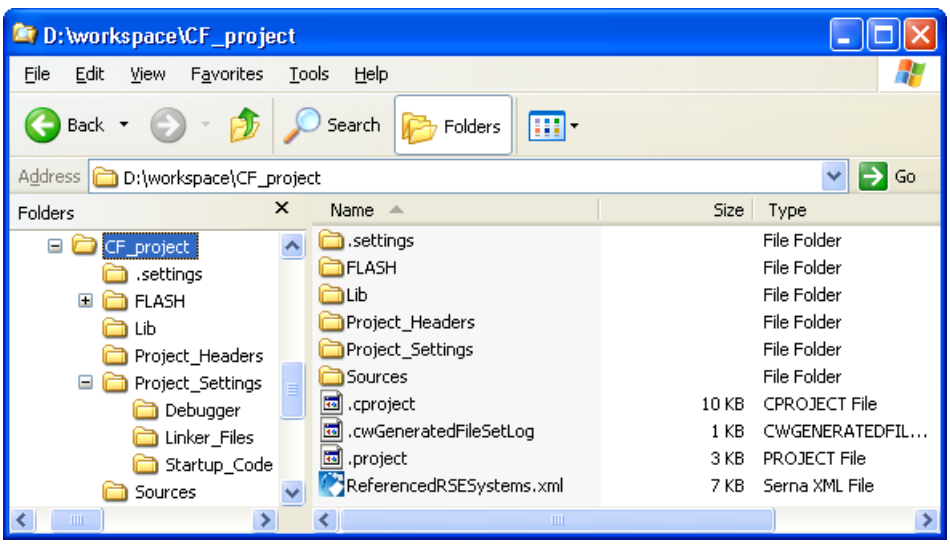

Figure 3-2. Contents of CF_project Directory

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

NOTE

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

3.1.2 Analysis of Groups in CodeWarrior Projects View

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

The default groups and their usual functions are:

- `FLASH`

The FLASH group contains all of the files that the CodeWarrior IDE uses to build the program. It also stores any files generated by the build process, such as any binaries (.args, .elf, and .elf.s19), and a map file (.elf.xmap). The CodeWarrior IDE uses this directory to manage the build process, so you should not tamper with anything in this directory. This directory's name is based on the build configuration, so if you switch to a different build configuration, its name changes.

- Lib

  The Lib group contains the C-source code file for the chosen MCU derivative. For this example, the mcf51ac128a.c file supports the mcf51ac128a derivative. This file defines symbols that you use to access the MCU's registers and bits within a register. It also defines symbols for any on-chip peripherals and their registers. After the first build, you can expand this file to see all of the symbols that it defines.

- Project_Headers

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

- Project_Settings

  The Project_Settings group consists of the following sub-folders:

  - Debugger

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

  - Linker_Files

    This group contains the linker file.

  - Startup_Code

    This group contains the source code that manages the MCU's initialization and startup functions. For CFV1 derivatives, these functions appear in the source file startcf.c.

- Sources

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

3.1.3 Analysis of Files in CodeWarrior Projects View

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

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

- `main.c`,

  located in <*project_directory*>\ `Sources`

- `Startcf.c`, and

  located in <*project_directory*>\ `Project_Settings\Startup_Code`

- `mcf51ac128a.c`

  located in <*project_directory*>\ `Lib`

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

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

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

   The **Clean** dialog box appears.

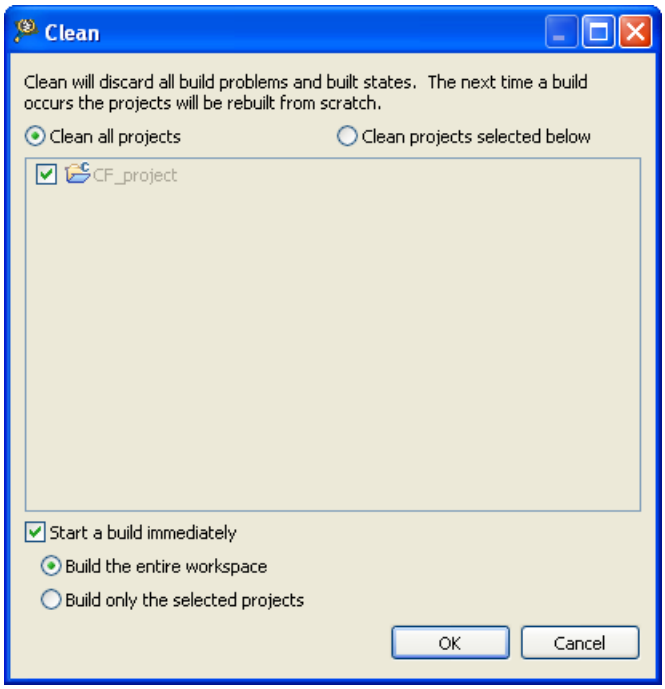

Figure 3-3. Clean Dialog Box

2. Select the **Clean projects selected below** option and select the project you want to re-build.
3. Clear the **Start a build immediately** checkbox. Click **OK** .

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

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

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

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

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

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

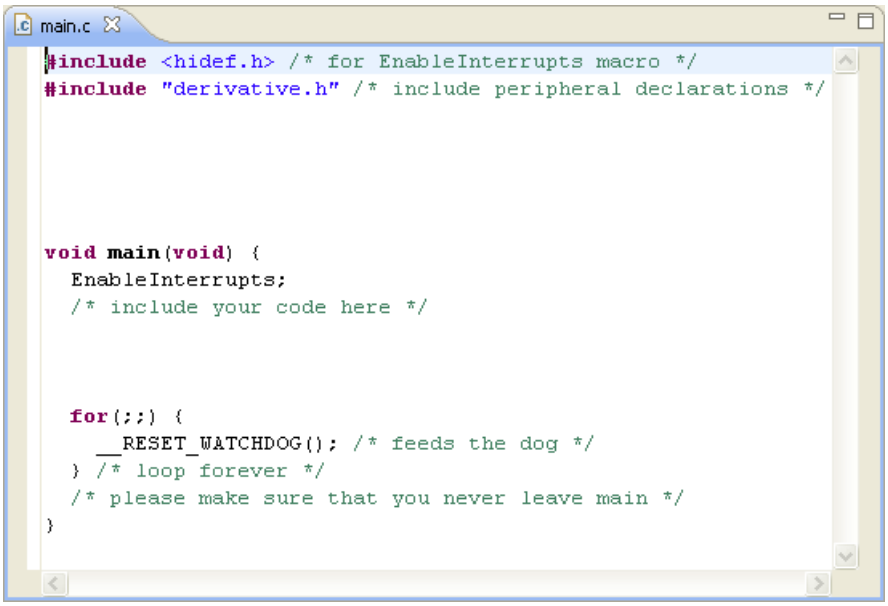

Figure 3-4. Default main.c File

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

3.2 Highlights

The CodeWarrior build tools provide the following features:

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

3.3 CodeWarrior Integration of Build Tools

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

3.3.1 Combined or Separated Installations

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

NOTE

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

3.3.2 ColdFire Compiler Build Settings Panels

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

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

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

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

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

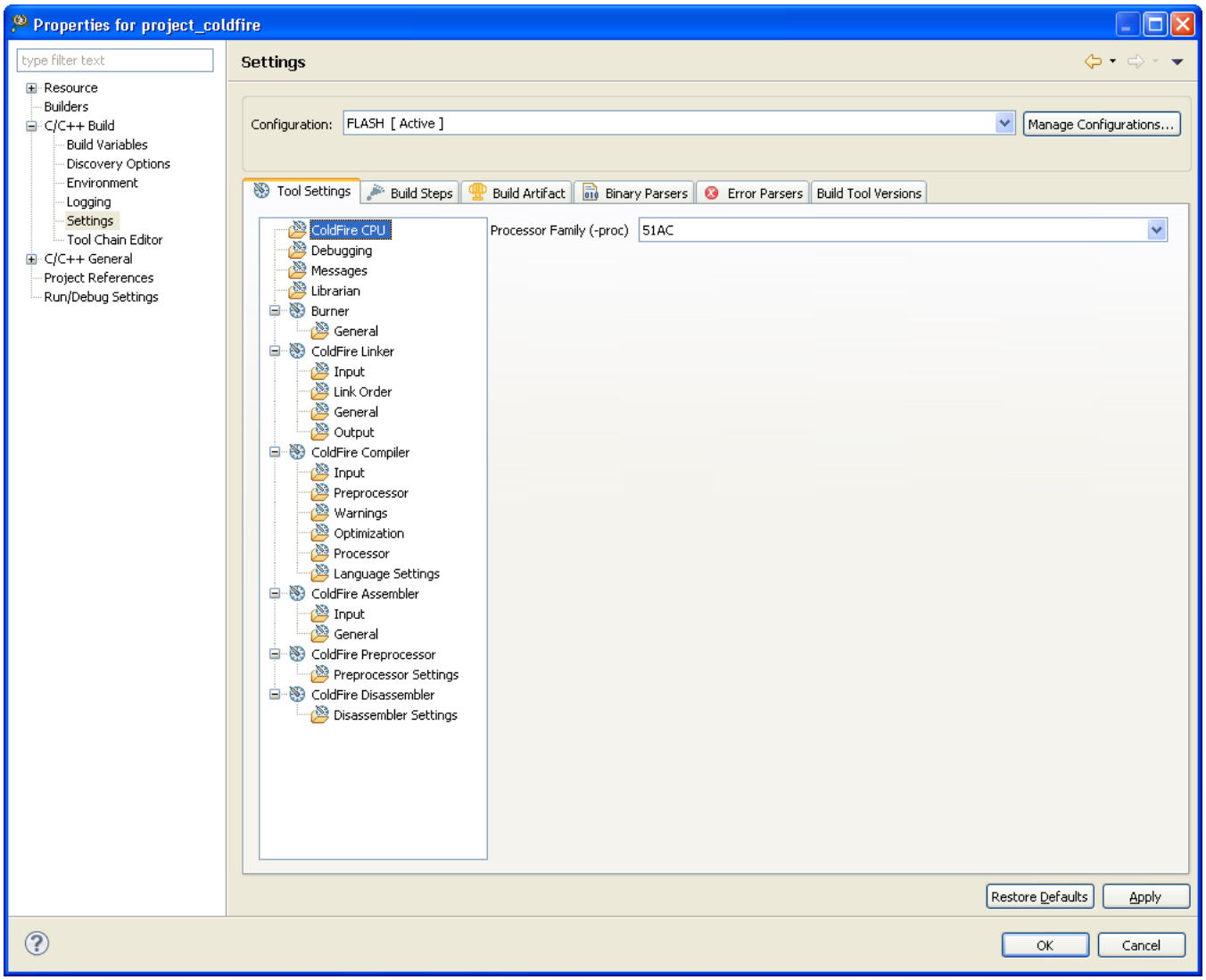

Figure 3-5. Project's Properties Dialog Box

The options are grouped by tool, such as general options, linker options, ColdFire compiler options, and so on. Scroll down the list, and click on the option whose settings you wish to examine and modify.

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

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

Table 3-1. Build Properties for ColdFire

Build Tool	Build Properties Panels
ColdFire Compiler	ColdFire Compiler > Input
	ColdFire Compiler > Preprocessor

Table continues on the next page...

Table 3-1. Build Properties for ColdFire (continued)

Build Tool	Build Properties Panels
	ColdFire Compiler > Warnings
	ColdFire Compiler > Optimization
	ColdFire Compiler > Processor
	ColdFire Compiler > Language Settings

3.3.2.1 ColdFire Compiler

Use this panel to specify the command, options, and expert settings for the build tool compiler. Additionally, the **ColdFire Compiler** tree control includes the general, include file search path settings. The following figure shows the ColdFire Compiler settings.

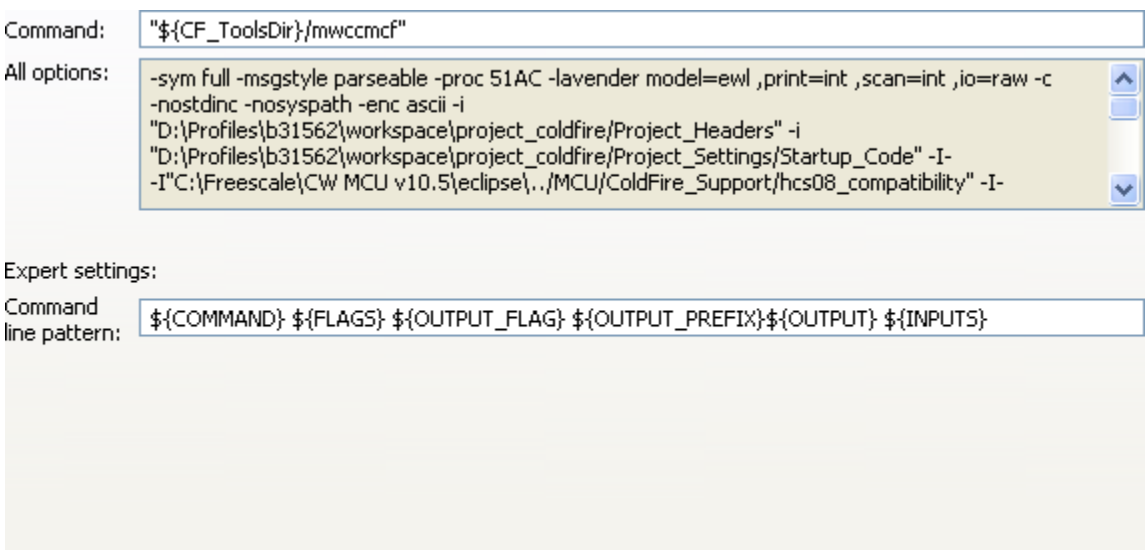

Figure 3-6. Tool Settings - ColdFire Compiler

The following table lists and describes the compiler options for the ColdFire connection.

Table 3-2. Tool settings - ColdFire Compiler Options

Option	Description
Command	Shows the location of the compiler executable file. Specify additional command line options for the compiler; type in custom flags that are not otherwise available in the UI.
All options	Shows the actual command line the compiler will be called with.
Expert settings Command line pattern	Shows the expert settings command line parameters; default is `${COMMAND} ${FLAGS}${OUTPUT_FLAG}${OUTPUT_PREFIX}${OUTPUT} ${INPUTS}`.

3.3.2.2 ColdFire Compiler > Input

Use this panel to specify additional files the **ColdFire Compiler** should use. You can specify multiple additional libraries and library search paths. Also, you can change the order in which the IDE uses or searches the libraries.

The following figure shows the **Input** panel.

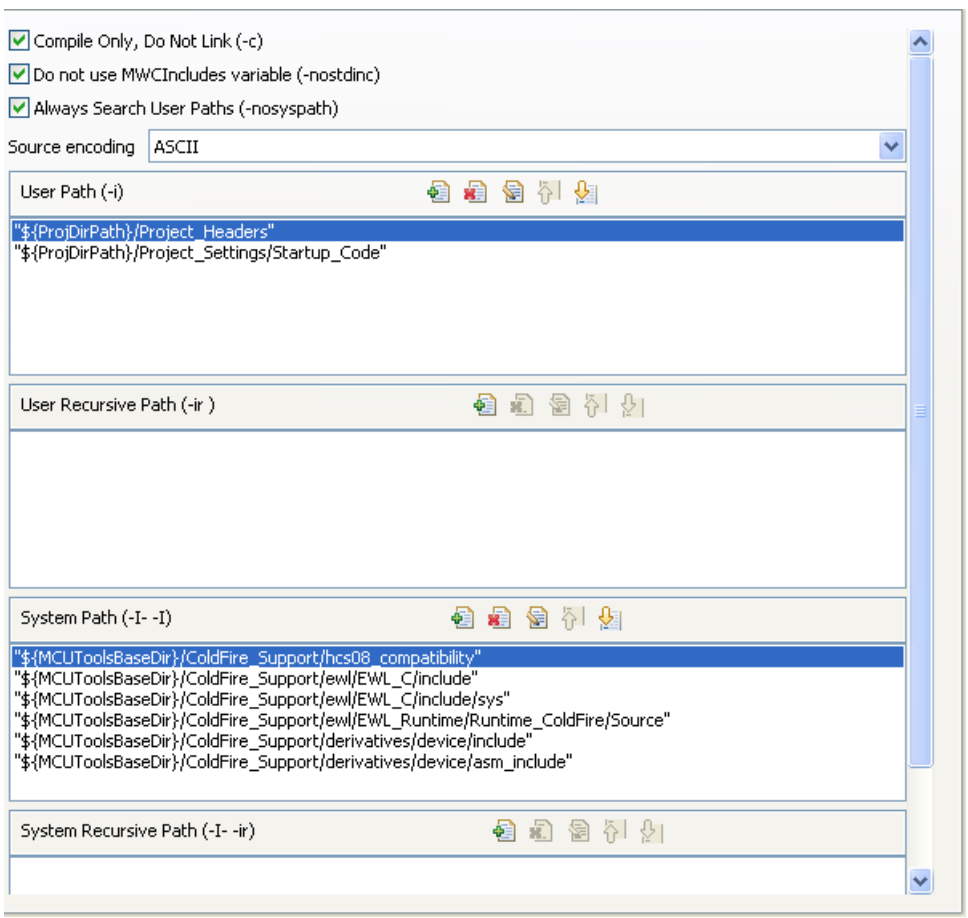

Figure 3-7. Tool Settings - ColdFire Compiler > Input

The following table lists and describes the input options for the ColdFire compiler.

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

Option	Description
Compile Only, Do Not Link (`-c`)	Check if you want to compile the file.
Do not use MWCIncludes variable (`-nostdinc`)	Check if you do not want to use MWCIncludes variable.
Always Search User Paths (`-nosyspath`)	Check if you want to always search user paths.

Table continues on the next page...

Table 3-3. Tool Settings - ColdFire Compiler > Input Options (continued)

Option	Description
Source encoding	Lists the type of source encoding.
User Path (`-i`)	Lists the available user paths.
User Recursive Path (`-ir`)	Lists the available user recursive paths.
System Path (`-I- -I`)	Lists the available system paths.
System Recursive Path (`-I- -ir`)	Lists the available system recursive paths.

3.3.2.3 ColdFire Compiler > Preprocessor

Use this panel to specify the preprocessor settings for the ColdFire Compiler. The following figure shows the **Preprocessor** panel.

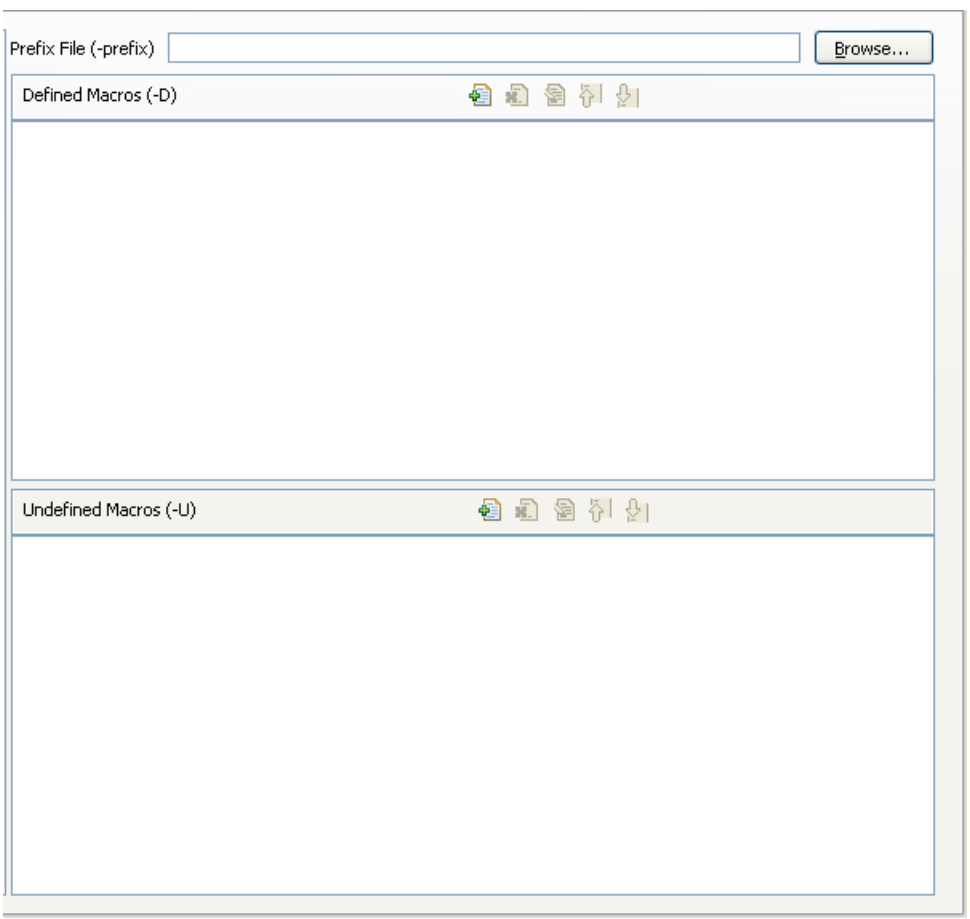

Figure 3-8. Tool Settings - ColdFire Compiler > Preprocessor

The following table lists and describes the preprocessor options for the ColdFire compiler.

Table 3-4. Tool Settings - ColdFire Compiler > Preprocessor Options

Option	Description
Prefix File (-prefix)	Use this option to add the contents of a text file or precompiled header as a prefix to all source files.
Defined Macros (-D)	Use this option to specify the defined macros.
Undefined Macros (-U)	Use this option to specify the undefined macros.

3.3.2.4 ColdFire Compiler > Warnings

Use this panel to specify the settings for ColdFire compiler to format the listing file, as well as error and warning messages.

The following figure shows the **Warnings** panel.

Figure 3-9. Tool Settings - ColdFire Compiler > Warnings

The following table lists and describes the warnings options for the ColdFire compiler.

Table 3-5. Tool settings - ColdFire Compiler > Warnings Options

Option	Description
Treat All Warnings As Errors	Check to treat all warnings as errors. The compiler will stop if it generates a warning message.
Illegal Pragmas	Check to notify the presence of illegal pragmas.
Possible Errors	Check to suggest possible errors.
Extended Error Checking	Check if you want to do an extended error checking.
Hidden virtual functions	Check to generate a warning message if you declare a non-virtual member function that prevents a virtual function, that was defined in a superclass, from being called and is equivalent to pragma `warn_hidevirtual` and the command-line option `-warnings hidevirtual`.
Implicit Arithmentic Conversions	Check to warn of implict arithmetic conversions.
Im plicit Integer to Float Conversions	Check to warn of implict conversion of an integer variable to floating-point type.
Implicit Float to Integer conversions	Check to warn of implict conversions of a floating-point varaible to integer type.
Implict Signed/Unsigned Conversions	Check to enable warning of implict conversions between signed and unsigned variables.
Pointer/Integral Conversions	Check to enable warnings of conversions between pointer and integers.
Unused Arguments	Check to warn of unused arguments in a function.
Unused Variables	Check to warn of unused variables in the code.
Unused Result From Non-Void-Returning Function	Check to warn of unused result from nonvoid-returning functions.
Missing `return' Statement	Check to warn of when a function lacks a return statement.
Expression Has No Side Effect	Check to issue a warning message if a source statement does not change the program's state. This is equivalent to the pragma `warn_no_side_effect`, and the command-line option `-warnings unusedexpr`.
Extra Commas	Check to issue a warning message if a list in an enumeration terminates with a comma. The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 ("C99") standard and is equivalent to pragma `warn_extracomma` and the command-line `option -warnings extracomma`.
Empty Declarations	Check to warn of empty declarations.
Inconsistent `class'/`struct' Usage	Check to warn of inconsistent usage of class or struct.
Include File Capitalization	Check to issue a warning message if the name of the file specified in a #include "file" directive uses different letter case from a file on disk and is equivalent to pragma `warn_filenamecaps` and the command-line option `-warnings filecaps`.

Table continues on the next page...

Table 3-5. Tool settings - ColdFire Compiler > Warnings Options (continued)

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

3.3.2.5 ColdFire Compiler > Optimization

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

The following figure shows the **Optimization** panel.

Figure 3-10. Tool Settings - ColdFire Compiler > Optimization

The following table lists and defines each option of the **Optimization** panel.

Table 3-6. Tool Settings - ColdFire Compiler > Optimization Options

Option	Description
Optimization Level (`-opt`)	Specify the optimizations that you want the compiler to apply to the generated object code: • **0** - Disable optimizations. This setting is equivalent to specifying the `-O0` command-line option. The compiler generates unoptimized, linear assembly-language code. • **1** - The compiler performs all target independent optimizations, such as function inlining. This setting is equivalent to specifying the `-O1` command-line option. The compiler omits all target-specific optimizations and generates linear assembly-language code. • **2** - The compiler performs all optimizations (both target-independent and target-specific). This setting is equivalent to specifying the `-O2` command-line option. The compiler outputs optimized, non-linear assembly-language code. • **3** - The compiler performs all the level 2 optimizations and iterates over target-independent and specific optimizations. This setting is equivalent to specifying the `-O3` command-line option. At this optimization level, the compiler generates code that is usually faster than the code generated from level 2 optimizations. • **4** - Like level 3, but with more comprehensive optimizations from levels 1 and 2. Equivalent to `#pragma optimization_level 4`. At this optimization level, the compiler generates code that is usually faster than the code generated from level 2 optimizations.
Speed vs Size	Use to specify an **Optimization Level** greater than **0** . • **Speed -** The compiler optimizes object code at the specified **Optimization Level** such that the resulting binary file has a faster execution speed, as opposed to a smaller executable code size. • **Size -** The compiler optimizes object code at the specified **Optimization Level** such that the resulting binary file has a smaller executable code size, as opposed to a faster execution speed. This setting is equivalent to specifying the `-Os` command-line option.
IPA	Use this option to control the interprocedural analysis optimization. • **Off** - No interprocedural analysis, but still performs function-level optimization. Equivalent to the "no deferred inlining" compilation policy of older compilers. • **File** - Completely parse each translation unit before generating any code or data. Equivalent to the "deferred inlining" option of older compilers. Also performs an early dead code and dead data analysis in this mode. Objects with unreferenced internal linkages will be dead-stripped in the compiler rather than in the linker.
Inlining	Enables inline expansion. If there is a `#pragmaINLINE` before a function definition, all calls of this function are replaced by the code of this function, if possible. Using the `-`

Table continues on the next page...

Table 3-6. Tool Settings - ColdFire Compiler > Optimization Options (continued)

Option	Description
	`Oi=c0` option switches off inlining. Functions marked with the `#pragma INLINE` are still inlined. To disable inlining, use the `-Oi=OFF` option.
Bottom-up Inlining	Check to control the bottom-up function inlining method. When active, the compiler inlines function code starting with the last function in the chain of functions calls, to the first one.

3.3.2.6 ColdFire Compiler > Processor

Use this panel to specify processor behavior. You can specify the file paths and define macros.

The following figure shows the **Processor** panel.

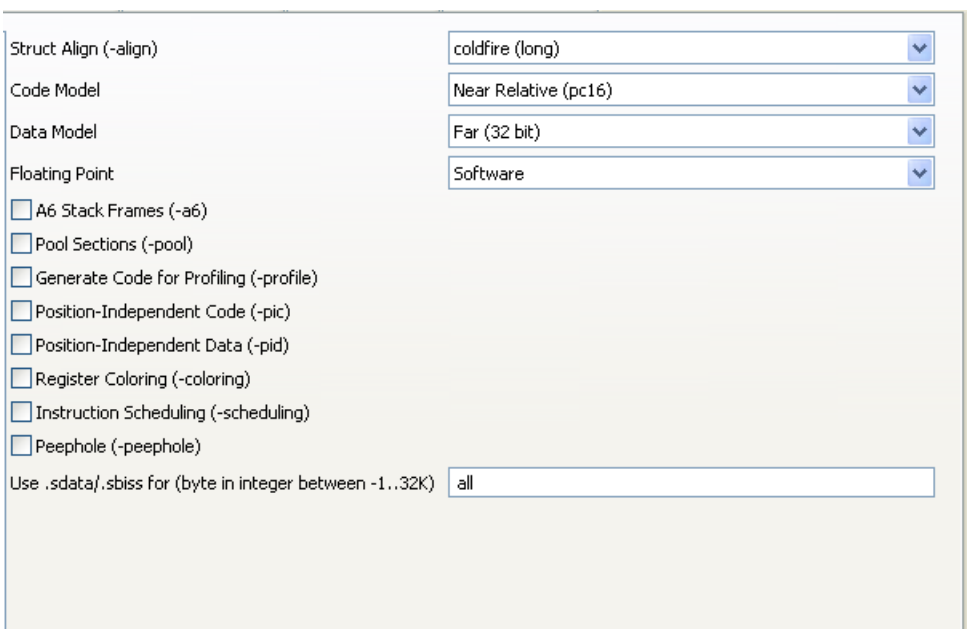

Figure 3-11. Tool Settings - ColdFire Compiler > Processor

The following table lists and defines each option of the **Processor** panel.

Table 3-7. Tool Settings - ColdFire Compiler > Processor Options

Option	Description
Struct Align (`-align`)	Specifies record and structure alignment in memory: • **byte** - Aligns all fields on 1 byte boundaries • **68k (word)** - Aligns all fields on word boundaries • **coldfire (long)** - Aligns all fields on long word boundaries

Table continues on the next page...

Table 3-7. Tool Settings - ColdFire Compiler > Processor Options (continued)

Option	Description
	Default: **coldfire (long)** This panel element corresponds to the options align pragma. **NOTE:** When you compile and link, ensure that alignment is the same for all files and libraries.
Code Model	Specifies access addressing for data and instructions in the object code: • **Smart (16/32 bit)** - Relative (16-bit) for function calls in the same segment; otherwise absolute (32-bit) • **Far (32 bit)** - Absolute for all function calls. Generates a 32-bit absolute references to the code and data. • **Near (16 bit)** - Generates a 16-bit references to the code and data. • **Near Relative (pc16)** - Generates a 16-bits relative references to code. Default: **Near Relative (pc16)**
Data Model	Specifies global-data storage and reference: • **Far (32 bit)** - Storage in far data space; available memory is the only size limit • **Near (16 bit)** - Storage in near data space; size limit is 64K Default: **Far (32 bit)** This panel element corresponds the `far_data` pragma
Floating Point	Specifies handling method for floating point operations: • **Software** - C runtime library code emulates floating-point operations. • **Hardware** - Processor hardware performs floating point operations; only appropriate for processors that have floating-point units. Default: **Software** For software selection, your project must include the appropriate `FP_ColdFire` C runtime library file. Grayed out if your target processor lacks an FPU.
A6 Stack Frames (`-a6`)	Clear to disable call-stack tracing; generates faster and smaller code. By default, the option is checked.
Pool Sections (`-pool`)	Check to collect all string constants into a single data object so your program needs one data section for all of them.
Generate Code for Profiling (`-profile`)	Check to enable the processor generate code for use with a profiling tool. Checking this box corresponds to using the command-line option `-profile`. Clearing this checkbox is equivalent to using the command-line option `-noprofile`
Position-Independent Code (`-pic`)	Check to generate position independent code (PIC) that is non relocatable.
Position-Independent Data (`-pid`)	Check to generate non-relocatable position-independent data (PID). PID is available with 16- and 32-bit addressing.
Register Coloring (`-coloring`)	Clear to enable the Compiler force all local variables to be stack-based except for compiler generated temporaries.
Instruction Scheduling (`-scheduling`)	Clear to prevent from scheduling instructions. Only applied at optimization levels 2 and 3.

Table continues on the next page...

Table 3-7. Tool Settings - ColdFire Compiler > Processor Options (continued)

Option	Description
Peephole (`-peephole`)	Clear to prevent the compiler from compiling long instruction sequences into compact ones. When `ON` it does not affect debugging unless the resulting instruction is a memory-to-memory operation which might make a variable used as temporary disappear.
Use `.sdata/.sbiss` for (byte in integer between `-1..32K`)	The options are: • All data - Select this option button to store all data items in the small data address space • All data smaller than - Select this option button to specify the maximum size for items stored in the small data address space; enter the maximum size in the text box. Using the small data area speeds data access, but has ramifications for the hardware memory map. The default settings specify not using the small data area. By default, all data smaller than is checked.

3.3.2.7 ColdFire Compiler > Language Settings

Use this panel direct the ColdFire compiler to apply specific processing modes to the language source code. You can compile source files with just one collection at a time. To compile source files with multiple collections, you must compile the source code sequentially. After each compile iteration change the collection of settings that the ColdFire compiler uses.

The following figure shows the **Language Settings** panel.

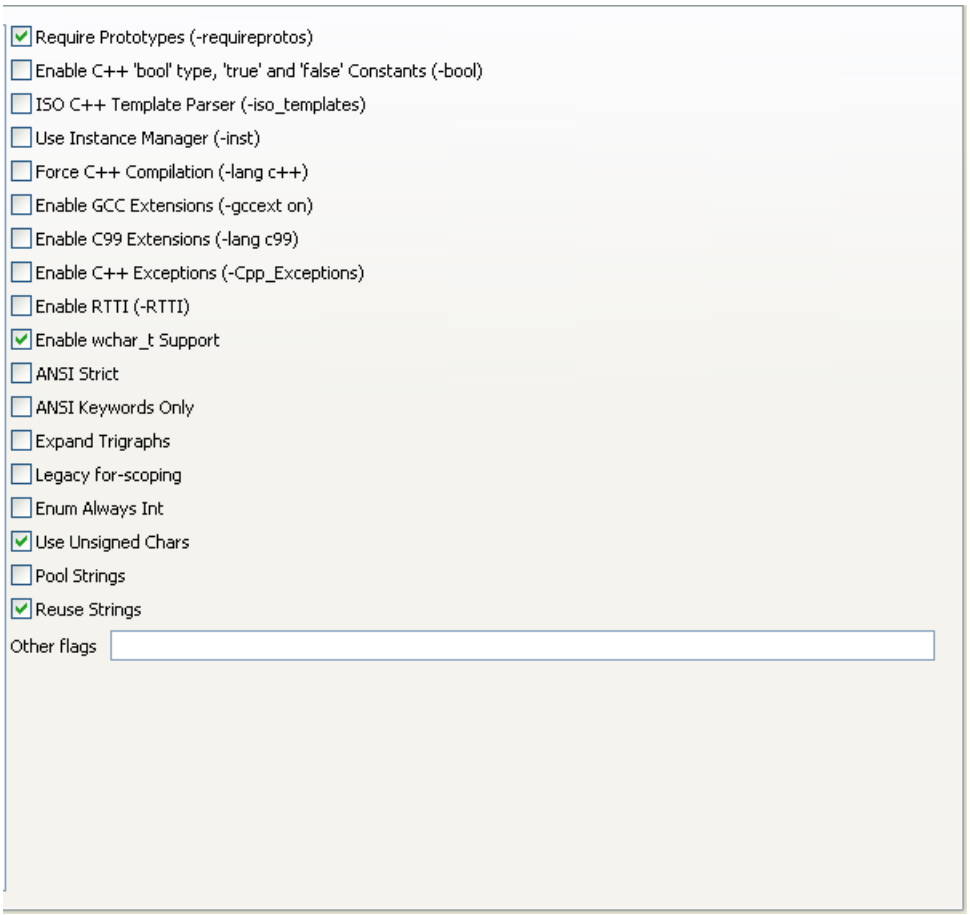

Figure 3-12. Tool settings - ColdFire Compiler > Language Settings

The following table lists and defines each option of the **Language Settings** panel.

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

Option	Description
Require Prototypes (`-requireprotos`)	Check to enforce the requirement of function prototypes. the compiler generates an error message if you define a previously referenced function that does not have a prototype. If you define the function before it is referenced but do not give it a prototype, this setting causes the compiler to issue a warning message.
Enable `C++` `` `bool` ' type, `` `true` ' and `` `false` ' Constants (`-bool`)	Check to enable the C++ compiler recognize the bool type and its true and false values specified in the ISO/IEC 14882-2003 C++ standard; is equivalent to pragma `bool` and the command-line option `-bool`.
ISO C++ Template Parser (`-iso_templates`)	Check to follow the ISO/IEC 14882-2003 standard for C++ to translate templates, enforcing more careful use of the typename and template keywords. The compiler also follows stricter rules for resolving names during declaration and instantiation and is equivalent to pragma `parse_func_templ` and the command-line `option -iso_templates`.

Table continues on the next page...

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

Option	Description
Use Instance Manager (`-inst`)	Check to reduce compile time by generating any instance of a C++ template (or non-inlined inline) function only once.
Force C++ Compilation (`-lang c++`)	Check to translates all C source files as C++ source code and is equivalent to pragma `cplusplus` and the command-line option `-lang c++`.
Enable GCC Extensions (`-gccext on`)	Check to recognize language features of the GNU Compiler Collection (GCC) C compiler that are supported by CodeWarrior compilers; is equivalent to pragma `gcc_extensions` and the command-line option `-gcc_extensions`.
Enable C99 Extensions (`-lang c99`)	Check to recognize ISO/IEC 9899-1999 ("C99") language features; is equivalent to pragma `c99` and the command-line option - `dialect c99`.
Enable C++ Exceptions (`-Cpp_Exceptions`)	Check to generate executable code for C++ exceptions; is equivalent to pragma `exceptions` and the command-line option - `cpp_exceptions`.
Enable `RTTI` (-RTTI)	Check to allow the use of the C++ runtime type information (RTTI) capabilities, including the `dynamic_cast` and `typeid` operators; is equivalent to pragma RTTI and the command-line option -RTTI.
Enable `wchar_t` Support	Check to enable C++ compiler recognize the wchar_t data type specified in the ISO/IEC 14882-2003 C++ standard; is equivalent to pragma `wchar_type` and the command-line option `-wchar_t`.
ANSI Strict	Check to enable C compiler operate in strict ANSI mode. In this mode, the compiler strictly applies the rules of the ANSI/ISO specification to all input files. This setting is equivalent to specifying the `- ansi` command-line option. The compiler issues a warning for each ANSI/ISO extension it finds.
ANSI Keywords Only	Check to generate an error message for all non-standard keywords (ISO/IEC 9899-1990 C, §6.4.1). If you must write source code that strictly adheres to the ISO standard, enable this setting; is equivalent to pragma `only_std_keywords` and the command-line option `-stdkeywords`.
Expand Trigraphs	Check to recognize trigraph sequences (ISO/IEC 9899-1990 C, §5.2.1.1); is equivalent to **pragma** `trigraphs` **and the command-line option** `-trigraphs`.
Legacy for-scoping	Check to generate an error message when the compiler encounters a variable scope usage that the ISO/IEC 14882-2003 C++ standard disallows, but is allowed in the C++ language specified in The Annotated C++ Reference Manual ("ARM"); is equivalent to pragma `ARM_scoping` and the command-line option `-for_scoping`.
Enum Always Int	Check to use signed integers to represent enumerated constants and is equivalent to pragma `enumsalwaysint` and the command-line option `-enum`.
Use Unsigned Chars	Check to treat char declarations as unsigned char declarations and is equivalent to pragma `unsigned_char` and the command-line option `-char unsigned`.

Table continues on the next page...

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

Option	Description
Pool Strings	Check to collect all string constants into a single data section in the object code it generates and is equivalent to pragma `pool_strings` and the command-line `option -strings pool`.
Reuse Strings	Check to store only one copy of identical string literals and is equivalent to opposite of the pragma `dont_reuse_strings` and the command-line `option -string reuse`.
Other flags	Specify additional command line options for the compiler; type in custom flags that are not otherwise available in the UI.

3.3.3 CodeWarrior Tips and Tricks

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

  NOTE

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

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

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

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

Chapter 4
Using Build Tools on Command Line

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

4.1 Configuring Command-Line Tools

This topic contains the following sections:

4.1.1 Setting CodeWarrior Environment Variables

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

Table 4-1. Environment variables for CodeWarrior command-line tools

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

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

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

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

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

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

```
rem Use ; to separate directory paths

set CWFolder=C:\Freescale\CW MCU v10.x

set MWCIncludes=%CWFolder%\MCU\ColdFire_Support\ewl\EWL_C\Include

set MWCIncludes=%MWCIncludes%;%CWFolder%\MCU\ColdFire_Support\ewl\EWL_C++\Include

set MWCIncludes=%MWCIncludes%;%CWFolder%\MCU\ColdFire_Support\ewl\EWL_C\Include\sys

set MWLibraries=%CWFolder%\MCU\ColdFire_Suppport\ewl\lib
```

4.1.2 Setting PATH Environment Variable

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

Listing: Example - Setting PATH

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

4.2 Invoking Command-Line Tools

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

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

```
tool options files
```

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

The CodeWarrior command-line tools are:

- `mwasmcf.exe` (coldfire assembler) - the assembler translate asm files into object files
- `mwccmcf.exe` (coldfire compiler) - the compiler translates C/C++ compilation units into object files.
- `mwldmcf.exe` (coldfire linker) - the linker binds object files to create executables or libraries.

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

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

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

4.3 Getting Help

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

```
tool -help
```

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

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

```
tool -help | more
```

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

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

```
tool -help usage
```

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

4.3.1 Parameter Formats

Parameters in an option are formatted as follows:

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

4.3.2 Option Formats

Options are formatted as follows:

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

  When the option's name is "`-xxx+`", however, the parameter must directly follow the option, without the "`+`" character (as in "`-xxx45`") and with no space separator.

- An option given as "`-[no]xxx`" may be issued as "`-xxx`" or "`-noxxx`".

  The use of "`-noxxx`" reverses the meaning of the option.

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

4.3.3 Common Terms

These common terms appear in many option descriptions:

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

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

4.4 File Name Extensions

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

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

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

Listing: Example of using linker command files

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

NOTE

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

Chapter 5
Command-Line Options for Standard C Conformance

This chapter lists the following command-line options for standard C conformance:

- -ansi
- -stdkeywords
- -strict

5.1 -ansi

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

Syntax

```
-ansi keyword
```

The arguments for `keyword` are:

`off`

Turns ISO conformance off. Same as

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

`on | relaxed`

Turns ISO conformance on in relaxed mode. Same as

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

`strict`

Turns ISO conformance on in strict mode. Same as

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

5.2 -stdkeywords

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

Syntax

```
-stdkeywords on | off
```

Remarks

Default setting is `off`.

5.3 -strict

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

Syntax

```
-strict on | off
```

Remarks

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

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

The default setting is `off`.

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

This chapter lists the following command-line options for standard C++ conformance:

6.1 -ARM

Deprecated. Use -for_scoping instead.

6.2 -bool

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

Syntax

```
-bool on | off
```

Remarks

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

6.3 -Cpp_exceptions

Controls the use of C++ exceptions.

Syntax

```
-cpp_exceptions on | off
```

Remarks

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

6.4 -dialect

Specifies the source language.

Syntax

```
-dialect keyword

-lang keyword
```

The arguments for *keyword* are:

`c`

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

`c99`

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

`c++ | cplus`

Always treat source as the C++ language.

`ec++`

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

`objc`

Always treat source as the Objective-C language.

6.5 -for_scoping

Controls legacy scope behavior in for loops.

Syntax

```
-for_scoping
```

Remarks

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

6.6 -instmgr

Controls whether the instance manager for templates is active.

Syntax

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

The options for *keyword* are:

`off`

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

`on`

Turns on the C++ instance manager.

`file=` *path*

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

Remarks

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

6.7 -iso_templates

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

Syntax

```
-iso_templates on | off
```

Remarks

Default setting is `on`.

6.8 -RTTI

Controls the availability of runtime type information (RTTI).

Syntax

```
-RTTI on | off
```

Remarks

Default setting is `on`.

6.9 -som

```
Obsolete. This option is no longer available.
```

6.10 -som_env_check

```
Obsolete. This option is no longer available.
```

6.11 -wchar_t

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

Syntax

```
-wchar_t on | off
```

Remarks

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

Chapter 7
Command-Line Options for Language Translation

This chapter lists the following command-line options for language translation:

Table 7-1. Command-line Options for Language Translation

-char	-mapcr	-msext
-defaults	-MM	-once
-encoding	-MD	-pragma
-flag	-MMD	-relax_pointers
-gccext	-Mfile	-requireprotos
-gcc_extensions	-MMfile	-search
-M	-MDfile	-trigraphs
-make	-MMDfile	-nolonglong

7.1 -char

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

Syntax

```
-char keyword
```

The arguments for *keyword* are:

`signed`

`char` data items are signed.

`unsigned`

`char` data items are unsigned.

Remarks

The default is `unsigned`.

7.2 -defaults

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

Syntax

```
-defaults


-nodefaults
```

Remarks

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

Use `-nodefaults` to disable the use of additional environment variables.

7.3 -encoding

Specifies the default source encoding used by the compiler.

Syntax

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

The options for *keyword* are:

`ascii`

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

`autodetect | multibyte | mb`

Scan file for multibyet encoding.

`system`

Uses local system format.

`UTF[8 | -8]`

Unicode Transformation Format (UTF).

`SJIS | Shift-JIS | ShiftJIS`

Shift Japanese Industrial Standard (Shift-JIS) format.

`EUC[JP | -JP]`

Japanese Extended UNIX Code (EUCJP) format.

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

International Organization of Standards (ISO) Japanese format.

Remarks

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

7.4 -flag

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

Syntax

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

Remarks

For example, this option setting

```
-flag require_prototypes
```

is equivalent to

```
#pragma require_prototypes on
```

This option setting

```
-flag no-require_prototypes
```

is the same as

```
#pragma require_prototypes off
```

7.5 -gccext

Enables GCC (GNU Compiler Collection) C language extensions.

Syntax

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

Remarks

See GCC Extensions for a list of language extensions that the compiler recognizes when this option is `on`.

The default setting is `off`.

7.6 -gcc_extensions

Equivalent to the `-gccext` option.

Syntax

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

7.7 -M

Scans the source files for dependencies and emits a Makefile, without generating the object code.

Syntax

```
-M
```

Remarks

This command is global and case-sensitive.

7.8 -make

Scans the source files for dependencies and emits a Makefile, without generating the object code.

Syntax

```
-make
```

Remarks

This command is global.

7.9 -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.

7.10 -MM

Scans source files for dependencies and emits a Makefile, without generating the object code or listing system `#include` files.

Syntax

```
-MM
```

Remarks

This command is global and case-sensitive.

7.11 -MD

Scans the source files for dependencies and emits a Makefile, generates the object code, and writes a dependency map.

Syntax

```
-MD
```

Remarks

This command is global and case-sensitive.

7.12 -MMD

Scans the source files for dependencies and emits a Makefile, generates the object code, writes a dependency map, without listing system `#include` files.

Syntax

```
-MMD
```

Remarks

This command is global and case-sensitive.

7.13 -Mfile

Like `-M`, but writes dependency map to the specified file.

Syntax

```
-Mfile file
```

Remarks

This command is global and case-sensitive.

7.14 -MMfile

Like -MM, but writes dependency map to the specified file.

Syntax

```
-MMfile file
```

Remarks

This command is global and case-sensitive.

7.15 -MDfile

Like `-MD`, but writes dependency map to the specified file.

Syntax

```
-MDfile file
```

Remarks

This command is global and case-sensitive.

7.16 -MMDfile

Like `-MMD`, but writes dependency map to the specified file.

Syntax

```
-MMDfile file
```

Remarks

This command is global and case-sensitive.

7.17 -msext

Allows Microsoft® Visual C++ extensions.

Syntax

```
-msext on | off
```

Remarks

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

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

When `off`, this command disables the extensions. It is default on non-x86 targets.

7.18 -once

Prevents header files from being processed more than once.

Syntax

```
-once
```

Remarks

You can also add `#pragmaonceon` in a prefix file.

7.19 -pragma

Defines a pragma for the compiler.

Syntax

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

The arguments are:

`name`

Name of the pragma.

`setting`

Arguments to give to the pragma

Remarks

For example, this command-line option

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

is equivalent to inserting this directive in source code

```
#pragma c99 on
```

7.20 -relax_pointers

Relaxes the pointer type-checking rules in C.

Syntax

```
-relax_pointers
```

Remarks

This option is equivalent to

```
#pragma mpwc_relax on
```

7.21 -requireprotos

Controls whether or not the compiler should expect function prototypes.

Syntax

```
-r[equireprotos]
```

7.22 -search

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

Syntax

```
-search
```

7.23 -trigraphs

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

Syntax

```
-trigraphs on | off
```

Remarks

Default setting is `off`.

7.24 -nolonglong

Disables the 'long long' support.

Syntax

```
-nolonglong
```

Chapter 8
Command-Line Options for Diagnostic Messages

This chapter lists the following command-line options for diagnostic messages:

Table 8-1. Command-line Options for Diagnostic Messages

-disassemble	-nofail	-version
-help	-progress	-timing
-maxerrors	-S	-warnings
-maxwarnings	-stderr	-wraplines
-msgstyle	-verbose	

8.1 -disassemble

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

Syntax

```
-dis[assemble]
```

Remarks

This option is global.

8.2 -help

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

Syntax

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

The options for *keyword* are:

`all`

Show all standard options

`group=` *keyword*

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

`[no]compatible`

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

`[no]deprecated`

Shows deprecated options

`[no]ignored`

Shows ignored options

`[no]meaningless`

Shows options meaningless for this target

`[no]normal`

Shows only standard options

`[no]obsolete`

Shows obsolete options

`[no]spaces`

Inserts blank lines between options in printout.

`opt[ion]=` *name*

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

`search=` *keyword*

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

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

Categorizes groups of options by tool; default.

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

`usage`

Displays usage information.

8.3 -maxerrors

Specifies the maximum number of errors messages to show.

Syntax

```
-maxerrors max
```

`max`

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

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

8.4 -maxwarnings

Specifies the maximum number of warning messages to show.

Syntax

```
-maxerrors max
```

`max`

Specifies the number of warning messages. Common values are:

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

8.5 -msgstyle

Controls the style used to show error and warning messages.

Syntax

```
-msgstyle keyword
```

The options for *keyword* are:

`gcc`

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

`ide`

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

`mpw`

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

`parseable`

Uses context-free machine parseable message style.

`std`

Uses standard message style. This is the default.

`enterpriseIDE`

Uses Enterprise-IDE message style.

8.6 -nofail

Continues processing after getting error messages in earlier files.

Syntax

```
-nofail
```

8.7 -progress

Shows progress and version information.

Syntax

```
-progress
```

8.8 -S

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

Syntax

```
-S
```

8.9 -stderr

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

Syntax

```
-stderr
```

```
-nostderr
```

Remarks

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

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

8.10 -verbose

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

Syntax

```
-v[erbose]
```

Remarks

This option also gives progress and version information.

8.11 -version

Displays version, configuration, and build data.

Syntax

```
-v[ersion]
```

8.12 -timing

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

Syntax

```
-timing
```

8.13 -warnings

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

Syntax

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

The options for `keyword` are:

`off`

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

```
#pragma
warning off
```

`on`

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

```
#pragma
warning on
```

`[no]cmdline`

Passed to all tools.

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

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

```
#pragma
warning_errors
```

`all`

Turns on all warning messages and require prototypes.

`[no]pragmas | [no]illpragmas`

Issues warning messages on invalid pragmas. Equivalent to

```
#pragma
```

```
warn_illpragma
```

```
[no]empty[decl]
```

Issues warning messages on empty declarations. Equivalent to

```
#pragma
warn_emptydecl
```

```
[no]possible | [no]unwanted
```

Issues warning messages on possible unwanted effects. Equivalent to

```
#pragma
warn_possunwanted
```

```
[no]unusedarg
```

Issues warning messages on unused arguments. Equivalent to

```
#pragma
warn_unusedarg
```

```
[no]unusedvar
```

Issues warning messages on unused variables. Equivalent to

```
#pragma
warn_unusedvar
```

```
[no]unused
```

Same as

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

```
[no]extracomma | [no]comma
```

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

```
#pragma
warn_extracomma
```

`[no]pedantic | [no]extended`

Pedantic error checking.

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

Issues warning messages on hidden virtual functions. Equivalent to

```
#pragma
warn_hidevirtual
```

`[no]implicit[conv]`

Issues warning messages on implicit arithmetic conversions. Implies

```
-warn impl_float2int,impl_signedunsigned
```

`[no]impl_int2float`

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

```
#pragma
warn_impl_i2f_conv
```

`[no]impl_float2int`

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

```
#pragma
warn_impl_f2i_conv
```

`[no]impl_signedunsigned`

Issues warning messages on implicit signed/unsigned conversions.

`[no]relax_i2i_conv`

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

`[no]notinlined`

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

```
#pragma
warn_notinlined
```

`[no]largeargs`

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

```
#pragma
warn_largeargs
```

`[no]structclass`

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

```
#pragma
warn_structclass
```

`[no]padding`

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

```
#pragma
warn_padding
```

`[no]notused`

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

```
#pragma
```

```
warn_resultnotused
```

`[no]missingreturn`

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

```
#pragma
warn_missingreturn
```

`[no]unusedexpr`

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

```
#pragma
warn_no_side_effect
```

`[no]ptrintconv`

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

`[no]anyptrintconv`

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

```
#pragma
warn_ptr_int_conv
```

`[no]undef[macro]`

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

```
#pragma
warn_undefmacro
```

`[no]filecaps`

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

```
#pragma
warn_filenamecaps
```

```
[no]sysfilecaps
```

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

```
#pragma
warn_filenamecaps_system
```

```
[no]tokenpasting
```

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

```
#pragma
warn_illtokenpasting
```

```
[no]alias_ptr_conv
```

Generates the warnings for potentially dangerous pointer casts.

```
display | dump
```

Displays list of active warnings.

8.14 -wraplines

Controls the word wrapping of messages.

Syntax

```
-wraplines
```

```
-nowraplines
```

Chapter 9
Command-Line Options for Preprocessing

This chapter lists the following command-line options for preprocessing:

Table 9-1. Coomand-line Options for Preprocessing

-convertpaths	-I-	-prefix
-cwd	-I+	-noprecompile
-D+	-include	-nosyspath
-define	-ir	-stdinc
-E	-P	-U+
-EP	-precompile	-undefine
-gccincludes	-preprocess	-allow_macro_redefs
-gccdepends	-ppopt	

9.1 -convertpaths

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

Syntax

```
-[no]convertpaths
```

Remarks

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

- Mac OS® - colon " `:`" (`:sys:stat.h`)

- UNIX - forward slash " /" (sys/stat.h)
- Windows® operating systems - backward slash " \" (sys\stat.h)

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

NOTE

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

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

9.2 -cwd

Controls where a search begins for # include files.

Syntax

```
-cwd keyword
```

The options for *keyword* are:

explicit

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

include

Begins searching in directory of referencing file.

proj

Begins searching in current working directory (default).

source

Begins searching in directory that contains the source file.

Remarks

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

9.3 -D+

Same as the `-define` option.

Syntax

```
-D+
name
```

The parameters are:

`name`

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

9.4 -define

Defines a preprocessor symbol.

Syntax

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

The parameters are:

`name`

The symbol name to define.

`value`

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

9.5 -E

Tells the command-line tool to preprocess source files.

Syntax

```
-E
```

Remarks

This option is global and case sensitive.

9.6 -EP

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

Syntax

```
-EP
```

Remarks

This option is global and case sensitive.

9.7 -gccincludes

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

Syntax

```
-gccinc[ludes]
```

Remarks

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

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

This command is global.

9.8 -gccdepends

Writes dependency file.

Syntax

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

Remarks

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

This command is global.

9.9 -I-

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

Syntax

```
-I-
```

```
-i-
```

Remarks

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

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

This command is global.

9.10 -I+

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

Syntax

```
-I+path
```

```
-i path
```

The parameters are:

`path`

The non-recursive access path to append.

Remarks

This command is global and case-sensitive.

9.11 -include

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

Syntax

```
-include file
```

`file`

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

Remarks

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

9.12 -ir

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

Syntax

```
-ir path
```

`path`

The recursive access path to append.

9.13 -P

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

Syntax

```
-P
```

Remarks

This option is global and case-sensitive.

9.14 -precompile

Precompiles a header file from selected source files.

Syntax

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

`file`

If specified, the precompiled header name.

`dir`

If specified, the directory to store the header file.

`""`

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

Remarks

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

```
-precompile
filesource
```

is equivalent to

```
-c -o
filesource
```

9.15 -preprocess

Preprocesses the source files. This command is global.

Syntax

```
-preprocess
```

9.16 -ppopt

Specifies options affecting the preprocessed output.

Syntax

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

The arguments for *keyword* are:

`[no]break`

Emits file and line breaks. This is the default.

`[no]line`

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

`[no]full[path]`

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

`[no]pragma`

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

`[no]comment`

Controls whether comments are kept or stripped.

`[no]space`

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

Remarks

The default settings is `break`.

9.17 -prefix

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

Syntax

```
-prefix file
```

9.18 -noprecompile

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

Syntax

```
-noprecompile
```

9.19 -nosyspath

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

Syntax

```
-nosyspath
```

Remarks

This command is global.

9.20 -stdinc

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

Syntax

```
-stdinc
```

```
-nostdinc
```

Remarks

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

9.21 -U+

Same as the -undefine option.

Syntax

```
-U+
name
```

9.22 -undefine

Undefines the specified symbol name.

Syntax

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

```
-U+name
```

`name`

The symbol name to undefine.

Remarks

This option is case-sensitive.

9.23 -allow_macro_redefs

Allows macro redefinitions without an error or warning.

Syntax

```
-allow_macro_redefs
```

Chapter 10
Command-Line Options for Library and Linking

This chapter lists the following command-line options for library and linking:

10.1 -keepobjects

Retains or deletes object files after invoking the linker.

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 option is global.

NOTE

Object files are always kept when compiling.

10.2 -nolink

Compiles the source files, without linking.

Syntax

```
-nolink
```

Remarks

This command is global.

10.3 -o

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

Syntax

```
-o file | dir
```

`file`

The output file name.

`dir`

The directory to store object files or text output.

Chapter 11
Command-Line Options for Object Code

This chapter lists the following command-line options for object code:

11.1 -c

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

Syntax

```
-c
```

Remarks

This option is global.

11.2 -codegen

Instructs the compiler to compile without generating object code.

Syntax

```
-codegen
```

```
-nocodegen
```

Remarks

This option is global.

11.3 -enum

Specifies the default size for enumeration types.

Syntax

```
-enum keyword
```

The arguments for *keyword* are:

`int`

Uses `int` size for enumerated types.

`min`

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

11.4 -min_enum_size

Specifies the size, in bytes, of enumerated types.

Syntax

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

Remarks

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

11.5 -ext

Specifies which file name extension to apply to object files.

Syntax

```
-ext extension
```

`extension`

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

- Limited to a maximum length of 14 characters
- Extensions specified without a leading period replace the source file's extension. For example, if *extension* is "`o`" (without quotes), then `source.cpp` becomes `source.o`.
- Extensions specified with a leading period (`.`*extension*) are appended to the object files name. For example, if `extension` is "`.o`" (without quotes), then `source.cpp` becomes `source.cpp.o`.

Remarks

This command is global. The default is none.

11.6 -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.

Chapter 12
Command-Line Options for Optimization

This chapter lists the following command-line options for optimization:

12.1 -inline

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

Syntax

```
-inline keyword
```

The options for *keyword* are:

`off | none`

Turns off inlining.

`on | smart`

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

`auto`

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

`noauto`

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

`deferred`

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

`level=n`

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

`all`

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

`[no]bottomup`

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

12.2 -ipa

Controls the interprocedural analysis optimization.

Syntax

```
-ipa file | function | off | program | program-final
```

The options are as follows:

```
-ipa off or -ipa function
```

Turns off the interprocedural analysis. This is default.

```
-ipa file
```

Turns on the interprocedural analysis at file level. For example:

```
mwccmcf -c -ipa file file1.c file2.c
```

This applies optimization to file `file1.c` and `file2.c` but not across both files. This command generates object files `file1.o` and `file2.o` which can later be linked to generate executable file.

```
-ipa program
```

Turns on the interprocedural analysis at program level use. For example:

```
mwccmcf -o myprog.elf -ipa program file1.c file2.c
```

This generates object code, applies this optimization among all resulting object code files to link it to generate out file `myprog.elf`.

To separate compile and linking steps for IPA, follow the steps listed below:

```
mwccmcf -c -ipa program file1.c
```

```
mwccmcf -c -ipa program file2.c
```

Compiles `file1.c` and `file2.c` into regular object files and also intermediate optimized obj files called `file1.irobj` and `file2.irobj`.

To link the object files, refer to the .o files or .irobj files like:

```
mwccmcf -o myprog.elf -ipa program file1.o file2.o
```

or

```
mwccmcf -o myprog.elf -ipa program file1.irobj
file2.irobj
```

```
-ipa program-final | program2
```

Invokes the linker directly.

For example:

```
mwccmcf -ipa program-final file1.irobj file2.irobj


mwldmcf -o myprog.elf file1.o file2.o
```

12.3 -O

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

Syntax

```
-O
```

Remarks

Provided for backwards compatibility.

12.4 -O+

Controls optimization settings.

Syntax

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

The *keyword* arguments are:

`0`

Equivalent to `-opt off`.

`1`

Equivalent to `-opt level=1`.

`2`

Equivalent to `-opt level=2`.

`3`

Equivalent to `-opt level=3`.

`4`

Equivalent to `-opt level=4,intrinsics`.

`p`

Equivalent to `-opt speed`.

`s`

Equivalent to `-opt space`.

Remarks

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

12.5 -opt

Specifies code optimization options to apply to object code.

Remarks

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

The *keyword* arguments are:

`off | none`

Suppresses all optimizations. This is the default.

```
on
```

Same as `-opt level=2`

```
all | full
```

Same as `-opt speed,level=4,intrinsics,noframe`

```
l[evel]=num
```

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

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

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

```
[no]space
```

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

```
[no]speed
```

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

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

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

```
[no]deadcode
```

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

```
[no]deadstore
```

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

`[no]lifetimes`

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

`[no]loop[invariants]`

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

`[no]prop[agation]`

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

`[no]strength`

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

`[no]dead`

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

`display | dump`

Displays complete list of active optimizations.

Chapter 13
ColdFire Command-Line Options

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

13.1 Naming Conventions

The following table lists the names of the CodeWarrior command line tools.

Table 13-1. Command line tools

Tool	Tasks
`mwccmcf.exe`	This tool translates the C and C++ source code to a ColdFire object code.
`mwasmmcf.exe`	This tool translates the ColdFire assembly language to an object code.
`mwldmcf.exe`	This tool links the object code to a loadable image file.

13.2 Diagnostic Command-Line Options

This topic lists the following diagnostic command-line options:

13.2.1 -g

Generates the debugging information.

Syntax

```
-g
```

Remarks

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

```
-sym full
```

13.2.2 -sym

Specifies the global debugging options.

Syntax

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

The options for the *keyword* are:

`off`

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

`on`

Generates the debugging information.

`full[path]`

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

13.3 Library and Linking Command-Line Options

This topic lists the following library and linking command-line options:

Table 13-2. Library and Linking Command-line Options

-deadstrip	-shared	-srec	-rbin
-force_active	-application	-sreceol	-rbingap
-main	-sdata	-sreclength	-keep
-map	-show	-brec	-list
-library	-dispaths	-breclength	-lavender

13.3.1 -deadstrip

Enables the dead-stripping of the unused code.

Syntax

```
-[no]dead[strip]
```

Remarks

This is a linker option.

13.3.2 -force_active

Specifies a list of symbols as undefined. Useful to the force linking static libraries.

Syntax

```
-force_active symbol[,...]
```

Remarks

This is a linker option.

13.3.3 -main

Sets the main entry point for the application or shared library.

Syntax

```
-m[ain] symbol
```

Remarks

The maximum length of *symbol* is 63 characters. The *default* is `__start`. To specify the no entry point, use:

```
-main ""
```

This is a linker option.

13.3.4 -map

Generates a link map file.

Syntax

```
-map [keyword[,...]]
```

The choices for *keyword* are:

`closure`

Calculates the symbol closures.

`unused`

Lists the unused symbols.

`keep`

Keeps the link map on failure.

Remarks

If multiple `symbol =.;` directives exist for the same symbol in the linker command file, the symbol appears multiple times in the map file. The last entry of a symbol in the map file always has the actual symbol address.

This is a linker option.

13.3.5 -library

Generates a static library.

Syntax

```
-library
```

Remarks

This is a global option.

13.3.6 -shared

Generates a shared library.

Syntax

```
-shared
```

Remarks

This is a global option.

13.3.7 -application

Generates an application.

Syntax

```
-application
```

Remarks

This is a global option. This is default.

13.3.8 -sdata

Places the data objects ' `all`' or smaller than ' `size`' bytes in the `.sdata` section.

Syntax

```
-sdata all|size
```

Remarks

The *size* value specifies the maximum size, in bytes, of all the objects in the small data section (typically named " `.sdata`"). The linker places the objects that are greater than this size in the data section (typically named ". `data`") instead.

The *default* value for *size* is `8`.

This is both a linker and a compiler option.

13.3.9 -show

Specifies the information to list in a disassembly.

Syntax

```
-show keyword[,...]
```

The options for the *keyword* are:

`only | none`

Shows no disassembly. Begins a list of options with `only` or `none` to prevent the default information from appearing in the disassembly.

`all`

Shows everything. This is default.

`[no]code | [no]text`

Shows disassembly of the code sections. This is default.

`[no]comments`

Shows comment field in the code. Implies `-show code`. This is default.

`[no]extended`

Shows the extended mnemonics. Implies `-show code`. This is default.

`[no]data`

Shows data with `-show verbose`, shows hex dumps of the sections. This is default.

`[no]debug | [no]sym`

Shows the symbolics information. This is default.

`[no]exceptions`

Shows the exception tables. Implies `-show data`. This is default.

`[no]headers`

Shows the ELF headers. This is default.

`[no]hex`

Shows addresses and opcodes in the code disassembly. Implies `-show code`. This is default.

`[no]names`

Shows the symbol table. This is default.

`[no]relocs`

Shows resolved relocations in the code and relocation tables. This is default.

`[no]source`

Shows source in the disassembly. Implies `-show code`, with `-show verbose`, displays the entire source file in the output, else shows only four lines around each function. This is default.

`[no]xtables`

Shows the exception tables. This is default.

`[no]verbose`

Shows verbose information, including the hex dump of program segments in the applications. This is default.

Remarks

This is a linker option.

13.3.10 -dispaths

Disassembler file paths mapping.

Syntax

```
-dispaths =
```

Remarks

Useful to map libraries sources, `src=dest`. This is a linker option.

13.3.11 -srec

Generates an S-record file.

Syntax

```
-srec
```

Remarks

Ignored when generating static libraries. This is a linker option.

13.3.12 -sreceol

Specifies the end-of-line style to use in an S-record file.

Syntax

```
-sreceol keyword
```

The options for the *keyword* are:

`mac`

Uses the Mac OS®-style end-of-line format.

`dos`

Uses the Microsoft® Windows®-style end-of-line format. This is the *default* choice.

`unix`

Uses a UNIX-style end-of-line format.

Remarks

This option also generates an S-record file if the `-srec` option has not already been specified.

This is a linker option.

13.3.13 -sreclength

Specify the length of S-records (should be a multiple of 4).

Syntax

```
-sreclength length
```

The options for the *length* are from 8 to 252. The default is 64.

Remarks

Implies `-srec`. This is a linker option.

13.3.14 -brec

Generates a B-record file - *.bin file. Each line of this binary file will contain a 4-byte address (big endian), a 4-byte length and a number of data bytes as specified by the length. There is no checksum related to a line. If data at a specified address overpass a maximum length a new line will be generated. The maximum length of a line can be specified using the -breclength option below.

Syntax

```
-brec
```

Remarks

This option is ignored if generating the static libraries.

This is a linker option.

13.3.15 -breclength

Specifies the length of B-records (should be a multiple of 4).

Syntax

```
-breclength length
```

The options for the *length* is any multiple of four from 8 to 252. The default is 64.

Remarks

This option implies `-brec`. This is a linker option.

13.3.16 -rbin

Generates a raw binary image of the application (the exact content that will be downloaded on the hardware board) - *.rbin file. If there are addresses that are not defined as part of the application they will be filled with 0x00 in the raw binary image. The gap between defined addresses should not exceed the maximum gap which can be configured using the -rbingap option below. If this gap is exceeded the raw binary file generation is dropped.

Syntax

```
-rbin
```

Remarks

This is a linker option.

13.3.17 -rbingap

Specifies maximum gap in the Raw-Binary files.

Syntax

```
-rbingap gap
```

Remarks

The range for '`gap`' is `0x0 - 0xffffffff`. The default value is 65536.

This is a linker option.

13.3.18 -keep

Keeps the local symbols (such as relocations and output segment names) generating during linking.

Syntax

```
-keep[local] on|off
```

Remarks

The default is `on`. This is a linker option.

13.3.19 -list

Generates the assembly listing.

Syntax

```
-list[ing]
```

Remarks

This is a linker option.

13.3.20 -lavender

Specifies a library selection.

Syntax

```
-lavender keyword[,...]
```

The options for `keyword` are:

`model=string`

Selects a library model for `string`, maximum length is 511 chars. The default is none.

`print=string`

Selects a `printf` formatter for `string`, maximum length is 511 chars;. The default is none.

`scan=string`

Selects a `scanf` formatter for `string`, maximum length is 511 chars. The default is none.

`io=string`

Selects an IO mode for `string`, maximum length is 511 chars. The default is none.

Remarks

This is a library option.

13.4 Code Generation Command-Line Options

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

Table 13-3. Code Generation Command-line Options

-abi	-pic	-intsize	-coloring
-a6	-pid	-model	-scheduling
-align	-processor	-pool	
-fp	-profile	-peephole	

13.4.1 -abi

Specifies an application-binary interface.

Syntax

```
-abi keyword
```

The options for the *keyword* are:

`standard`

Standard (MPW) ABI

`register`

Registers ABI. This is default.

`compact`

Compact ABI

13.4.2 -a6

Generates the A6 stack frames.

Syntax

```
-[no]a6
```

13.4.3 -align

Specifies the structure and array alignment options.

Syntax

```
-align keyword[,...]
```

The options for the *keyword* are:

`68k`

MC68K (word) alignment (two bytes unless field is one byte).

`coldfire`

Sets the long word alignment (four bytes unless field is smaller). This is default.

`byte`

Sets the alignment on bytes.

`word`

Sets the word alignment (2 bytes unless field issmaller).

`long`

Sets the long word alignment (four bytes unless field is smaller). This is default.

`native=68k`

Sets the native alignment value to 68k

`native=coldfire`

Sets the native alignment value to coldfire

`mc68k`

Deprecated: MC68K alignment (two bytes unless field is one byte)

`mc68k4byte`

Deprecated: MC68K 4-byte alignment (four bytes unless field is smaller); default

`power[pc]`

Deprecated: PowerPC alignment (align based on size of field)

`power[pc]`

Deprecated: PowerPC alignment (align based on size of field)

`array[members]`

Aligns array members.

13.4.4 -fp

Specifies the floating point options.

Syntax

```
-fp
keyword
```

The options for the *keyword* are:

`soft[ware]`

Generates the calls to software FP emulation library. This is *default*.

`hard[ware]`

Generates the FPU instructions.

13.4.5 -pic

Generates the position-independent code references.

Syntax

```
-[no]pic
```

13.4.6 -pid

Generates the position-independent data references.

Syntax

```
-[no]pid
```

13.4.7 -processor

Generates and links the object code for a specific processor.

Syntax

```
-proc[essor] generic
```

13.4.8 -profile

Generates code for profiling.

Syntax

```
-[no]profile
```

13.4.9 -intsize

Specifies the size of(int).

Syntax

```
-intsize keyword
```

The options for the `keyword` are:

`2`

Uses two-byte integers

`4`

Four-byte integers. This is default.

13.4.10 -model

Specifies the code and data model options.

```
-model keyword[,...]
```

The options for the `keyword` are:

`near`

Generates a 16-bit references to the code and data.

`far`

Generates a 32-bits absolute references to the code and data.

`nearCode`

Generates a 16-bits absolute references to code.

`nearRelCode`

Generates a 16-bits relative references to code.

`farCode`

Generates a 32-bits absolute references to code.

`nearData`

Generates a 16-bit absolute references to data.

`farData`

Generates a 32-bit absolute references to data. This is default.

`smartCode|codeSmart`

Generates a 16- or 32-bit absolute references to the code depending upon whether the callee can be seen from the current scope or file. This is default.

13.4.11 -pool

Concatenates the objects into single sections.

Syntax

```
-[no]pool[_section]
```

13.4.12 -peephole

Uses the peephole optimization.

Syntax

```
-[no]peephole
```

Remarks

This is default.

13.4.13 -coloring

Uses register coloring for the locals.

Syntax

```
-[no]coloring
```

Remarks

This is default.

13.4.14 -scheduling

Uses an instruction scheduling (requires -O2 or above).

Syntax

```
-[no]scheduling
```

Remarks

This is default.

Chapter 14
ELF Linker and Command Language

This chapter explains the CodeWarrior Executable and Linking Format (ELF) Linker. Beyond making a program file from your project's object files, the linker has several extended functions for manipulating program code in different ways. You can define variables during linking, control the link order down to the level of single functions, and change the alignment.

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

This chapter consists of these sections:

14.1 Deadstripping

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

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

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

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

- To disable deadstripping for particular symbols, enter the symbol names in the **Force Active Symbols** text box of the **ColdFire Linker** Settings Panel.
- To disable deadstripping for individual sections of the linker command file, use the `KEEP_SECTION()` directive. As code does not directly reference interrupt-vector tables, a common use for this directive is disabling deadstripping for these interrupt-vector tables. The subsection Closure Segments provides additional information about the `KEEP_SECTION()` directive.

NOTE

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

14.2 Defining Sections in Source Code

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

Use the `__declspec` directive to specify where to place a single definition in object code. The following listing shows an example.

Listing: Using the __declspec directive to specify where to place definitions

```
#pragma define_section data_type ".myData" far_absolute RW
__declspec(data_type) int red;

__declspec(data_type) int sky;
```

14.3 Executable files in Projects

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

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

14.4 S-Record Comments

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

14.5 LCF Structure

Linker command files consist of three kinds of segments, which *must* be in this order:

- A Memory Segment, which begins with the `MEMORY{}` directive
- Optional Closure Segments, which begin with the `FORCE_ACTIVE{}`, `KEEP_SECTION{}`, or `REF_INCLUDE{}` directives
- A Sections Segment, which begins with the `SECTIONS{}` directive

The topics covered here:

- Memory Segment
- Closure Segments
- Sections Segment

14.5.1 Memory Segment

Use the memory segment to divide available memory into segments. The following listing shows the pattern.

Listing: Example Memory Segment

```
MEMORY {
    segment_1 (RWX): ORIGIN = 0x80001000, LENGTH = 0x19000

    segment_2 (RWX): ORIGIN = AFTER(segment_1), LENGTH = 0
```

```
    segment_x (RWX): ORIGIN = memory address, LENGTH = segment size
     and so on...
}
```

In this pattern:

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

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

The explanation of the `MEMORY` directive, at MEMORY, includes more information about the memory segment.

14.5.2 Closure Segments

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

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

For example, suppose that:

- Symbol `_abc` references symbols `_def` and `_ghi`,
- Symbol `_def` references symbols `_jkl` and `_mno`, and
- Symbol `_ghi` references symbol `_pqr`

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

The three closure-segment directives have specific uses:

- `FORCE_ACTIVE` - Use this directive to make the linker include a symbol that it otherwise would not include.

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

The following listing shows an example of each directive.

Listing: Example Closure Sections

```
# 1st closure segment keeps 3 symbols in link

FORCE_ACTIVE {break_handler, interrupt_handler, my_function}

# 2nd closure segment keeps 2 sections in link

KEEP_SECTION {.interrupt1, .interrupt2}

# 3rd closure segment keeps file-dependent section in link

REF_INCLUDE {.version}
```

14.5.3 Sections Segment

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

Listing: Example Sections Segment

```
SECTIONS {
   .section_name : #The section name, for your reference,

   {               # must begin with a period.

      filename.c  (.text) #Put .text section from filename.c,

      filename2.c (.text) #then put .text section from filename2.c,

      filename.c  (.data) #then put .data section from filename.c,

      filename2.c (.data) #then put .data section from filename2.c,

      filename.c  (.bss)  #then put .bss section from filename.c,

      filename2.c (.bss)  #then put .bss section from filename2.c.

      . = ALIGN (0x10);   #Align next section on 16-byte boundary.

   } > segment_1          #Map these contents to segment_1.

   .next_section_name:

   {

      more content descriptions
```

```
   } > segment_x          #End of .next_section_name definition
}                         #End of sections segment
```

The explanation of the SECTIONS directive, at SECTIONS, includes more information about the sections segment.

14.6 LCF Syntax

This section explains LCF commands, including practical ways to use them. Sub-sections are:

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

14.6.1 Variables, Expressions, and Integrals

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

```
_dec_num = 99999999;

_hex_num_ = 0x9011276;
```

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

```
_symbolicname = some_expression;
```

NOTE

There must be a semicolon at the end of a symbol assignment statement. A symbol assignment is valid only at the start of an expression, so you cannot use something like this:

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

- Absolute expression - the symbol contains the value that it will have in the output file.
- Relocatable expression - the value expression is a fixed offset from the base of a section.

LCF syntax for expressions is very similar to the syntax of the C programming language:

- All integer types are `long` or `unsigned long`.
- Octal integers begin with a leading zero; other digits are 0 through 7, as these symbol assignments show:

```
_octal_number = 01374522;

_octal_number2 = 032405;
```

- Decimal integers begin with any non-zero digit; other digits are 0 through 9, as these symbol assignments show:

```
_dec_num = 99999999;

_decimal_number = 123245;

_decvalfour = 9011276;
```

- Hexadecimal integers begin with a zero and the letter x; other digits are 0 through f, as these symbol assignments show:

```
_hex_number = 0x999999FF;

_firstfactorspace = 0X123245EE;

_fifthhexval = 0xFFEE;
```

- Negative integers begin with a minus sign:

```
_decimal_number = -123456;
```

14.6.2 Arithmetic, Comment Operators

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

Table 14-1. LCF Arithmetic Operators

Precedence	Operators
1	- ~ !

Table continues on the next page...

Table 14-1. LCF Arithmetic Operators (continued)

Precedence	Operators
2	* / %
3	+ -
4	>> <<
5	== != > < <= >=
6	&
7	|
8	&&
9	||

To add comments to your file, use the pound character, C-style slash and asterisk characters, or C++-style double-slash characters, in any of these formats:

```
#  This is a one-line comment

/* This is a

              multiline comment */

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

14.6.3 Alignment

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

Listing: ALIGN Keyword Example

```
file.c (.text)
. = ALIGN (0x10);

file.c (.data)    # aligned on 16-byte boundary.
```

Listing: ALIGNALL Command Example

```
file.c (.text)
ALIGNALL (0x10);  #everything past this point aligned

                  # on 16 byte boundary

file.c (.data)
```

NOTE

If one segment entry imposes an alignment requirement, that segment's starting address must conform to that requirement.

Otherwise, there could be conflicting section alignment in the code the linker produces.

For more alignment information, refer to the topics ALIGN and ALIGNALL.

14.6.4 Specifying Files and Functions

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

Listing: Standard Source-File Specification

```
SECTIONS {
  .example_section :

    {

       main.c  (.text)

       file2.c (.text)

       file3.c (.text)

       # and so forth
```

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

```
*  (.text)
```

in a section definition tells the system to include the `.text` section from each file.

Furthermore the `*` wildcard does not duplicate sections already specified; you need not replace existing lines of the code. In Listing: Standard Source-File Specification, replacing the `# and so forth` comment line with

```
*  (.text)
```

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

Once '*' is used any other use of file (.text) will be ignored because the linker makes a single pass thru the .lcf.

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

Listing: Function Placement Example

```
SECTIONS {
  .program_section :

    {

      OBJECT (beta, main.c)    # Function beta is 1st section item

      OBJECT (alpha, main.c)   # Function alpha is 2nd section_item

      * (.text)  # Remaining_items are .text sections from all files

    } > ROOT
```

NOTE

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

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

For more information about specifying files and functions, see the explanations OBJECT and SECTIONS.

14.6.5 Stack and Heap

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

Listing: Stack Setup Operations

```
_stack_address = __END_BSS;
_stack_address = _stack_address & ~7; /*align top of stack by 8*/

__SP_INIT = _stack_address + 0x4000;  /*set stack to 16KB*/
```

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

Listing: Heap Setup Operations

```
___heap_addr = __SP_INIT; /* heap grows opposite stack */
___heap_size = 0x50000; /* heap size set to 500KB */
```

14.6.6 Static Initializers

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

In your linker command file, use lines similar to the lines listed below, to tell the linker where to put the table of static initializers (relative to the ' .' location counter):

```
___sinit__ = .;

STATICINIT
```

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

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

14.6.7 Exception Tables

You need exception tables only for C++ code. To create one, add the `EXCEPTION` command to the end of your code section - the following listing is an example.

The program identifies the two symbols `__exception_table_start__` and `__exception_table_end__` at runtime.

Listing: Creating an Exception Table

```
__exception_table_start__ = .;
EXCEPTION

__exception_table_end__ = .;
```

14.6.8 Position-Independent Code and Data

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

In addition, your LCF must define these six symbols:

```
__START_PICTABLE __END_PICTABLE__PICTABLE_SIZE__START_PIDTABLE__END_PIDTABLE __PIDTABLE_SIZE
```

The following listing is an example definition for PIC and PID.

Listing: PIC, PID Section Definition

```
.pictables :
{

. = ALIGN(0x8);
```

```
__START_PICTABLE = .;
*(.picdynrel)__END_PICTABLE = .;
__PICTABLE_SIZE = __END_PICTABLE - __START_PICTABLE;
__START_PIDTABLE = .;
*(.piddynrel)__END_PIDTABLE = .;
__PIDTABLE_SIZE = __END_PIDTABLE - __START_PIDTABLE;
} >> DATA
```

14.6.9 ROM-RAM Copying

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

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

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

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

Listing: Partial LCF for ROM-to-RAM Copy

```
# ROM location: address 0x0
# RAM location: address 0x800000
# For clarity, no alignment directives in this listing
MEMORY {
  TEXT (RX) : ORIGIN = 0x0, LENGTH = 0
  DATA (RW) : ORIGIN = 0x800000, LENGTH = 0
}
SECTIONS{
  .main :
  {
    *(.text)
    *(.rodata)
  } > TEXT
```

```
# Locate initialized data in ROM area at end of .main.
  .main_data : AT( ADDR(.main) + SIZEOF(.main) )
  {
    *(.data)
    *(.sdata)
    *(.sbss)
  } > DATA
  .uninitialized_data:
  {
    *(SCOMMON)
    *(.bss)
    *(COMMON)
  } >> DATA
```

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

- ROM start address
- RAM start address
- size

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

Listing: LCF Copy Table for Runtime ROM Copy

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

 .romp : AT (_romp_at)

 {

   __S_romp = _romp_at;

   WRITEW(_main_ROM);          #ROM start address

   WRITEW(ADDR(.main_data));   #RAM start address

   WRITEW(SIZEOF(.main_data)); #size

   WRITEW(0);

   WRITEW(0);

   WRITEW(0);

 }
```

```
 __SP_INIT = . + 0x4000;  # set stack to 16kb
 __heap_addr = __SP_INIT; # heap grows opposite stack direction
 __heap_size = 0x10000;   # set heap to 64kb
}                         # end SECTIONS segment
                          # end LCF
```

14.6.10 Writing Data Directly to Memory

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

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

The system inserts the data at the section's current address. The following listing shows an example.

Listing: Embedding Data Directly into Output

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

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

Listing: Embedding a Binary File into Output

```
    _musicStart = .;
    INCLUDE music.mid

    _musicEnd = .;
}  > DATA
```

14.7 Commands, Directives, and Keywords

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

Table 14-2. LCF Functions, Keywords, Directives, and Commands

. (location counter)	ADDR	ALIGN
ALIGNALL	EXCEPTION	FORCE_ACTIVE
INCLUDE	KEEP_SECTION	MEMORY
OBJECT	REF_INCLUDE	SECTIONS
SIZEOF	SIZEOF_ROM	WRITEB
WRITEH	WRITEW	WRITES0COMMENT
ZERO_FILL_UNINITIALIZED		

14.7.1 . (location counter)

Denotes the current output location.

Remarks

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

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

Example

The code of the following listing moves the location counter to a position 0x10000 bytes past the symbol `__start`.

Listing: Moving the Location Counter

```
..data :
{
     *.(data)
     *.(bss)
     *.(COMMON)
     __start = .;
     . = __start + 0x10000;
     __end = .;
```

```
} > DATA
```

14.7.2 ADDR

Returns the address of the named section or memory segment.

```
ADDR (sectionName | segmentName)
```

Parameters

`sectionName`

Identifier for a file section.

`segmentName`

Identifier for a memory segment

Example

The code of the following listing uses the `ADDR` function to assign the address of `ROOT` to the symbol `__rootbasecode`.

Listing: ADDR() Function

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

}

SECTIONS{

  .code :

  {

  __rootbasecode = ADDR(ROOT);

     *.(text);

  } > ROOT


}
```

14.7.3 ALIGN

Returns the location-counter value, aligned on a specified boundary.

```
ALIGN(alignValue)
```

Parameter

`alignValue`

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

Remarks

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

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

14.7.4 ALIGNALL

Forces minimum alignment of all objects in the current segment to the specified value.

```
ALIGNALL(alignValue);
```

Parameter

`alignValue`

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

Remarks

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

Example

The following listing is an example use for `ALIGNALL()` command.

Listing: ALIGNALL Example

```
.code :
{
    ALIGNALL(16);  // Align code on 16-byte boundary
    *    (.init)
    *    (.text)
    ALIGNALL(64);  //align data on 64-byte boundary
    *    (.rodata)
} > .text
```

14.7.5 EXCEPTION

Creates the exception table index in the output file.

```
EXCEPTION
```

Remarks

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

Example

The following listing shows the code for creating an exception table.

Listing: Creating an Exception Table

```
__exception_table_start__ = .;
 EXCEPTION

 __exception_table_end__ = .;
```

14.7.6 FORCE_ACTIVE

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

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

Parameter

`symbol`

Any defined symbol.

14.7.7 INCLUDE

Includes a specified binary file in the output file.

```
INCLUDE filename
```

Parameter

`filename`

Name of a binary file. The path to the binary file needs to be specified using the `-L` linker option. For example,

```
mwldmcf ... -LC:\path_to_my_include_file.
```

Remarks

For more information and an example, see the subsection Writing Data Directly to Memory.

14.7.8 KEEP_SECTION

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

```
KEEP_SECTION{ sectionType[, sectionType] }
```

Parameter

`sectionType`

Identifier for any user-defined or predefined section.

14.7.9 MEMORY

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

```
MEMORY { memory_spec[, memory_spec] }
```

Parameters

`memory_spec`

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

`segmentName`

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

`accessFlags`

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

`address`

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

`length`

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

`fileName`

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

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

Remarks

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

For each memory segment, the `ORIGIN` keyword introduces the starting address, and the `LENGTH` keyword introduces the length value.

There is no overflow checking for the autolength feature. To prevent overflow, you should use the `AFTER` keyword to specify the segment's starting address.

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

For more information, see the subsection Memory Segment.

Example

The following listing is an example use of the `MEMORY` directive.

Listing: MEMORY Directive Example

```
MEMORY {
    TEXT (RX)   :  ORIGIN = 0x00003000, LENGTH = 0
```

```
    DATA (RW)   :  ORIGIN = AFTER(TEXT), LENGTH = 0
    }
```

14.7.10 OBJECT

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

```
OBJECT (function, filename | *)
```

Parameters

`function`

Represents the mangled function name (for C language, the used mangling is prefixing the function name with one '_' character)

`filename`

Represents the source file or object file name where the function should be taken from. The actual search is performed through the object files (the input of the linker) – the back-mapping from object files to source files is performed by the linker in order to satisfy some constructions in the linker command file.

`*`

Represents all object files, meaning that the function will be searched through all the object files provided to the linker

Remarks

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

14.7.11 REF_INCLUDE

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

```
REF_INCLUDE{ sectionType[, sectionType] }
```

Parameter

`sectionType`

Identifier for any user-defined or predefined section.

Remarks

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

14.7.12 SECTIONS

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

```
SECTIONS { section_spec[, section_spec] }
```

Parameters

`section_spec`

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

`sectionName`

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

`AT (loadAddress)`

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

`contents`

Statements that assign a value to a symbol or specify section placement, including input sections. Subsections Arithmetic, Comment Operators, Specifying Files and Functions, Alignment, and . (location counter) explain possible `contents` statements.

`segmentName`

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

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

Remarks

For more information, see the subsection Sections Segment.

Example

The following listing is an example sections-segment definition.

Listing: SECTIONS Directive Example

```
SECTIONS {
  .text : {

            _textSegmentStart = .;

            alpha.c (.text)

            . = ALIGN (0x10);

            beta.c (.text)

            _textSegmentEnd = .;

  }

  .data : { *(.data) }

  .bss  : { *(.bss)

            *(COMMON)

  }
}
```

14.7.13 SIZEOF

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

```
SIZEOF(segmentName | sectionName)
```

Parameters

`segmentName`

Name of a segment; must start with a period.

`sectionName`

Name of a section; must start with a period.

14.7.14 SIZEOF_ROM

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

```
SIZEOF_ROM (segmentName)
```

Parameter

`segmentName`

Name of a ROM segment; must start with a period.

Remarks

Always returns the value 0 until the ROM is built. Accordingly, you should use `SIZEOF_ROM` only within an expression inside a `WRITEB`, `WRITEH`, `WRITEW`, or `AT` function.

Furthermore, you need `SIZEOF_ROM` only if you use the `COMPRESS` option on the memory segment. Without compression, there is no difference between the return values of `SIZEOF_ROM` and `SIZEOF`.

14.7.15 WRITEB

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

```
WRITEB (expression);
```

Parameter

`expression`

Any expression that returns a value `0x00` to `0xFF`.

14.7.16 WRITEH

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

```
WRITEH (expression);
```

Parameter

`expression`

Any expression that returns a value 0x0000 to 0xFFFF

14.7.17 WRITEW

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

```
WRITEW (expression);
```

Parameter

expression

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

14.7.18 WRITES0COMMENT

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

```
WRITES0COMMENT "comment"
```

Parameter

comment

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

Remarks

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

```
S0aa0000bbbbbbbbbbbbbbbbdd
```

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

This command does not null-terminate the ASCII string.

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

Example

This example shows that multi-line `S0` comments are valid:

```
WRITES0COMMENT "Line 1 comment


Line 2 comment"
```

14.7.19 ZERO_FILL_UNINITIALIZED

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

```
ZERO_FILL_UNINITIALIZED
```

Remarks

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

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

Example

The code of the following listing tells the linker to write uninitialized data to the binary files as zeros.

Listing: ZERO_FILL_UNINITIALIZED Example

```
MEMORY {
  TEXT  (RX) :ORIGIN = 0x00030000, LENGTH = 0

  DATA  (RW) :ORIGIN = AFTER(TEXT), LENGTH = 0

}

ZERO_FILL_UNINITIALIZED
```

```
SECTIONS {
  .main_application:
  {
    *(.text)
    .=ALIGN(0x8);
    *(.rodata)
    .=ALIGN(0x8);
  } > TEXT
...
}
```

Chapter 15
C Compiler

This chapter explains the CodeWarrior implementation of the C programming language. The topics included are as follows:

15.1 Extensions to Standard C

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

15.1.1 Controlling Standard C Conformance

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

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

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

Table 15-1. Controlling conformance to the ISO/IEC 9899-1990 C language

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

15.1.2 C++-style Comments

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

Listing: C++ Comments

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

15.1.3 Unnamed Arguments

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

Listing: Unnamed Arguments

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

15.1.4 Extensions to the Preprocessor

When ANSI strictness is off, the C compiler allows a `#` to prefix an item that is not a macro argument. It also allows an identifier after an `#endif` directive. Listing: Using # in Macro Definitions and Listing: Identifiers After #endif show examples.

Listing: Using # in Macro Definitions

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

#define add2(x) #x "2"

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

Listing: Identifiers After #endif

```
#ifdef __CWCC__
  /* . . . */

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

#ifdef __CWCC__

  /* . . . */

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

15.1.5 Non-Standard Keywords

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

15.2 C99 Extensions

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

- Controlling C99 Extensions
- Trailing Commas in Enumerations
- Compound Literal Values
- Designated Initializers
- Predefined Symbol __func__
- Implicit Return From main()
- Non-constant Static Data Initialization
- Variable Argument Macros
- Extra C99 Keywords
- C++-Style Comments
- C++-Style Digraphs

- Empty Arrays in Structures
- Hexadecimal Floating-Point Constants
- Variable-Length Arrays
- Unsuffixed Decimal Literal Values
- C99 Complex Data Types

15.2.1 Controlling C99 Extensions

The following table shows how to control C99 extensions.

Table 15-2. Controlling C99 extensions to the C language

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

15.2.2 Trailing Commas in Enumerations

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

Listing: Trailing comma in enumeration example

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

15.2.3 Compound Literal Values

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

Listing: Example of a Compound Literal

```
#pragma c99 on
struct my_struct {

  int i;

  char c[2];

} my_var;

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

15.2.4 Designated Initializers

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

Listing: Example of Designated Initializers

```
#pragma c99 on
struct X {

    int a,b,c;

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

union U {

    char a;

    long b;

} u = { .b = 1234567 };

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

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

15.2.5 Predefined Symbol __func__

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

Listing: Predefined symbol __func__

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

15.2.6 Implicit Return From main()

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

```
  return 0;
```

15.2.7 Non-constant Static Data Initialization

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

15.2.8 Variable Argument Macros

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

Listing: Variable argument macros example

```
#define MYLOG(...) fprintf(myfile, __VA_ARGS__)
#define MYVERSION 1

#define MYNAME "SockSorter"

int main(void)

{

    MYLOG("%d %s\n", MYVERSION, MYNAME);
```

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

15.2.9 Extra C99 Keywords

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

Table 15-3. Extra C99 Keywords

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

15.2.10 C++-Style Comments

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

```
//
```

and continues till the end of a source code line.

A C-style comment begins with

```
/*
```

ends with

```
*/
```

and may span more than one line.

15.2.11 C++-Style Digraphs

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

Table 15-4. C++-Style Digraphs

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

15.2.12 Empty Arrays in Structures

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

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

```
struct {
  int r;

  char arr[];

} s;
```

15.2.13 Hexadecimal Floating-Point Constants

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

Examples:

```
0x2f.3a2p3
```

```
0xEp1f
```

```
0x1.8p0L
```

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

15.2.14 Variable-Length Arrays

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

Listing: Example of C99 Variable Length Array usage

```
#pragma c99 on
void f(int n) {

   int arr[n];

   /* ... */

}
```

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

Listing: Bad Example of C99 Variable Length Array usage

```
#pragma c99 on
int n;

int arr[n];

// ERROR: variable length array

// types can only be used in local or
```

```
// function prototype scope.
```

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

Listing: Bad Example of C99 usage in Function Prototype

```
#pragma c99 on
template<typename T> int f(int n, int A[n][n]);

{

};

// ERROR: variable length arrays

// cannot be used in function template prototypes

// or local template variables
```

15.2.15 Unsuffixed Decimal Literal Values

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

Listing: Examples of C99 Unsuffixed Constants

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

sizeof(4294967295u) == sizeof(unsigned long)

#pragma c99 off

sizeof(4294967295)  == sizeof(unsigned long)

sizeof(4294967295u) == sizeof(unsigned long)
```

15.2.16 C99 Complex Data Types

The compiler supports the C99 complex and imaginary data types when the `C99 extensions` option is enabled. Listing: C99 Complex Data Type shows an example.

Listing: C99 Complex Data Type

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

NOTE

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

15.3 GCC Extensions

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

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

15.3.1 Controlling GCC Extensions

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

Table 15-5. Controlling GCC extensions to the C language

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

15.3.2 Initializing Automatic Arrays and Structures

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

Listing: Initializing arrays and structures with non-constant values

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

15.3.3 sizeof() Operator

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

Listing: Using the sizeof() operator with void and function types

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

15.3.4 Statements in Expressions

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

Listing: Using statements and definitions in expressions

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

{

    return POW2(4);

}
```

15.3.5 Redefining Macros

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

Listing: Redefining a macro without undefining first

```
#define SOCK_MAXCOLOR 100
#undef SOCK_MAXCOLOR

#define SOCK_MAXCOLOR 200 /* OK: this macro is previously undefined. */

#define SOCK_MAXCOLOR 300
```

15.3.6 typeof() Operator

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

Listing: Using the typeof() operator

```
int *ip;
/* Variables iptr and jptr have the same type. */

typeof(ip) iptr;

int *jptr;

/* Variables i and j have the same type. */

typeof(*ip) i;

int j;
```

15.3.7 Void and Function Pointer Arithmetic

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

15.3.8 __builtin_constant_p() Operator

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

15.3.9 Forward Declarations of Static Arrays

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

Listing: Forward declaration of an empty array

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

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

15.3.10 Omitted Operands in Conditional Expressions

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

Listing: Using the shorter form of the conditional expression

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

15.3.11 __builtin_expect() Operator

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

This compile-time operator takes two arguments:

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

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

Listing: Example for __builtin_expect() operator

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

```
        }
    }
}
```

15.3.12 Void Return Statements

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

Listing: Returning void

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

15.3.13 Minimum and Maximum Operators

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

Listing: Example of Minimum and Maximum Operators

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

15.3.14 Local Labels

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

Listing: Example of using Local Labels

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

Chapter 16
C++ Compiler

This chapter explains the CodeWarrior implementation of the C++ programming language. The topics included here are as follows:

16.1 C++ Compiler Performance

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

16.1.1 Precompiling C++ Source Code

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

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

16.1.2 Using Instance Manager

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

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

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

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

Table 16-1. Controlling the C++ instance manager

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

16.2 Extensions to Standard C++

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

- __PRETTY_FUNCTION__ Identifier
- Standard and Non-Standard Template Parsing

16.2.1 __PRETTY_FUNCTION__ Identifier

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

16.2.2 Standard and Non-Standard Template Parsing

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

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

Listing: Using typename keyword

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

  T::name *ptr; // ERROR: an attempt to multiply T::name by ptr

  typename T::name *ptr; // OK

}
```

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

Listing: Using template keyword

```
template <typename T> void f(T* ptr)
{

  ptr->f<int>(); // ERROR: f is less than int

  ptr->template f<int>(); // OK

}
```

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

Listing: Binding non-dependent identifiers

```
void f(char);
template <typename T> void tmpl_func()

{
```

```
  f(1); // Uses f(char); f(int), below, is not defined yet.

  g(); // ERROR: g() is not defined yet.

}

void g();

void f(int);
```

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

Listing: Qualifying template arguments in base classes

```
template <typename T> struct Base
{

  void f();

}

template <typename T> struct Derive: Base<T>

{

  void g()

  {

    f(); // ERROR: Base<T>::f() is not visible.

    Base<T>::f(); // OK

  }

}
```

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

Listing: Function call with type-dependent argument

```
void f(char);
template <typename T> void type_dep_func()

{

  f(1); // Uses f(char), above; f(int) is not declared yet.

  f(T()); // f() called with a type-dependent argument.

}

void f(int);

struct A{};

void f(A);
```

```
int main()

{

  type_dep_func<int>(); // Calls f(char) twice.

  type_dep_func<A>(); // Calls f(char) and f(A);

  return 0;

}
```

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

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

```
static void f(int); // f() is internal.
template <typename T> void type_dep_fun_ext()

{

  f(T()); // f() called with a type-dependent argument.

}

int main()

{

  type_dep_fun_ext<int>(); // ERROR: f(int) must be external.

}
```

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

Listing: Assembly statements cannot depend on template arguments

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

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

}
```

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

Listing: Address of Template-id Supported

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

...

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

16.3 Implementation-Defined Behavior

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

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

NOTE

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

Table 16-2. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882-2003 C++, §A)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
Nesting levels of compound statements, iteration control structures, and selection control structures	256	Unlimited
Nesting levels of conditional inclusion	256	256
Pointer, array, and function declarators (in any combination) modifying an arithmetic, structure, union, or incomplete type in a declaration	256	Unlimited
Nesting levels of parenthesized expressions within a full expression	256	Unlimited
Number of initial characters in an internal identifier or macro name	1024	Unlimited
Number of initial characters in an external identifier	1024	Unlimited
External identifiers in one translation unit	65536	Unlimited
Identifiers with block scope declared in one block	1024	Unlimited
Macro identifiers simultaneously defined in one translation unit	65536	Unlimited
Parameters in one function definition	256	Unlimited
Arguments in one function call	256	Unlimited
Parameters in one macro definition	256	256
Arguments in one macro invocation	256	256
Characters in one logical source line	65536	Unlimited
Characters in a character string literal or wide string literal (after concatenation)	65536	Unlimited
Size of an object	262144	2 GB
Nesting levels for `# include` files	256	256

Table continues on the next page...

Table 16-2. Implementation Quantities for C/C++ Compiler (ISO/IEC 14882-2003 C++, §A) (continued)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
`Case` labels for a `switch` statement (excluding those for any nested `switch` statements)	16384	Unlimited
Data members in a single class, structure, or union	16384	Unlimited
Enumeration constants in a single enumeration	4096	Unlimited
Levels of nested class, structure, or union definitions in a single struct-declaration-list	256	Unlimited
Functions registered by `atexit()`	32	64
Direct and indirect base classes	16384	Unlimited
Direct base classes for a single class	1024	Unlimited
Members declared in a single class	4096	Unlimited
Final overriding virtual functions in a class, accessible or not	16384	Unlimited
Direct and indirect virtual bases of a class	1024	Unlimited
Static members of a class	1024	Unlimited
Friend declarations in a class	4096	Unlimited
Access control declarations in a class	4096	Unlimited
Member initializers in a constructor definition	6144	Unlimited
Scope qualifications of one identifier	256	Unlimited
Nested external specifications	1024	Unlimited
Template arguments in a template declaration	1024	Unlimited
Recursively nested template instantiations	17	64 (adjustable upto 30000 using #pragma template_depth(<n>))
Handlers per try block	256	Unlimited
Throw specifications on a single function declaration	256	Unlimited

16.4 GCC Extensions

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

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

Listing: Using the :: operator in class declarations

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

};
```

Chapter 17
Precompiling

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

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

This section shows you how to use and create precompiled files. The topics listed here are as follows:

17.1 What can be Precompiled

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

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

  You cannot precompile libraries or other binary files.

- Precompiled files must have a `.mch` filename extension.

- The file must not contain any statements that generate data or executable code.

  However, the file may define static data.

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

17.2 Using a Precompiled File

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

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

The following listing shows an example.

Listing: Using a precompiled file

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

#include "wool.h /* Regular header file. */

/* ... */
```

17.3 Creating Precompiled File

This section shows how to create and manage precompiled files:

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

17.3.1 Precompiling File on Command Line

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

1. Start a command line shell.

2. Issue this command

   mwcc h_file.pch -precompile *p_file*.mch

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

The file is precompiled.

17.3.2 Updating Precompiled File Automatically

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

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

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

17.3.3 Preprocessor Scope in Precompiled Files

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

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

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

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

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

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

Listing: Pragma Settings in 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 the following listing includes the precompiled version of the listing displayed above.

Listing: Pragma Settings in an Included Precompiled File

```
/* test.c */
/* Far data is disabled. */

#pragma far_data off

/* This precompiled file sets far_data on. */

#include "my_pch.mch"

/* far_data is still off but xxx is still a far variable. */
```

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

Chapter 18
Addressing

The compiler offers different alternatives to data and code addressing. Since these models are generic to the compilation unit a set of pragmas and declaration specifiers are made available to specialize addressing.

This chapter explains following topics:

18.1 Data

The near data model uses 16-bits absolute addressing for `.data` and `.bss`. The far data model uses 32-bits absolute addressing for `.data` and `.bss`.

It it possible to define a memory region of global data that uses relative addressing (`.sdata/.sbss`), together these regions have a maximum size of 64K and uses A5 as reference. Using A5-relative data reduces code size.

The compiler, using the .sdata threshold size, will assign initialized data to `.sdata` sections and uninitialized (or set to zero) data to .sbss sections. Larger data objects will get assigned to `.data` or `.bss`.

It is possible to override the `.sdata` thereshold for a variable by using the far or near declaration specifier. Note that these near and far specifiers apply to declarations since they affect relocations.

The LCF needs to define the A5-relative base `__SDA_BASE` between `.sdata` and `.sbss` for linktime relocations.

Listing: LCF Example

```
/* visible declarations, typically in a .h include file */
extern far  type varA; /* all references to varA will be 32-bits
absolute */

extern near type varB; /* all references to varB will be 16-bits
absolute */

extern type varC; /* all references to varC will use the current
addressing setting */

/* definitions */

far   type varA; /* varA will be in a .bss section */

near  type varB; /* varB will be in a .bss section */

      type varC; /* varC will use the current segment setting */

const type varD; /* varD will be in a .rodata section */

/* LCF example */

.data :

      {

      ___DATA_RAM    = . ;

      *(.exception)

      .              = ALIGN(0x10);

      __exception_table_start__ = .;

      EXCEPTION

      __exception_table_end__   = .;

      ___sinit__    = .;

      STATICINIT

      *(.data)

      .             = ALIGN (0x10);

      ___DATA_END   = .;

      __START_SDATA = .;

      *(.sdata)

      .             = ALIGN (0x10);

      __END_SDATA   = .;

      __SDA_BASE    = .;

      .             = ALIGN(0x10);

      } >> sram

.bss :

      {

      . = ALIGN(0x10);
```

```
    __START_SBSS = .;
    *(.sbss)
    *(SCOMMON)
   __END_SBSS   = .;
  . = ALIGN(0x10);
  __START_BSS  = .;
  *(.bss)
  *(COMMON)
  __END_BSS    = .;
  ___BSS_START = __START_SBSS;
  ___BSS_END   = __END_BSS;
  . = ALIGN(0x10);
  } >> sram
```

The global data model can also be overriden using local pragmas, these pragmas affect addressing and in some cases section assignment.

This section contains the following topics:

- Addressing and Section Assignment
- Addressing Only

18.1.1 Addressing and Section Assignment

The following pragmas are used for addressing and section assignment:

- near_data
- far_data
- pcrelstrings

18.1.1.1 near_data

Syntax

```
#pragma near_data on | off | reset
```

Description

This pragma makes the subsequent data declarations near absolute, if the declaration is also a definition the variable gets assigned to a `.data` and `.bss` section regardless of the `.sdata` threshold.

18.1.1.2 far_data

Syntax

```
#pragma far_data on | off | reset
```

Description

This pragma makes the subsequent data declarations far absolute, if the declaration is also a definition the variable gets assigned to a . `data` and `.bss` section regardless of the `.sdata` threshold.

18.1.1.3 pcrelstrings

Syntax

```
#pragma pcrelstrings on | off | reset
```

Description

This pragma merges the strings defined in a function to its `.text` section, accesses will be pc-relative.

18.1.2 Addressing Only

The following pragmas are for addressing only:

- pcrelconstdata
- pcreldata

18.1.2.1 pcrelconstdata

Syntax

```
#pragma pcrelconstdata on | off | reset
```

Description

This pragma uses the pc-relative addressing for `constdata` (`.rodata`).

18.1.2.2 pcreldata

Syntax

```
#pragma pcreldata on | off | reset
```

Description

This pragma uses the pc-relative addressing for data (`.data` and `.bss`).

18.2 Code

The compiler basic code models are near/far absolute. The *smartCode* model uses PC relative addressing if the target function is defined in the same compilation unit. The last mode is *nearRelCode*, it makes all junction calls PC-relative.

The global code model can also be overriden using declaration specifiers or local pragmas, these pragmas affect addressing. These pragmas are in effect until another selection is made or the end of the compilation unit is reached.

The pragmas listed here for code addressing are as follows:

- near_code
- near_rel_code
- smart_code
- far_code

18.2.1 near_code

Syntax

```
#pragma near_code
```

Remarks

This pragma uses the near absolute addressing for subsequent function calls.

18.2.2 near_rel_code

Syntax

```
#pragma near_rel_code
```

Remarks

This pragma uses the pc-relative addressing for subsequent function calls.

18.2.3 smart_code

Syntax

```
#pragma smart_code
```

Remarks

This pragma uses the pc-relative addressing if the function is defined in the same compilation unit.

18.2.4 far_code

Syntax

```
#pragma far_code
```

Remarks

This pragma uses the far absolute addressing for subsequent function calls.

18.3 Sections

You can also add user sections, the recommended syntax to assign objects to a user defined section is `__declspec(sectname)`.

Listing: Syntax __declspec(sectname)

```
void __declspec(bootsection) _bootoperation ( void );
#pragma define_section sectname [objecttype] istring [ustring]
[addrmode] [accmode] [align]
```

where,

`<objecttype>` binds sectname to object type

`code_type` executable object code

`data_type` non-constant data size > small data threshold

`sdata_type` non-constant data size <= small data threshold

`const_type` constant data

`string_type` string data

`all_types` all code and data (default)

`<sectname>` identifer by which this user-defined section is referenced in the source, i.e. via `#pragma section <sectname> begin` or `__declspec(<sectname>)`

`<istring>` section name string for -initialized- data assigned to `<section>` e.g. `.data` (also applies to uninitialized data if `<ustring>` is omitted)

`<ustring>` elf section name for -uninitialized- data assigned to `<section>`

`<addrmode>` one of the following, which indicates how the section is addressed:

`standard` target-specific standard method

`near_absolute` 16-bit absolute address

`far_absolute` 32-bit absolute address

`near_code` 16-bit offset from target-specific code base register

`far_code` 32-bit offset from target-specific code base register

`near_data` 16-bit offset from target-specific data base register

`far_data` 32-bit offset from target-specific data base register

`<accmode>` one of the following, which indicates the attributes of the section:

`R` readable

`RW` readable and writable

`RX` readable and executable

`RWX` readable, writable and executable

`<align>` section alignment, default is 0 => (optimize for speed || aligncode) ? 4 : 2

The ELF sections have default configurations, these can be modified by redefining them. Compiler generated sections:

```
"text",sect_text, ".text", 0, "standard", "RX", 0

"data",sect_data, ".data", ".bss", "standard", "RW", 0

"sdata",sect_sdata, ".sdata", ".sbss", "near_data", "RW", 0

"const",sect_const, ".rodata", 0, "standard", "R", 0

"string",sect_string, ".rodata", 0, "standard", "R", 0

"abs",sect_abs, ".abs", 0, "far_absolute", "R", 0
```

This section contains the following topics:

18.3.1 xMAP

The map file lists the different segments with their defines and entry points. It provides at the end a summary of the program segments.

18.3.2 __declspec(bare)

This declaration specifier turns off the prefixing of the linkname with an _ (underscore).

18.3.3 Bitfield Ordering

The default bitfield ordering is big endian, meaning the first bits allocated take the highest bits as seen for a value held in a register. Pragma `reverse_bitfields` changes this allocation to the lower bits. When binary compatibility is required it is useful to specify an invariant declaration. Using an `__attribute__` in the struct/class declaration serves that purpose.

Listing: Example - bitfield ordering

```
 typedef struct {
          unsigned long a:1;

          unsigned long b:3;

  } __attribute__((bitfields(little_endian)))   bf_le;

  typedef struct {

          unsigned long a:1;

          unsigned long b:3;

 } __attribute__((bitfields(big_endian)))   bf_be;

 bf_le le = { 1, 4 };

 bf_be be = { 1, 4 };

struct __attribute__((bitfields(big_endian)))

{

        unsigned long a:1;

        unsigned long b:3;

} bebf = { 1, 4 };

struct __attribute__((bitfields(little_endian)))

{

        unsigned long a:1;

        unsigned long b:3;

} lebf = { 1, 4 };
```

18.4 Far and Near Addressing

The compiler reserves the A5 address register to contain the base address of small static data at runtime. By default the compiler generates instructions that use addressing modes that are relative to the A5 register when referring to data in the `sbss` and `sdata` sections.

Use non-standard keywords to specify absolute addressing instead of A5-relative addressing. To specify that executable code should use absolute far addressing modes to refer to a variable, declare the variable with the `far` or `__far` keyword preceding its declaration. To specify that executable code should use near addressing modes to refer to a variable, declare the variable with `near` or `__near` preceding its declaration. The following listing shows an example.

Listing: Specifying addressing modes

```
static int i;
far static char c;

near static short j;

void f(void)

{

  i = 10; /* A5-relative addressing */

  c = '!'; /* 32-bit absolute addressing */

  j = 5; /* 16-bit absolute addressing */

}
```

Chapter 19
Intermediate Optimizations

After it translates a program's source code into its intermediate representation, the compiler optionally applies optimizations that reduce the program's size, improve its execution speed, or both. The topics in this chapter explain these optimizations and how to apply them:

19.1 Interprocedural Analysis

Most compiler optimizations are applied only within a function. The compiler analyzes a function's flow of execution and how the function uses variables. It uses this information to find shortcuts in execution and reduce the number of registers and memory that the function uses. These optimizations are useful and effective but are limited to the scope of a function.

The CodeWarrior compiler has a special optimization that it applies at a greater scope. Widening the scope of an optimization offers the potential to greatly improve performance and reduce memory use. *Interprocedural analysis* examines the flow of execution and data within entire files and programs to improve performance and reduce size.

19.1.1 Invoking Interprocedural Analysis

The following table explains how to control interprocedural analysis.

Table 19-1. Controlling interprocedural analysis

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose an item in the **IPA** option of the **C/C++ Language Settings** settings panel.
source code	`#pragma ipa program file | on | function | off`
command line	`-ipa file | program | program-final | function | off`

19.1.2 Function-Level Optimization

Interprocedural analysis may be disabled by setting it to either `off` or `function`. If IPA is disabled, the compiler generates instructions and data as it reads and analyzes each function. This setting is equivalent to the "no deferred codegen" mode of older compilers.

19.1.3 File-Level Optimization

When interprocedural analysis is set to optimize at the file level, the compiler reads and analyzes an entire file before generating instructions and data.

At this level, the compiler generates more efficient code for inline function calls and C++ exception handling than when interprocedural analysis is off. The compiler is also able to increase character string reuse and pooling, reducing the size of object code. This is equivalent to the " deferred inlining" and "deferred codegen" options of older compilers.

The compiler also safely removes static functions and variables that are not referred to within the file, which reduces the amount of object code that the linker must process, resulting in better linker performance.

19.1.4 Program-Level Optimization

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

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

19.1.5 Program-Level Requirements

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

- Dependencies Among Source Files
- Function and Top-level Variable Declarations
- Type Definitions
- Unnamed Structures and Enumerations in C

19.1.5.1 Dependencies Among Source Files

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

19.1.5.2 Function and Top-level Variable Declarations

Because the compiler treats all files that compose a program as if they were a single, large source file, you must make sure all non-static declarations for variables or functions with the same name are identical. See the following listing for an example of declarations that prevent the compiler from applying program-level analysis.

Listing: Declaration conflicts in program-level interprocedural analysis

```
/* file1.c */
```

```
extern int i;

extern int f();

int main(void)

{

   return i + f();

}

/* file2.c */

short i;       /* Conflict with variable i in file1.c. */

extern void f(); /* Conflict with function f() in file1.c */
```

The following listing fixes this problem by renaming the conflicting symbols.

Listing: Fixing declaration conflicts for program-level interprocedural analysis

```
/* file1.c */
extern int i1;

extern int f1();

int main(void)

{

   return i1 + f1();

}

/* file2.c */

short i2;

extern void f2();
```

19.1.5.3 Type Definitions

Because the compiler examines all source files for a program, make sure all definitions for a type are the same. See the following listing for an example of conflicting type definitions:

Listing: Type definitions conflicts in program-level interprocedural analysis

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

a_rec a;

/* fileB.c */

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

a_rec b;
```

The following listing shows a suggested solution for C:

Listing: Fixing type definitions conflicts in C

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

a1_rec a;

/* fileB.c */

struct a2_rec { char c; };

a2_rec b;
```

The following listing shows a suggested solution for C++:

Listing: Fixing type definitions conflicts in C++

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

a_rec a;

/* fileB.c */

namespace { struct a_rec { char c; }; }

a_rec b;
```

19.1.5.4 Unnamed Structures and Enumerations in C

The C language allows anonymous `struct` and `enum` definitions in type definitions. Using such definitions prevents the compiler from properly applying program-level interprocedural analysis. Make sure to give names to structures and enumerations in type definitions. The following listing shows an example of unnamed structures and enumerations.

Listing: Unnamed structures and enumerations in C

```
/* In C, the types x_rec and y_enum each represent a structure and an
enumeration with no name.
   In C++ these same statements define a type x_rec and y_enum,

   a structure named x_rec and an enumeration named y_enum.

*/

typedef struct { int a, b, c; } x_rec;

typedef enum { Y_FIRST, Y_SECOND, Y_THIRD } y_enum;
```

The following listing shows a suggested solution.

Listing: Naming structures and enumerations in C

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

19.2 Intermediate Optimizations

After it translates a function into its intermediate representation, the compiler may optionally apply some optimizations. The result of these optimizations on the intermediate representation will either reduce the size of the executable code, improve the executable code's execution speed, or both.

- Dead Code Elimination
- Expression Simplification
- Common Subexpression Elimination
- Copy Propagation
- Dead Store Elimination
- Live Range Splitting
- Loop-Invariant Code Motion
- Strength Reduction
- Loop Unrolling

19.2.1 Dead Code Elimination

The dead code elimination optimization removes expressions that are not accessible or are not referred to. This optimization reduces size and increases execution speed.

The following table explains how to control the optimization for dead code elimination.

Table 19-2. Controlling dead code elimination

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 1**, **Level 2**, **Level 3**, or **Level 4** in the **OptimizationsLevel** settings panel.
source code	`#pragma opt_dead_code on | off | reset`
command line	`-opt [no]deadcode`

In the following listing, the call to `func1()` will never execute because the `if` statement that it is associated with will never be true.

Listing: Before dead code elimination

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

Consequently, the compiler can safely eliminate the call to `func1()`, as shown in the following listing.

Listing: After dead code elimination

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

19.2.2 Expression Simplification

The expression simplification optimization attempts to replace arithmetic expressions with simpler expressions. Additionally, the compiler also looks for operations in expressions that can be avoided completely without affecting the final outcome of the expression. This optimization reduces size and increases speed.

The following table explains how to control the optimization for expression simplification.

Table 19-3. Controlling expression simplification

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 1**, **Level 2**, **Level 3**, or **Level 4** in the **Optimization Level** settings panel.
source code	There is no pragma to control this optimization.
command line	`-opt level=1`, `-opt level=2`, `-opt level=3`, `-opt level=4`

For example, the following listing contains a few assignments to some arithmetic expressions:

- addition to zero
- multiplication by a power of 2

- subtraction of a value from itself
- arithmetic expression with two or more literal values

Listing: Before expression simplification

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

    *result1 = x + 0;

    *result2 = x * 2;

    *result3 = x - x;

    *result4 = 1 + x + 4;

}
```

The following listing shows source code that is equivalent to expression simplification. The compiler has modified these assignments to:

- remove the addition to zero
- replace the multiplication of a power of 2 with bit-shift operation
- replace a subtraction of x from itself with 0
- consolidate the additions of 1 and 4 into 5

Listing: After expression simplification

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

    *result1 = x;

    *result2 = x << 1;

    *result3 = 0;

    *result4 = 5 + x;

}
```

19.2.3 Common Subexpression Elimination

Common subexpression elimination replaces multiple instances of the same expression with a single instance. This optimization reduces size and increases execution speed.

The following table explains how to control the optimization for common subexpression elimination.

Table 19-4. Controlling common subexpression elimination

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 2**, **Level 3**, or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_common_subs on | off | reset`
command line	`-opt [no]cse`

For example, in the following listing, the subexpression `x * y` occurs twice.

Listing: Before common subexepression elimination

```
void func_from(int* vec, int size, int x, int y, int value)
{

    if (x * y < size)

    {

        vec[x * y - 1] = value;

    }

}
```

The following listing shows equivalent source code after the compiler applies common subexpression elimination. The compiler generates instructions to compute `x * y` and stores it in a hidden, temporary variable. The compiler then replaces each instance of the subexpression with this variable.

Listing: After common subexpression elimination

```
void func_to(int* vec, int size, int x, int y, int value)
{

    int temp = x * y;

    if (temp < size)

    {

        vec[temp - 1] = value;

    }

}
```

19.2.4 Copy Propagation

Copy propagation replaces variables with their original values if the variables do not change. This optimization reduces runtime stack size and improves execution speed.

The following table explains how to control the optimization for copy propagation.

Table 19-5. Controlling copy propagation

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 2**, **Level 3**, or **Level 4** in the **Global Optimizations** settings panel.
source code	`#pragma opt_propagation on | off | reset`
command line	`-opt [no]prop[agation]`

For example, in the following listing, the variable `j` is assigned the value of `x`.

Listing: Before copy propagation

```
void func_from(int* a, int x)
{

    int i;

    int j;

    j = x;

    for (i = 0; i < j; i++)

    {

        a[i] = j;

    }

}
```

But `j`'s value is never changed, so the compiler replaces later instances of `j` with `x`, as shown in the following listing.

Listing: After copy propagation

```
void func_to(int* a, int x)
{

    int i;

    int j;

    j = x;

    for (i = 0; i < x; i++)

    {

        a[i] = x;

    }

}
```

By propagating `x`, the compiler is able to reduce the number of registers it uses to hold variable values, allowing more variables to be stored in registers instead of slower memory. Also, this optimization reduces the amount of stack memory used during function calls.

19.2.5 Dead Store Elimination

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

The following table explains how to control the optimization for dead store elimination.

Table 19-6. Controlling dead store elimination

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_dead_assignments on | off | reset`
command line	`-opt [no]deadstore`

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

Listing: Before dead store elimination

```
void func_from(int x, int y)
{

    x = y * y;

    otherfunc1(y);

    x = getresult();

    otherfunc2(y);

}
```

In the following listing the compiler can safely remove the first assignment to `x` since the result of this assignment is never used.

Listing: After dead store elimination

```
void func_to(int x, int y)
{

    otherfunc1(y);

    x = getresult();
```

```
    otherfunc2(y);
}
```

19.2.6 Live Range Splitting

Live range splitting attempts to reduce the number of variables used in a function. This optimization reduces a function's runtime stack size, requiring fewer instructions to invoke the function. This optimization potentially improves execution speed.

The following table explains how to control the optimization for live range splitting.

Table 19-7. Controlling live range splitting

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	There is no pragma to control this optimization.
command line	`-opt level=3`, `-opt level=4`

For example, in the following listing three variables, `a`, `b`, and `c`, are defined. Although each variable is eventually used, each of their uses is exclusive to the others. In other words, `a` is not referred to in the same expressions as `b` or `c`, `b` is not referred to with `a` or `c`, and `c` is not used with `a` or `b`.

Listing: Before live range splitting

```
void func_from(int x, int y)
{
    int a;
    int b;
    int c;
    a = x * y;
    otherfunc(a);
    b = x + y;
    otherfunc(b);
    c = x - y;
    otherfunc(c);
}
```

In the following listing, the compiler has replaced `a`, `b`, and `c`, with a single variable. This optimization reduces the number of registers that the object code uses to store variables, allowing more variables to be stored in registers instead of slower memory. This optimization also reduces a function's stack memory.

Listing: After live range splitting

```
void func_to(int x, int y)
{

    int a_b_or_c;

    a_b_or_c = x * y;

    otherfunc(temp);

    a_b_or_c = x + y;

    otherfunc(temp);

    a_b_or_c = x - y;

    otherfunc(temp);

}
```

19.2.7 Loop-Invariant Code Motion

Loop-invariant code motion moves expressions out of a loop if the expressions are not affected by the loop or the loop does not affect the expression. This optimization improves execution speed.

The following table explains how to control the optimization for loop-invariant code motion.

Table 19-8. Controlling loop-invariant code motion

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_loop_invariants on | off | reset`
command line	`-opt [no]loop[invariants]`

For example, in the following listing, the assignment to the variable `circ` does not refer to the counter variable of the `for` loop, `i`. But the assignment to `circ` will be executed at each loop iteration.

Listing: Before loop-invariant code motion

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

The following listing shows source code that is equivalent to how the compiler would rearrange instructions after applying this optimization. The compiler has moved the assignment to `circ` outside the `for` loop so that it is only executed once instead of each time the `for` loop iterates.

Listing: After loop-invariant code motion

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

19.2.8 Strength Reduction

Strength reduction attempts to replace slower multiplication operations with faster addition operations. This optimization improves execution speed but increases code size.

The following table explains how to control the optimization for strength reduction.

Table 19-9. Controlling strength reduction

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_strength_reduction on | off | reset`
command line	`-opt [no]strength`

For example, in the following listing, the assignment to elements of the `vec` array use a multiplication operation that refers to the `for` loop's counter variable, `i`.

Listing: Before strength reduction

```
void func_from(int* vec, int max, int fac)
{

    int i;

    for (i = 0; i < max; ++i)

    {

        vec[i] = fac * i;

    }

}
```

In the following listing, the compiler has replaced the multiplication operation with a hidden variable that is increased by an equivalent addition operation. Processors execute addition operations faster than multiplication operations.

Listing: After strength reduction

```
void func_to(int* vec, int max, int fac)
{

    int i;

    int strength_red;

    hidden_strength_red = 0;

    for (i = 0; i < max; ++i)

    {

        vec[i] = hidden_strength_red;

        hidden_strength_red = hidden_strength_red + i;

    }

}
```

19.2.9 Loop Unrolling

Loop unrolling inserts extra copies of a loop's body in a loop to reduce processor time executing a loop's overhead instructions for each iteration of the loop body. In other words, this optimization attempts to reduce the ratio of the time that the processor takes to execute a loop's completion test and the branching instructions, compared to the time the processor takes to execute the loop's body. This optimization improves execution speed but increases code size.

The following table explains how to control the optimization for loop unrolling.

Table 19-10. Controlling loop unrolling

Turn control this option from here...	use this setting
CodeWarrior IDE	Choose **Level 3** or **Level 4** in the **Optimization Level** settings panel.
source code	`#pragma opt_unroll_loops on | off | reset`
command line	`-opt level=3`, `-opt level=4`

For example, in the following listing, the `for` loop's body is a single call to a function, `otherfunc()`. For each time the loop's completion test executes

```
  for (i = 0; i < MAX; ++i)
```

the function executes the loop body only once.

Listing: Before loop unrolling

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

{

    int i;

    for (i = 0; i < MAX; ++i)

    {

        otherfunc(vec[i]);

    }

}
```

In the following listing, the compiler has inserted another copy of the loop body and rearranged the loop to ensure that variable `i` is incremented properly. With this arrangement, the loop's completion test executes once for every 2 times that the loop body executes.

Listing: After loop unrolling

```
const int MAX = 100;
void func_to(int* vec)

{

    int i;

    for (i = 0; i < MAX;)

    {

        otherfunc(vec[i]);

        ++i;

        otherfunc(vec[i]);

        ++i;

    }

}
```

19.3 Inlining

Inlining replaces instructions that call a function and return from it with the actual instructions of the function being called. Inlining function makes your program faster because it executes the function code immediately without the overhead of a function call and return. However, inlining can also make your program larger because the compiler may insert the function's instructions many times throughout your program.

The rest of this section explains how to specify which functions to inline and how the compiler performs the inlining:

- Choosing Which Functions to Inline
- Inlining Techniques

19.3.1 Choosing Which Functions to Inline

The compiler offers several methods to specify which functions are eligible for inlining.

To specify that a function is eligible to be inlined, precede its definition with the `inline`, `__inline__`, or `__inline` keyword. To allow these keywords in C source code, turn off **ANSI Keywords Only** in the CodeWarrior IDE's **C/C++ Language** settings panel or turn off the `only_std_keywords` pragma in your source code.

To verify that an eligible function has been inlined or not, use the **Non-Inlined Functions** option in the IDE's **C/C++ Warnings** panel or the `warn_notinlined` pragma. The following listing shows an example:

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

```
#pragma only_std_keywords off
inline int attempt_to_inline(void)

{

    return 10;

}
```

To specify that a function must never be inlined, follow its definition's specifier with `__attribute__((never_inline))`. The following listing shows an example.

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

```
int never_inline(void) __attribute__((never_inline))
{

   return 20;

}
```

To specify that no functions in a file may be inlined, including those that are defined with the `inline`, `__inline__`, or `__inline` keywords, use the `dont_inline` pragma. The following listing shows an example:

Listing: Specifying that no functions may be inlined

```
#pragma dont_inline on
/* Will not be inlined. */

inline int attempt_to_inline(void)

{

    return 10;

}

/* Will not be inlined. */

int never_inline(void) __attribute__((never_inline))

{

   return 20;

}

#pragma dont_inline off

/* Will be inlined, if possible. */

inline int also_attempt_to_inline(void)
```

```
{
    return 10;
}
```

The kind of functions that are never inlined are as follows:

- functions with variable argument lists
- functions defined with `__attribute__((never_inline))`
- functions compiled with `#pragma optimize_for_size on` or the **Optimize For Size** setting in the IDE's **Optimization Level** panel
- functions which have their addresses stored in variables

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

The following functions also should never be inlined:

- functions that return class objects that need destruction
- functions with class arguments that need destruction

It can inline such functions if the class has a trivial empty constructor as in this case:

```
struct X {
    int n;
    X(int a) { n = a; }
    ~X() {}
};
inline X f(X x) { return X(x.n + 1); }
int main()
{
    return f(X(1)).n;
}
```

This does not depend on "ISO C++ templates".

19.3.2 Inlining Techniques

The depth of inlining explains how many levels of function calls the compiler will inline. The **Inlining** setting in the IDE's **C/C++ Build > Settings > ColdFire Compiler > Optimization** panel and the `inline_depth` pragma control inlining depth.

Normally, the compiler only inlines an eligible function if it has already translated the function's definition. In other words, if an eligible function has not yet been compiled, the compiler has no object code to insert. To overcome this limitation, the compiler can perform interprocedural analysis (IPA) either in file or program mode. This lets the

compiler evaluate all the functions in a file or even the entire program before inlining the code. The **IPA** setting in the IDE's **C/C++ Build > Settings > ColdFire Compiler Language** settings panel and the `ipa` pragma control this capability.

The compiler normally inlines functions from the first function in a chain of function calls to the last function called. Alternately, the compiler may inline functions from the last function called to the first function in a chain of function calls. The **Bottom-up Inlining** option in the IDE's **C/C++ Build > Settings > ColdFire Compiler Language** settings panel and the `inline_bottom_up` and `inline_bottom_up_once` pragmas control this reverse method of inlining.

Some functions that have not been defined with the `inline`, `__inline__`, or `__inline` keywords may still be good candidates to be inlined. Automatic inlining allows the compiler to inline these functions in addition to the functions that you explicitly specify as eligible for inlining. The **Auto-Inline** option in the IDE's **C/C++ Build > Settings > ColdFire Compiler > Optimization** panel and the `auto_inline` pragma control this capability.

When inlining, the compiler calculates the complexity of a function by counting the number of statements, operands, and operations in a function to determine whether or not to inline an eligible function. The compiler does not inline functions that exceed a maximum complexity. The compiler uses three settings to control the extent of inlined functions:

- maximum auto-inlining complexity: the threshold for which a function may be auto-inlined
- maximum complexity: the threshold for which any eligible function may be inlined
- maximum total complexity: the threshold for all inlining in a function

The `inline_max_auto_size`, `inline_max_size`, and `inline_max_total_size` pragmas control these thresholds, respectively.

Chapter 20
ColdFire Runtime Libraries

The CodeWarrior tool chain includes libraries conforming to ISO/IEC-standards for C and C++ runtime libraries, and other code. CodeWarrior tools come with prebuilt configurations of these libraries.

This chapter explains how to use prebuilt libraries. This chapter consists of these sections:

20.1 EWL for C and C++ Development

EWL introduces a new library set aiming at reducing the memory footprint taken by IO operations and introduces a simpler memory allocator.

The IO operations are divided in three categories:

- Printing
- Scanning
- File operations

The printing and scanning formatters for EWL are grouped in an effort to provide only the support required for the application:

`int` - integer and string processing

`int_FP` - integer, string and floating point

`int_LL` - integer (including long long) and string

`int_FP_LL` - all but wide chars

`c9x` - all including wide char

The buffered IO can be replaced by raw IO, this works solely when printf and scanf are used to perform IO, all buffering is bypassed and writing direct to the device is used. EWL libraries contain prebuilt versions for all formatters and IO modes. Selecting a model combination enables correct compiling and linking. The EWL layout for ColdFire is built per core architecture. It is composed of:

`libm.a` - math support (c9x or not)

`libc.a` - non c9x std C libs

`libc99.a` - c9x libs

`librt.a` - runtime libraries

`libc++.a` - non-c9x matching c++ libraries

`libstdc++.a` - c9x/c++ compliant libs

`fp_coldfire.a` - FPU emulation libraries

Selecting an EWL model for the libraries frees the user from adding libraries to the project, the linker will determine from the settings the correct library set, these settings are : `processor, pid/pic, hard/soft FPU`. Although the library names are known to the toolset, but their location is not. The environment variable MWLibraries can specify the path to the lib hirerarchy root, otherwise the -L linker option needs to point to the hierarchy

```
export MWLibraries=d:/CodeWarrior/ColdFire_Support/ewl
mwldmcf ... -Ld:\CodeWarrior\ColdFire_Support\ewl
```

Note that, when using EWL and not using `-nostdlib`, MWLibraryFiles must not be defined since it conflicts with the known library names.

NOTE

For more information on the Embedded Warrior Library (EWL) for C or C++, see the following manuals: CodeWarrior Development Tools EWL C Reference and CodeWarrior Development Tools EWL C++ Reference.

This section consists of the following topics:

- How to rebuild the EWL Libraries
- Volatile Memory Locations
- Predefined Symbols and Processor Families
- UART Libraries
- Memory, Heaps, and Other Libraries

20.1.1 How to rebuild the EWL Libraries

The files (viz. `libm.a, libc.a.` ...etc.) are present in the ewl\lib folder under a sub-folder v<x>, where <x> is a number indicating the coldfire architecture. If the architecture supports `PIC/PID` then the library files for them will be present in a sub-folder (called pi) for that architecture.

The process to rebuild the EWL library files is given below.

NOTE

The user should have access to a "make" utility within DOS.

1. Open a DOS command prompt.
2. Define the CWINSTALL environment variable ...

   For example, if the you CWINSTALL is in the folder `C:\Freescale\CW MCU v10.x` then you can define `CWINSTALL` as follows:

   set `CWINSTALL='C:/Program Files/Freescale/CW MCU V10.x/MCU'`

   NOTE

   The single quote character (') is important because there are spaces in the path.

3. Change your working directory to the " `ewl`" folder, for example,

   cd `C:\Freescale\CW MCU v10.x\MCU\ColdFire_Support\ewl`

4. To clean the existing library files ... `<path_to_make_utility>\make -f makefile clean`

   You could skip the < `path_to_make_utility`> if you have " `make`" in your PATH variable.

5. All the library files can be re-built using the following command:

   make `-s -f makefile PLATFORM=CF all cleanobj`

   The "PLATFORM=CF" indicates that EWL is being re-built for ColdFire.

6. After the make command is complete, check the lib folder and its sub-folders for the presence of `libm.a, libc.a,` ...etc.

The EWL library files are rebuilt.

20.1.2 Volatile Memory Locations

Typically memory mapped devices use a volatile declaration modifier to prevent the compiler from optimizing out loads and stores. The current implementation of the optimizer uses read-modify-write instructions where possible instead of a load, operation, store instruction sequence. If this presents a problem for a particular device, you need to turn off the peephole optimizer for routines accessing it, as shown below:

```
#pragma peephole off
void foo (void)
{...}
#pragma peephole reset
```

20.1.3 Predefined Symbols and Processor Families

The compiler defines a few symbols to support the ColdFire processors and families. Symbol __COLDFIRE__ is always set to represent the currently selected processor. All processors and families are symbolically represented with __MCFzzzz__ where zzzz represents either a family or a processor part number. ColdFire processors are combined into functional families. Selecting a family brings in scope the parts belonging to that set. This selection is occurs in the code generation settings panel. Once a family and processor is selected, it defines a serie of built-in macros making it possible for users to specialize their code by family and processors within a family.

Family Parts

```
521x 5211 5212 5213

5214_6 5214 5216

5221x 52210 52211 52212 52213

521x0 52100 52110

5222x 52221 52223

5223x 52230 52231 52232 52233 52234 52235 52236

528x 5280 5281 5282

5206e 5206e

5207_8 5207 5208

523x 5235
```

```
524x 5249

525x 5251 5253

5270_1 5270 5271

5272 5272

5274_5 5274 5275

5307 5307

532x 5327 5328 5329

537x 5371 5372 5373

5407 5407

5445x 54450 54451 54452 54453 54454 54455

547x 5470 5471 5472 5473 5474 5475

548x 5480 5481 5482 5483 5484 5485
```

For example, selecting a 54455 brings in scope all other processors belonging to that family as well as the family, the resulting set of built-in symbols is { `__MCF5445x__`, `__MCF54450__`, `__MCF54451__`, `__MCF54452__`, `__MCF54453__`, `__MCF54454__`, `__MCF54455__` }. The processor values are all distinct and the family value is simply defined.

```
#ifdef __MCF5445x__

/** this is true for **/

#if __COLDFIRE__ == __MCF54455__

/** this is true **/

#elif __COLDFIRE__ == __MCF54454__

/** this id false since the 54455 was selected **/

#endif

#endif

#ifdef __MCF547x__


/** this is false, we didn't select that family **/

#endif

#if __COLDFIRE__ == __MCF5445x__

/** this is false since __COLDFIRE__ represents a part and
not a family **/

#endif
```

This topic contains the following topic:

20.1.3.1 Codegen Settings

These settings are augmented with control for register allocation, instruction scheduling and folding (peephole). Register allocation places local variables in registers, to achieve this it performs variable lifetime analysis and leads to some variables used as temporaries to disappear. Variable lifetimes are recorded in the debugging informatin to provide accurate runtime values. Turning off this setting forces all locals to be stack based except compiler generated temporaries.

Peephole optimization replaces simple instructions sequences into more complex ones, reducing code size and often speeds up code. Instruction scheduling is performed at optimization level 2 and up, it schedules instructions to minimize latency.

The PC-relative strings setting was removed from the codegen setting to provide consistent behavior with all PC-relative pragmas: `pcrelstrings`, `pcreldata`, `pcrelconstdata`. The `__declspec(pcrel)` declaration specifier can also be used to force individual declaration to use pc relative addressing.

20.1.3.1.1 Alignment

Data accesses for ColdFire parts achieve better performance when long integers are aligned on a modulo-4 boundary. This alignment corresponds to the mc68k-4bytes or coldfire in the settings panel, it is the preferred alignment and is now the default for creating new projects.

20.1.3.1.2 Backward Compatibility

The struct alignment setting from the code generation panel controls structure alignment and is used by the compiler to indicate the preferred section alignment in the elf object. This default is overriden by the align_all directive in the `lcf`.

`Pragma align` accepts a *native* value that defaults to coldfire starting with this release, the previous value for native was 68k. A new pragma is introduced to provide backward native compatibility by setting 'native's value:

```
#pragma native_coldfire_alignment on|off|reset
```

When legacy 68k alignment is required for BINARY compatibility, EWL needs to be recompiled with native_coldfire_alignmentoff.

20.1.3.2 Sections

The compiler by default creates an ELF section per variable and function, for compilation units where the number of sections is reached the compiler can concatenate objects into single sections. The 'pool sections' checkbox from the code generation settings panel controls this behavior, the command line equivalent is -pool[_section]

The Embedded Warrior Libraries provides the libraries described in the ISO/IEC standards for C and C++. EWL also provides some extensions to the standard libraries.

- UART Libraries
- Memory, Heaps, and Other Libraries

20.1.4 UART Libraries

The ColdFire Embedded Warrior Libraries (EWL) supports console I/O through the serial port. This support includes:

- Standard C-library I/O.
- All functions that do not require disk I/O.
- Memory functions malloc() and free().

To use C or C++ libraries for console I/O, you must include a special serial UART driver library in your project. These driver library files are in folder: ColdFire_Support\Legacy.

The following table lists target boards and corresponding UART library files.

Table 20-1. Serial I/O UART Libraries

Board	Filename
CF5206e SBC	mot_sbc_5206e_serial\Bin\UART_SBC_5206e_Aux.a
CF5206e LITE	mot_5206e_lite_serial\Bin \UART_5206e_lite_Aux.a
CF5307 SBC	mot_sbc_5307_serial\Bin\UART_SBC_5307_Aux.a
CF5407 SBC	mot_sbc_5407_serial\Bin\UART_SBC_5407_Aux.a
CF5249 SBC	mot_sbc_5249_serial\Bin\UART_SBC_5249_Aux.a

20.1.5 Memory, Heaps, and Other Libraries

The heap you create in your linker command file becomes the default heap, so it does not need initialization. Additional memory and heap points are:

- To have the system link memory-management code into your code, call `malloc()` or `new()`.
- Initialize multiple memory pools to form a large heap.
- To create each memory pool, call `init_alloc()`. (You do not need to initialize the memory pool for the default heap.)

The memory allocation scheme in EWL requires the following symbols to be defined in the LCF file: `___mem_limit and ___stack_safety`.

For example,

```
___mem_limit    = ___HEAP_END;
 ___stack_safety = 16;
```

You may be able to use another standard C library with CodeWarrior projects. You should check the `stdarg.h` file in this other standard library and in your runtime libraries. Additional points are:

- The CodeWarrior C and C++ C/C++ compiler generates correct variable-argument functions only with the header file that the EWL include.
- You may find that other implementations are also compatible.
- You may also need to modify the runtime to support a different standard C library; you must include `__va_arg.c`.
- Other C++ libraries are not compatible.

NOTE

If you are working with any kind of embedded OS, you may need to customize EWL to work properly with that OS.

20.2 Runtime Libraries

Every ColdFire project must include a runtime library, which provides basic runtime support, basic initialization, system startup, and the jump to the main routine. RAM-based debug is the primary reason behind runtime-library development for ColdFire boards, so you probably must modify a library for your application.

Find your setup and then include the appropriate runtime library file:

- For a C project, use the file that starts with C_.
- For a C++ project, use the file that starts with Cpp_.
- All these files are in folder: `\ColdFire_Support\Runtime\Runtime_ColdFire`.

NOTE

ABI corresponds directly to the parameter-passing setting of the **ColdFire Processor Settings** panel (Standard, Compact or Register). If your target supports floating points, you should use an FPU-enabled runtime library file.

The topics covered here are as follows:

- Position-Independent Code
- Board Initialization Code
- Custom Modifications

20.2.1 Position-Independent Code

To use position-independent code or position-independent data in your program, you must customize the runtime library. Follow these steps:

1. Load project file `EWL_RuntimeCF.mcp`, from the folder `\ColdFire_Support\ewl\EWL_Runtime`.
2. Modify runtime functions.
   a. Open file `CF_startup.c`.
   b. As appropriate for your application, change or remove runtime function `__block_copy_section`. (This function relocates the PIC/PID sections in the absence of an operating system.)
   c. As appropriate for your application, change or remove runtime function `__fix_addr_references`. (This function creates the relocation tables.)
3. Change the prefix file.
   a. Open the C/C++ preference panel for your target.
   b. Make sure this panel specifies prefix file `PICPIDRuntimePrefix.h`.
4. Recompile the runtime library for your target.

Once you complete this procedure, you are ready to use the modified runtime library in your PIC/PID project. Source-file comments and runtime-library release notes may provide additional information.

20.2.2 Board Initialization Code

The CodeWarrior development tools come with several basic, assembly-language hardware initialization routines. Some of these initialization routines might be useful in your programs.

You do not need to include these initialization routines when you are debugging. The debugger and the debug kernel perform the board initialization.

We recommend that program your application to do as much initialization as possible, minimizing the initializations that the configuration file performs. This helps the debugging process by placing more of the code to be debugged in Flash memory or ROM rather than RAM.

20.2.3 Custom Modifications

A library file supports specific functionality. You might want to eliminate calls to library files that are not needed. For example, if your target-device memory is particularly small, you might want to delete library files that contain functionality that your application does not use. Follow this guidance:

- **Configuration settings** - You can change configuration settings in projects files or makefiles. Generally, modifying flags in the configuration header file `ansi_prefix.CF.size.h` is sufficient to modify the working set. Sometimes, however, you must also modify the `ansi_prefix.CF.h` header file.
- **Projects** - The easiest way to create a new project is to start from a copy of a project file that is compliant with the rules and standards you want for your project. Turning off flags, such as the flag for floating point support, forces you to remove some files from the project file list. But this is appropriate, as your project will not need those files. Although changing the basic configuration can require editing all targets of all project files, usually modifying the single targets your application uses is sufficient.
- **Makefiles** - Makefile targets already are set up to build any library; the `CFLAGS` macro defines the basic configuration. Target `all` does not include all targets, but a commented variation of all these targets is present in every makefile.

Chapter 21
Declaration Specifications

Declaration specifications describe special properties to associate with a function or variable at compile time. You insert these specifications in the object's declaration.

21.1 Syntax for Declaration Specifications

The syntax for a declaration specification is

```
__declspec(spec [ options ]) function-declaration;
```

where *spec* is the declaration specification, *options* represents possible arguments for the declaration specification, and *function-declaration* represents the declaration of the function. Unless otherwise specified in the declaration specification's description, a function's definition does not require a declaration specification.

21.2 Declaration Specifications

This topic lists the following declaration specifications:

21.2.1 __declspec(register_abi)

Specifies that a function should use `register_abi` for the calling convention.

Syntax

`__declspec (register_abi)` function_prototype;

Remarks

Declaring a function's prototype with this declaration specification tells the compiler use the specified calling convention. Also the user can specify `compact_abi` or `standard_abi`, if such a calling function is desired. This is especially useful for pure assembly functions.

21.2.2 __declspec(never_inline)

Specifies that a function must not be inlined.

Syntax

```
__declspec (never_inline) function_prototype;
```

Remarks

Declaring a function's prototype with this declaration specification tells the compiler not to inline the function, even if the function is later defined with the `inline`, `__inline__`, or `__inline` keywords.

21.3 Syntax for Attribute Specifications

The syntax for an attribute specification is

```
__attribute__((list-of-attributes))
```

where *list-of-attributes* is a comma-separated list of zero or more attributes to associate with the object. Place an attribute specification at the end of the declaration and definition of a function, function parameter, or variable. The following listing shows an example.

Listing: Example of an attribute specification

```
int f(int x __attribute__((unused))) __attribute__((never_inline));
int f(int x __attribute__((unused))) __attribute__((never_inline))

{
```

```
   return 20;
}
```

21.4 Attribute Specifications

This topic lists the following attribute specifications:

- __attribute__((deprecated))
- __attribute__((force_export))
- __attribute__((malloc))
- __attribute__((noalias))
- __attribute__((returns_twice))
- __attribute__((unused))
- __attribute__((used))

21.4.1 __attribute__((deprecated))

Specifies that the compiler must issue a warning when a program refers to an object.

Syntax

```
variable-declaration __attribute__((deprecated));

variable-definition __attribute__((deprecated));

function-declaration __attribute__((deprecated));

function-definition __attribute__((deprecated));
```

Remarks

This attribute instructs the compiler to issue a warning when a program refers to a function or variable. Use this attribute to discourage programmers from using functions and variables that are obsolete or will soon be obsolete.

Listing: Example of deprecated attribute

```
int velocipede(int speed) __attribute__((deprecated));
int bicycle(int speed);

int f(int speed)

{

  return velocipede(speed); /* Warning. */
```

```
}
int g(int speed)
{
  return bicycle(speed * 2); /* OK */
}
```

21.4.2 __attribute__((force_export))

Prevents a function or static variable from being dead-stripped.

Syntax

```
function-declaration __attribute__((force_export));
function-definition __attribute__((force_export));
variable-declaration __attribute__((force_export));
variable-definition __attribute__((force_export));
```

Remarks

This attribute specifies that the linker must not dead-strip a function or static variable even if the linker determines that the rest of the program does not refer to the object.

21.4.3 __attribute__((malloc))

Specifies that the pointers returned by a function will not point to objects that are already referred to by other variables.

Syntax

```
function-declaration __attribute__((malloc));
function-definition __attribute__((malloc));
```

Remarks

This attribute specification gives the compiler extra knowledge about pointer aliasing so that it can apply stronger optimizations to the object code it generates.

21.4.4 __attribute__((noalias))

Prevents access of data object through an indirect pointer access.

Syntax

```
function-parameter __attribute__((noalias));

variable-declaration __attribute__((noalias));

variable-definition __attribute__((noalias));
```

Remarks

This attribute specifies to the compiler that a data object is only accessed directly, helping the optimizer to generate a better code. The sample code in the following listing will not return a correct result if `ip` is pointed to `a`.

Listing: Example of the noalias attribute

```
extern int a __attribute__((noalias));
int f(int *ip)

{

  a = 1;

  *ip = 0;

  return a;   // optimized to return 1;

}
```

21.4.5 __attribute__((returns_twice))

Specifies that a function may return more than one time because of multithreaded or non-linear execution.

Syntax

```
function-declaration __attribute__((returns_twice));

function-definition __attribute__((returns_twice));
```

Remarks

This attribute specifies to the compiler that the program's flow of execution might enter and leave a function without explicit function calls and returns. For example, the standard library's `setjmp()` function allows a program to change its execution flow arbitrarily.

With this information, the compiler limits optimizations that require explicit program flow.

21.4.6 __attribute__((unused))

Specifies that the programmer is aware that a variable or function parameter is not referred to.

Syntax

```
function-parameter __attribute__((unused));

variable-declaration __attribute__((unused));

variable-definition __attribute__((unused));
```

Remarks

This attribute specifies that the compiler should not issue a warning for an object if the object is not referred to. This attribute specification has no effect if the compiler's unused warning setting is off.

Listing: Example of the unused attribute

```
void f(int a __attribute__((unused))) /* No warning for a. */
{

  int b __attribute__((unused)); /* No warning for b. */

  int c; /* Possible warning for c. */

  return 20;

}
```

21.4.7 __attribute__((used))

Prevents a function or static variable from being dead-stripped.

Syntax

```
function-declaration __attribute__((used));

function-definition __attribute__((used));

variable-declaration __attribute__((used));

variable-definition __attribute__((used));
```

Remarks

This attribute specifies that the linker must not dead-strip a function or static variable even if the linker determines that the rest of the program does not refer to the object.

Chapter 22
Predefined Macros

The compiler preprocessor has predefined macros (some refer to these as predefined symbols). The compiler simulates the variable definitions, that describe the compile-time environment and the properties of the target processor.

This chapter lists the following predefined macros that all CodeWarrior compilers make available:

22.1 __COUNTER__

Preprocessor macro that expands to an integer.

Syntax

```
__COUNTER__
```

Remarks

The compiler defines this macro as an integer that has an intial value of 0 incrementing by 1 every time the macro is used in the translation unit.

The value of this macro is stored in a precompiled header and is restored when the precompiled header is used by a translation unit.

22.2 __cplusplus

Preprocessor macro defined if compiling C++ source code.

Syntax

```
__cplusplus
```

Remarks

The compiler defines this macro when compiling C++ source code. This macro is undefined otherwise.

22.3 __CWCC__

Preprocessor macro defined as the version of the CodeWarrior compiler.

Syntax

```
__CWCC__
```

Remarks

CodeWarrior compilers issued after 2006 define this macro with the compiler's version. For example, if the compiler version is 4.2.0, the value of `__CWCC__` is `0x4200`.

CodeWarrior compilers issued prior to 2006 used the pre-defined macro `__MWERKS__`. The `__MWERKS__` predefined macro is still functional as an alias for `__CWCC__`.

The ISO standards do not specify this symbol.

22.4 __DATE__

Preprocessor macro defined as the date of compilation.

Syntax

```
__DATE__
```

Remarks

The compiler defines this macro as a character string representation of the date of compilation. The format of this string is

```
"Mmm dd yyyy"
```

where *Mmm* is the three-letter abbreviation of the month, *dd* is the day of the month, and *yyyy* is the year.

22.5 __embedded_cplusplus

Defined as 1 when compiling embedded C++ source code, undefined otherwise.

Syntax

```
__embedded_cplusplus
```

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the Embedded C++ proposed standard. The compiler does not define this macro otherwise.

22.6 __FILE__

Preprocessor macro of the name of the source code file being compiled.

Syntax

```
__FILE__
```

Remarks

The compiler defines this macro as a character string literal value of the name of the file being compiled, or the name specified in the last instance of a `#line` directive.

22.7 __func__

Predefined variable of the name of the function being compiled.

Prototype

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

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__func__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body. This variable is also undefined when C99 (ISO/IEC 9899-1999) or GCC (GNU Compiler Collection) extension settings are off.

22.8 __FUNCTION__

Predefined variable of the name of the function being compiled.

Prototype

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

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__FUNCTION__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body. This variable is also undefined when C99 (ISO/IEC 9899-1999) or GCC (GNU Compiler Collection) extension settings are off.

22.9 __ide_target()

Preprocessor operator for querying the IDE about the active build target.

Syntax

```
__ide_target("target_name")
```

`target-name`

The name of a build target in the active project in the CodeWarrior IDE.

Remarks

Expands to `1` if *target_name* is the same as the active build target in the CodeWarrior IDE's active project, otherwise, expands to `0`. The ISO standards do not specify this symbol.

22.10 __LINE__

Preprocessor macro of the number of the line of the source code file being compiled.

Syntax

```
__LINE__
```

Remarks

The compiler defines this macro as an integer value of the number of the line of the source code file that the compiler is translating. The `#line` directive also affects the value that this macro expands to.

22.11 __MWERKS__

Deprecated. Preprocessor macro defined as the version of the CodeWarrior compiler.

Syntax

```
__MWERKS__
```

Remarks

Replaced by the built-in preprocessor macro `__CWCC__`.

CodeWarrior compilers issued after 1995 define this macro with the compiler's version. For example, if the compiler version is 4.0, the value of `__MWERKS__` is `0x4000`.

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

The ISO standards do not specify this symbol.

22.12 __PRETTY_FUNCTION__

Predefined variable containing a character string of the "unmangled" name of the C++ function being compiled.

Syntax

Prototype

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

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__PRETTY_FUNCTION__`. This name, *function-name*, is the same identifier that appears in source code, not the "mangled" identifier that the compiler and linker use. The C++ compiler "mangles" a function name by appending extra characters to the function's identifier to denote the function's return type and the types of its parameters.

The ISO/IEC 14882-2003 C++ standard does not specify this symbol. This implicit variable is undefined outside of a function body. This symbol is only defined if the GCC extension setting is on.

22.13 __profile__

Preprocessor macro that specifies whether or not the compiler is generating object code for a profiler.

Syntax

```
__profile__
```

Remarks

Defined as 1 when generating object code that works with a profiler, otherwise, undefined. The ISO standards does not specify this symbol.

22.14 __STDC__

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

Syntax

```
__STDC__
```

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the ISO/IEC 9899-1990 and ISO/IEC 9899-1999 standards. Otherwise, the compiler does not define this macro.

22.15 __TIME__

Preprocessor macro defined as a character string representation of the time of compilation.

Syntax

```
__TIME__
```

Remarks

The compiler defines this macro as a character string representation of the time of compilation. The format of this string is

```
"hh:mm:ss"
```

where *hh* is a 2-digit hour of the day, *mm* is a 2-digit minute of the hour, and *ss* is a 2-digit second of the minute.

22.16 __optlevelx

Optimization level exported as a predefined macro.

Syntax

```
__optlevel0

__optlevel1

__optlevel2
```

__optlevel3

__optlevel4

Remarks

Using these macros, user can conditionally compile code for a particular optimization level. The following table lists the level of optimization provided by the __optlevelx macro.

Table 22-1. Optimization Levels

Macro	Optimization Level
__optlevel0	O0
__optlevel1	O1
__optlevel2	O2
__optlevel3	O3
__optlevel4	O4

Example

The listing below shows an example of __optlevelx macro usage.

Listing: Example usage of __optlevel macro

```
int main()
{

#if __optlevel0

... // This code compiles only if this code compiled with Optimization

level 0

#elif __optlevel1

... // This code compiles only if this code compiled with Optimization

level 1

#elif __optlevel2

... // This code compiles only if this code compiled with Optimization

level 2

#elif __optlevel3

... // This code compiles only if this code compiled with Optimization

level 3

#elif __optlevel4

... // This code compiles only if this code compiled with Optimization

level 4

#endif
```

```
}
```

Chapter 23
ColdFire Predefined Symbols

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

This chapter lists the following predefined symbols made available by the CodeWarrior compiler for ColdFire processors:

23.1 __COLDFIRE__

Preprocessor macro defined to describe the target ColdFire processor.

Syntax

```
#define __COLDFIRE__ processor_code
```

Parameter

`processor`

MCFxxxx

where, `xxxx` is the processor number.

Remarks

The compiler defines this macro to describe the ColdFire processor that the compiler is generating object code for.

23.2 __STDABI__

Preprocessor macro defined to describe the compiler's parameter-passing setting.

Syntax

```
#define __STDABI__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the compiler is set to use standard parameter-passing code generation, 0 otherwise.

23.3 __REGABI__

Preprocessor macro defined to describe the compiler's parameter-passing setting.

Syntax

```
#define __REGABI__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the compiler is set to use register-based parameter-passing code generation, 0 otherwise.

23.4 __fourbyteints__

Preprocessor macro defined to describe the compiler's int size setting.

Syntax

```
#define __fourbyteints__ 0 | 1
```

Remarks

The compiler defines this macro to be 1 if the compiler is set to use 4 bytes int , 0 otherwise.

23.5 __HW_FPU__

Preprocessor macro defined to describe the compiler's hardware and software floating point setting.

Syntax

```
#define __HW_FPU__ 0 | 1
```

Remarks

Set to 1 if the target has hardware floating point unit and the floating point is selected to be handled using the hardware capabilities. In this situation, the compiler generates code to handle floating point computations by using the FPU instruction set (note that any compliance / non-compliance with floating point standards is provided by the hardware floating point unit capabilities - one should check the compliance on the hardware FPU description side).

Set to 0 if the floating point is selected to be handled using software floating point library. In this situation, the compiler will generate code to handle floating point computations by using software emulation of floating point operations, implemented using runtime library functions. The software emulation is in compliance with the IEEE 754 floating point standard.

Chapter 24
Using Pragmas

The #pragma preprocessor directive specifies option settings to the compiler to control the compiler and linker's code generation.

24.1 Checking Pragma Settings

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

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

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

If *setting-name* is `on`, `__option(`*setting-name*`)` returns 1. If *setting-name* is `off`, `__option(`*setting-name*`)` returns 0. If *setting-name* is not the name of a pragma, `__option(`*setting-name*`)` returns false. If *setting-name* is the name of a pragma that does not accept the `on`, `off`, and `reset` arguments, the compiler issues a warning message.

The following listing shows an example.

Listing: Using the __option() preprocessor function

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

#else

#include "custom.h" /* Use the specialized declarations. */

#endif
```

24.2 Saving and Restoring Pragma Settings

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

Remembering which pragmas to save and restore is tedious and error-prone. Fortunately, the compiler has mechanisms that save and restore pragma settings at compile time. Pragma settings may be saved and restored at two levels:

- all pragma settings
- some individual pragma settings

Settings may be saved at one point in a compilation unit (a source code file and the files that it includes), changed, then restored later in the same compilation unit. Pragma settings cannot be saved in one source code file then restored in another unless both source code files are included in the same compilation unit.

Pragmas `push` and `pop` save and restore, respectively, most pragma settings in a compilation unit. Pragmas `push` and `pop` may be nested to unlimited depth. The following listing shows an example.

Listing: Using push and pop to save and restore pragma settings

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

#pragma optimize_for_size off

void fast_func_A(void)

{

/* ... */

}

/* Settings for slow_func(). */

#pragma push /* Save file settings. */

#pragma optimization_size 0

void slow_func(void)

{

/* ... */

}
```

```
#pragma pop /* Restore file settings. */

void fast_func_B(void)

{

/* ... */

}
```

Pragmas that accept the `reset` argument perform the same actions as pragmas `push` and `pop`, but apply to a single pragma. A pragma's `on` and `off` arguments save the pragma's current setting before changing it to the new setting. A pragma's `reset` argument restores the pragma's setting. The `on`, `off`, and `reset` arguments may be nested to an unlimited depth. The following listing shows an example.

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

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

void fast_func_A(void)

{

/* ... */

}

/* Setting for smallslowfunc(). */

#pragma opt_unroll_loops off

void small_func(void)

{

/* ... */

}

/* Restore previous setting. */

#pragma opt_unroll_loops reset

void fast_func_B(void)

{

/* ... */

}
```

24.3 Determining which Settings are Saved and Restored

Not all pragma settings are saved and restored by pragmas `push` and `pop`. Pragmas that do not change compiler settings are not affected by `push` and `pop`. For example, pragma `message` cannot be saved and restored.

The following listing shows an example that checks if the `ANSI_strict` pragma setting is saved and restored by pragmas `push` and `pop`.

Listing: Testing if pragmas push and pop save and restore a setting

```
/* Preprocess this source code. */
#pragma ANSI_strict on

#pragma push

#pragma ANSI_strict off

#pragma pop

#if __option(ANSI_strict)

#error "Saved and restored by push and pop."

#else

#error "Not affected by push and pop."

#endif
```

24.4 Invalid Pragmas

If you enable the compiler's setting for reporting invalid pragmas, the compiler issues a warning when it encounters a pragma it does not recognize. For example, the pragma statements in The following listing generate warnings with the invalid pragmas setting enabled.

Listing: Invalid Pragmas

```
#pragma silly_data off       // WARNING: silly_data is not a pragma.
#pragma ANSI_strict select  // WARNING: select is not defined

#pragma ANSI_strict on      // OK
```

The following table shows how to control the recognition of invalid pragmas.

Table 24-1. Controlling invalid pragmas

To control this option from here...	use this setting
CodeWarrior IDE	**Illegal Pragmas** in the **C/C++ Build > Settings > ColdFire Compiler > Warnings panel**
source code	`#pragma warn_illpragma`
command line	`-warnings illpragmas`

24.5 Pragma Scope

The scope of a pragma setting is limited to a compilation unit (a source code file and the files that it includes).

At the beginning of compilation unit, the compiler uses its default settings. The compiler then uses the settings specified by the CodeWarrior IDE's build target or in command-line options.

The compiler uses the setting in a pragma beginning at the pragma's location in the compilation unit. The compiler continues using this setting:

- until another instance of the same pragma appears later in the source code
- until an instance of pragma `pop` appears later in the source code
- until the compiler finishes translating the compilation unit

Chapter 25
Pragmas for Standard C Conformance

This chapter lists the following pragmas for standard C conformance:

25.1 ANSI_strict

Controls the use of non-standard language features.

Syntax

```
#pragma ANSI_strict on | off | reset
```

Remarks

If you enable the pragma `ANSI_strict`, the compiler generates an error message if it encounters some CodeWarrior extensions to the C language defined by the ISO/IEC 9899-1990 ("C90") standard:

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

25.2 c99

Controls the use of a subset of ISO/IEC 9899-1999 ("C99") language features.

Syntax

```
#pragma c99 on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts many of the language features described by the ISO/IEC 9899-1999 standard:

- More rigid type checking.
- Trailing commas in enumerations.
- GCC/C99-style compound literal values.
- Designated initializers.
- `__func__` predefined symbol.
- Implicit `return 0;` in `main()`.
- Non- `const` static data initializations.
- Variable argument macros (`__VA_ARGS__`).
- `bool` and `_Bool` support.
- `long long` support (separate switch).
- `restrict` support.
- `//` comments.
- `inline` support.
- Digraphs.
- `_Complex` and `_Imaginary` (treated as keywords but not supported).
- Empty arrays as last struct members.
- Designated initializers
- Hexadecimal floating-point constants.
- Variable length arrays are supported within local or function prototype scope (as required by the C99 standard).
- Unsuffixed decimal constant rules.
- `++bool--` expressions.
- `(T) (int-list)` are handled/parsed as cast-expressions and as literals.
- `__STDC_HOSTED__` is `1`.

25.3 c9x

Equivalent to `#pragma c99`.

25.4 ignore_oldstyle

Controls the recognition of function declarations that follow the syntax conventions used before ISO/IEC standard C (in other words, "K&R" style).

Syntax

```
#pragma ignore_oldstyle on | off | reset
```

Remarks

If you enable this pragma, the compiler ignores old-style function declarations and lets you prototype a function any way you want. In old-style declarations, you specify the types of arguments on separate lines instead of the function's argument list. For example, the code in the following listing defines a prototype for a function with an old-style definition.

Listing: Mixing Old-style and Prototype Function Declarations

```
int f(char x, short y, float z);
#pragma ignore_oldstyle on

f(x, y, z)

char x;

short y;

float z;

{

  return (int)x+y+z;

}

#pragma ignore_oldstyle reset
```

This pragma does not correspond to any panel setting. By default, this setting is disabled.

25.5 only_std_keywords

Controls the use of ISO/IEC keywords.

Syntax

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

Remarks

The compiler recognizes additional reserved keywords. If you are writing source code that must follow the ISO/IEC C standards strictly, enable the pragma `only_std_keywords`.

25.6 require_prototypes

Controls whether or not the compiler should expect function prototypes.

Syntax

```
#pragma require_prototypes on | off | reset
```

Remarks

This pragma affects only non-static functions.

If you enable this pragma, the compiler generates an error message when you use a function that does not have a preceding prototype. Use this pragma to prevent error messages, caused by referring to a function before you define it. For example, without a function prototype, you might pass data of the wrong type. As a result, your code might not work as you expect even though it compiles without error.

In the following listing, the `main()` function calls `PrintNum()` with an integer argument, even though the `PrintNum()` takes an argument of the type `float`.

Listing: Unnoticed Type-mismatch

```
#include <stdio.h>
void main(void)

{

  PrintNum(1);  /* PrintNum() tries to interpret the

                   integer as a float. Prints 0.000000. */

}

void PrintNum(float x)

{

    printf("%f\n", x);

}
```

When you run this program, you could get this result:

```
  0.000000
```

Although the compiler does not complain about the type mismatch, the function does not give the result you desired. Since `PrintNum()` does not have a prototype, the compiler does not know how to generate the instructions to convert the integer, to a floating-point number before calling `PrintNum()`. Consequently, the function interprets the bits it received as a floating-point number and prints nonsense.

A prototype for `PrintNum()`, as in the following listing, gives the compiler sufficient information about the function to generate instructions to properly convert its argument to a floating-point number. The function prints what you expected.

Listing: Using a Prototype to Avoid Type-mismatch

```
#include <stdio.h>
void PrintNum(float x); /* Function prototype. */

void main(void)

{

    PrintNum(1);        /*  Compiler converts int to float.

}                           Prints 1.000000. */

void PrintNum(float x)

{

    printf("%f\n", x);

}
```

In other situations where automatic conversion is not possible, the compiler generates an error message if an argument does not match the data type required by a function prototype. Such a mismatched data type error is easier to locate at compile time than at runtime.

Chapter 26
Pragmas for C++

This chapter list the following pragmas for C++:

Table 26-1. Pragmas for C++

access_errors	always_inline	arg_dep_lookup	ARM_conform
ARM_scoping	array_new_delete	auto_inline	bool
cplusplus	cpp1x	cpp_extensions	debuginline
def_inherited	defer_codegen	defer_defarg_parsing	direct_destruction
direct_to_som	dont_inline	ecplusplus	exceptions
extended_errorcheck	inline_bottom_up	inline_bottom_up_once	inline_depth
inline_max_auto_size	inline_max_size	inline_max_total_size	internal
iso_templates	new_mangler	no_conststringconv	no_static_dtors
nosyminline	old_friend_lookup	old_pods	old_vtable
opt_classresults	parse_func_templ	parse_mfunc_templ	RTTI
suppress_init_code	template_depth	thread_safe_init	warn_hidevirtual
warn_no_explicit_virtual	warn_no_typename	warn_notinlined	warn_structclass
wchar_type			

26.1 access_errors

Controls whether or not to change invalid access errors to warnings.

Syntax

```
#pragma access_errors on | off | reset
```

Remarks

If you enable this pragma, the compiler issues an error message instead of a warning when it detects invalid access to protected or private class members.

This pragma does not correspond to any IDE panel setting. By default, this pragma is `on`.

26.2 always_inline

Controls the use of inlined functions.

Syntax

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

Remarks

This pragma is deprecated. We recommend that you use the `inline_depth()` pragma instead.

26.3 arg_dep_lookup

Controls C++ argument-dependent name lookup.

Syntax

```
#pragma arg_dep_lookup on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler uses argument-dependent name lookup.

This pragma does not correspond to any IDE panel setting. By default, this setting is `on`.

26.4 ARM_conform

This pragma is no longer available. Use `ARM_scoping` instead.

26.5 ARM_scoping

Controls the scope of variables declared in the expression parts of `if`, `while`, `do`, and `for` statements.

Syntax

```
#pragma ARM_scoping on | off | reset
```

Remarks

If you enable this pragma, any variables you define in the conditional expression of an `if`, `while`, `do`, or `for` statement remain in scope until the end of the block that contains the statement. Otherwise, the variables only remain in scope until the end of that statement. The following listing shows an example.

Listing: Example of Using Variables Declared in for Statement

```
for(int i=1; i<1000; i++) { /* . . . */ }
return i;  // OK if ARM_scoping is on, error if ARM_scoping is off.
```

26.6 array_new_delete

Enables the operator `new[]` and `delete[]` in array allocation and deallocation operations, respectively.

Syntax

```
#pragma array_new_delete on | off | reset
```

Remarks

By default, this pragma is `on`.

26.7 auto_inline

Controls which functions to inline.

Syntax

```
#pragma auto_inline on | off | reset
```

Remarks

If you enable this pragma, the compiler automatically chooses functions to inline for you, in addition to functions declared with the `inline` keyword.

Note that if you enable the `dont_inline` pragma, the compiler ignores the setting of the `auto_inline` pragma and does not inline any functions.

26.8 bool

Determines whether or not `bool`, `true`, and `false` are treated as keywords in C++ source code.

Syntax

```
#pragma bool on | off | reset
```

Remarks

If you enable this pragma, you can use the standard C++ `bool` type to represent `true` and `false`. Disable this pragma if `bool`, `true`, or `false` are defined in your source code.

Enabling the `bool` data type and its `true` and `false` values is not equivalent to defining them in source code with `typedef`, `enum`, or `#define`. The C++ `bool` type is a distinct type defined by the ISO/IEC 14882-2003 C++ Standard. Source code that does not treat `bool` as a distinct type might not compile properly.

26.9 cplusplus

Controls whether or not to translate subsequent source code as C or C++ source code.

Syntax

```
#pragma cplusplus on | off | reset
```

Remarks

If you enable this pragma, the compiler translates the source code that follows as C++ code. Otherwise, the compiler uses the suffix of the filename to determine how to compile it. If a file name ends in `.c`, `.h`, or `.pch`, the compiler automatically compiles it as C code, otherwise as C++. Use this pragma only if a file contains both C and C++ code.

NOTE

The CodeWarrior C/C++ compilers do not distinguish between uppercase and lowercase letters in file names and file name extensions except on UNIX-based systems.

26.10 cpp1x

Controls whether or not to enable support to experimental features made available in the 1x version of C++ standard.

Syntax

```
#pragma cpp1x on | off | reset
```

Remarks

If you enable this pragma, you can use the following extensions to the 1x or 05 version of the C++ standard that would otherwise be invalid:

- Enables support for `__alignof__`.
- Enables support for `__decltype__`, which is a reference type preserving typeof.
- Enables support for `nullptr`.
- Enables support to allow `>>` to terminate nested template argument lists.
- Enables support for `__static_assert`.

NOTE

This pragma enables support to experimental and unvalidated implementations of features that may or may not be available in the final version of the C++ standard. The features should not be used for critical or production code.

26.11 cpp_extensions

Controls language extensions to ISO/IEC 14882-2003 C++.

Syntax

```
#pragma cpp_extensions on | off | reset
```

Remarks

If you enable this pragma, you can use the following extensions to the ISO/IEC 14882-2003 C++ standard that would otherwise be invalid:

- Anonymous `struct` & `union` objects. The following listing shows an example.
  Listing: Example of Anonymous struct & union Objects

```
#pragma cpp_extensions on
void func()
```

```
{
  union {
    long  hilo;
    struct { short hi, lo; }; //  anonymous struct
  };
  hi=0x1234;
  lo=0x5678;  //  hilo==0x12345678
}
```

- Unqualified pointer to a member function. The following listing shows an example.
  Listing: Example of an Unqualified Pointer to a Member Function

```
#pragma cpp_extensions on
struct RecA { void f(); }

void RecA::f()

{

  void (RecA::*ptmf1)() = &RecA::f; // ALWAYS OK

  void (RecA::*ptmf2)() = f; // OK if you enable cpp_extensions.

}
```

- Inclusion of `const` data in precompiled headers.

26.12 debuginline

Controls whether the compiler emits debugging information for expanded inline function calls.

Syntax

```
#pragma debuginline on | off | reset
```

Remarks

If the compiler emits debugging information for inline function calls, then the debugger can step to the body of the inlined function. This behavior more closely resembles the debugging experience for un-inlined code.

NOTE

Since the actual "call" and "return" instructions are no longer present when stepping through inline code, the debugger will immediately jump to the body of an inlined function and "return" before reaching the return statement for the function.

Thus, the debugging experience of inlined functions may not be as smooth as debugging un-inlined code.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

26.13 def_inherited

Controls the use of `inherited`.

Syntax

```
#pragma def_inherited on | off | reset
```

Remarks

The use of this pragma is deprecated. It lets you use the non-standard `inherited` symbol in C++ programming by implicitly adding

```
typedef base inherited;
```

as the first member in classes with a single base class.

NOTE

The ISO/IEC 14882-2003 C++ standard does not support the `inherited` symbol. Only the CodeWarrior C++ language implements the `inherited` symbol for single inheritance.

26.14 defer_codegen

Obsolete pragma. Replaced by interprocedural analysis options. See Interprocedural Analysis.

26.15 defer_defarg_parsing

Defers the parsing of default arguments in member functions.

Syntax

```
#pragma defer_defarg_parsing on | off
```

Remarks

To be accepted as valid, some default expressions with template arguments will require additional parenthesis. For example, the following listing results in an error message.

Listing: Deferring parsing of default arguments

```
template<typename T,typename U> struct X { T t; U u; };
struct Y {

   // The following line is not accepted, and generates

   // an error message with defer_defarg_parsing on.

   void f(X<int,int> = X<int,int>());

};
```

The following listing does not generate an error message.

Listing: Correct default argument deferral

```
template<typename T,typename U> struct X { T t; U u; };
struct Y {

   // The following line is OK if the default

   // argument is parenthesized.

   void f(X<int,int> = (X<int,int>()) );

};
```

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

26.16 direct_destruction

This pragma is obsolete. It is no longer available.

26.17 direct_to_som

This pragma is obsolete. It is no longer available.

26.18 dont_inline

Controls the generation of inline functions.

Syntax

```
#pragma dont_inline on | off | reset
```

Remarks

If you enable this pragma, the compiler does not inline any function calls, even those declared with the `inline` keyword or within a class declaration. Also, it does not automatically inline functions, regardless of the setting of the `auto_inline` pragma, described in auto_inline. If you disable this pragma, the compiler expands all inline function calls, within the limits you set through other inlining-related pragmas.

26.19 ecplusplus

Controls the use of embedded C++ features.

Syntax

```
#pragma ecplusplus on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler disables the non-EC++ features of ISO/IEC 14882-2003 C++ such as templates, multiple inheritance, and so on.

26.20 exceptions

Controls the availability of C++ exception handling.

Syntax

```
#pragma exceptions on | off | reset
```

Remarks

If you enable this pragma, you can use the `try` and `catch` statements in C++ to perform exception handling. If your program does not use exception handling, disable this setting to make your program smaller.

You can throw exceptions across any code compiled by the CodeWarrior C/C++ compiler with `#pragma exceptions on`.

You cannot throw exceptions across libraries compiled with `#pragma exceptions off`. If you throw an exception across such a library, the code calls `terminate()` and exits.

This pragma corresponds to the **Enable C++ Exceptions** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is `on`.

26.21 extended_errorcheck

Controls the issuing of warning messages for possible unintended logical errors.

Syntax

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

Remarks

If you enable this pragma, the C++ compiler generates a warning message for the possible unintended logical errors.

It also issues a warning message when it encounters a delete operator for a class or structure that has not been defined yet. The following listing shows an example.

Listing: Attempting to delete an undefined structure

```
#pragma extended_errorcheck on
struct X;

int func(X *xp)

{

     delete xp;    // Warning: deleting incomplete type X

}
```

An empty `return` statement in a function that is not declared `void`. For example, the following listing results in a warning message.

Listing: A non-void function with an empty return statement

```
int MyInit(void)
{

  int err = GetMyResources();

  if (err != -1)

  {

    err = GetMoreResources();

  }
```

```
  return; /* WARNING: empty return statement */
}
```

The following listing shows how to prevent this warning message.

Listing: A non-void function with a proper return statement

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err != -1)
  {
    err = GetMoreResources();
  }
  return err; /* OK */
}
```

26.22 inline_bottom_up

Controls the bottom-up function inlining method.

Syntax

```
  #pragma inline_bottom_up on | off | reset
```

Remarks

Bottom-up function inlining tries to expand up to eight levels of inline leaf functions. The maximum size of an expanded inline function and the caller of an inline function can be controlled by the pragmas shown in the following listings.

Listing: Maximum Complexity of an Inlined Function

```
// Maximum complexity of an inlined function
#pragma inline_max_size(
max )          // default
max == 256
```

Listing: Maximum Complexity of a Function that Calls Inlined Functions

```
// Maximum complexity of a function that calls inlined functions
#pragma inline_max_total_size(
max )    // default
max == 10000
```

where *max* loosely corresponds to the number of instructions in a function.

If you enable this pragma, the compiler calculates inline depth from the last function in the call chain up to the first function that starts the call chain. The number of functions the compiler inlines from the bottom depends on the values of `inline_depth`, `inline_max_size`, and `inline_max_total_size`. This method generates faster and smaller source code for some (but not all) programs with many nested inline function calls.

If you disable this pragma, top-down inlining is selected, and the `inline_depth` setting determines the limits for top-down inlining. The `inline_max_size` and `inline_max_total_size` pragmas do not affect the compiler in top-down mode.

26.23 inline_bottom_up_once

Performs a single bottom-up function inlining operation.

Syntax

```
#pragma inline_bottom_up_once on | off | reset
```

Remarks

By default, this pragma is `off`.

26.24 inline_depth

Controls how many passes are used to expand inline function calls.

Syntax

```
#pragma inline_depth(n)
#pragma inline_depth(smart)
```

Parameters

`n`

Sets the number of passes used to expand inline function calls. The number *n* is an integer from `0` to `1024` or the `smart` specifier. It also represents the distance allowed in the call chain from the last function up. For example, if `d` is the total depth of a call chain, then functions below a depth of `d-n` are inlined if they do not exceed the following size settings:

```
#pragma inline_max_size(n);
```

```
#pragma inline_max_total_size(n);
```

The first pragma sets the maximum function size to be considered for inlining; the second sets the maximum size to which a function is allowed to grow after the functions it calls are inlined. Here, *n* is the number of statements, operands, and operators in the function, which turns out to be roughly twice the number of instructions generated by the function. However, this number can vary from function to function. For the `inline_max_size` pragma, the default value of *n* is 256; for the `inline_max_total_size` pragma, the default value of *n* is 10000.

`smart`

The `smart` specifier is the default mode, with four passes where the passes 2-4 are limited to small inline functions. All inlineable functions are expanded if `inline_depth` is set to 1-1024.

26.25 inline_max_auto_size

Determines the maximum complexity for an auto-inlined function.

Syntax

```
#pragma inline_max_auto_size ( complex )
```

Parameters

`complex`

The `complex` value is an approximation of the number of statements in a function, the current default value is 15. Selecting a higher value will inline more functions, but can lead to excessive code bloat.

Remarks

This pragma does not correspond to any panel setting.

26.26 inline_max_size

Sets the maximum number of statements, operands, and operators used to consider the function for inlining.

Syntax

```
#pragma inline_max_size ( size )
```

Parameters

`size`

The maximum number of statements, operands, and operators in the function to consider it for inlining, up to a maximum of 256.

Remarks

This pragma does not correspond to any panel setting.

26.27 inline_max_total_size

Sets the maximum total size a function can grow to when the function it calls is inlined.

Syntax

```
#pragma inline_max_total_size ( max_size )
```

Parameters

`max_size`

The maximum number of statements, operands, and operators the inlined function calls that are also inlined, up to a maximum of 7000.

Remarks

This pragma does not correspond to any panel setting.

26.28 internal

Controls the internalization of data or functions.

Syntax

```
#pragma internal on | off | reset

#pragma internal list name1 [, name2 ]*
```

Remarks

When using the `#pragma internal on` format, all data and functions are automatically internalized.

Use the `#pragma internal list` format to tag specific data or functions for internalization. It applies to all names if it is used on an overloaded function. You cannot use this pragma for C++ member functions or static class members.

The following listing shows an example:

Listing: Example of an Internalized List

```
extern int f(), g;
#pragma internal list f,g
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

26.29 iso_templates

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler and issue warning messages for missing typenames.

Syntax

```
#pragma iso_templates on | off | reset
```

Remarks

This pragma combines the functionality of pragmas parse_func_templ, parse_mfunc_templ and warn_no_typename.

This pragma ensures that your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions. The compiler issues a warning message if a typenames required by the C++ standard is missing but can still be determined by the compiler based on the context of the surrounding C++ syntax.

By default, this pragma is `on`.

26.30 new_mangler

Controls the inclusion or exclusion of a template instance's function return type, to the mangled name of the instance.

Syntax

```
#pragma new_mangler on | off | reset
```

Remarks

The C++ standard requires that the function return type of a template instance to be included in the mangled name, which can cause incompatibilities. Enabling this pragma within a prefix file resolves those incompatibilities.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

26.31 no_conststringconv

Disables the deprecated implicit const string literal conversion (ISO/IEC 14882-2003 C++, §4.2).

Syntax

```
#pragma no_conststringconv on | off | reset
```

Remarks

When enabled, the compiler generates an error message when it encounters an implicit const string conversion.

Listing: Example of const string conversion

```
#pragma no_conststringconv on
char *cp = "Hello World"; /* Generates an error message. */
```

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

26.32 no_static_dtors

Controls the generation of static destructors in C++.

Syntax

```
#pragma no_static_dtors on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate destructor calls for static data objects. Use this pragma to generate smaller object code for C++ programs that never exit (and consequently never need to call destructors for static objects).

This pragma does not correspond to any panel setting. By default, this setting is disabled.

26.33 nosyminline

Controls whether debug information is gathered for inline/template functions.

Syntax

```
#pragma nosyminline on | off | reset
```

Remarks

When on, debug information is not gathered for inline/template functions.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

26.34 old_friend_lookup

Implements non-standard C++ friend declaration behavior that allows friend declarations to be visible in the enclosing scope.

```
#pragma old_friend_lookup on | off | reset
```

Example

This example shows friend declarations that are invalid without #pragma old_friend_lookup.

Listing: Valid and invalid declarations without #pragma old_friend_lookup

```
class C2;
void f2();

struct S {

     friend class C1;

     friend class C2;

     friend void f1();

     friend void f2();

};

C1 *cp1;    // error, C1 is not visible without namespace declaration

C2 *cp2;    // OK

int main()
```

```
{
     f1();  // error, f1() is not visible without namespace declaration
     f2();  // OK
}
```

26.35 old_pods

Permits non-standard handling of classes, structs, and unions containing pointer-to-pointer members

Syntax

```
#pragma old_pods on | off | reset
```

Remarks

According to the ISO/IEC 14882:2003 C++ Standard, classes/structs/unions that contain pointer-to-pointer members are now considered to be plain old data (POD) types.

This pragma can be used to get the old behavior.

26.36 old_vtable

This pragma is no longer available.

26.37 opt_classresults

Controls the omission of the copy constructor call for class return types if all return statements in a function return the same local class object.

Syntax

```
#pragma opt_classresults on | off | reset
```

Remarks

The following listing shows an example.

Listing: Example #pragma opt_classresults

```
#pragma opt_classresults on
struct X {

  X();

  X(const X&);

  // ...

};

X f() {

  X x; // Object x will be constructed in function result buffer.

  // ...

  return x; // Copy constructor is not called.

}
```

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

26.38 parse_func_templ

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler.

Syntax

```
#pragma parse_func_templ on | off | reset
```

Remarks

If you enable this pragma, your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions.

This option actually corresponds to the **ISO C++ Template Parser** option (together with pragmas parse_func_templ and warn_no_typename). By default, this pragma is disabled.

26.39 parse_mfunc_templ

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler for member function bodies.

Syntax

```
#pragma parse_mfunc_templ on | off | reset
```

Remarks

If you enable this pragma, member function bodies within your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

26.40 RTTI

Controls the availability of runtime type information.

Syntax

```
#pragma RTTI on | off | reset
```

Remarks

If you enable this pragma, you can use runtime type information (or RTTI) features such as `dynamic_cast` and `typeid`.

NOTE

Note that `*type_info::before(const type_info&)` is not implemented.

26.41 suppress_init_code

Controls the suppression of static initialization object code.

Syntax

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

Remarks

If you enable this pragma, the compiler does not generate any code for static data initialization such as C++ constructors.

WARNING!

Using this pragma can produce erratic or unpredictable behavior in your program.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

26.42 template_depth

Controls how many nested or recursive class templates you can instantiate.

```
#pragma template_depth(n)
```

Remarks

This pragma lets you increase the number of nested or recursive class template instantiations allowed. By default, *n* equals 64; it can be set from 1 to 30000. You should always use the default value unless you receive the error message

```
template too complex or recursive
```

This pragma does not correspond to any panel setting.

26.43 thread_safe_init

Controls the addition of extra code in the binary to ensure that multiple threads cannot enter a static local initialization at the same time.

Syntax

```
#pragma thread_safe_init on | off | reset
```

Remarks

A C++ program that uses multiple threads and static local initializations introduces the possibility of contention over which thread initializes static local variable first. When the pragma is `on`, the compiler inserts calls to mutex functions around each static local initialization to avoid this problem. The C++ runtime library provides these mutex functions.

Listing: Static local initialization example

```
int func(void) {
  // There may be synchronization problems if this function is

  // called by multiple threads.

  static int countdown = 20;

  return countdown--;

}
```

NOTE

This pragma requires runtime library functions which may not be implemented on all platforms, due to the possible need for operating system support.

The following listing shows another example.

Listing: Example thread_safe_init

```
#pragma thread_safe_init on
void thread_heavy_func()

{

  // Multiple threads can now safely call this function:

  // the static local variable will be constructed only once.

  static std::string localstring = thread_unsafe_func();

}
```

NOTE

When an exception is thrown from a static local initializer, the initializer is retried by the next client that enters the scope of the local.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

26.44 warn_hidevirtual

Controls the recognition of a non-virtual member function that hides a virtual function in a superclass.

Syntax

```
#pragma warn_hidevirtual on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if you declare a non-virtual member function that hides a virtual function in a superclass. One function hides another if it has the same name but a different argument type. The following listing shows an example.

Listing: Hidden Virtual Functions

```
class A {
  public:

    virtual void f(int);
```

```
    virtual void g(int);
};
class B: public A {
  public:
    void f(char);        // WARNING: Hides A::f(int)
    virtual void g(int); // OK: Overrides A::g(int)
};
```

The ISO/IEC 14882-2003 C++ Standard does not require this pragma.

NOTE

A warning message normally indicates that the pragma name is not recognized, but an error indicates either a syntax problem or that the pragma is not valid in the given context.

26.45 warn_no_explicit_virtual

Controls the issuing of warning messages if an overriding function is not declared with a virtual keyword.

Syntax

```
#pragma warn_no_explicit_virtual on | off | reset
```

Remarks

The following listing shows an example.

Listing: Example of warn_no_explicit_virtual pragma

```
#pragma warn_no_explicit_virtual on
struct A {
  virtual void f();
};
struct B {
  void f();
  // WARNING: override B::f() is declared without virtual keyword
}
```

Tip

This warning message is not required by the ISO/IEC 14882-2003 C++ standard, but can help you track down unwanted overrides.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

26.46 warn_no_typename

Controls the issuing of warning messages for missing `typenames`.

Syntax

```
#pragma warn_no_typename on | off | reset
```

Remarks

The compiler issues a warning message if a `typenames` required by the C++ standard is missing but can still be determined by the compiler based on the context of the surrounding C++ syntax.

This pragma does not correspond to any panel setting. This pragma is enabled by the ISO/IEC 14882-2003 C++ template parser.

26.47 warn_notinlined

Controls the issuing of warning messages for functions the compiler cannot inline.

Syntax

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

Remarks

The compiler issues a warning message for non-inlined, (i.e., on those indicated by the `inline` keyword or in line in a class declaration) inline function calls.

26.48 warn_structclass

Controls the issuing of warning messages for the inconsistent use of the `class` and `struct` keywords.

Syntax

```
#pragma warn_structclass on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if you use the `class` and `struct` keywords in the definition and declaration of the same identifier.

Listing: Inconsistent use of class and struct

```
class X;
struct X { int a; }; // WARNING
```

Use this warning when using static or dynamic libraries to link with object code produced by another C++ compiler that distinguishes between class and structure variables in its name " mangling."

26.49 wchar_type

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

Syntax

```
#pragma wchar_type on | off | reset
```

Remarks

If you enable this pragma, `wchar_t` is treated as a built-in type. Otherwise, the compiler does not recognize this type.

This pragma corresponds to the **Enable wchar_t Support** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is enabled.

Chapter 27
Pragmas for Language Translation

This chapter lists the following pragmas for language translation:

Table 27-1. Pragmas for Language Translation

asmpoundcomment	asmsemicolcomment	const_strings
dollar_identifiers	gcc_extensions	mark
mpwc_newline	mpwc_relax	multibyteaware
multibyteaware_preserve_literals	text_encoding	trigraphs
unsigned_char		

27.1 asmpoundcomment

Controls whether the "#" symbol is treated as a comment character in inline assembly.

Syntax

```
#pragma asmpoundcomment on | off | reset
```

Remarks

Some targets may have additional comment characters, and may treat these characters as comments even when

```
#pragma asmpoundcomment off
```

is used.

Using this pragma may interfere with the function-level inline assembly language.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

27.2 asmsemicolcomment

Controls whether the " ;" symbol is treated as a comment character in inline assembly.

Syntax

```
#pragma asmsemicolcomment on | off | reset
```

Remarks

Some targets may have additional comment characters, and may treat these characters as comments even when

```
#pragma asmsemicolcomment off
```

is used.

Using this pragma may interfere with the assembly language of a specific target.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

27.3 const_strings

Controls the `const`-ness of character string literals.

Syntax

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

Remarks

If you enable this pragma, the type of string literals is an array `const char[`*n*`]`, or `const wchar_t[`*n*`]` for wide strings, where *n* is the length of the string literal plus 1 for a terminating `NUL` character. Otherwise, the type `char[`*n*`]` or `wchar_t[`*n*`]` is used.

27.4 dollar_identifiers

Controls the 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 except underscores, alphabetic, numeric character, and universal characters (`\uxxxx`, `\Uxxxxxxxx`) in an identifier.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

27.5 gcc_extensions

Controls the acceptance of GNU C language extensions.

Syntax

```
#pragma gcc_extensions on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts GNU C extensions in C source code. This includes the following non-ANSI C extensions:

- Initialization of automatic `struct` or `array` variables with non- `const` values.
- Illegal pointer conversions
- `sizeof( void ) == 1`
- `sizeof( function-type ) == 1`
- Limited support for GCC statements and declarations within expressions.
- Macro redefinitions without a previous `#undef`.
- The GCC keyword `typeof`
- Function pointer arithmetic supported
- `void*` arithmetic supported
- Void expressions in return statements of void
- `__builtin_constant_p (`*expr*`)` supported
- Forward declarations of arrays of incomplete type
- Forward declarations of empty static arrays
- Pre-C99 designated initializer syntax (deprecated)
- shortened conditional expression (*c* `?:` *y*)
- `long __builtin_expect (long exp, long c)` now accepted

27.6 mark

Adds an item to the **Function** pop-up menu in the IDE editor.

Syntax

```
#pragma mark itemName
```

Remarks

This pragma adds *itemName* to the source file's **Function** pop-up menu. If you open the file in the CodeWarrior Editor and select the item from the **Function** pop-up menu, the editor brings you to the pragma. Note that if the pragma is inside a function definition, the item does not appear in the **Function** pop-up menu.

If *itemName* begins with " `--`", a menu separator appears in the IDE's **Function** pop-up menu:

```
#pragma mark --
```

This pragma does not correspond to any panel setting.

27.7 mpwc_newline

Controls the use of newline character convention.

Syntax

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

Remarks

If you enable this pragma, the compiler translates `'\n'` as a Carriage Return (0x0D) and `'\r'` as a Line Feed (0x0A). Otherwise, the compiler uses the ISO standard conventions for these characters.

If you enable this pragma, use ISO standard libraries that were compiled when this pragma was enabled.

If you enable this pragma and use the standard ISO standard libraries, your program will not read and write `'\n'` and `'\r'` properly. For example, printing `'\n'` brings your program's output to the beginning of the current line instead of inserting a newline.

This pragma does not correspond to any IDE panel setting. By default, this pragma is disabled.

27.8 mpwc_relax

Controls the compatibility of the `char*` and `unsigned char*` types.

Syntax

```
#pragma mpwc_relax on | off | reset
```

Remarks

If you enable this pragma, the compiler treats `char*` and `unsigned char*` as the same type. Use this setting to compile source code written before the ISO C standards. Old source code frequently uses these types interchangeably.

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

NOTE

Turning this option on may prevent the compiler from detecting some programming errors. We recommend not turning on this option.

The following listing shows how to use this pragma to relax function pointer checking.

Listing: Relaxing function pointer checking

```
#pragma mpwc_relax on
extern void f(char *);

/* Normally an error, but allowed. */

extern void(*fp1)(void *) = &f;

/* Normally an error, but allowed. */

extern void(*fp2)(unsigned char *) = &f;
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

27.9 multibyteaware

Controls how the **Source encoding** option in the IDE is treated

Syntax

```
#pragma multibyteaware on | off | reset
```

Remarks

This pragma is deprecated. See `#pragma text_encoding` for more details.

27.10 multibyteaware_preserve_literals

Controls the treatment of multibyte character sequences in narrow character string literals.

Syntax

```
#pragma multibyteaware_preserve_literals on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

27.11 text_encoding

Identifies the character encoding of source files.

Syntax

```
#pragma text_encoding ( "name" | unknown | reset [, global] )
```

Parameters

`name`

The IANA or MIME encoding name or an OS-specific string that identifies the text encoding. The compiler recognizes these names and maps them to its internal decoders:

```
system US-ASCII ASCII ANSI_X3.4-1968

ANSI_X3.4-1968 ANSI_X3.4 UTF-8 UTF8 ISO-2022-JP

CSISO2022JP ISO2022JP CSSHIFTJIS SHIFT-JIS

SHIFT_JIS SJIS EUC-JP EUCJP UCS-2 UCS-2BE

UCS-2LE UCS2 UCS2BE UCS2LE UTF-16 UTF-16BE

UTF-16LE UTF16 UTF16BE UTF16LE UCS-4 UCS-4BE

UCS-4LE UCS4 UCS4BE UCS4LE 10646-1:1993

ISO-10646-1 ISO-10646 unicode

global
```

Tells the compiler that the current and all subsequent files use the same text encoding. By default, text encoding is effective only to the end of the file.

Remarks

By default, `#pragmatext_encoding` is only effective through the end of file. To affect the default text encoding assumed for the current and all subsequent files, supply the "global" modifier.

27.12 trigraphs

Controls the use trigraph sequences specified in the ISO standards.

Syntax

```
#pragma trigraphs on | off | reset
```

Remarks

If you are writing code that must strictly adhere to the ANSI standard, enable this pragma.

Table 27-2. Trigraph table

Trigraph	Character
??=	#
??/	\
??'	^
??(	[
??)	]
??!	|
??<	{
??>	}
??-	~

NOTE

Use of this pragma may cause a portability problem for some targets.

Be careful when initializing strings or multi-character constants that contain question marks.

Listing: Example of Pragma trigraphs

```
char c = '????'; /* ERROR: Trigraph sequence expands to '??^ */
char d = '\?\?\?\?'; /* OK */
```

27.13 unsigned_char

Controls whether or not declarations of type `char` are treated as `unsigned char`.

Syntax

```
#pragma unsigned_char on | off | reset
```

Remarks

If you enable this pragma, the compiler treats a `char` declaration as if it were an `unsigned char` declaration.

NOTE

If you enable this pragma, your code might not be compatible with libraries that were compiled when the pragma was disabled. In particular, your code might not work with the ISO standard libraries included with CodeWarrior.

Chapter 28
Pragmas for Diagnostic Messages

This chapter lists the following pragmas for diagnostic messages:

Table 28-1. Pragmas for Diagnostic Messages

extended_errorcheck	maxerrorcount	message	showmessagenumber
show_error_filestack	suppress_warnings	sym	unused
warning	warning_errors	warn_any_ptr_int_conv	warn_emptydecl
warn_extracomma	warn_filenamecaps	warn_filenamecaps_system	warn_hiddenlocals
warn_illpragma	warn_illtokenpasting	warn_illunionmembers	warn_impl_f2i_conv
warn_impl_i2f_conv	warn_impl_s2u_conv	warn_implicitconv	warn_largeargs
warn_missingreturn	warn_no_side_effect	warn_padding	warn_pch_portability
warn_possunwant	warn_ptr_int_conv	warn_resultnotused	warn_undefmacro
warn_uninitializedvar	warn_possiblyuninitializedvar	warn_unusedarg	warn_unusedvar

28.1 extended_errorcheck

Controls the issuing of warning messages for possible unintended logical errors.

Syntax

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

Remarks

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

- An integer or floating-point value assigned to an `enum` type. The following listing shows an example.
  Listing: Assigning to an Enumerated Type

```
enum Day { Sunday, Monday, Tuesday, Wednesday,
           Thursday, Friday, Saturday } d;

d = 5; /* WARNING */

d = Monday; /* OK */

d = (Day)3; /* OK */
```

- An empty `return` statement in a function that is not declared `void`. For example, the following listing results in a warning message.

  Listing: A non-void function with an empty return statement

```
int MyInit(void)
{

  int err = GetMyResources();

  if (err != -1)

  {

    err = GetMoreResources();

  }

  return; /* WARNING: empty return statement */

}
```

  The following listing shows how to prevent this warning message.

  Listing: A non-void function with a proper return statement

```
int MyInit(void)
{

  int err = GetMyResources();

  if (err != -1)

  {

    err = GetMoreResources();

  }

  return err; /* OK */

}
```

28.2 maxerrorcount

Limits the number of error messages emitted while compiling a single file.

Syntax

```
#pragma maxerrorcount( num | off )
```

Parameters

num

Specifies the maximum number of error messages issued per source file.

`off`

Does not limit the number of error messages issued per source file.

Remarks

The total number of error messages emitted may include one final message:

```
Too many errors emitted
```

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

28.3 message

Tells the compiler to issue a text message to the user.

Syntax

```
#pragma message( msg )
```

Parameter

msg

Actual message to issue. Does not have to be a string literal.

Remarks

On the command line, the message is sent to the standard error stream.

28.4 showmessagenumber

Controls the appearance of warning or error numbers in displayed messages.

Syntax

```
#pragma showmessagenumber on | off | reset
```

Remarks

When enabled, this pragma causes messages to appear with their numbers visible. You can then use the warning pragma with a warning number to suppress the appearance of specific warning messages.

This pragma does not correspond to any panel setting. By default, this pragma is off.

28.5 show_error_filestack

Controls the appearance of the current # include file stack within error messages occurring inside deeply-included files.

Syntax

```
#pragma show_error_filestack on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is on.

28.6 suppress_warnings

Controls the issuing of warning messages.

Syntax

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

Remarks

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

This pragma does not correspond to any panel setting. By default, this pragma is off.

28.7 sym

Controls the generation of debugger symbol information for subsequent functions.

Syntax

```
#pragma sym on | off | reset
```

Remarks

The compiler pays attention to this pragma only if you enable the debug marker for a file in the IDE project window. If you disable this pragma, the compiler does not put debugging information into the source file debugger symbol file (SYM or DWARF) for the functions that follow.

The compiler always generates a debugger symbol file for a source file that has a debug diamond next to it in the IDE project window. This pragma changes only which functions have information in that symbol file.

This pragma does not correspond to any panel setting. By default, this pragma is enabled.

28.8 unused

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

Syntax

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

var_name

The name of a variable.

Remarks

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

In C++, you cannot use this pragma with functions defined within a class definition or with template functions.

Listing: Example of Pragma unused() in C

```
#pragma warn_unusedvar on
#pragma warn_unusedarg on

static void ff(int a)

{

  int b;

#pragma unused(a,b)

/* Compiler does not warn that a and b are unused. */
```

```
}
```

Listing: Example of Pragma unused() in C++

```
#pragma warn_unusedvar on
#pragma warn_unusedarg on

static void ff(int /* No warning */)

{

  int b;

#pragma unused(b)

/* Compiler does not warn that b is unused. */

}
```

This pragma does not correspond to any CodeWarrior IDE panel setting.

28.9 warning

Controls which warning numbers are displayed during compiling.

Syntax

```
#pragma warning on | off | reset (num [, ...])
```

This alternate syntax is allowed but ignored (message numbers do not match):

```
#pragma warning(warning_type : warning_num_list [, warning_type: warning_num_list,    ...])
```

Parameters

num

The number of the warning message to show or suppress.

warning_type

Specifies one of the following settings:

- `default`
- `disable`
- `enable`

warning_num_list

The *warning_num_list* is a list of warning numbers separated by spaces.

Remarks

Use the pragma `showmessagenumber` to display warning messages with their warning numbers.

This pragma only applies to CodeWarrior front-end warnings. Using the pragma for the Power Architecture back-end warnings returns invalid message number warning.

The CodeWarrior compiler allows, but ignores, the alternative syntax for compatibility with Microsoft® compilers.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

28.10 warning_errors

Controls whether or not warnings are treated as errors.

Syntax

```
#pragma warning_errors on | off | reset
```

Remarks

If you enable this pragma, the compiler treats all warning messages as though they were errors and does not translate your file until you resolve them.

28.11 warn_any_ptr_int_conv

Controls if the compiler generates a warning message when an integral type is explicitly converted to a pointer type or vice versa.

Syntax

```
#pragma warn_any_ptr_int_conv on | off | reset
```

Remarks

This pragma is useful to identify potential 64-bit pointer portability issues. An example is shown in.

Listing: Example of warn_any_ptr_int_conv

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

```
void func() {

   i = (short)ip;

   /* WARNING: short type is not large enough to hold pointer. */

}

#pragma warn_any_ptr_int_conv on

void bar() {

   i  = (int)ip; /* WARNING: pointer to integral conversion. */

   ip = (short *)i; /* WARNING: integral to pointer conversion. */

}
```

28.12 warn_emptydecl

Controls the recognition of declarations without variables.

Syntax

```
#pragma warn_emptydecl on | off | reset
```

Remarks

If you enable this pragma, the compiler displays a warning message when it encounters a declaration with no variables.

Listing: Examples of empty declarations in C and C++

```
#pragma warn_emptydecl on
int ; /* WARNING: empty variable declaration. */

int i; /* OK */


long j;; /* WARNING */

long j; /* OK */
```

Listing: Example of empty declaration in C++

```
#pragma warn_emptydecl on
extern "C" {

}; /* WARNING */
```

28.13 warn_extracomma

Controls the recognition of superfluous commas in enumerations.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a trailing comma in enumerations. For example, the following listing is acceptable source code but generates a warning message when you enable this setting.

Listing: Warning about extra commas

```
#pragma warn_extracomma on
enum { mouse, cat, dog, };

/* WARNING: compiler expects an identifier after final comma. */
```

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

This pragma corresponds to the **Extra Commas** setting in the CodeWarrior IDE's **C/C++ Warnings** settings panel. By default, this pragma is disabled.

28.14 warn_filenamecaps

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

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message when an `#include` directive capitalizes a filename within a user include differently from the way the filename appears on a disk. It also detects use of "8.3" DOS filenames in Windows® operating systems when a long filename is available. Use this pragma to avoid porting problems to operating systems with case-sensitive file names.

By default, this pragma only checks the spelling of user includes such as the following:

```
#include "file"
```

For more information on checking system includes, see `warn_filenamecaps_system`.

28.15 warn_filenamecaps_system

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

Syntax

```
#pragma warn_filenamecaps_system on | off | reset
```

Remarks

If you enable this pragma along with `warn_filenamecaps`, the compiler issues a warning message when an `#include` directive capitalizes a filename within a system include differently from the way the filename appears on a disk. It also detects use of "8.3" DOS filenames in Windows® systems when a long filename is available. This pragma helps avoid porting problems to operating systems with case-sensitive file names.

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

```
#include <file>
```

Use this pragma along with the warn_filenamecaps pragma.

NOTE

Some SDKs (Software Developer Kits) use "colorful" capitalization, so this pragma may issue a lot of unwanted messages.

28.16 warn_hiddenlocals

Controls the recognition of a local variable that hides another local variable.

Syntax

```
#pragma warn_hiddenlocals on | off | reset
```

Remarks

When `on`, the compiler issues a warning message when it encounters a local variable that hides another local variable. An example appears in the following listing.

Listing: Example of hidden local variables warning

```
#pragma warn_hiddenlocals on
void func(int a)

{

   {

      int a; /* WARNING: this 'a' obscures argument 'a'.

   }

}
```

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this setting is `off`.

28.17 warn_illpragma

Controls the recognition of invalid pragma directives.

Syntax

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

Remarks

If you enable this pragma, the compiler displays a warning message when it encounters a pragma it does not recognize.

28.18 warn_illtokenpasting

Controls whether or not to issue a warning message for improper preprocessor token pasting.

Syntax

```
#pragma warn_illtokenpasting on | off | reset
```

Remarks

An example of this is shown below:

```
#define PTR(x) x##* / PTR(y)
```

Token pasting is used to create a single token. In this example, `y` and `x` cannot be combined. Often the warning message indicates the macros uses "`##`" unnecessarily.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

28.19 warn_illunionmembers

Controls whether or not to issue a warning message for invalid union members, such as unions with reference or non-trivial class members.

Syntax

```
#pragma warn_illunionmembers on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

28.20 warn_impl_f2i_conv

Controls the issuing of warning messages for implicit `float`-to- `int` conversions.

Syntax

```
#pragma warn_impl_f2i_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting floating-point values to integral values. The following listing provides an example.

Listing: Example of Implicit float-to-int Conversion

```
#pragma warn_impl_f2i_conv on
float f;

signed int si;

int main()

{

    f  = si; /* WARNING */

#pragma warn_impl_f2i_conv off

    si = f; /* OK */

}
```

28.21 warn_impl_i2f_conv

Controls the issuing of warning messages for implicit `int`-to- `float` conversions.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting integral values to floating-point values. The following listing shows an example.

Listing: Example of implicit int-to-float conversion

```
#pragma warn_impl_i2f_conv on
float f;

signed int si;

int main()

{

    si = f; /* WARNING */

#pragma warn_impl_i2f_conv off

    f  = si; /* OK */

}
```

28.22 warn_impl_s2u_conv

Controls the issuing of warning messages for implicit conversions between the `signed int` and `unsigned int` data types.

Syntax

```
#pragma warn_impl_s2u_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting either from `signed int` to `unsigned int` or vice versa. The following listing provides an example.

Listing: Example of implicit conversions between signed int and unsigned int

```
#pragma warn_impl_s2u_conv on
signed int si;
```

```
unsigned int ui;

int main()

{

    ui = si; /* WARNING */

    si = ui; /* WARNING */

#pragma warn_impl_s2u_conv off

    ui = si; /* OK */

    si = ui; /* OK */

}
```

28.23 warn_implicitconv

Controls the issuing of warning messages for all implicit arithmetic conversions.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message for all implicit arithmetic conversions when the destination type might not represent the source value. The following listing provides an example.

Listing: Example of Implicit Conversion

```
#pragma warn_implicitconv on
float f;

signed int si;

unsigned int ui;

int main()

{

    f  = si; /* WARNING */

    si = f; /* WARNING */

    ui = si; /* WARNING */

    si = ui; /* WARNING */

}
```

NOTE

This option "opens the gate" for the checking of implicit conversions. The sub-pragmas `warn_impl_f2i_conv`, `warn_impl_i2f_conv`, and `warn_impl_s2u_conv` control the classes of conversions checked.

28.24 warn_largeargs

Controls the issuing of warning messages for passing non-"int" numeric values to unprototyped functions.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message if you attempt to pass a non-integer numeric value, such as a `float` or `long long`, to an unprototyped function when the `require_prototypes` pragma is disabled.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

28.25 warn_missingreturn

Issues a warning message when a function that returns a value is missing a `return` statement.

Syntax

```
#pragma warn_missingreturn on | off | reset
```

Remarks

An example is shown in the following figure.

Listing: Example of warn_missingreturn pragma

```
#pragma warn_missingreturn on
int func()

{

  /* WARNING: no return statement. */
```

```
}
```

28.26 warn_no_side_effect

Controls the issuing of warning messages for redundant statements.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a statement that produces no side effect. To suppress this warning message, cast the statement with `(void)`. The following listing provides an example.

Listing: Example of Pragma warn_no_side_effect

```
#pragma warn_no_side_effect on
void func(int a,int b)

{

   a+b; /* WARNING: expression has no side effect */

   (void)(a+b); /* OK: void cast suppresses warning. */

}
```

28.27 warn_padding

Controls the issuing of warning messages for data structure padding.

Syntax

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

Remarks

If you enable this pragma, the compiler warns about any bytes that were implicitly added after an ANSI C `struct` member to improve memory alignment. Refer to the appropriate *Targeting* manual for more information on how the compiler pads data structures for a particular processor or operating system.

This pragma corresponds to the **Pad Bytes Added** setting in the CodeWarrior IDE's **C/C++ Warnings** settings panel. By default, this setting is `off`.

28.28 warn_pch_portability

Controls whether or not to issue a warning message when `#pragmaonceon` is used in a precompiled header.

Syntax

```
#pragma warn_pch_portability on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when you use `#pragma once on` in a precompiled header. This helps you avoid situations in which transferring a precompiled header from machine to machine causes the precompiled header to produce different results. For more information, see pragma `once`.

This pragma does not correspond to any panel setting. By default, this setting is `off`.

28.29 warn_possunwant

Controls the recognition of possible unintentional logical errors.

Syntax

```
#pragma warn_possunwant on | off | reset
```

Remarks

If you enable this pragma, the compiler checks for common, unintended logical errors:

- An assignment in either a logical expression or the conditional portion of an `if`, `while`, or `for` expression. This warning message is useful if you use `=` when you mean to use `==`. The following listing shows an example.
  Listing: Confusing = and == in Comparisons

  ```
  if (a=b) f(); /* WARNING: a=b is an assignment. */
  if ((a=b)!=0) f(); /* OK: (a=b)!=0 is a comparison. */

  if (a==b) f(); /* OK: (a==b) is a comparison. */
  ```
- An equal comparison in a statement that contains a single expression. This check is useful if you use `==` when you meant to use `=`. The following listing shows an example.

Listing: Confusing = and == Operators in Assignments

```
a == 0;         // WARNING: This is a comparison.
a = 0;          // OK: This is an assignment, no warning
```

- A semicolon (;) directly after a `while`, `if`, or `for` statement.

  For example, The following listing generates a warning message.

Listing: Empty statement

```
i = sockcount();
while (--i);  /* WARNING: empty loop. */

    matchsock(i);
```

If you intended to create an infinite loop, put white space or a comment between the `while` statement and the semicolon. The statements in the following suppress the above error or warning messages.

Listing: Intentional empty statements

```
while (i++) ; /* OK: White space separation. */
while (i++) /* OK: Comment separation */ ;
```

28.30 warn_ptr_int_conv

Controls the recognition the conversion of pointer values to incorrectly-sized integral values.

Syntax

```
#pragma warn_ptr_int_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message if an expression attempts to convert a pointer value to an integral type that is not large enough to hold the pointer value.

Listing: Example for #pragma warn_ptr_int_conv

```
#pragma warn_ptr_int_conv on
char *my_ptr;

char too_small = (char)my_ptr;  /* WARNING: char is too small. */
```

See also warn_any_ptr_int_conv.

28.31 warn_resultnotused

Controls the issuing of warning messages when function results are ignored.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a statement that calls a function without using its result. To prevent this, cast the statement with `(void)`. The following listing provides an example.

Listing: Example of Function Calls with Unused Results

```
#pragma warn_resultnotused on
extern int bar();

void func()

{

   bar(); /* WARNING: result of function call is not used. */

   void(bar()); /* OK: void cast suppresses warning. */

}
```

This pragma does not correspond to any panel setting. By default, this pragma is off.

28.32 warn_undefmacro

Controls the detection of undefined macros in `#if` and `#elif` directives.

Syntax

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

Remarks

The following listing provides an example.

Listing: Example of Undefined Macro

```
#if BADMACRO == 4 /* WARNING: undefined macro. */
```

Use this pragma to detect the use of undefined macros (especially expressions) where the default value 0 is used. To suppress this warning message, check if macro defined first.

NOTE

A warning message is only issued when a macro is evaluated. A short-circuited "`&&`" or "`||`" test or unevaluated "`?:`" will not produce a warning message.

28.33 warn_uninitializedvar

Controls the compiler to perform some dataflow analysis and emits a warning message whenever there is a usage of a local variable and no path exists from any initialization of the same local variable.

Usages will not receive a warning if the variable is initialized along any path to the usage, even though the variable may be uninitialized along some other path. `warn_possiblyunitializedvar` pragma is introduced for such cases. Refer to pragma warn_possiblyuninitializedvar for more details.

Syntax

```
#pragma warn_uninitializedvar on | off | reset
```

Remarks

This pragma has no corresponding setting in the CodeWarrior IDE. By default, this pragma is `on`.

28.34 warn_possiblyuninitializedvar

It is a distinct pragma from `warn_uninitializedvar`, which uses a slightly different process to detect the uninitialized variables.

It will give a warning whenever local variables are used before being initialized along any path to the usage. As a result, you get more warnings.

However, some of the warnings will be false ones. The warnings will be false when all of the paths with uninitialized status turn out to be paths that can never actually be taken.

Syntax

```
#pragma warn_possiblyuninitializedvar on | off | reset
```

Remarks

This pragma has no corresponding setting in the CodeWarrior IDE. By default, this pragma is `off`.

NOTE

`warn_possiblyuninitializedvar` is superset of `warn_uninitializedvar`.

28.35 warn_unusedarg

Controls the recognition of unreferenced arguments.

Syntax

```
#pragma warn_unusedarg on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters an argument you declare but do not use.

This check helps you find arguments that you either misspelled or did not use in your program. The following listing shows an example.

Listing: Warning about unused function arguments

```
void func(int temp, int error);
{

  error = do_something(); /* WARNING: temp is unused. */

}
```

To prevent this warning, you can declare an argument in a few ways:

- Use the pragma `unused`, as in the following listing.
  Listing: Using pragma unused() to prevent unused argument messages

  ```
  void func(int temp, int error)
  {

    #pragma unused (temp)

    /* Compiler does not warn that temp is not used. */

    error=do_something();

  }
  ```
- Do not give the unused argument a name. The following listing shows an example.

  The compiler allows this feature in C++ source code. To allow this feature in C source code, disable ANSI strict checking.

  Listing: Unused, Unnamed Arguments

```
void func(int /* temp */, int error)
{

  /* Compiler does not warn that "temp" is not used. */

  error=do_something();

}
```

28.36 warn_unusedvar

Controls the recognition of unreferenced variables.

Syntax

```
#pragma warn_unusedvar on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a variable you declare but do not use.

This check helps you find variables that you either misspelled or did not use in your program. The following listing shows an example.

Listing: Unused Local Variables Example

```
int error;
void func(void)

{

  int temp, errer; /* NOTE: errer is misspelled. */

  error = do_something(); /* WARNING: temp and errer are unused. */

}
```

If you want to use this warning but need to declare a variable that you do not use, include the pragma `unused`, as in the following listing.

Listing: Suppressing Unused Variable Warnings

```
void func(void)
{

  int i, temp, error;

  #pragma unused (i, temp) /* Do not warn that i and temp */

  error = do_something();    /* are not used */

}
```

Chapter 29
Pragmas for Preprocessing

This chapter lists the following pragmas options for preprocessing:

Table 29-1. Pragmas for Preprocessing

check_header_flags	faster_pch_gen	flat_include
fullpath_file	fullpath_prepdump	keepcomments
line_prepdump	macro_prepdump	msg_show_lineref
msg_show_realref	notonce	old_pragma_once
once	pop, push	pragma_prepdump
precompile_target	simple_prepdump	space_prepdump
srcrelincludes	syspath_once	

29.1 check_header_flags

Controls whether or not to ensure that a precompiled header's data matches a project's target settings.

Syntax

```
#pragma check_header_flags on | off | reset
```

Remarks

This pragma affects precompiled headers only.

If you enable this pragma, the compiler verifies that the precompiled header's preferences for `double` size, `int` size, and floating point math correspond to the build target's settings. If they do not match, the compiler generates an error message.

If your precompiled header file depends on these settings, enable this pragma. Otherwise, disable it.

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `off`.

29.2 faster_pch_gen

Controls the performance of precompiled header generation.

Syntax

```
#pragma faster_pch_gen on | off | reset
```

Remarks

If you enable this pragma, generating a precompiled header can be much faster, depending on the header structure. However, the precompiled file can also be slightly larger.

This pragma does not correspond to any panel setting. By default, this setting is `off`.

29.3 flat_include

Controls whether or not to ignore relative path names in `#include` directives.

Syntax

```
#pragma flat_include on | off | reset
```

Remarks

For example, when `on`, the compiler converts this directive

```
#include <sys/stat.h>
```

to

```
#include <stat.h>
```

Use this pragma when porting source code from a different operating system, or when a CodeWarrior IDE project's access paths cannot reach a given file.

By default, this pragma is `off`.

29.4 fullpath_file

Controls if __FILE__ macro expands to a full path or the base file name.

Syntax

```
#pragma fullpath_file on | off | reset
```

Remarks

When this pragma on, the __FILE__ macro returns a full path to the file being compiled, otherwise it returns the base file name.

29.5 fullpath_prepdump

Shows the full path of included files in preprocessor output.

Syntax

```
#pragma fullpath_prepdump on | off | reset
```

Remarks

If you enable this pragma, the compiler shows the full paths of files specified by the #include directive as comments in the preprocessor output. Otherwise, only the file name portion of the path appears.

29.6 keepcomments

Controls whether comments are emitted in the preprocessor output.

Syntax

```
#pragma keepcomments on | off | reset
```

Remarks

This pragma corresponds to the **Keep comments** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is off.

29.7 line_prepdump

Shows `#line` directives in preprocessor output.

Syntax

```
#pragma line_prepdump on | off | reset
```

Remarks

If you enable this pragma, `#line` directives appear in preprocessing output. The compiler also adjusts line spacing by inserting empty lines.

Use this pragma with the command-line compiler's `-E` option to make sure that `#line` directives are inserted in the preprocessor output.

This pragma corresponds to the **Use #line** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

29.8 macro_prepdump

Controls whether the compiler emits `#define` and `#undef` directives in preprocessing output.

Syntax

```
#pragma macro_prepdump on | off | reset
```

Remarks

Use this pragma to help unravel confusing problems like macros that are aliasing identifiers or where headers are redefining macros unexpectedly.

29.9 msg_show_lineref

Controls diagnostic output involving `#line` directives to show line numbers specified by the `#line` directives in error and warning messages.

Syntax

```
#pragma msg_show_lineref on | off | reset
```

Remarks

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `on`.

29.10 msg_show_realref

Controls diagnostic output involving `#line` directives to show actual line numbers in error and warning messages.

Syntax

```
#pragma msg_show_realref on | off | reset
```

Remarks

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `on`.

29.11 notonce

Controls whether or not the compiler lets included files be repeatedly included, even with `#pragma once on`.

Syntax

```
#pragma notonce
```

Remarks

If you enable this pragma, files can be repeatedly `#include`d, even if you have enabled `#pragma once on`. For more information, refer to the pragma once.

This pragma does not correspond to any CodeWarrior IDE panel setting.

29.12 old_pragma_once

This pragma is no longer available.

29.13 once

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

Syntax

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

Remarks

Use this pragma to ensure that the compiler includes header files only once in a source file. This pragma is especially useful in precompiled header files.

There are two versions of this pragma:

```
#pragma once
```

and

```
#pragma once on
```

Use `#pragma once` in a header file to ensure that the header file is included only once in a source file. Use `#pragma once on` in a header file or source file to ensure that *any* file is included only once in a source file.

Beware that when using `#pragma once on`, precompiled headers transferred from one host machine to another might not give the same results during compilation. This inconsistency is because the compiler stores the full paths of included files to distinguish between two distinct files that have identical file names but different paths. Use the `warn_pch_portability` pragma to issue a warning message when you use `#pragma once on` in a precompiled header.

Also, if you enable the `old_pragma_once on` pragma, the `once` pragma completely ignores path names.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

29.14 pop, push

Saves and restores pragma settings.

Syntax

```
#pragma push
```

```
#pragma pop
```

Remarks

The pragma `push` saves all the current pragma settings. The pragma `pop` restores all the pragma settings that resulted from the last `push` pragma. For example, see in the following listing.

Listing: push and pop example

```
#pragma ANSI_strict on
#pragma push /* Saves all compiler settings. */

#pragma ANSI_strict off

#pragma pop /* Restores ANSI_strict to on. */
```

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

Tip

Pragmas directives that accept `on` | `off` | `reset` already form a stack of previous option values. It is not necessary to use `#pragma pop` or `#pragma push` with such pragmas.

29.15 pragma_prepdump

Controls whether pragma directives in the source text appear in the preprocessing output.

Syntax

```
#pragma pragma_prepdump on | off | reset
```

29.16 precompile_target

Specifies the file name for a precompiled header file.

Syntax

```
#pragma precompile_target filename
```

Parameters

filename

A simple file name or an absolute path name. If *filename* is a simple file name, the compiler saves the file in the same folder as the source file. If *filename* is a path name, the compiler saves the file in the specified folder.

Remarks

If you do not specify the file name, the compiler gives the precompiled header file the same name as its source file.

The following listing shows sample source code from a precompiled header source file. By using the predefined symbols `__cplusplus` and the pragma `precompile_target`, the compiler can use the same source code to create different precompiled header files for C and C++.

Listing: Using #pragma precompile_target

```
#ifdef __cplusplus
  #pragma precompile_target "MyCPPHeaders"

#else

  #pragma precompile_target "MyCHeaders"

#endif
```

This pragma does not correspond to any panel setting.

29.17 simple_prepdump

Controls the suppression of comments in preprocessing output.

Syntax

```
  #pragma simple_prepdump on | off | reset
```

Remarks

By default, the compiler adds comments about the current include file being in preprocessing output. Enabling this pragma disables these comments.

29.18 space_prepdump

Controls whether or not the compiler removes or preserves whitespace in the preprocessor's output.

Syntax

```
#pragma space_prepdump on | off | reset
```

Remarks

This pragma is useful for keeping the starting column aligned with the original source code, though the compiler attempts to preserve space within the line. This pragma does not apply to expanded macros.

29.19 srcrelincludes

Controls the lookup of `#include` files.

Syntax

```
#pragma srcrelincludes on | off | reset
```

Remarks

When `on`, the compiler looks for `#include` files relative to the previously included file (not just the source file). When `off`, the compiler uses the CodeWarrior IDE's access paths or the access paths specified with the `-ir` option.

Use this pragma when multiple files use the same file name and are intended to be included by another header file in that directory. This is a common practice in UNIX programming.

29.20 syspath_once

Controls how included files are treated when `#pragmaonce` is enabled.

Syntax

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

Remarks

When this pragma and pragma `once` are set to `on`, the compiler distinguishes between identically-named header files referred to in `#include <file-name>` and `#include "file-name"`.

When this pragma is `off` and pragma once is `on`, the compiler will ignore a file that uses a

```
#include <file-name>
```

directive if it has previously encountered another directive of the form

```
#include "file-name"
```

for an identically-named header file.

shows an example.

This pragma does not correspond to any panel setting. By default, this setting is `on`.

Listing: Pragma syspath_once example

```
#pragma syspath_once off
#pragma once on /* Include all subsequent files only once. */

#include "sock.h"

#include <sock.h> /* Skipped because syspath_once is off. */
```

Chapter 30
Pragmas for Code Generation

This chapter lists the following pragmas for code generation:

Table 30-1. Pragmas for Code Generation

aggressive_inline	dont_reuse_strings	enumsalwaysint
errno_name	explicit_zero_data	float_constants
instmgr_file	longlong	longlong_enums
min_enum_size	pool_strings	readonly_strings
reverse_bitfields	store_object_files	

30.1 aggressive_inline

Specifies the size of enumerated types.

Syntax

```
#pragma aggressive_inline on | off | reset
```

Remarks

The IPA-based inliner (`-ipa` file) will inline more functions when this option is enabled. This option can cause code bloat in programs that overuse inline functions. Default is off.

30.2 dont_reuse_strings

Controls whether or not to store identical character string literals separately in object code.

Syntax

```
#pragma dont_reuse_strings on | off | reset
```

Remarks

Normally, C and C++ programs should not modify character string literals. Enable this pragma if your source code follows the unconventional practice of modifying them.

If you enable this pragma, the compiler separately stores identical occurrences of character string literals in a source file.

If this pragma is disabled, the compiler stores a single instance of identical string literals in a source file. The compiler reduces the size of the object code it generates for a file if the source file has identical string literals.

The compiler always stores a separate instance of a string literal that is used to initialize a character array. The code listed below shows an example.

Although the source code contains 3 identical string literals, `"cat"`, the compiler will generate 2 instances of the string in object code. The compiler will initialize `str1` and `str2` to point to the first instance of the string and will initialize `str3` to contain the second instance of the string.

Using `str2` to modify the string it points to also modifies the string that `str1` points to. The array `str3` may be safely used to modify the string it points to without inadvertently changing any other strings.

Listing: Reusing string literals

```
#pragma dont_reuse_strings off
void strchange(void)

{

  const char* str1 = "cat";

  char* str2 = "cat";

  char str3[] = "cat";

  *str2 = 'h'; /* str1 and str2 point to "hat"! */

  str3[0] = 'b';

  /* OK: str3 contains "bat", *str1 and *str2 unchanged.

}
```

30.3 enumsalwaysint

Specifies the size of enumerated types.

Syntax

```
#pragma enumsalwaysint on | off | reset
```

Remarks

If you enable this pragma, the C/C++ compiler makes an enumerated type the same size as an `int`. If an enumerated constant is larger than `int`, the compiler generates an error message. Otherwise, the compiler makes an enumerated type the size of any integral type. It chooses the integral type with the size that most closely matches the size of the largest enumerated constant. The type could be as small as a `char` or as large as a `long long`.

The following listing shows an example.

Listing: Example of Enumerations the Same as Size as int

```
enum SmallNumber { One = 1, Two = 2 };
  /* If you enable enumsalwaysint, this type is

     the same size as an int. Otherwise, this type is

     the same size as a char. */

enum BigNumber

  { ThreeThousandMillion = 3000000000 };

  /* If you enable enumsalwaysint, the compiler might

     generate an error message. Otherwise, this type is

     the same size as a long long. */
```

30.4 errno_name

Tells the optimizer how to find the `errno` identifier.

Syntax

```
#pragma errno_name id | ...
```

Remarks

When this pragma is used, the optimizer can use the identifier `errno` (either a macro or a function call) to optimize standard C library functions better. If not used, the optimizer makes worst-case assumptions about the effects of calls to the standard C library.

NOTE

The EWL C library already includes a use of this pragma, so you would only need to use it for third-party C libraries.

If `errno` resolves to a variable name, specify it like this:

```
#pragma errno_name _Errno
```

If `errno` is a function call accessing ordinarily inaccessible global variables, use this form:

```
#pragma errno_name ...
```

Otherwise, do not use this pragma to prevent incorrect optimizations.

This pragma does not correspond to any panel setting. By default, this pragma is unspecified (worst case assumption).

30.5 explicit_zero_data

Controls the placement of zero-initialized data.

Syntax

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

Remarks

Places zero-initialized data into the initialized data section instead of the BSS section when `on`.

By default, this pragma is `off`.

30.6 float_constants

Controls how floating pointing constants are treated.

Syntax

```
#pragma float_constants on | off | reset
```

Remarks

If you enable this pragma, the compiler assumes that all unqualified floating point constant values are of type `float`, not `double`. This pragma is useful when porting source code for programs optimized for the " `float`" rather than the " `double`" type.

When you enable this pragma, you can still explicitly declare a constant value as double by appending a "D" suffix.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

30.7 instmgr_file

Controls where the instance manager database is written, to the target data directory or to a separate file.

Syntax

```
#pragma instmgr_file "name"
```

Remarks

When the **Use Instance Manager** option is on, the IDE writes the instance manager database to the project's data directory. If the `#pragma instmgr_file` is used, the database is written to a separate file.

Also, a separate instance file is always written when the command-line tools are used.

NOTE

Should you need to report a bug, you can use this option to create a separate instance manager database, which can then be sent to technical support with your bug report.

30.8 longlong

Controls the availability of the `long long` type.

Syntax

```
#pragma longlong on | off | reset
```

Remarks

When this pragma is enabled and the compiler is translating "C90" source code (ISO/IEC 9899-1990 standard), the compiler recognizes a data type named `long long`. The `long long` type holds twice as many bits as the `long` data type.

This pragma does not correspond to any CodeWarrior IDE panel setting.

By default, this pragma is on for processors that support this type. It is `off` when generating code for processors that do not support, or cannot turn on, the `long long` type.

30.9 longlong_enums

Controls whether or not enumerated types may have the size of the `long long` type.

Syntax

```
#pragma longlong_enums on | off | reset
```

Remarks

This pragma lets you use enumerators that are large enough to be `long long` integers. It is ignored if you enable the enumsalwaysint pragma.

This pragma does not correspond to any panel setting. By default, this setting is enabled.

30.10 min_enum_size

Specifies the size, in bytes, of enumeration types.

Syntax

```
#pragma min_enum_size 1 | 2 | 4
```

Remarks

Turning on the `enumsalwaysint` pragma overrides this pragma. The default is 1.

30.11 pool_strings

Controls how string literals are stored.

Syntax

```
#pragma pool_strings on | off | reset
```

Remarks

If you enable this pragma, the compiler collects all string constants into a single data object so your program needs one data section for all of them. If you disable this pragma, the compiler creates a unique data object for each string constant. While this decreases the number of data sections in your program, on some processors it also makes your program bigger because it uses a less efficient method to store the address of the string.

This pragma is especially useful if your program is large and has many string constants or uses the CodeWarrior Profiler.

NOTE

If you enable this pragma, the compiler ignores the setting of the `pcrelstrings` pragma.

30.12 readonly_strings

Controls whether string objects are placed in a read-write or a read-only data section.

Syntax

```
#pragma readonly_strings on | off | reset
```

Remarks

If you enable this pragma, literal strings used in your source code are output to the read-only data section instead of the global data section. In effect, these strings act like `constchar*`, even though their type is really `char*`.

This pragma does not correspond to any IDE panel setting.

30.13 reverse_bitfields

Controls whether or not the compiler reverses the bitfield allocation.

Syntax

```
#pragma reverse_bitfields on | off | reset
```

Remarks

This pragma reverses the bitfield allocation, so that bitfields are arranged from the opposite side of the storage unit from that ordinarily used on the target. The compiler still orders the bits within a single bitfield such that the lowest-valued bit is in the right-most position.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

Limitation

Please be aware of the following limitations when this pragma is set to `on`:

- The data types of the bit-fields must be the same data type.
- The structure (`struct`) or `class` must not contain non-bit-field members; however, the structure (`struct`) can be the member of another structure.

30.14 store_object_files

Controls the storage location of object data, either in the target data directory or as a separate file.

Syntax

```
#pragma store_object_files on | off | reset
```

Remarks

By default, the IDE writes object data to the project's target data directory. When this pragma is on, the object data is written to a separate object file.

NOTE

For some targets, the object file emitted may not be recognized as actual object data.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

Chapter 31
Pragmas for Optimization

This chapter lists the following pragmas for optimization:

Table 31-1. Pragmas for Optimization

global_optimizer	ipa	ipa_inline_max_auto_size
ipa_not_complete	opt_common_subs	opt_dead_assignments
opt_dead_code	opt_lifetimes	opt_loop_invariants
opt_propagation	opt_strength_reduction	opt_strength_reduction_strict
opt_unroll_loops	opt_vectorize_loops	optimization_level
optimize_for_size	optimizewithasm	pack
strictheaderchecking		

31.1 global_optimizer

Controls whether the Global Optimizer is invoked by the compiler.

Syntax

```
#pragma global_optimizer on | off | reset
```

Remarks

In most compilers, this #pragma determines whether the Global Optimizer is invoked (configured by options in the panel of the same name). If disabled, only simple optimizations and back-end optimizations are performed.

NOTE

This is not the same as `#pragmaoptimization_level`. The Global Optimizer is invoked even at `optimization_level0` if `#pragmaglobal_optimizer` is enabled.

31.2 ipa

Specifies how to apply interprocedural analysis optimizations.

Syntax

```
#pragma ipa program | file | on | function | off
```

Remarks

See Interprocedural Analysis.

Place this pragma at the beginning of a source file, before any functions or data have been defined. There are three levels of interprocedural analysis:

- program-level: the compiler translates all source files in a program then optimizes object code for the entire program
- file-level: the compiler translates each file and applies this optimization to the file
- function-level: the compiler does not apply interprocedural optimization

The options `file` and `on` are equivalent. The options `function` and `off` are equivalent.

31.3 ipa_inline_max_auto_size

Determines the maximum complexity for an auto-inlined function.

Syntax

```
#pragma ipa_inline_max_auto_size (intval)
```

Parameters

`intval`

The `intval` value is an approximation of the number of statements in a function, the current default value is 500, which is approximately equal to 100 satement function. Selecting a zero value will disable the IPA auto inlining.

Remarks

The size of the code objects that are not referenced by address and are only called once is specified above a certain threshold using this pragma, preventing them from being marked as inline.

31.4 ipa_not_complete

Controls the usage of **Complete Program IPA** mode by the compiler.

Syntax

```
#pragma ipa_not_complete on | off | reset
```

Remarks

In **Complete Program IPA** mode, the compiler assumes that the IPA graph is complete and that there are no external entry points other than `main()`, static initialization or force export functions. The compiler will not use this mode if the

The **Complete Program IPA** mode is not used by the compiler if:

- the program has no `main()` and no force export functions.
- the pragma is on the context of `main()` or force export functions.

NOTE

The compiler will be more aggressive in the **Complete Program IPA** mode.

Any `extern` object that is not `main()`, static initialization code or `force export` and not directly or indirectly used, will be deadstipped by the compiler and will not appear in the object and/or executable files. By default, this setting is `off`.

31.5 opt_common_subs

Controls the use of common subexpression optimization.

Syntax

```
#pragma opt_common_subs on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces similar redundant expressions with a single expression. For example, if two statements in a function both use the expression

```
a * b * c + 10
```

the compiler generates object code that computes the expression only once and applies the resulting value to both statements.

The compiler applies this optimization to its own internal representation of the object code it produces.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer pragma.

31.6 opt_dead_assignments

Controls the use of dead store optimization.

Syntax

```
#pragma opt_dead_assignments on | off | reset
```

Remarks

If you enable this pragma, the compiler removes assignments to unused variables before reassigning them.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

31.7 opt_dead_code

Controls the use of dead code optimization.

Syntax

```
#pragma opt_dead_code on | off | reset
```

Remarks

If you enable this pragma, the compiler removes a statement that other statements never execute or call.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

31.8 opt_lifetimes

Controls the use of lifetime analysis optimization.

Syntax

```
#pragma opt_lifetimes on | off | reset
```

Remarks

If you enable this pragma, the compiler uses the same processor register for different variables that exist in the same routine but not in the same statement.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

31.9 opt_loop_invariants

Controls the use of loop invariant optimization.

Syntax

```
#pragma opt_loop_invariants on | off | reset
```

Remarks

If you enable this pragma, the compiler moves all computations that do not change inside a loop to outside the loop, which then runs faster.

This pragma does not correspond to any panel setting.

31.10 opt_propagation

Controls the use of copy and constant propagation optimization.

Syntax

```
#pragma opt_propagation on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces multiple occurrences of one variable with a single occurrence.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

31.11 opt_strength_reduction

Controls the use of strength reduction optimization.

Syntax

```
#pragma opt_strength_reduction on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces array element arithmetic instructions with pointer arithmetic instructions to make loops faster.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

31.12 opt_strength_reduction_strict

Uses a safer variation of strength reduction optimization.

Syntax

```
#pragma opt_strength_reduction_strict on | off | reset
```

Remarks

Like the opt_strength_reduction pragma, this setting replaces multiplication instructions that are inside loops with addition instructions to speed up the loops. However, unlike the regular strength reduction optimization, this variation ensures that the optimization is only applied when the array element arithmetic is not of an unsigned type that is smaller than a pointer type.

This pragma does not correspond to any panel setting. The default varies according to the compiler.

31.13 opt_unroll_loops

Controls the use of loop unrolling optimization.

Syntax

```
#pragma opt_unroll_loops on | off | reset
```

Remarks

If you enable this pragma, the compiler places multiple copies of a loop's statements inside a loop to improve its speed.

This pragma does not correspond to any panel setting. By default, this settings is related to the global_optimizer level.

31.14 opt_vectorize_loops

Controls the use of loop vectorizing optimization.

Syntax

```
#pragma opt_vectorize_loops on | off | reset
```

Remarks

If you enable this pragma, the compiler improves loop performance.

NOTE

Do not confuse loop vectorizing with the vector instructions available in some processors. Loop vectorizing is the rearrangement of instructions in loops to improve performance. This optimization does not optimize a processor's vector data types.

By default, this pragma is `off`.

31.15 optimization_level

Controls global optimization.

Syntax

```
#pragma optimization_level 0 | 1 | 2 | 3 | 4 | reset
```

Remarks

This pragma specifies the degree of optimization that the global optimizer performs.

To select optimizations, use the pragma `optimization_level` with an argument from `0` to `4`. The higher the argument, the more optimizations performed by the global optimizer. The `reset` argument specifies the previous optimization level.

For more information on the optimization the compiler performs for each optimization level, refer to the *Targeting* manual for your target platform.

31.16 optimize_for_size

Controls optimization to reduce the size of object code.

```
#pragma optimize_for_size on | off | reset
```

Remarks

This setting lets you choose what the compiler does when it must decide between creating small code or fast code. If you enable this pragma, the compiler creates smaller object code at the expense of speed. It also ignores the `inline` directive and generates function calls to call any function declared `inline`. If you disable this pragma, the compiler creates faster object code at the expense of size.

31.17 optimizewithasm

Controls optimization of assembly language.

Syntax

```
#pragma optimizewithasm on | off | reset
```

Remarks

If you enable this pragma, the compiler also optimizes assembly language statements in C/C++ source code.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

31.18 pack

Stores data to reduce data size instead of improving execution performance.

Syntax

```
#pragma pack()

#pragma pack(0 | n | push | pop)
```

n

One of these integer values: 1, 2, 4, 8, or 16.

Remarks

Use this pragma to align data to use less storage even if the alignment might affect program performance or does not conform to the target platform's application binary interface (ABI).

If this pragma's argument is a power of 2 from 1 to 16, the compiler will store subsequent data structures to this byte alignment.

The `push` argument saves this pragma's setting on a stack at compile time. The `pop` argument restores the previously saved setting and removes it from the stack. Using this pragma with no argument or with 0 as an argument specifies that the compiler will use ABI-conformant alignment.

Not all processors support misaligned accesses, which could cause a crash or incorrect results. Even on processors which allow misaligned access, your program's performance might be reduced. Your program may have better performance if it treats the packed structure as a byte stream, then packs and unpacks each byte from the stream.

NOTE

Pragma `pack` is implemented somewhat differently by most compiler vendors, especially when used with bitfields. If you need portability, you are probably better off using explicit shift and mask operations in your program instead of bitfields.

31.19 strictheaderchecking

Controls how strict the compiler checks headers for standard C library functions.

Syntax

```
#pragma strictheaderchecking on | off | reset
```

Remarks

The 3.2 version compiler recognizes standard C library functions. If the correct prototype is used, and, in C++, if the function appears in the " `std`" or root namespace, the compiler recognizes the function, and is able to optimize calls to it based on its documented effects.

When this #pragma is on (default), in addition to having the correct prototype, the declaration must also appear in the proper standard header file (and not in a user header or source file).

This pragma does not correspond to any panel setting. By default, this pragma is on.

Chapter 32
ColdFire Pragmas

This chapter lists pragmas for the CodeWarrior compiler for ColdFire architectures.

32.1 ColdFire V1 Pragmas

This topic lists the following ColdFire V1 pragmas:

Table 32-1. ColdFire V1 Pragmas

aligncode	CODE_SEG	CONST_SEG
DATA_SEG	hw_longlong	opt_tail_call
near_rel_code	no_register_coloring	scheduling
STRING_SEG	opt_cse_calls	TRAP_PROC

32.1.1 aligncode

Syntax

```
#pragma aligncode on | off | reset
```

Description

The compiler, by default, pads .text sections to a multiple of 4 and sets these sections sh_addralign to 4 for performance reasons. Setting this pragma to off changes these values to 2.

32.1.2 CODE_SEG

Specifies the addressing mode and location of object code for functions.

Syntax

```
#pragma CODE_SEG [ modifier ] [ name ]

#pragma CODE_SEG DEFAULT
```

Parameters

`modifier`

This optional parameter specifies the addressing mode to use:

- `__NEAR_SEG`, `__SHORT_SEG`: use 16-bit addressing
- `__FAR_SEG`: use 32-bit addressing

`name`

A section name. You must use a section name defined in your project's linker command file.

Description

To specify the section in which to store a function and the addressing mode to refer to it, place this pragma before the function definition. Use this pragma when porting source code from HC08 architectures to ColdFire V1 architectures.

Use `DEFAULT` to use the parameters specified by the previous use of this pragma.

The following listing shows an example.

Listing: CODE_SEG example

```
void f(void);
void h(void);

/* Use far addressing to refer to this function. */

/* Store function in section "text". */

#pragma CODE_SEG __FAR_SEG text

void f(void){
```

```
h();
}
/* Use near addressing to refer to this function. */
/* Store function in section "MYCODE2". */
#pragma CODE_SEG __NEAR_SEG MYCODE2
void h(void){
f();
}
/* Use previous pragma CODE_SEG parameters: */
/* __FAR_SEG text */
#pragma CODE_SEG DEFAULT
```

32.1.3 CONST_SEG

Specifies the addressing mode and location of object code for constant data.

Syntax

```
#pragma CONST_SEG [ modifier ] [ name ]
#pragma CODE_SEG DEFAULT
```

Parameters

`modifier`

This optional parameter specifies the addressing mode to use:

- `__NEAR_SEG`, `__SHORT_SEG`: use 16-bit addressing
- `__FAR_SEG`: use 32-bit addressing

`name`

A section name. You must use a section name defined in your project's linker command file.

Description

To specify the section in which to store constant data and the addressing mode to refer to this data, place this pragma before definitions that define constant and literal values. Use this pragma when porting source code for HC08 architectures to ColdFire V1 architectures.

Use `DEFAULT` to use the parameters specified in the previous use of this pragma.

The following listing shows an example.

Listing: CONST_SEG example

```
/* Place socks_left in section rodata. */
const int socks_left = 20;

/* Place socks_right in section MYCONST. */

#pragma CONST_SEG MYCONST

int socks_right = 30;
```

32.1.4 DATA_SEG

Specifies the addressing mode and location of object code for variable data.

Syntax

```
#pragma DATA_SEG [ modifier ] [ name ]

#pragma DATA_SEG DEFAULT
```

Parameters

`modifier`

This optional parameter specifies the addressing mode to use:

- `__NEAR_SEG`, `__SHORT_SEG`: use 16-bit addressing
- `__FAR_SEG`: use 32-bit addressing

`name`

A section name. You must use a section name defined in your project's linker command file.

Description

To specify the section in which to store variable data and the addressing mode to refer to this data, place this pragma before variable definitions. Use this pragma when porting source code for HC08 architectures to ColdFire V1 architectures.

Use `DEFAULT` to use the parameters specified in the previous use of this pragma.

The following listing shows an example.

Listing: DATA_SEG example

```
/* Place socks in section sdata. */
struct {
```

```
   int left;
   int right;
} socks = { 1, 2 };
/* Place socks_total in section sbss.
int socks_total;
/* Place socks_flag in section MYDATA.
#pragma DATA_SEG MYDATA
int socks_flag;
```

32.1.5 hw_longlong

Syntax

```
pragma# hw_longlong on | off | reset
```

Description

This pragma controls inline long long arithmetic support, when turned off all long long arithmetic operations are performed via intrinsincs calls. Default is on.

32.1.6 opt_tail_call

Syntax

```
pragma# opt_tail_call on | off | reset
```

Description

This pragma is on when opt level is 2 or more, the compiler will replace tail calls with jmp

32.1.7 near_rel_code

Syntax

```
pragma# near_rel_code
```

Description

The compiler will use PC-relative addressing for all function calls unless the called function was declared far or near.

32.1.8 no_register_coloring

Syntax

```
pragma# no_register_coloring on | off | reset
```

Description

disables allocation of local variables to registers.

32.1.9 scheduling

Syntax

```
pragma# scheduling on | off | reset
```

Description

enables instruction scheduling for that processor, optimization level 2 and above.

32.1.10 STRING_SEG

Specifies the addressing mode and location of object code for constant character strings.

Syntax

```
#pragma STRING_SEG [ modifier ] [ name ]

#pragma STRING_SEG DEFAULT
```

Parameters

`modifier`

This optional parameter specifies the addressing mode to use:

- `__NEAR_SEG`, `__SHORT_SEG`: use 16-bit addressing
- `__FAR_SEG`: use 32-bit addressing

`name`

A section name. You must use a section name defined in your project's linker command file.

Description

To specify the section in which to store constant character strings and the addressing mode to refer to this data, place this pragma before character string literal values. Use this pragma when porting source code for HC08 architectures to ColdFire V1 architectures.

Use `DEFAULT` to use the parameters specified in the previous use of this pragma.

The following listing shows an example.

Listing: DATA_SEG example

```
/* Place "longitude" and "altitude" in section rodata. */
const char* s1 = "longitude";

char* s2 = "altitude";

/* Place "latitude" in section sdata. */

char s3[50] = "latitude";

#pragma STRING_SEG MYSTRINGS

/* Place "liberty" and "fraternity" in section MYSTRINGS. */

const char* s4 = "liberty";

char* s5 = "fraternity";

/* "equality" will go in section sdata. */

char s6[50] = "equality";
```

32.1.11 opt_cse_calls

Syntax

```
pragma# opt_cse_calls on | off | reset
```

Description

under optimize for size and for optimization levels 2 and above, the compiler will treat multiple calls to a function as a common sub-expression and replace occurrences with indirect calls.

32.1.12 TRAP_PROC

Generates instructions to allow a function to handle a processor exception.

Syntax

```
#pragma TRAP_PROC
```

Remarks

Functions that act as interrupt service routines require special executable code to handle function entry and exit. Use this pragma to specify to the compiler that the subsequent function defintion will be used to handle processor exceptions.

Listing: Using the TRAP_PROC Pragma to Mark an Interrupt Function

```
#include <hidef.h> /* For interrupt macro. */
#pragma TRAP_PROC

void MyInterrupt(void) {

  DisableInterrupts;

  /* ... */

  EnableInterrupts;

}
```

32.2 ColdFire Diagnostic Pragmas

This topic lists the ColdFire diagnostic pragmas.

- SDS_debug_support

32.2.1 SDS_debug_support

Deprecated.

32.3 ColdFire Library and Linking Pragmas

This topic lists the following ColdFire library and linking pragmas:

32.3.1 define_section

Specifies a predefined section or defines a new section for compiled object code.

```
#pragma define_section sname ".istr" [.ustr] [addrmode] [accmode]
```

Parameters

`sname`

Identifier for source references to this user-defined section. Make sure this name does not conflict with names recognized by the `__declspec` directive.

`istr`

Section-name string for *initialized* data assigned to this section. Double quotes must surround this parameter value, which must begin with a period. (Also applies to *uninitialized* data if there is no `ustr` value.)

`ustr`

Optional: ELF section name for uninitialized data assigned to this section. Must begin with a period. Default value is the `istr` value.

`addrmode`

Optional: any of these address-mode values:

- `standard` - 32-bit absolute address (default)
- `near_absolute` - 16-bit absolute address
- `far_absolute` - 32-bit absolute address
- `near_code` - 16-bit offset from the PC address
- `far_code` - 32-bit offset from the PC address
- `near_data` - 16-bit offset from the A5 register address
- `far_data` - 32-bit offset from the A5 register address

`accmode`

Optional: any of these letter combinations:

- `R` - readable
- `RW` - readable and writable

- `RX` - readable and executable
- `RWX` - readable, writable, and executable (default)

(No other letter orders are valid: WR, XR, or XRW would be an error.)

Remarks

The compiler predefines the common PowerPC sections that the following table lists.

Table 32-2. ColdFire Predefined Sections

Applicability	Definition Pragmas
Absolute Addressing Mode	`#pragma define_section text ".text" far_absolute RX`
	`#pragma define_section data ".data" ".bss" far_absolute RW`
	`#pragma define_section sdata ".sdata" ".sbss" near_data RW`
	`#pragma define_section const ".rodata" far_absolute R`
C++, Regardless of Addressing Mode	`#pragma define_section exception ".exception" far_absolute R`
	`#pragma define_section exceptlist ".exceptlist" far_absolute R`
PID Addressing Mode	`#pragma define_section text ".text" far_absolute RX`
	`#pragma define_section data ".data" ".bss" far_data RW`
	`#pragma define_section sdata ".sdata" ".sbss" near_data RW`
	`#pragma define_section const ".rodata" far_absolute R`
PIC Addressing Mode	`#pragma define_section text ".text" far_code RX`
	`#pragma define_section data ".data" ".bss" far_absolute RW`
	`#pragma define_section sdata ".sdata" ".sbss" near_data RW`
	`#pragma define_section const ".rodata" far_code R`

Another use for `#pragma define_section` is redefining the attributes of predefined sections:

- To force 16-bit absolute addressing for all data, use

```
#pragma define_section data ".data" near_absolute
```

- To force 32-bit absolute addressing for exception tables, use:

```
#pragma define_section exceptlist ".exceptlist" far_code

#pragma define_section exception ".exception" far_code
```

You should put any such attribute-redefinition pragmas, in a prefix file or other header that your program's all source files will include.

NOTE

The ELF linker's **Section Mappings** settings panel must map any user-defined compiler section to an appropriate segment.

32.3.2 force_active

Syntax

```
#pragma force_active on | off | reset
```

Remarks

In source code, use `__declspec(force_export)`, `__attrbute__((force_export))`, or `__attrbute__((used))`.

In a linker command file, use the `FORCE_ACTIVE` command.

32.4 ColdFire Code Generation Pragmas

This topic lists the following ColdFire code generation pragmas:

Table 32-3. Pragmas for Code Generation

codeColdFire	const_multiply	emac
explicit_zero_data	inline_intrinsics	interrupt
native_coldfire_alignment	options	readonly_strings
section		

32.4.1 codeColdFire

Controls organization and generation of ColdFire object code.

```
#pragma codeColdFire processor
```

Parameter

`processor`

`_MCFxxxx_`

where, xxxx is the processor number.

32.4.2 const_multiply

Enables support for constant multiplies, using shifts and add/subtracts.

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

Remarks

The default value is `on`.

32.4.3 emac

Enables EMAC assembly instructions in inline assembly.

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

Remarks

Enables inline-assembly instructions `mac`, `msac`, `macl`, `msacl`, `move`, and `movclr` for the ColdFire EMAC unit.

The default value is `OFF`.

32.4.4 explicit_zero_data

Specifies storage area for zero-initialized data.

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

Remarks

The default value `OFF` specifies storage in the `.sbss` or `.bss` section. The value `ON` specifies storage in the `.data` section. The value `reset` specifies storage in the most-recent previously specified section.

Example

```
#pragma explicit_zero_data on

int in_data_section = 0;

#pragma explicit_zero_data off

int in_bss_section = 0;
```

32.4.5 inline_intrinsics

Controls support for inline intrinsic optimizations `strcopy` and `strlen`.

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

Remarks

In the `strcopy` optimization, the system copies the string via a set of move-immediate commands to the source address. The system applies this optimization if the source is a string constant of fewer than 64 characters, and optimizing is set for speed.

In the `strlen` optimization, a move-immediate of the length of the string to the result, replaces the function call. The system applies this optimization, if the source is a string constant.

The default value is `ON`.

32.4.6 interrupt

Controls compilation for interrupt-routine object code.

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

Remarks

For the value `ON`, the compiler generates special prologus and epilogues for the functions this pragma encapsulates. The compiler saves or restores all modified registers (both nonvolatile and scratch). Functions return via `RTE` instead of `RTS`.

You also can also use `__declspec(interrupt)` to mark a function as an interrupt routine. This directive also allows you to specify an optional status register mask at runtime.

32.4.7 native_coldfire_alignment

Sets value of the native alignment option.

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

Remarks

The default value is on.

Example

```
#pragma native_coldfire_alignment on

#if __option(native_ coldfire_ alignment)

...

#endif
```

For more information, please see "TN256: Converting ColdWarrior® ColdFire® Projects to CodeWarrior ColdFire v7.x".

32.4.8 options

Specifies how to align structure and class data.

Syntax

```
#pragma options align=alignment
```

Parameter

`alignment`

Specifies the boundary on which structure and class data is aligned in memory.

Table 32-4. Structs and Classes Alignment

If *alignment* is ...	The compiler ...
`Byte`	Aligns every field on a byte boundary.
`68k/mac68k`	Aligns every field on 2-byte boundaries, unless a field is 1 byte long, This is standard alignment for 68K. This is word align alignment in the code generation panel.
`coldfire / mac68k4byte`	Aligns every field on 4-byte boundaries, unless a field is 1 or 2 bytes long. This is the standard alignment for ColdFire. This is long alignment in the code generation panel.

Table continues on the next page...

Table 32-4. Structs and Classes Alignment (continued)

If *alignment* is ...	The compiler ...
`native`	Aligns every field using the alignment defined with pragma `native_coldfire_alignment`.
`packed`	Aligns every field on a 1-byte boundary. This is byte alignment in the code generation panel.
`reset`	Resets to the value in the previous `#pragma` options *align* statement.

NOTE

There is a space between `options` and `align`.

Remarks

The use of this pragma is deprecated. Fields are aligned on 4-byte boundaries and doubles are aligned on 8-byte boundaries. This pragma does not correspond to any panel setting.

32.4.9 readonly_strings

Enables the compiler to place strings in the `.rodata` section.

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

Remarks

The default value is `on`.

For the `off` value, the compiler puts strings in initialized data sections `.data` or `.sdata`, according to the string size.

32.4.10 section

Activates or deactivates a user-defined or predefined section.

```
#pragma section sname begin | end
```

Parameters

`sname`

Identifier for a user-defined or predefined section.

`begin`

Activates the specified section from this point in program execution.

`end`

Deactivates the specified section from this point in program execution; the section returns to its default state.

Remarks

Each call to this pragma must include a begin parameter or an end parameter, but not both.

You may use this pragma with `#pragma push` and `#pragma pop` to ease complex or frequent changes to section settings.

NOTE

A simpler alternative to `#pragma section` is the `__declspec()` declaration specifier.

32.5 ColdFire Optimization Pragmas

This topic lists the following ColdFire optimization pragmas:

- no_register_coloring
- opt_cse_calls
- opt_unroll_count
- opt_unroll_instr_count
- profile
- scheduling

32.5.1 no_register_coloring

Disables allocation of local variables to registers.

```
#pragma no_register_coloring on | off | reset
```

32.5.2 opt_cse_calls

When compiling with both optimization level 2 or more and optimize for size, this option is set. The compiler treats multiple calls to a function as a common sub-expression and replace occurrences with indirect calls.

```
#pragma opt_cse_calls on | off | reset
```

32.5.3 opt_unroll_count

Limits the number of times a loop can be unrolled; fine-tunes the loop-unrolling optimization.

```
#pragma opt_unroll_count [ 0..127 | reset ]
```

Remarks

The default value is 8.

32.5.4 opt_unroll_instr_count

Limits the number of pseudo-instructions; fine-tunes the loop-unrolling optimization.

```
#pragma opt_unroll_instr_count [ 0..127 | reset ]
```

Remarks

There is not always a one-to-one mapping between pseudo-instructions and actual ColdFire instructions.

The default value is 100.

32.5.5 profile

Organizes object code for the profiler library and enables simple profiling.

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

32.5.6 scheduling

Enables instruction scheduling for that processor, optimization level 2 and above.

```
#pragma scheduling on | off | reset
```

Chapter 33
Appendices

This section includes the following chapter:

Chapter 34
Calling Assembly Functions

This chapter describes the process to create an inline and pure assembly functions and how to call functions from the assembly language. This topics discussed here are as follows:

- Inline Assembly
- Calling Inline Assembly Language Functions
- Calling Pure Assembly Language Functions
- Calling Functions from Assembly Language

34.1 Inline Assembly

This topic explains support for inline assembly language programming. Inline assembly language are assembly language instructions and directives embedded in C and C++ source code. The topics discussed here are as follows:

- Inline Assembly Syntax
- Inline Assembly Directives

34.1.1 Inline Assembly Syntax

Syntax explanation topics are:

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

34.1.1.1 Statements

All internal assembly statements must follow this syntax:

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

Other rules for statements are:

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

```
move.l   b, DO
MOVE.L   b, d0
```

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

NOTE

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

The following listings are valid examples of inline assembly code:

Listing: Function-Level Sample

```
long int b;
struct mystruct {

  long int a;

} ;

static asm long f(void)     // Legal asm qualifier

{

  move.l   struct(mystruct.a)(A0),D0 // Accessing a struct.

  add.l    b,D0   // Using a global variable, put return value

                  // in D0.

  rts             // Return from the function:

                  // result = mystruct.a + b

}
```

Listing: Statement-Level Sample

```
long square(short a)
{

   asm {

     move.w  a,d0    // fetch function argument `a'

     mulu.w  d0,d0   // multiply

     return          // return from function (result is in D0)

   }

}
```

NOTE

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

34.1.1.2 Additional Syntax Rules

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

- Each label must end with a colon; labels may contain the `@` character. For example, `x1:` and `@x2:` would be valid labels, but `x3` would not - it lacks a colon.
- Comments must use C/ C++ syntax: either starting with double slash characters (`//`) or enclosed by slash and asterisk characters (`/*` ... `*/`).
- To refer to a field in a structure, use the `struct` construct:

  ```
  struct(structTypeName.fieldName) structAddress
  ```

  For example, suppose that `A0` points to structure `WindowRecord`. This instruction moves the structure's `refCon` field to `D0`:

  ```
  move.l  struct(WindowRecord.refCon) (A0), D0
  ```
- To refer to a global variable, merely use its name, as in the statement

  ```
  move.w      x,d0    // Move x into d0
  ```

34.1.1.3 Preprocessor Features

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

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

34.1.1.4 Local Variables and Arguments

Handling of local variables and arguments depends on the level of inline assembly. However, *when register coloring is selected in the code generation settings for optimization level 1 or greater*, you can force variables to stay in a register by using the symbol $.

- Function-level
- Statement-level

34.1.1.4.1 Function-level

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

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

```
asm void alpha(short n)
{
  move.w      4(sp),d0 //  n
  // . . .
}
```

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

1. Declare your variables as you would in a normal C function.
2. Use the fralloc directive. It makes space on the stack for the local stack variables. Additionally, with the statement link #*x*,a6, this directive reserves registers for the local register variables.
3. In your assembly, you can refer to the local variables and variable arguments by name.

4. Finally, use the `frfree` directive to free the stack storage and restore the reserved registers. (It is somewhat easier to use a C wrapper and statement level assembly.)

The code listed below is an example of using local variables and function arguments in function-level inline assembly.

Listing: Function-level Local Variables, Function Arguments

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

34.1.1.4.2 Statement-level

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

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

Listing: Statement-Level Local Variables, Function Arguments

```
long square(short a)
{
  long result=0;
  asm {
    move.w  a,d0      // fetch function argument `a'
    mulu.w  d0,d0     // multiply
    move.l  d0,result // store in local `result' variable
  }
  return result;
```

```
}
```

34.1.1.5 Returning from Routine

Every inline assembly function (not statement level) should end with a return statement. Use the `rts` statement for ordinary C functions, as the following listing shows:

Listing: Assembly Function Return

```
asm void f(void)
{   add.l      d4, d5}       // Error, no RTS statement

asm void g(void)

{   add.l      d4, d5

    rts}                     // OK
```

For statement-level returns, refer to the return and naked topics.

34.1.2 Inline Assembly Directives

This chapter lists and describes the special assembler directives that the Power Architecture inline assembler accepts.

Table 34-1. Inline Assembly Directives

dc	ds	entry
fralloc	frfree	machine
naked	opword	return

NOTE

Except for `dc` and `ds`, the inline assembly directives are available only for function/routine level.

34.1.2.1 dc

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

```
dc[.(b|w|l)] constexpr (,constexpr)*
```

Parameters

b

Byte specifier, which lets you specify any C (or Pascal) string constant.

w

Word specifier (default).

l

Longword specifier.

constexpr

Name for block of constant expressions.

Example

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

34.1.2.2 ds

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

```
ds[.(b|w|l)] size
```

Parameters

b

Byte specifier.

w

Word specifier (the default).

l

Longword specifier.

`size`

Number of bytes, words, or longwords in the block.

Example

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

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

34.1.2.3 entry

Defines an entry point into the current function. Use the `extern` qualifier to declare a global entry point and use the `static` qualifier to declare a local entry point. If you leave out the qualifier, `extern` is assumed (refer to the listing displayed below).

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

Parameters

`extern`

Specifier for a global entry point (the default).

`static`

Specifier for a local entry point.

`name`

Name for the new entry point.

Example

The following listing defines the new local entry point `MyEntry` for function `MyFunc`.

Listing: Entry Directive Example

```
static long MyEntry(void);
static asm long MyFunc(void)

{

    move.l  a,d0

    bra.s   L1

    entry   static MyEntry

    move.l  b,d0

L1: rts
```

```
}
```

34.1.2.4 fralloc

Lets you declare local variables in an assembly function.

```
fralloc [+]
```

Parameter

`+`

Optional ColdFire-register control character.

Remarks

This directive makes space on the stack for your local stack variables. It also reserves registers for your local register variables (with the statement `link #`*x*`,a6`).

Without the `+` control character, this directive pushes modified registers onto the stack.

With the `+` control character, this directive pushes all register arguments into their Power Architecture registers.

Counterpart to the `frfree` directive.

34.1.2.5 frfree

Frees the stack storage area; also restores the registers (with the statement unlk a6) that `fralloc` reserved.

```
frfree
```

34.1.2.6 machine

Specifies the CPU for which the compiler generates its inline-assembly instructions.

```
machine processor
```

Parameter

`processor`

```
MCFxxxxx
```

where, xxxx is the processor number.

Remarks

If you use this directive to specify a target processor, additional inline-assembler instructions become available - instructions that pertain only to that processor. For more information, see the Freescale processor user's manual

34.1.2.7 naked

Suppresses the compiler-generated stackframe setup, cleanup, and return code.

```
  naked
```

Remarks

Functions with this directive cannot access local variables by name. They should not contain C code that implicitly or explicitly uses local variables or memory.

Counterpart to the `return` directive.

Example

The following listing is an example use of this directive.

Listing: Naked Directive Example

```
long square(short)
{
  asm{
    naked              // no stackframe or compiler-generated rts
    move.w  4(sp),d0   // fetch function argument from stack
    mulu.w  d0,d0      // multiply
    rts                // return from function: result in D0
  }
}
```

34.1.2.8 opword

Writes machine-instruction constants directly into the executable file, without any error checking.

```
opword constant[,constant]
```

Parameter

`constant`

Any appropriate machine-code value.

Example

`opword` 0x4E75 - which is equivalent to 'rts'.

34.1.2.9 return

Inserts a compiler-generated sequence of stackframe cleanup and return instructions. Counterpart to the `naked` directive.

```
return instruction[, instruction]
```

Parameter

`instruction`

Any appropriate C instruction.

34.2 Calling Inline Assembly Language Functions

The following listing demonstrates how to create an inline assembly function in a C source file. This example adds two integers and returns the result.

Note that you are passing the two parameters in registers D0, D1, and the result is returned in D0.

Listing: Sample Code - An Inline Assembly Function

```
asm int my_inlineasm_func(int a, int b)
{

subq.l   #4,a7

move.l   d1,(a7)

add.l    (a7),d0

addq.l   #4,a7
```

```
rts
}
```

The following listing shows the C calling statement for this inline-assembly-language function.

Listing: Sample Code - Inline Assembly Function Call

```
rez = my_inlineasm_func(10, 1);
```

34.3 Calling Pure Assembly Language Functions

This topic describes how to call the pure assembly language functions. This topic explains:

- Function Definition
- Function Use

34.3.1 Function Definition

The label which is associated with the start of the function should be the link name of the function. The following table shows an example of the link name of a pure assembly language function.

Table 34-2. Example of the Link Name for an Assembly Language Function

C Code	Link Name
my_asm_func	_my_asm_func

The labels defined in an assembly file have local scope. To access them from the other file (the .c file) they should be marked as global. For example,

```
.global  _my_asm_func
```

NOTE

Make sure that both on the caller and the callee side, the function has same signature and calling convention.

The following listing is an example of a pure assembly language function.

Listing: Sample Code - A Pure Assembly Language Function

```
.global _my_asm_func
.text

_my_asm_func:

 subq.l   #4,a7

 move.l   d1,(a7)

 add.l    (a7),d0

 addq.l   #4,a7

 rts
```

34.3.2 Function Use

To call a function, declare the function prototype in the `.c` file:

```
int my_asm_func(int a, int b);
```

The function call is as for any regular C function call. For example:

```
my_asm_func(5, 2);
```

34.4 Calling Functions from Assembly Language

Assembly language programs can call the functions written in either C or assembly language. From within the assembly language instructions, you can call the C functions. For example, if the C function definition is:

```
int my_c_func(int i)
{

return i++;

}
```

the assembly language calling statement should be:

```
jsr _my_c_func
```

NOTE

The function is called using its link name.

Index

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

W

X

Z

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number CWMCUCFCMPREF
Revision 10.6, 08/2014