CodeWarrior Development Studio for Power Architecture® Processors Build Tools Reference

Revised: 2 March 2012

How to Contact Us

Corporate Headquarters	Freescale Semiconductor, Inc. 6501 William Cannon Drive West Austin, Texas 78735 U.S.A.
World Wide Web	`http://www.freescale.com/codewarrior`
Technical Support	`http://www.freescale.com/support`

Table of Contents

1

Introduction

This reference 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 reference covers the CodeWarrior compiler and its linker, versions 4.0 and higher.

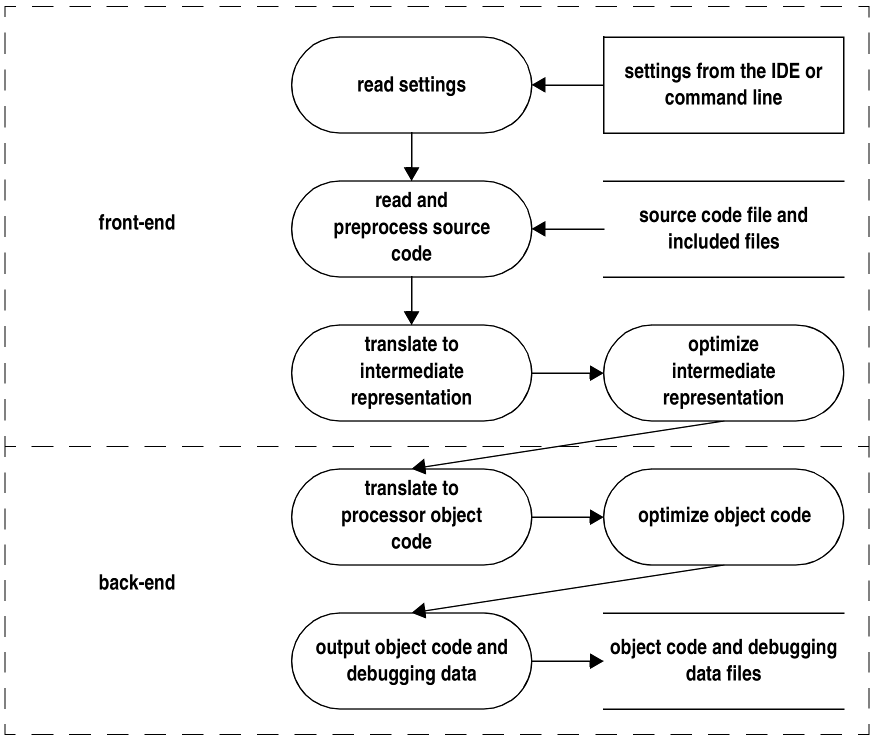
This chapter explains the processes that CodeWarrior build tools use to create software:

- Compiler Architecture
- Linker Architecture

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. Figure 1.1 shows the steps the compiler takes.

Figure 1.1 CodeWarrior compiler steps


front-end
read settings
settings from the IDE or command line
read and preprocess source code
source code file and included files
translate to intermediate representation
optimize intermediate representation
back-end
translate to processor object code
optimize object code
output object code and debugging data
object code and debugging data files

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

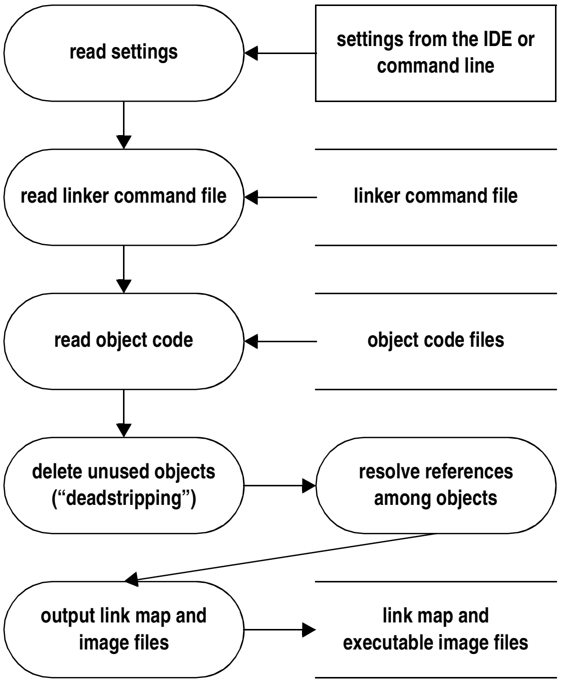
Linker Architecture

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.

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

Figure 1.2 CodeWarrior linker steps


read settings
settings from the IDE or command line
read linker command file
linker command file
read object code
object code files
delete unused objects ("deadstripping")
resolve references among objects
output link map and image files
link map and executable image files

- 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

2

Using Build Tools with the CodeWarrior IDE

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

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

- Choosing Tools and Files
- IDE Options and Pragmas
- IDE Settings Panels

Choosing Tools and Files

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

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

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

IDE Options and Pragmas

Use IDE settings and directives in source code to configure the build tools.

The CodeWarrior compiler follows these steps to determine the settings to apply to each file that the compiler translates under the IDE:

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

  The compiler applies pragma directives and updates their settings as pragma directives are encountered in this source code.
- the compiler translates the source code file and the files that it includes

  The compiler applies pragma settings as it encounters them.

IDE Settings Panels

These CodeWarrior IDE settings panels control compiler and linker behavior:

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

C/C++ Language Settings Panel

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

Table 2.1 C/C++ Language Settings Panel

This item...	controls this behavior	and is equivalent to these options
Force C++ Compilation	Checked—translates all C source files as C++ source code. Clear—uses the filename's extension to determine whether to use the C or C++ compiler. The entries in the IDE's **File Mappings** settings panel specify the suffixes that the compiler assigns to each compiler.	pragma `cplusplus` and the command-line option `-lang c++`

Table 2.1 C/C++ Language Settings Panel (*continued*)

This item...	controls this behavior	and is equivalent to these options
ISO C++ Template Parser	Checked—follows the ISO/IEC 14882-1998 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. Clear—the C+++ compiler does not expect template source code to follow the ISO C++ standard as closely.	pragma `parse_func_templ` and the command-line option `-iso_templates`
Use Instance Manager	Checked—reduces compile time by generating any instance of a C++ template (or non-inlined inline) function only once. Clear—generates a new instance of a template or non-inlined function each time it appears in source code. Control where the instance database is stored using `#pragma instmgr_file`.	command-line option `-instmgr`
Enable C++ Exceptions	Checked—generates executable code for C++ exceptions. Clear—generates smaller, faster executable code. Enable the **Enable C++ Exceptions** setting if you use the `try`, `throw`, and `catch` statements specified in the ISO/IEC 14882-1998 C++ standard. Otherwise, disable this setting to generate smaller and faster code.	pragma `exceptions` and the command-line option `-cpp_exceptions`

Table 2.1 C/C++ Language Settings Panel (*continued*)

This item...	controls this behavior	and is equivalent to these options
Enable RTTI	Checked—allows the use of the C++ runtime type information (RTTI) capabilities, including the `dynamic_cast` and `typeid` operators. Clear—the compiler generates smaller, faster object code but does not allow runtime type information operations.	pragma `RTTI` and the command-line option `-RTTI`
Enable bool Support	Checked—the C++ compiler recognizes the `bool` type and its `true` and `false` values specified in the ISO/IEC 14882-1998 C++ standard. Clear—the compiler does not recognize this type or its values.	pragma `bool` and the command-line option `-bool`
Enable wchar_t Support	Checked—the C++ compiler recognizes the `wchar_t` data type specified in the ISO/IEC 14882-1998 C++ standard. Clear—the compiler does not recognize this type. Turn off this option when compiling source code that defines its own `wchar_t` type.	pragma `wchar_type` and the command-line option `-wchar_t`
EC++ Compatibility Mode	Checked—expects C++ source code files to contain Embedded C++ source code. Clear—the compiler expects regular C++ source code in C++ source files.	pragma `ecplusplus` and the command-line option `-dialect ec++`

Table 2.1 C/C++ Language Settings Panel (*continued*)

This item...	controls this behavior	and is equivalent to these options
Inline Depth	Don't Inline—Inlines no functions, not even C or C++ functions declared `inline`. Smart—Inlines small functions to a depth of 2 to 4 inline functions deep. 1 to 8—Inlines to the depth specified by the numerical selection.	The **Don't Inline** item corresponds to the pragma `dont_inline` and the command-line option `-inline off`. The **Smart** and **1** to **8** items correspond to the `pragma inline_depth` and the command-line option `-inline level=`*n*, where *n* is `1` to `8`.
IPA	Specifies the Interprocedural Analysis (IPA) policy. 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. **Program—**completely parse the entire program before optimizing and generating code, providing many optimization benefits. For example, the compiler can auto-inline functions that are defined in another translation unit.	command line option `-ipa`

Table 2.1 C/C++ Language Settings Panel (*continued*)

This item...	controls this behavior	and is equivalent to these options
Auto-Inline	Checked—the compiler chooses which functions to inline. Also inlines C++ functions declared `inline` and member functions defined within a class declaration. Clear—the compiler only considers functions declared with `inline`.	pragma `auto_inline` and the command-line option `-inline auto`
Bottom-up Inlining	Checked—performs inline analysis from the last function to the first function in a chain of function calls. Clear—inline analysis begins at the first function in a chain of function calls.	pragma `inline_bottom_up` and the command-line option `-inline bottomup`

Table 2.1 C/C++ Language Settings Panel (*continued*)

This item...	controls this behavior	and is equivalent to these options
ANSI Strict	Checked—Only recognizes source code that conforms to the ISO/IEC 9899-1990 standard for C. Clear—recognize several CodeWarrior extensions to the C language: • unnamed arguments in function definitions • a # not followed by a macro directive • using an identifier after a `#endif` directive • using typecasted pointers as lvalues • converting points to type of the same size • arrays of zero length in structures • the `D` constant suffix • enumeration constant definitions that cannot be represented as signed integers when the **Enums Always Int** option is on in the IDE's **C/C++ Language** settings panel or the `enumsalwaysint` pragma is on • a C++ `main()` function that does not return an integer value	pragma `ANSI_strict` and the command-line option `-ansi strict`
ANSI Keywords Only	Checked—(ISO/IEC 9899-1990 C, §6.4.1) generates an error message for all non-standard keywords. If you must write source code that strictly adheres to the ISO standard, enable this setting. Clear—the compiler recognizes only these non-standard keywords: `far`, `inline`, `__inline__`, `__inline`, and `pascal`.	pragma `only_std_keywords` and the command-line option `-stdkeywords`

Table 2.1 C/C++ Language Settings Panel (*continued*)

This item...	controls this behavior	and is equivalent to these options
Expand Trigraphs	Checked—recognizes trigraph sequences (ISO/IEC 9899-1990 C, §5.2.1.1). Clear—ignores trigraph characters. Many common character constants look like trigraph sequences, and this extension lets you use them without including escape characters.	pragma `trigraphs` and the command-line option `-trigraphs`
Legacy for-scoping	Checked—generates an error message when the compiler encounters a variable scope usage that the ISO/IEC 14882-1998 C++ standard disallows, but is allowed in the C++ language specified in *The Annotated C++ Reference Manual* (“ARM”). Clear—allows scope rules specified in ARM.	pragma `ARM_scoping` and the command-line option `-for_scoping`
Require Function Prototypes	Checked—enforces 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. Clear—do not require prototypes.	pragma `require_prototypes` and the command-line option `-requireprotos`
Enable C99 Extensions	Checked—recognizes ISO/IEC 9899-1999 (“C99”) language features. Clear—recognizes only ISO/IEC 9899-1990 (“C90”) language features.	pragma `c99` and the command-line option `-dialect c99`

Table 2.1 C/C++ Language Settings Panel (*continued*)

This item...	controls this behavior	and is equivalent to these options
Enable GCC Extensions	Checked—recognizes language features of the GNU Compiler Collection (GCC) C compiler that are supported by CodeWarrior compilers. Clear—do not recognize GCC extensions	pragma `gcc_extensions` and the command-line option `-gcc_extensions`
Enums Always Int	Checked—uses signed integers to represent enumerated constants. Clear—uses smallest possible integer type to represent enumerated constants.	pragma `enumsalwaysint` and the command-line option `-enum`
Use Unsigned Chars	Checked—treats `char` declarations as `unsigned char` declarations. Clear—`char` declarations are `signed char` declarations	pragma `unsigned_char` and the command-line option `-char unsigned`
Pool Strings	Checked—collects all string constants into a single data section in the object code it generates. Clear—creates a unique section for each string constant.	pragma `pool_strings` and the command-line option `-strings pool`
Reuse Strings	Checked—stores only one copy of identical string literals. Clear—stores each string literal separately.	opposite of the pragma `dont_reuse_strings` and the command-line option `-string reuse`

C/C++ Preprocessor Panel

The C/C++ Preprocessor settings panel controls the operation of the CodeWarrior compiler's preprocessor.

Table 2.2 C/C++ Preprocessor Panel

This item...	controls this behavior
Prefix Text	Contains source code that the compiler inserts at the beginning of each translation unit. A translation unit is the combination of a source code file and all the files that it includes.
Source encoding	Allows you to specify the default encoding of source files. The compiler recognizes Multibyte and Unicode source text. To replicate the obsolete option **Multi-Byte Aware**, set this option to **System** or **Autodetect**. Additionally, options that affect the preprocess request appear in this panel.
Use prefix text in precompiled header	Checked—inserts the source code in the **Prefix Text** field at the beginning of a precompiled header file. Clear—does not insert **Prefix Text** contents in a precompiled header file. Defaults to clear to correspond with previous versions of the compiler that ignore the prefix file when building precompiled headers. If any pragmas are imported from old C/C++ Language Panel settings, this option is enabled.
Emit file changes	Checked—notification of file changes (or #line changes) appear in the output. Clear—no file changes appear in output.
Emit #pragmas	Checked—pragma directives appear in the preprocessor output. Essential for producing reproducible test cases for bug reports. Clear—pragma directives do not appear in preprocessor output.

Table 2.2 C/C++ Preprocessor Panel (*continued*)

This item...	**controls this behavior**
Show Full Paths	Checked—show the full path of a file's name. Clear—show the base filename.
Keep comments	Checked—comments appear in the preprocessor output. Clear—comments do not appear in preprocessor output.
Use #line	Checked—file changes appear in comments (as before) or in #line directives. Clear—file changes do not appear in comments or in #line directives.
Keep whitespace	Checked—whitespace is copied to preprocessor output. This is useful for keeping the starting column aligned with the original source, though the compiler attempts to preserve space within the line. This does not apply when macros are expanded. Clear—whitespace is stripped in preprocessor output.

C/C++ Warnings Panel

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

Table 2.3 C/C++ Warnings Panel

This item	controls this behavior	and is equivalent to these options
Illegal Pragmas	Checked—issues a warning message if the compiler encounters an unrecognized pragma. Clear—no action for unrecognized pragma directives.	pragma `warn_illpragma` pragma and the command-line option `-warnings illpragmas`
Possible Errors	Checked—issues warning messages for common, usually-unintended logical errors: in conditional statements, using the assignment (=) operator instead of the equality comparison (==) operator, in expression statements, using the == operator instead of the = operator, placing a semicolon (`;`) immediately after a `do`, `while`, `if`, or `for` statement.	pragma `warn_possunwant` and the command-line option `-warnings possible`
Extended Error Checking	Checked—issues warning messages for common programming errors: mis-matched return type in a function’s definition and the return statement in the function’s body, mismatched assignments to variables of enumerated types.	pragma `extended_errorcheck` and the command-line option `-warnings extended`
Hidden Virtual Functions	Checked—generates a warning message if you declare a non-virtual member function that prevents a virtual function, that was defined in a superclass, from being called.	pragma `warn_hidevirtual` and the command-line option `-warnings hidevirtual`

Table 2.3 C/C++ Warnings Panel

This item	controls this behavior	and is equivalent to these options
Implicit Arithmetic Conversions	Checked—issues a warning message when the compiler applies implicit conversions that may not give results you intend: assignments where the destination is not large enough to hold the result of the conversion, a signed value converted to an unsigned value, an integer or floating-point value is converted to a floating-point or integer value, respectively.	pragma `warn_implicitconv` and the command-line option `-warnings implicitconv`
Float To Integer	Checked—issues a warning message for implicit conversions from floating point values to integer values.	pragma `warn_impl_f2i_conv` and the command-line option `-warnings impl_float2int`
Signed/Unsigned	Checked—issues a warning message for implicit conversions from a signed or unsigned integer value to an unsigned or signed value, respectively.	pragma `warn_impl_s2u_conv` and the command-line option `-warnings signedunsigned`
Integer To Float	Checked—issues a warning message for implicit conversions from integer to floating-point values.	pragma `warn_impl_i2f_conv` and the command-line option `-warnings impl_int2float`
Pointer/Integral Conversions	Checked—issues a warning message for implicit conversions from pointer values to integer values and from integer values to pointer values.	pragmas `warn_any_ptr_int_conv` and `warn_ptr_int_conv` and the command-line option `-warnings ptrintconv,anyptrinvconv`
Unused Variables	Checked—issues a warning message for local variables that are not referred to in a function.	pragma `warn_unusedvar` and the command-line option `-warnings unusedvar`

Table 2.3 C/C++ Warnings Panel

This item	controls this behavior	and is equivalent to these options
Unused Arguments	Checked—issues a warning message for function arguments that are not referred to in a function.	pragma `warn_unusedarg` and the command-line option `-warnings unusedarg`
Missing 'return' Statements	Checked—issues a warning message if a function that is defined to return a value has no `return` statement.	pragma `warn_missingreturn` and the command-line option `-warnings missingreturn`
Expression Has No Side Effect	Checked—issues a warning message if a statement does not change the program's state.	pragma `warn_no_side_effect` and the command-line option `-warnings unusedexpr`
Enable All	Checked—turns on all warning options.	
Disable All	Checked—turns off all warning options.	
Extra Commas	Checked—issues a warning message if a list in an enumeration terminates with a comma. The compiler ignores terminating commas in enumerations when compiling source code that conforms to the ISO/IEC 9899-1999 ("C99") standard.	pragma `warn_extracomma` and the command-line option `-warnings extracomma`
Inconsistent 'class'/ 'struct' Usage	Checked—issues a warning message if the class and struct keywords are used interchangeably in the definition and declaration of the same identifier in C++ source code.	pragma `warn_structclass` and the command-line option `-warnings structclass`
Empty Declarations	Checked—issues a warning message if a declaration has no variable name.	pragma `warn_emptydecl` and the command-line option `-warnings emptydecl`

Table 2.3 C/C++ Warnings Panel

This item	controls this behavior	and is equivalent to these options
Include File Capitalization	Checked—issues a warning message if the name of the file specified in a `#include "file"` directive uses different letter case from a file on disk.	pragma `warn_filenamecaps` and the command-line option `-warnings filecaps`
Check System Includes	Checked—issues a warning message if the name of the file specified in a `#include <file>` directive uses different letter case from a file on disk.	pragma `warn_filenamecaps_system` and the command-line option `-warnings sysfilecaps`
Pad Bytes Added	Checked—issues a warning message when the compiler adjusts the alignment of components in a data structure.	pragma `warn_padding` and the command-line option `-warnings padding`
Undefined Macro in #if	Checked—issues a warning message if an undefined macro appears in `#if` and `#elif` directives.	pragma `warn_undefmacro` and the command-line option `-warnings undefmacro`
Non-Inlined Functions	Checked—issues a warning message if a call to a function defined with the `inline`, `__inline__`, or `__inline` keywords could not be replaced with the function body.	pragma `warn_notinlined` and the command-line option `-warnings notinlined`
Treat All Warnings As Errors	Checked—issues warning messages as error messages.	pragma `warning_errors` pragma and the command-line option `-warnings error`

3

Using Build Tools on the Command Line

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

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

Configuring Command-Line Tools

- Setting CodeWarrior Environment Variables
- Setting the PATH Environment Variable

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. Table 3.1 describes these environment variables.

Table 3.1 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 "<" and ">" 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 3.1 Setting environment variables in Microsoft® Windows® operating systems

```
rem Use ; to separate directory paths

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

set MWCIncludes=%CWFolder%\MSL_Common\Include
set MWCIncludes=%MWCIncludes%;%CWFolder%\\Include
set MWLibraries=%CWFolder%\Support\;%CWFolder%\\Runtime
```

Setting the PATH Environment Variable

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

Listing 3.2 Example of setting PATH

```
set CWFolder=C:\Program Files\Freescale\CodeWarrior
set PATH=%PATH%\%CWFolder%\Bin;%CWFolder%\Toolset
```

Invoking Command-Line Tools

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

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

```
tool options files
```

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

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

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

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

Getting Help

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

```
tool -help
```

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

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

```
tool -help | more
```

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

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

```
tool -help usage
```

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

Parameter Formats

Parameters in an option are formatted as follows:

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

Option Formats

Options are formatted as follows:

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

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

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

Common Terms

These common terms appear in many option descriptions:

- A "cased" option is considered case-sensitive. By default, no options are case-sensitive.
- "compatibility" indicates that the option is borrowed from another vendor's tool and its behavior may only approximate its counterpart.
- A "global" option has an effect over the entire command line and is parsed before any other options. When several global options are specified, they are interpreted in order.
- A "deprecated" option will be eliminated in the future and should no longer be used. An alternative form is supplied.
- An "ignored" option is accepted by the tool but has no effect.
- A "meaningless" option is accepted by the tool but probably has no meaning for the target operating system.
- An "obsolete" option indicates a deprecated option that is no longer available.

- A “substituted” option has the same effect as another option. This points out a preferred form and prevents confusion when similar options appear in the help.
- Use of “default” in the help text indicates that the given value or variation of an option is used unless otherwise overridden.

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

File Name Extensions

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

The command-line version of the CodeWarrior C/C++ compiler accepts non-standard file extensions as source code but also emits a warning message. By default, the compiler assumes that a file with any extensions besides `.c`, `.h`, `.pch` is C++ source code. The linker ignores all files that it 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 (Listing 3.3).

Listing 3.3 Example of using linker command files

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

4

Command-Line Options for Standard C Conformance

-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
```

-stdkeywords

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

Syntax

```
-stdkeywords on | off
```

Remarks

Default setting is `off`.

-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 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

The default setting is `off`.

5

Command-Line Options for Standard C++ Conformance

-ARM

Deprecated. Use -for_scoping instead.

-bool

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

Syntax

```
-bool on | off
```

Remarks

When `on`, the compiler recognizes the `true` and `false` keywords in expressions of type `bool`. When off, the compiler does recognizes the keywords, forcing the source code to provide definitions for these names. The default is `on`.

-Cpp_exceptions

Controls the use of C++ exceptions.

Syntax

```
-cpp_exceptions on | off
```

Remarks

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

-dialect

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.

-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`.

-instmgr

Controls whether the instance manager for templates is active.

Syntax

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

The options for *keyword* are:

`off`

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

`on`

Turn on the C++ instance manager.

`file=`*`path`*

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

Remarks

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

NOTE The instance manager feature is not supported by the DSi compiler.

-iso_templates

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

Syntax

```
-iso_templates on | off
```

Remarks

Default setting is `on`.

-RTTI

Controls the availability of runtime type information (RTTI).

Syntax

```
-RTTI on | off
```

Remarks

Default setting is `on`.

-som

Obsolete. This option is no longer available.

-som_env_check

Obsolete. This option is no longer available.

-wchar_t

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

Syntax

```
-wchar_t on | off
```

Remarks

The `-wchar 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`.

6

Command-Line Options for Language Translation

-char

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

Syntax

`-char keyword`

The arguments for *keyword* are:

`signed`

`char` data items are signed.

`unsigned`

`char` data items are unsigned.

Remarks

The default is `signed`.

-defaults

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

Syntax

`-defaults`

`-nodefaults`

Remarks

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

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

-encoding

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.f

`EUC[JP | -JP]`

Japanese Extended UNIX Code (EUCJP) format.

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

International Organization of Standards (ISO) Japanese format.

Remarks

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

-flag

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
```

-gccext

Enables GCC (Gnu Compiler Collection) C language extensions.

Syntax

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

Remarks

See "GCC Extensions" on page 260 for a list of language extensions that the compiler recognizes when this option is `on`.

The default setting is `off`.

-gcc_extensions

Equivalent to the `-gccext` option.

Syntax

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

-M

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

Syntax

```
-M
```

Remarks

This command is global and case-sensitive.

-make

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

Syntax

```
-make
```

Remarks

This command is global.

-mapcr

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

Syntax

```
-mapcr
-nomapcr
```

Remarks

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

-MM

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

Syntax

```
-MM
```

Remarks

This command is global and case-sensitive.

-MD

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

Syntax

```
-MD
```

Remarks

This command is global and case-sensitive.

-MMD

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

Syntax

```
-MMD
```

Remarks

This command is global and case-sensitive.

-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

-once

Prevents header files from being processed more than once.

Syntax

```
-once
```

Remarks

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

-pragma

Defines a pragma for the compiler.

Syntax

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

The arguments are:

`name`

Name of the 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
```

-relax_pointers

Relaxes the pointer type-checking rules in C.

Syntax

```
-relax_pointers
```

Remarks

This option is equivalent to

```
#pragma mpwc_relax on
```

-requireprotos

Controls whether or not the compiler should expect function prototypes.

Syntax

```
-r[equireprotos]
```

-search

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

Syntax

```
-search
```

-trigraphs

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

Syntax

```
-trigraphs on | off
```

Remarks

Default setting is `off`.

7

Command-Line Options for Diagnostic Messages

-disassemble

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

Syntax

`-dis[assemble]`

Remarks

This option is global.

-help

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

Syntax

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

The options for *keyword* are:

`all`

Show all standard options

`group=`*`keyword`*

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

`[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.

-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 (default).
- `n` - Maximum number of errors to show, such as `-maxwarnings`.

-maxwarnings

Specifies the maximum number of warning messages to show.

Syntax

`-maxwarnings` *`max`*

`max`

Specifies the number of warning messages. Common values are:

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

-msgstyle

Controls the style used to show error and warning messages.

Syntax

`-msgstyle` *`keyword`*

The options for *keyword* are:

`gcc`

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

`ide`

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

`mpw`

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

`parseable`

Uses context-free machine parseable message style.

`std`

Uses standard message style. This is the default.

`enterpriseIDE`

Uses Enterprise-IDE message style.

-nofail

Continues processing after getting error messages in earlier files.

Syntax

```
-nofail
```

-progress

Shows progress and version information.

Syntax

```
-progress
```

-S

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

Syntax

```
-S
```

-stderr

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.

-verbose

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

Syntax

```
-v[erbose]
```

Remarks

This option also gives progress and version information.

-version

Displays version, configuration, and build data.

Syntax

```
-v[ersion]
```

-timing

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

Syntax

```
-timing
```

-warnings

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

Syntax

`-w[arnings] 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. Refer Table 7.1 for a list of warning messages turned on by the `-w[arnings] on` command.

Equivalent to #pragma warning on

`most`

Turns on most warnings.

`all`

Turns on almost all warnings and require prototypes.

`full`

Turns on all warning messages and require prototypes. This option is likely to generate spurious warnings.

NOTE `-warnings full` should be used before using any other options that affect warnings. For example, use `-warnings full -warnings noanyptrintconv` instead of `-warnings noanyptrintconv -warnings full`.

`[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`

`[no]pragmas | [no]illpragmas`

Issues warning messages on invalid pragmas. Enabled when `most` is used. Equivalent to

`#pragma warn_illpragma`

`[no]empty[decl]`

Issues warning messages on empty declarations. Enabled when `most` is used. Equivalent to

`#pragma warn_emptydecl`

`[no]possible | [no]unwanted`

Issues warning messages on possible unwanted effects. Enabled when `most` is used. Equivalent to

`#pragma warn_possunwanted`

`[no]unusedarg`

Issues warning messages on unused arguments. Enabled when `most` is used. Equivalent to

`#pragma warn_unusedarg`

`[no]unusedvar`

Issues warning messages on unused variables. Enabled when `most` is used. Equivalent to

`#pragma warn_unusedvar`

`[no]unused`

Same as

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

Enabled when `most` is used.

`[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. Enabled when `most` is used. Equivalent to

`#pragma warn_extracomma`

`[no]extended`

Extended error checking. Enabled when `most` is used. Equivalent to either:

`#pragma extended_errorcheck`

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

Issues warning messages on hidden virtual functions. Enabled when `most` is used. Equivalent to

`#pragma warn_hidevirtual`

`[no]implicit[conv]`

Issues warning messages on implicit arithmetic conversions. Enabled when `all` is used. Implies

`-warn impl_float2int,impl_signedunsigned`

`[no]impl_int2float`

Issues warning messages on implicit integral to floating conversions. Enabled when `all` is used. Equivalent to

`#pragma warn_impl_i2f_conv`

`[no]impl_float2int`

Issues warning messages on implicit floating to integral conversions. Enabled when `all` is used. Equivalent to

`#pragma warn_impl_f2i_conv`

`[no]impl_signedunsigned`

Issues warning messages on implicit signed/unsigned conversions. Enabled when `all` is used.

`[no]notinlined`

Issues warning messages for functions declared with the `inline` qualifier that are not inlined. Enabled when `full` is used. Equivalent to

`#pragma warn_notinlined`

`[no]largeargs`

Issues warning messages when passing large arguments to unprototyped functions. Enabled when `most` is used. Equivalent to

`#pragma warn_largeargs`

`[no]structclass`

Issues warning messages on inconsistent use of `class` and `struct`. Enabled when `most` is used. Equivalent to

`#pragma warn_structclass`

`[no]padding`

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

`#pragma warn_padding`

`[no]notused`

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

`#pragma warn_resultnotused`

`[no]missingreturn`

Issues warning messages when a return without a value in non-void-returning function occurs. Enabled when `most` is used. 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. Enabled when `full` is used.

`[no]anyptrintconv`

Issues warning messages on any conversion of pointers to integers. Enabled when `full` is used. Equivalent to

`#pragma warn_ptr_int_conv`

`[no]undef[macro]`

Issues warning messages on the use of undefined macros in `#if` and `#elif` conditionals. Enabled when `full` is used. Equivalent to

`#pragma warn_undefmacro`

`[no]filecaps`

Issues warning messages when `#include ""` directives use incorrect capitalization. Enabled when `most` is used. Equivalent to

`#pragma warn_filenamecaps`

`[no]sysfilecaps`

Issue warning messages when `#include <>` statements use incorrect capitalization. Enabled when `most` is used. Equivalent to

`#pragma warn_filenamecaps_system`

`[no]tokenpasting`

Issue warning messages when token is not formed by the `##` preprocessor operator. Enabled when `most` is used. Equivalent to

`#pragma warn_illtokenpasting`

`[no]relax_i2i_conv`

Relax implicit arithmetic conversion warnings on certain implicit conversions. Equivalent to

`#pragma relax_i2i_conv`

`display | dump`

Display list of active warnings.

Remarks

Table 7.1 lists the equivalent command option of the warning messages turned on by the `-w[arnings] on` command.

Table 7.1 Warnings turned on by the `-w[arnings] on` command

on/most	all (includes most)	full (includes all and most)
[no]pragmas | [no]illpragmas	[no]implicit[conv]	[no]notinlined
[no]possible | [no]unwanted	[no]impl_int2float	[no]notused
[no]empty[decl]	[no]impl_float2int	[no]ptrintconv
[no]unusedarg	[no]impl_signedunsigned	[no]anyptrintconv
[no]unusedvar		[no]undef[macro]
[no]unused		[no]padding
[no]extracomma | [no]comma		
[no]extended		
[no]hidevirtual | [no]hidden[virtual]		
[no]largeargs		
[no]structclass		
[no]missingreturn		
[no]unusedexpr		
[no]filecaps		

Table 7.1 Warnings turned on by the `-w[arnings] on` command (*continued*)

on/most	all (includes most)	full (includes all and most)
[no]sysfilecaps		
[no]tokenpasting		

-wraplines

Controls the word wrapping of messages.

Syntax

```
-wraplines
```

```
-nowraplines
```

8 Command-Line Options for Preprocessing

-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>`.

-cwd

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

Syntax

`-cwd` *`keyword`*

The options for *keyword* are:

`explicit`

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

`include`

Begins searching in directory of referencing file.

`proj`

Begins searching in current working directory (default).

`source`

Begins searching in directory that contains the source file.

Remarks

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

-D+

Same as the `-define` option.

Syntax

`-D+`*`name`*

The parameters are:

`name`

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

-define

Defines a preprocessor symbol.

Syntax

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

The parameters are:

`name`

The symbol name to define.

`value`

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

-E

Tells the command-line tool to preprocess source files.

Syntax

```
-E
```

Remarks

This option is global and case sensitive.

-EP

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

Syntax

```
-EP
```

Remarks

This option is global and case sensitive.

-gccincludes

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

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.

-I-

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

Syntax

`-I-`

`-i-`

Remarks

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

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

This command is global.

-I+

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

Syntax

`-I+`*`path`*

```
-i path
```

The parameters are:

`path`

The non-recursive access path to append.

Remarks

This command is global and case-sensitive.

-include

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

Syntax

```
-include file
```

`file`

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

Remarks

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

-ir

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

Syntax

```
-ir path
```

`path`

The recursive access path to append.

-P

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

Syntax

`-P`

Remarks

This option is global and case-sensitive.

-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*`

-preprocess

Preprocesses the source files. This command is global.

Syntax

`-preprocess`

-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`.

-prefix

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

Syntax

```
-prefix file
```

-noprecompile

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

Syntax

```
-noprecompile
```

-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.

-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.

-U+

Same as the -undefine option.

Syntax

`-U+`*`name`*

-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

Command-Line Options for Library and Linking

-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.

-nolink

Compiles the source files, without linking.

Syntax

```
-nolink
```

Remarks

This command is global.

-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.

10

Command-Line Options for Object Code

-c

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

Syntax

```
-c
```

Remarks

This option is global.

-codegen

Instructs the compiler to compile without generating object code.

Syntax

```
-codegen
-nocodegen
```

Remarks

This option is global.

-enum

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.

-min_enum_size

Specifies the size, in bytes, of enumerated types.

Syntax

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

Remarks

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

-ext

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 setting is `.o`.

-strings

Controls how string literals are stored and used.

Remarks

`-str[ings]` *`keyword`*`[, ...]`

The *keyword* arguments are:

`[no]pool`

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

`[no]reuse`

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

`[no]readonly`

Make all string constants read-only. This is the default.

11

Command-Line Options for Optimization

-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 `-inline on`, `-inline auto`. This does not turn on the `aggressive_inline` feature.

-ipa

Controls Interprocedural Analysis (IPA) that lets the compiler generate better optimizations by evaluating all the functions and data objects in a file before generating code.

Syntax

```
-ipa file | function | off
```

`function | off`

Per-function optimization. This is the default option.

`file`

Per file optimization.

Remarks

See “Interprocedural Analysis” on page 281.

WARNING!

Using IPA mode from command-line tools is more complicated.

Use the `off` or `function` arguments to turn interprocedural analysis off. This is the default setting.

Use the `file` argument to apply interprocedural analysis at the file level. For example, if the name of the compiler is `mwcc`, this command:

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

generates object code and applies this optimization to file `file1.c` and then `file2.c`, but does not apply the optimization across both files. For each source file, this command generates a regular object code file (a file with a name that ends with “.o” or “.obj”), which is empty. It also generates an additional file ending with “.irobj”. This additional object code file contains the object code to which the compiler has applied interprocedural analysis.

This example compiles the same source files again, applies file-level analysis, then links object code into an output file named `myprog`:

```
mwcc -o myprog -ipa file -c file1.c file2.c
```

-O

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

Syntax

`-O`

Remarks

Provided for backwards compatibility.

-O+

Controls optimization settings.

Syntax

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

The *keyword* arguments are:

`0`

Equivalent to `-opt off`.

`1`

Equivalent to `-opt level=1`.

`2`

Equivalent to `-opt level=2`.

`3`

Equivalent to `-opt level=3`.

`4`

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

`p`

Equivalent to `-opt speed`.

`s`

Equivalent to `-opt space`.

Remarks

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

-opt

Specifies code optimization options to apply to object code.

Remarks

`-opt`*`keyword`* `[,...]`

The *keyword* arguments are:

`off | none`

Suppresses all optimizations. This is the default.

`on`

Same as `-opt level=2`

`all | full`

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

`l[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.

[no]peep[hole]

Peephole optimization. Equivalent to #pragma peephole.

[no]schedule

Performs instruction scheduling.

display | dump

Displays complete list of active optimizations.

12

Command-Line for Power Architecture Processors

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

Naming Conventions

Table 12.1 lists the names of the CodeWarrior command line tools.

Table 12.1 Power Architecture command line tools

This tool...	does these tasks...
`mwasmeppc`	translates assembly language source code into object code
`mwcceppc`	translates C and C++ source code into object code
`mwldeppc`	links object code into a loadable image file

Specifying Source File Locations

The build tools use several environment variables at build time to search for `include` files, libraries, and other source files. All of the variables mentioned here are lists which are separated by semicolons (“`;`”) in Windows operating systems and colons (“`:`”) in Solaris operating systems.

Unless `-nodefaults` is passed to on the command line, the compiler searches for an environment variable called `MWCEABIPPCIncludes` or `MWCIncludes` (in that order). These variables contain a list of system access paths to be searched after the system access paths specified by the user. The assembler also does this, using the variables `MWAsmEABIPPCIncludes` or `MWAsmIncludes`.

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

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

If you are only building for one target, it is okay to use `MWCIncludes`, `MWAsmIncludes`, `MWLibraries`, and `MWLibraryFiles`. The target-specific versions of the variables come in handy when targeting multiple targets, since the target-specific variables override the generic variables. Note that if the target-specific variable exists, the generic variable will not be used; the contents of the two variables will not be combined.

Licensing Command-Line Options

-fullLicenseSearch

Continues the search for a license file on the host computer.

Syntax

```
-fullLicenseSearch
```

Remarks

A license file unlocks features and capabilities in CodeWarrior tools. This option extends the normal search for a valid `license.dat` file.

Each time they are invoked, the command-line compiler, stand-alone assembler, and linker search on the host computer in this order until they find a valid license file in this order:

- the directory specified in a `-license` option
- the directory containing the command-line tool

- the current working directory
- the directory containing the CodeWarrior IDE

When this option is not used, the tool stops when it finds a valid license file. With this option, the tool searches all paths to read all valid licenses.

-license

Specifies a location on the host computer to search for a license file.

Syntax

```
-license location
```

where *location* is the path of a directory that contains a valid license file named `license.dat`.

Remarks

A license file unlocks features and capabilities in CodeWarrior tools.

Diagnostic Command-Line Options

-g

Generates DWARF 1.*x*-conforming debugging information.

Syntax

```
-g[dwarf]
```

Remarks

This option is global. This option is equivalent to

```
-sym dwarf-1,full
```

-gdwarf-2

Generates DWARF-2.*x*-conforming debugging information.

Syntax

```
-gdwarf-2
```

Remarks

This option is global. This option is equivalent to

```
-sym dwarf-2,full
```

-fmt

Equivalent to the `-format` option.

Syntax

```
-fmt x | nox
```

-format

Specifies the style of mnemonics to show in disassemblies.

Syntax

```
-format x | nox
```

Remarks

To show extended mnemonics in a disassembly, use

```
-format x
```

This option is the default.

To show regular mnemonics in a disassembly, use

```
-format nox
```

This is a linker option.

-listclosure

Controls the appearance of symbol closures in the linker's map file.

Syntax

```
-listclosure
-nolistclosure
```

Remarks

This option also generates a map file if the `-map` option has not already been specified.

This is a linker option.

-listdwarf

Controls the appearance of DWARF debugging information in the linker's map file.

Syntax

```
-listdwarf
-nolistdwarf
```

Remarks

This option also generates a map file if the `-map` option has not already been specified.

This is a linker option.

-map

Generates a text file that describes the contents of the linker's output file.

Syntax

`-map [`*`filename`*`]`

Remarks

The default value for *filename* is the name of the linker's output file with a `.MAP` file name extension.

This is a linker option.

-mapunused

Controls the appearance of a list of unused symbols in the map file.

Syntax

`-mapunused`

`-nomapunused`

Remarks

This option also generates a map file if the `-map` option has not already been specified.

This is a linker option.

-sym

Specifies global debugging options.

Syntax

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

The choices for *keyword* are:

`off`

Do not generate debugging information. This option is the default.

`on`

Generate DWARF-1-conforming debugging information.

`dwarf-1`

Generate DWARF-1-conforming debugging information.

`full[path]`

Store absolute paths of source files instead of relative paths.

`dwarf-2`

Generate DWARF-2-conforming debugging information.

-unused

Equivalent to the `-mapunused` option.

Syntax

```
-unused
-nounused
```

Library and Linking Command-Line Options

-codeaddr

Sets the runtime address of the executable code.

Syntax

```
-codeaddr addr
```

Remarks

The *addr* value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with `0x`. The default is `65536`.

The linker ignores this option if you invoke the linker with the `-lcf` option.

This is a linker option.

-ConvertArchiveToPartialLink

Extracts all objects from the library files (.a) and then puts them into a partially linked file (plf).

Syntax

```
-ConvertArchiveToPartialLink archives -o filename
```

where *archives* is the list of archive files, and *filename* is the name of the output PLF file.

Example

```
$mwldeppc.exe -ConvertArchiveToPartialLink
     MSL_C.PPCEABI.bare.E.UC.a Runtime.PPCEABI.E.UC.a -o
     XXX.plf
```

Remarks

This linker command can be used for a project with only archive files (MSL C archive) as the project would normally generate an empty `plf`.

Use `-o` option to specify the name of the output PLF file. If `-o` option is not provided to the linker then the linker will generate the file with a default `a.out` filename.

While working with this linker command, if we link any object file (`*.o`), other than archive (`*.a`), then the output file (`*.plf`) will even contain the contents of linked object file, along with the usual archive contents.

It has been observed that all `.plf` files converted from the MSL archives have the `.ctor` and `.dtor` section. `.plf` files converted from `Wii` archives do not have the `.ctor` and `.dtor` section.

While working with the CodeWarrior IDE:

- the output file is set by default to `debug.elf`, it should be changed to *.plf.
- the PLF also contains dwarf info because default dwarf info option is enabled.

-dataaddr

Sets the loading address of the data.

Syntax

`-dataaddr` *`addr`*

Remarks

The *addr* value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with `0x`. The default is the address after the code and large constant sections.

The linker ignores this option if the linker is invoked with the `-lcf` option.

This is a linker option.

-genbinary

Controls the generation of a binary file.

Syntax

`-genbinary` *`none`* `|` *`one`* `|` *`multiple`*

Remarks

To generate no binary file even if s-record generation is `on`, use

```
-genbinary none
```

This option is the default.

To generate a single binary file with all the loadable code and data, even if s-record generation is `off`, use

```
-genbinary one
```

To generate separate binary files for each MEMORY directive, even if s-record generation is off, use

```
-genbinary multiple
```

This is a linker option.

-heapaddr

Sets the runtime address of the heap.

Syntax

`-heapaddr` *`addr`*

Remarks

The *addr* value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with `0x`. The default is

`stack_address` `-` `(`*`heap_size`* `+` *`stack_size`*`)`

where *stack_address* is the address of the stack, *heap_size* is the size of the heap, and *stack_size* is the size of the stack.

This is a linker option.

-heapsize

Sets the runtime size of the heap, in kilobytes.

Syntax

`-heapsize` *`size`*

Remarks

The default value for *size* is `1024`.

This is a linker option.

-lcf

Uses the code and data addresses specified in a linker command file.

Syntax

`-lcf` *`filename`*

Remarks

The filename argument is the name of a linker command file. The file must have a `.lcf` file name extension. The linker ignores the `-codeaddr`, `-dataaddr`, `-sdataaddr`, and `-sdata2addr` options if it uses the `-lcf` option.

This is a linker option.

-library

Generates a static library.

Syntax

`-library`

Remarks

This option is global. This is a linker option.

-linkmode

Controls the performance of the linker.

Syntax

`-linkmode` *`keyword`*

The choices for *keyword* are:

`lessram`

Use little memory but take more processing time.

`normal`

Use a medium amount of memory for medium processing time. This is the default.

`moreram`

Use lots of memory to improve processing time.

Remarks

This is a linker option.

-main

Specifies the main entry point for the executable image.

Syntax

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

Remarks

The maximum length of *symbol* is 63 characters. The default is `__start`.

This is a linker option.

-model

Specifies the addressing mode that the linker uses when resolving references.

Syntax

`-model` *`keyword`*

The choices for *keyword* are:

`absolute`

Use absolute executable and data addressing. This choice is the default.

`sda_pic_pid`

Use position-independent addressing executable code and data.

Remarks

This is a linker option.

-noentry

Specifies no entry point for the executable image.

Syntax

`-noentry`

Remarks

The linker uses the main entry point to determine which objects/functions to add to your application that are referenced from that entry point. In absence of an entry point, the application will be empty (completely deadstripped) resulting in an linker error.

There are several ways to pass other entry points to the linker for objects that are not referenced from the main entry point.

- use the linker command file directives `TERM` or `INIT`
- use `__declspec(export)`
- use the lcf directives `FORCEFILES` or `FORCEACTIVE`

For example, if you have a simple reset vector function which simply calls your startup code (call the startup code __start and __reset for the reset vector function for this example), you could do the following :

- use `-m __start` at the command prompt
- use `ENTRY(__start)` in the Linker Command File
- use `INIT(__reset)` at the command prompt
- use `FORCEACTIVE(__reset)` in the Linker Command File

- use `__declspec(export) void __reset(void) {__start;}` in the source.

-nomain

Equivalent to [`-noentry`](#).

Syntax

```
-nomain
```

-opt_partial

Finishes a partial link operation.

Syntax

```
-opt_partial
```

Remarks

This option allows the use of a linker command file, creates tables for C++ static constructors, C++ static destructors, and C++ exceptions. This option also tells the linker to build an executable image even if some symbols cannot be resolved.

This is a linker option.

-partial

Does not report error messages for unresolved symbols.

Syntax

```
-partial
```

Remarks

This option tells the linker to build a reloadable object file even if some symbols cannot be resolved.

This is a linker option.

-r

Equivalent to `-partial`.

Syntax

```
-r
```

Remarks

This option tells the linker to build a reloadable object file even if some symbols cannot be resolved.

This is a linker option.

-r1

Equivalent to `-opt_partial`.

Syntax

```
-r1
```

Remarks

This option allows the use of a linker command file, creates tables for C++ static constructors, C++ static destructors, and C++ exceptions. This option tells the linker to build a reloadable object file even if some symbols cannot be resolved.

This is a linker option.

-r2

Equivalent to `-resolved_partial`.

Syntax

```
-r2
```

Remarks

This option first allows the use of a linker command file, creates tables for C++ static constructors, C++ static destructors, and C++ exceptions.

This is a linker option.

-resolved_partial

Finishes a partial link operation and issues error messages for unresolved symbols.

Syntax

```
-resolved_partial
```

Remarks

This option first allows the use of a linker command file, creates tables for C++ static constructors, C++ static destructors, and C++ exceptions.

This is a linker option

-sdataaddr

Sets the loading address of small data.

Syntax

`-sdataaddr` *`addr`*

Remarks

The *addr* value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with `0x`. The default is the address after the large data section.

The linker ignores this option if the linker is invoked with the `-lcf` option.

This is a linker option.

-sdata2addr

Sets the loading address of small constant data.

Syntax

`-sdata2addr` *`addr`*

Remarks

The *addr* value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with `0x`. The default is the address after the small data section.

The linker ignores this option if the linker is invoked with the `-lcf` option.

This is a linker option.

-sdatathreshold

Limits the size of the largest objects in the small data section.

Syntax

```
-sdata[threshold] size
```

Remarks

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

You can override this option for a variable in your source code like this

```
    __declspec(section ".sdata") extern int bigobj[25];
```

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

This is a linker option.

-sdata2threshold

Limits the size of the largest objects in the small constant data section.

Syntax

```
-sdata2[threshold] size
```

Remarks

The *size* value specifies the maximum size, in bytes, of all objects in the small constant data section (typically named ".sdata2"). The linker places constant objects that are greater than this size in the constant data section (typically named ".rodata") instead.

You can override this option for a variable in your source code like this

```
    __declspec(section ".sdata2") extern int bigobj[] =
    { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

The default for *size* is `8`.

This is a linker option.

-show

Specifies the information to list in a disassembly.

Syntax

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

The choices for *keyword* are:

`only | none`

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

`all`

Shows binary, executable code, detailed, data, extended, and exception information in the disassembly.

`binary | nobinary`

Shows or does not show address and op-code values.

`code | nocode`

Shows or does not show executable code sections.

`text | notext`

Equivalent to the `code` and `nocode` choices, respectively.

`data | nodata`

Shows or does not show data sections.

`detail | nodetail`

Shows or does not show extra information.

`extended | noextended`

Shows or does not show extended mnemonics.

`exceptions | noexceptions`

Shows or does not show C++ exception tables. This option also shows data sections.

`xtab[les] | noxtab[les]`

Equivalent to the `exceptions` and `noexceptions` choices, respectively.

`headers | noheaders`

Shows or does not show object header information.

`debug | nodebug`

Shows or does not show debugging information.

`dwarf | nodwarf`

Equivalent to the `debug` and `nodebug` choices, respectively.

`tables | notables`

Shows or does not show character string and symbol tables.

`source | nosource`

Interleaves the code dissassembly with c or c++ source code.

Remarks

The default setting for this option is

`-show binary,code,data,extended,headers,tables`

This is a linker option.

-sortsrec

Sort the records in an S-record file in ascending address order.

Syntax

```
-sortsrec
```

Remarks

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

-srec

Generates an S-record file.

Syntax

`-srec [`*`file-name`*`]`

Remarks

The default value for file-name is the name of the linker's output file with a `.mot` file name extension.

This is a linker option.

-sreceol

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

Syntax

`-sreceol `*`keyword`*

The choices for *keyword* are:

`mac`

Use Mac OS®-style end-of-line format.

`dos`

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

`unix`

Use 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.

-sreclength

Specify the length of S-records.

Syntax

`-sreclength `*`value`*

The choices for *value* are from 8 to 255. The default is 26.

Remarks

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

This is a linker option.

-stackaddr

Sets the runtime address of the stack.

Syntax

```
-stackaddr addr
```

Remarks

The *addr* value is an address, in decimal or hexadecimal format. Hexadecimal values must begin with `0x`. The default is `0x3dff0`.

This is a linker option.

-stacksize

Sets the runtime size of the stack, in kilobytes.

Syntax

```
-stacksize size
```

Remarks

The default value for *size* is `64`.

This is a linker option.

-strip_partial

Removes unreferenced objects on a partially linked image.

Syntax

```
-strip_partial
```

Remarks

Use this option with either the `-opt_partial` or `-resolved_partial` options.

This is a linker option.

-tune_relocations

Ensures that references made by the linker conform to the PowerPC EABI (Embedded Application Binary Interface) or position-independent ABI (Application Binary Interface).

Syntax

```
-tune_relocations
```

Remarks

Use this option only with the `-abi eabi` and `-abi sda_pic_pid` option to ensure that references in the executable image conform to these ABIs.

To conform to both of these ABIs, the linker will modify relocations that do not reach the desired executable code. The linker first converts near branch instructions to far branch instructions. Then it will convert absolute branches to PC-relative branches. For branches that cannot be converted to far or PC-relative addressing, the linker will generate branch islands.

To conform to the SDA PIC/PID ABI, the linker will generate the appropriate style of addressing.

This option is global. This is a linker option.

-xtables

Equivalent to `-show exceptions` or `-show noexceptions`.

Syntax

```
-xtables on | off
```

Remarks

This is a linker option.

-stdlib

Uses standard system library access paths as specified by the environment variable `%MWLibraries%` to add system libraries as specified by the environment variable `%MWLibraryFiles%` at the end of link order.

Syntax

```
-stdlib
```

```
-nostdlib
```

Remarks

This command is global. This is a linker option.

-L+

Adds a new library search path to the default settings list.

Syntax

`-L+`*`path`*

`-l `*`path`*

The parameters are:

`path`

The search path to append.

Remarks

This command is global and case-sensitive.

-lr

Adds a recursive library search path to the default settings list.

Syntax

```
-lr path
```

The parameters are:

`path`

The recursive library search path to append.

Remarks

This command is global. This is a linker option.

-l+

Adds a library by searching access path for a specified library filename.

Syntax

`-l+file`

The parameters are:

`file`

Name of the library file to search.

Remarks

The linker searches access path for the specified `lib<file>.<ext>`, where `<ext>` is a typical library extension. If the file is not found then search for `<file>.` This command is case-sensitive.

Code Generation Command-Line Options

-abi

Chooses which ABI (Application Binary Interface) to conform to.

Syntax

`-abi` *`keyword`*

The choices for *keyword* are:

`eabi`

Use the Power Architecture Embedded ABI. This choice is the default.

`SysV`

Use the UNIX System V ABI without GNU extensions.

`SuSE`

Use the SuSE® Linux ABI with GNU extensions.

`YellowDog`

Use the Yellow Dog™ Linux ABI with GNU extensions

`sda_pic_pid`

Use position-independent addressing executable code and data.

Remarks

This option is global.

-align

Specifies structure and array alignment.

Syntax

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

The choices for *keyword* are:

`power[pc]`

Use conventional Power Architecture alignment. This choice is the default.

`mac68k`

Use conventional Mac OS® 68K alignment.

`mac68k4byte`

Use Mac OS® 68K 4-byte alignment.

`array[members]`

Align members of arrays, too.

-altivec_move_block

Controls the use of Altivec instructions to optimize block moves.

Syntax

```
-altivec_move_block
-noaltivec_move_block
```

Remarks

The default setting is `-noaltivec_move_block`.

-big

Generates object code and links an executable image to use big-endian data formats.

Syntax

```
-big
```

Remarks

This is the default setting for the compiler and linker.

-common

Moves uninitialized data into a common section.

Syntax

```
-common on | off
```

Remarks

The default is `off`.

-fatext

Use `eppc.o` as the file name extension for object files.

Syntax

```
-fatext
```

Remarks

Normally, the compiler generates object code files that have a file name extension of `.o`. This option tells the compiler to use `eppc.o` as a file name extension instead. If the compiler is invoked with this option and the compiler invokes the linker, the linker will search for object files that use the `eppc.o` file name extension.

-fp

Controls floating-point code generation.

Syntax

`-fp *keyword*`

The choices for *keyword* are:

`none | off`

No floating point code generation.

`soft[ware]`

Use software libraries to perform floating-point operations. This is the default.

`hard[ware]`

Use the processor's built-in floating-point capabilities to perform floating-point operations.

`dpfp`

Use the processor's double-precision floating-point capabilities on the e500v2 processor.

`spfp`

Use software libraries for floating-point operations that use the `double` data type and use the e500 SPE-EFPU floating-point capabilities for other floating-point operations.

`spfp_only`

Use to have the compiler consider `double` and `long double` data types as `floating point`. This option is only supported for e200 (Zen or VLE) and e500v1 processors that support SPFP APU.

NOTE When you downgrade from `double` data type to a floating point you will lose precision and range. If your expected numbers are within the range supported by a `floating point` data type, then this option might dramatically speed up and shrink your code. Do not use this option if you have instances in your project that depend on the size of a `double` data type.

`fmadd`

Equivalent to `-fp hard -fp_contract`.

Remarks

When using the `-fp spfp_only` option, the size of a `double` data type changes to a `floating point` data type, if you have existing code that is expecting to find certain bits at certain locations of the exponent or significand, then you will have to change that code to expect 4 byte doubles. Your code can make a test as shown in Listing 12.1.

Listing 12.1 Example Test Code

```
if (sizeof(double) == 4) {
    ...
} else {
    ... }
```

The e500 and VLE library project files have targets and pre-built libraries (with `SP` in the name) that support this feature. Ensure you pick the right libraries to include in a project that supports this feature else you may call a function with a `8 byte double` parameter and only pass a `4 byte double` argument. The linker will report with a warning if you mix up the libraries - make sure you have linker warnings enabled.

If you have a library that doesn't use floating point, try setting it to `none` for the floating point model by using the `-fp none` option. Libaries with `none` floating point do not cause a warning when added to projects using another floating point model.

The sample code in Listing 12.2 assumes that you are using the `-fp spfp_only` option and have included `SP` libraries. Your existing code makes a call to a MSL math function and a user defined function that takes a `double` argument and returns a `double` data type.

Listing 12.2 Sample Code

```
#include <math.h>
extern double my_func(double);
extern double d1, d2;
void main()
{
        d1 = pow(d2, 2.0);
        d2 = my_func(d1);
}
```

Following can be observed while executing the sample code in Listing 12.2:

- `2.0` will be treated as a `4 byte double` constant (exactly like `2.0f`).
- Storage for `d1` and `d2` will be `4 bytes` each (exactly like floats).
- MSL will either inline or call a stub function for `pow` which will call `powf`.
- `my_func` will receive and return a `4 byte double`. As long as `my_func` doesn't do bit twiddling or require numbers not representable in a float, it will do its job correctly.

NOTE If you are using a Zen processor and are using the `-fp spfp_only` option, ensure passing `-spe_addl_vector` instead of `-spe_vector` in order to have the compiler generate Multiply-Add instructions.

-fp_contract

Generates fused multiply-addition instructions.

Syntax

```
-fp_contract
```

Remarks

This option is the same as the `-maf` option.

-func_align

Specifies alignment of functions in executable code.

Syntax

```
-func_align 4 | 8 | 16 | 32 | 64 | 128
```

Remarks

The default alignment is `4`. However, at an optimization level `4`, the alignment changes to `16`. If you are using `-func_align 4` (or `none`) and if you are compiling for VLE, then the linker will compress gaps between VLE functions:

- if those functions are not called by a Classic PPC function
- the function has an alignment greater than `4`.

NOTE Compression of the gaps will only happen on files compiled by the CodeWarrior compiler.

-gen-fsel

Deprecated. Use `-use_fsel` instead.

Syntax

```
-gen-fsel
-no-gen-fsel
```

-little

Generates object code and links an executable image to use little-endian data formats.

Syntax

```
-little
```

-maf

Controls the use of fused multiply-addition instructions.

Syntax

```
-maf on | off
```

Remarks

The `-maf on` option tells the compiler to generate fused multiply-addition operations instead of separate multiplication and addition instructions. The `-maf off` option tells the compiler to use separate multiplication and addition instructions.

-ordered-fp-compares

Controls the assumption of no unordered values in comparisons.

Syntax

```
-ordered-fp-compares
-no-ordered-fp-compares
```

Remarks

The default is `-no-ordered-fp-compares`.

-pool

Controls the grouping of similar-sized data objects.

Syntax

```
-pool[data] on | off
```

Remarks

Use this option to reduce the size of executable object code in functions that refer to many object of the same size. These similar-sized objects do not need to be of the same type. The compiler only applies this option to a function if the function refers to at least 3 similar-sized objects. The objects must be global or static.

At the beginning of the function, the compiler generates instructions to load the address of the first similar-sized object. The compiler then uses this address to generate 1 instruction for each subsequent reference to other similar-sized objects instead of the usual 2 instructions for loading an object using absolute addressing.

This option is equivalent to the `pool_data` pragma.

The default is `on`.

-processor

Generates and links object code for a specific processor.

Syntax

```
-proc[essor] keyword
```

The choices for *keyword* are:

```
401 | 403 | 405 | 505 | 509 | 5100 | 5200 | 555 | 56x |
601 | 602 | 603 | 603e | 604 | 604e | 740 | 74x | 750 |
75x | 801 | 821 | 823 | 850 | 85x | 860 | 86x | 87x | 88x
| 7400 | 744x | 7450 | 745x | 82xx| 8240 | 824x | 8260 |
827x | 8280 | 85xx | e300v1 | e300c1 | e300c2 | e300c3 |
e300c4 | e500v1 | e500v2 | e600 | Zen | 5565 | gekko |
generic
```

Remarks

The *keyword* parameter specifies the processor core. The default for *keyword* is `generic`. To identify which core your target device uses, refer to the product page of the target device on the Freescale web site.

If you specify the *keyword* as `e500v1`, `e500v2`, or `Zen`, the compiler uses `unsigned` as the default parameter for the `-char` switch.

The e300v1 *keyword* is deprecated.

-profile

Controls the appearance of calls to a profiler library at the entry and exit points of each function.

Syntax

```
-profile on | off
```

Remarks

The default is `off`.

-ppc_asm_to_vle

Converts regular Power Architecture assembler mnemonics to equivalent VLE (Variable Length Encoded) assembler mnemonics in the inline assembler.

Syntax

```
-ppc_asm_to_vle
```

Remarks

While translating assembly statements in C or C++ source code, the compiler will replace each regular Power Architecture assembler mnemonic with its matching VLE instruction if one exists. The mnemonics for VLE (Variable Length Encoding) instructions begin with “se_” or “e_”. The compiler’s inline assembler recognizes these mnemonics when the compiler is configured to generate VLE object code.

VLE instructions give extra flexibility in instruction encoding and alignment, allowing the compiler and linker to greatly reduce the size of runtime object code with only a small penalty in execution performance.

-rambuffer

Specifies a runtime address in which to store the executable image in RAM so that it may be transferred to flash memory.

Syntax

```
-rambuffer addr
```

Remarks

This option specifies information for a legacy flashing tool (some development boards that used the Power Architecture 821 processor). This tool required that the executable image must first be loaded to an area in RAM before being transferred to ROM. Do not use this option if your flash memory tool does not follow this behavior.

The CodeWarrior tools ignore this option if it is not used with the `-romaddr` option.

-readonlystrings

Places string constants in a read-only section.

Syntax

```
-readonlystrings
```

-relax_ieee

Controls the use of relaxed IEEE floating point operations.

Syntax

```
-relax_ieee
-norelax_ieee
```

Remarks

The default is `-relax_ieee`.

-romaddr

Generates a ROM image and specifies the image's starting address at runtime.

Syntax

`-romaddr` *`addr`*

-rostr

Equivalent to the `-readonlystrings` option.

Syntax

```
-rostr
```

-schedule

Controls the rearrangement of instructions to reduce the effects of instruction latency.

Syntax

```
-schedule on | off
```

Remarks

The default is `off`.

-spe_vector

Enables the SPE vector support.

Syntax

```
-spe_vector
```

Remarks

This option needs to be enabled when the floating point is set to SPFP or DPFP as both SPFP and DPFP require support from the SPE vector unit. If the option is not turned on, the compiler generates a warning and automatically enables the SPE vector generation.

-spe2_vector

Enables the SPE2 vector support

Syntax

```
-spe2_vector
```

Remarks

In order to use the SPE2 intrinsics:

- Include `<spe.h>` in the source file.
- From the EPPC Processor settings panel:
  - Select **Zen** from the **Processor** list box.
  - Select **spe2** from the **vector** list box.

NOTE SPE2 instructions are supported in standalone assembler and compiler's inline assembler. These instructions are currently not fully validated, users must use them at their own risks.

-spe_addl_vector

Enables the additional SPE fused multiply-add and multiply-subtract instuctions support.

Syntax

```
-spe_addl_vector
```

Remarks

The e200 z3 and z6 cores support 8 additional SPE fused multiply-add and multiply-subtract instructions. This option tells the compiler to generate the additional SPE instructions, when appropriate, for more optimized codes.

This option also turns on the `-spe_vector` option.

-strict_ieee

Specifies the use of strict IEEE floating point operations.

Syntax

```
-strict_ieee
```

Remarks

This option is the same as the `-norelax_ieee` option.

-use_lmw_stmw

Controls the use of multiple load and store instructions for function prologues and epilogues.

Syntax

```
-use_lmw_stmw on | off
```

Remarks

This option is only available for big-endian processors. This option is not available for big-endian e500v1 and e500v2 architectures when vector and double-precision floating-point instructions are used. The default is off.

-use_fsel

Controls the use of `fsel` instructions.

Syntax

```
-use_fsel on | off
```

Remarks

Do not turn on this option if the Power Architecture processor of your target platform does not have hardware floating-point capabilities that includes `fsel`. This option only has an effect if `-relax_ieee` is also specified on the command line. The default is `off`.

-use_isel

Controls the use of `isel` instructions.

Syntax

```
-use_isel on | off
```

Remarks

Do not turn on this option if the Power Architecture processor of your target platform does not implement the Freescale ISEL APU. The default is `off`.

-vector

Specifies AltiVec™ vector options.

Syntax

`-vector` *`keyword`*`[,...]`

The options for *keyword* are:

`on`

Generate AltiVec vectors and related instructions.

`off`

Do not generate AltiVec vectors and related instructions.

`vrsave`

Generate AltiVec vectors and instructions that use `VRSAVE` prologue and epilogue code.

`novrsave`

Do not use `VRSAVE` code. This option is the default.

-vle

Controls the use of the Variable Length Encoded (VLE) instruction set.

Syntax

```
-vle
```

Remarks

This option tells the compiler and linker to generate and lay out Variable Length Encoded (VLE) instructions, available on Zen variants of Power Architecture processors. VLE instructions give extra flexibility in instruction encoding and alignment, allowing the compiler and linker to greatly reduce the size of runtime object code with only a small penalty in execution performance.

This option also turns on the `-processor Zen` option.

Optimization Command-Line Options

-code_merging

Removes duplicated functions to reduce object code size.

Syntax

`-code_merging` *`keyword[,...]`*

The choices for *keyword* are:

`all`

Use the `all` argument to specify that the linker should remove all duplicate functions except one.

`safe`

Use the `safe` argument to specify that only duplicate functions marked as weak should be reduced to one function.

`aggressive`

Use the `aggressive` option to specify that the linker should ignore references to function addresses when considering which duplicate functions to remove.

`off`

Use the `off` argument to disable code merging optimization.

Remarks

This linker optimization removes duplicate copies of functions with identical executable code.

The linker does not apply this optimization to functions that have been declared with the `__declspec(no_linker_opts)` directive.

By default the code merging optimization is off.

-far_near_addressing

Simplifies address computations to reduce object code size and improve performance.

Syntax

`-far_near_addressing`

`-nofar_near_addressing`

Remarks

This linker optimization simplifies address computations in object code. If an address value is within the range that can be stored in the immediate field of the load immediate instruction, the linker replaces the address's two-instruction computation with a single instruction. An address value that is outside this range still requires two instructions to compute.

The ranges of values that may be stored in the immediate field is `-0x7fff` to `0x8000` for the regular `li` instruction and `-0x7ffff` to `0x80000` for `e_li`, the VLE (Variable Length Encoding) instruction.

The linker does not apply this optimization to functions that have been declared with the `__declspec(no_linker_opts)` directive.

-vle_bl_opt

Replaces branch instructions to reduce object code size.

Syntax

```
-ble_bl_opt
-noble_bl_opt
```

Remarks

This linker optimization replaces each 32-bit `e_bl` instruction with a 16-bit `se_bl` instruction for a function call when the span of memory between the calling function and called function is sufficiently close.

This optimization requires that the target processor has the Variable Length Encoding (VLE) extension.

The linker does not apply this optimization to functions that have been declared with the `__declspec(no_linker_opts)` directive.

-vle_enhance_merging

Removes duplicated functions that are called by functions that use VLE instructions to reduce object code size.

Syntax

```
-vle_enhance_merging
-novle_enhance_merging
```

Remarks

When applying the code merging optimization (`-code_merging`), this linker optimization ensures that function calls that use VLE (Variable Length Encoding) instructions will still be able to reach a function that has been removed. This optimization replaces the 16-bit `se_bl` instruction with a 32-bit `e_bl` instruction.

When this option is not used, the linker does not merge functions that are called by functions that use VLE instructions.

This optimization requires that the target processor has the Variable Length Encoding (VLE) extension. This optimization has no effect when the linker is not applying the code merging optimization.

The linker does not apply this optimization to functions that have been declared with the `__declspec(no_linker_opts)` directive.

-volatileasm

Controls whether or not inline assembly statements will be optimized.

Syntax

```
-volatileasm
```

```
-novolatileasm
```

13 Assembler

This chapter descibes the assembler:

- Syntax
- Directives
- Macros
- GNU Compatibility

Syntax

This section describes the syntax of assembly language statements. It consists of these topics:

- Assembly Language Statements
- Statement Syntax
- Symbols
- Constants
- Expressions
- Comments
- Data Alignment

Assembly Language Statements

The three types of assembly language statements are:

- Machine instructions
- Macro calls
- Assembler directives

Instructions, directives, and macro names are case insensitive: the assembler considers LWZ, Lwz, and lwz to be the same instruction

Remember these rules for assembly language statements:

1. The maximum length of a statement or an expanded macro is 512 characters.
2. A statement must reside on a single line. However, you can concatenate two or more lines by typing a backslash (\) character at the end of the line.
3. Each line of the source file can contain only one statement unless the assembler is running in GNU mode. (This mode allows multiple statements on one line, with semicolon separators.)

Statement Syntax

Listing 13.1 shows the syntax of an assembly language statement. Table 13.1 describes the elements of this syntax.

Listing 13.1 Statement Syntax

```
statement ::= [ symbol ] operation [ operand ] [ ,operand ]... [
comment ]

operation ::= machine_instruction | assembler_directive | macro_call

operand ::= symbol | constant | expression | register_name
```

Table 13.1 Syntax Elements

Element	Description
symbol	A combination of characters that represents a value.
machine_instruction	A machine instruction for your target processor.
assembler_directive	A special instruction that tells the assembler how to process other assembly language statements. For example, certain assembler directives specify the beginning and end of a macro.
macro_call	A statement that calls a previously defined macro.
constant	A defined value, such as a string of characters or a numeric value.
expression	A mathematical expression.

Table 13.1 Syntax Elements (*continued*)

Element	Description
register_name	The name of a register; these names are processor-specific.
comment	Text that the assembler ignores, useful for documenting your code.

Symbols

A *symbol* is a group of characters that represents a value, such as an address, numeric constant, string constant, or character constant. There is no length limit to symbols.

The syntax of a symbol is:

```
symbol ::= label | equate
```

In general, symbols have file-wide scope. This means:

1. You can access the symbol from anywhere in the file that includes the symbol definition.
2. You cannot access the symbol from another file.

However, it is possible for symbols to have a different scope, as described in the following sub-sections.

- Labels
- Non-Local Labels
- Local Labels
- Relocatable Labels
- Equates
- Case-Sensitive Identifiers

Labels

A *label* is a symbol that represents an address. A label's scope depends on whether the label is local or non-local.

The syntax of a label is:

```
label ::= local_label [ : ] | non-local_label[ : ]
```

The default settings are that each label ends with a colon (:), a label can begin in any column. However, if you port existing code that does not follow this convention, you should clear the **Labels must end with** ':' checkbox of the Assembler settings. After you clear the checkbox, you may use labels that do not end with colons, but such labels must begin in column 1.

Non-Local Labels

A *non-local label* is a symbol that represents an address and has file-wide scope. The first character of a non-local label must be a:

- letter (a-z or A-Z),
- period (.),
- question mark (?), or an
- underscore (_).

Subsequent characters can be from the preceding list or a:

- numeral (0-9), or
- dollar sign ($).

Local Labels

A *local label* is a symbol that represents an address and has local scope: the range forward and backward within the file to the points where the assembler encounters non-local labels.

The first character of a local label must be an at-sign (@). The subsequent characters of a local label can be:

- letters (a-z or A-Z)
- numerals (0-9)
- underscores (_)
- question marks (?)
- dollar sign. ($)
- periods (.)

NOTE You cannot export local labels; local labels do not appear in debugging tables.

Within an expanded macro, the scope of local labels works differently:

- The scope of local labels defined in macros does not extend outside the macro.
- A non-local label in an expanded macro does not end the scope of locals in the unexpanded source.

Listing 13.2 shows the scope of local labels in macros: the `@SKIP` label defined in the macro does not conflict with the `@SKIP` label defined in the main body of code.

Listing 13.2 Local Label Scope in a Macro

```
MAKEPOS .MACRO
        cmpwi    0,r3,0
        bge      @SKIP
        neg      r3,r3
@SKIP:                    ; Scope of this label is within
                          ; the macro
        .ENDM
START:
        lwz      r3,COUNT
        cmpw     0,r3, r4
        beq      @SKIP
        MAKEPOS
@SKIP:                    ; Scope of this label is START to
                          ; END excluding lines arising
                          ; from macro expansion
        addic    r3,r3,1
END:    blr
```

Relocatable Labels

The assembler assumes a flat 32-bit memory space. You can use the expressions of Table 13.2 to specify the relocation of a 32-bit label.

NOTE The assembler for your target processor may not allow all of these expressions.

Table 13.2 Relocatable Label Expressions

Expression	Represents
label	The offset from the address of the label to the base of its section, relocated by the section base address. It also is the PC-relative target of a branch or call. It is a 32-bit address.
label@l	The low 16-bits of the relocated address of the symbol.
label@h	The high 16-bits of the relocated address of the symbol. You can OR this with *label*`@l` to produce the full 32-bit relocated address.
label@ha	The adjusted high 16-bits of the relocated address of the symbol. You can add this to *label*`@l` to produce the full 32-bit relocated address.

Table 13.2 Relocatable Label Expressions

Expression	Represents
label@sdax	For labels in a small data section, the offset from the base of the small data section to the label. This syntax is not allowed for labels in other sections.
label@got	For processors with a global offset table, the offset from the base of the global offset table to the 32-bit entry for label.

Equates

An *equate* is a symbol that represents any value. To create an equate, use the `.equ` or `.set` directive.

The first character of an equate must be a:

- letter (a-z or A-Z),
- period (.),
- question mark (?), or
- underscore (_)

Subsequent characters can be from the preceding list or a:

- numeral (0-9) or
- dollar sign ($).

The assembler allows *forward equates*. This means that a reference to an equate can be in a file before the equate's definition. When an assembler encounters such a symbol whose value is not known, the assembler retains the expression and marks it as unresolved. After the assembler reads the entire file, it reevaluates any unresolved expressions. If necessary, the assembler repeatedly reevaluates expressions until it resolves them all or cannot resolve them any further. If the assembler cannot resolve an expression, it issues an error message.

NOTE The assembler must be able to resolve immediately any expression whose value affects the location counter.
If the assembler can make a reasonable assumption about the location counter, it allows the expression.

The code of Listing 13.3 shows a valid forward equate.

Listing 13.3 Valid Forward Equate

```
            .data
            .long  alloc_size
alloc_size  .set   rec_size + 4
; a valid forward equate on next line
rec_size    .set   table_start-table_end
            .text
;...
table_start:
; ...
table_end:
```

However, the code of Listing 13.4 is not valid. The assembler cannot immediately resolve the expression in the `.space` directive, so the effect on the location counter is unknown.

Listing 13.4 Invalid Forward Equate

```
;invalid forward equate on next line
rec_size    .set   table_start-table_end
            .space rec_size
            .text; ...
table_start:
; ...
table_end:
```

Case-Sensitive Identifiers

The **Case-sensitive identifiers** checkbox of the Assembler settings panel lets you control case-sensitivity for symbols:

- Check the checkbox to make symbols case sensitive — `SYM1`, `sym1`, and `Sym1` are three different symbols.
- Clear the checkbox to make symbols *not* case-sensitive — `SYM1`, `sym1`, and `Sym1` are the same symbol. (This is the default setting.)

Constants

The assembler recognizes three kinds of constants:

- Integer Constants
- Floating-Point Constants
- Character Constants

Integer Constants

Table 13.3 lists the notations for integer constants. Use the preferred notation for new code. The alternate notations are for porting existing code.

Table 13.3 Preferred Integer Constant Notation

Type	Preferred Notation	Alternate Notation
Hexadecimal	`$` followed by string of hexadecimal digits, such as `$deadbeef`.	`0x` followed by a string of hexadecimal digits, such as `0xdeadbeef`.
		`0` followed by a string of hexadecimal digits, ending with `h`, such as `0deadbeefh`.
Decimal	String of decimal digits, such as `12345678`.	String of decimal digits followed by `d`, such as `12345678d`.
Binary	`%` followed by a string of binary digits, such as `%01010001`.	String of binary digits followed by `b`, such as `01010001b`.

NOTE The assembler uses 32-bit signed arithmetic to store and manipulate integer constants.

Floating-Point Constants

You can specify floating-point constants in either hexadecimal or decimal format. The decimal format must contain a decimal point or an exponent. Examples are `1E-10` and `1.0`.

You can use floating-point constants only in data generation directives such as `.float` and `.double`, or in floating-point instructions. You cannot such constants in expressions.

Character Constants

Enclose a character constant in single quotes. However, if the character constant includes a single quote, use double quotes to enclose the character constant.

NOTE A character constant cannot include both single and double quotes.

The maximum width of a character constant is 4 characters, depending on the context. Examples are `'A'`, `'ABC'`, and `'TEXT'`.

A character constant can contain any of the escape sequences that Table 13.4 lists.

Table 13.4 Character Constant Escape Sequences

Sequence	Description
\b	Backspace
\n	Line feed (ASCII character 10)
\r	Return (ASCII character 13)
\t	Tab
\"	Double quote
\\	Backslash
\nnn	Octal value of \nnn

During computation, the assembler zero-extends a character constant to 32 bits. You can use a character constant anywhere you can use an integer constant.

Expressions

The assembler uses 32-bit signed arithmetic to evaluates expressions; it does not check for arithmetic overflow.

As different processors use different operators, the assembler uses an expression syntax similar to that of the C language. Expressions use C operators and follow C rules for parentheses and associativity.

NOTE To refer to the program counter in an expression, use a period (.), dollar sign ($), or asterisk (*).

Table 13.5 lists the expression operators that the assembler supports.

Table 13.5 Expression Operators

Category	Operator	Description
Binary	+	add
	-	subtract
	*	multiply
	/	divide
	%	modulo
	||	logical OR
	&&	logical AND
	|	bitwise OR
	&	bitwise AND
	^	bitwise XOR
	<<	shift left
	>>	shift right (zeros are shifted into high order bits)
	==	equal to
	!=	not equal to
	<=	less than or equal to
	>=	greater than or equal to
	<	less than
	>	greater than
Unary	+	unary plus
	-	unary minus
	~	unary bitwise complement

Table 13.5 Expression Operators (*continued*)

Category	Operator	Description
Alternate	<>	not equal to
	%	modulo
	|	logical OR
	||	logical XOR

Operator precedence is:

1. unary `+ - ~`
2. `* / %`
3. binary `+ -`
4. `<< >>`
5. `< <= > >=`
6. `== !=`
7. `&`
8. `^`
9. `|`
10. `&&`
11. `||`

Comments

There are several ways to specify comments:

1. Use either type of C-style comment, which can start in any column:

   ```
   // This is a comment.
   /* This is a comment. */
   ```

2. Start the comment with an asterisk (*) in the first column of the line.

NOTE The asterisk (*) must be the first character of the line for it to specify a comment. The asterisk has other meanings if it occurs elsewhere in a line.

3. Clear the **Allow space in operand field** checkbox of the Assembler settings panel. Subsequently, if you type a space in an operand field, all the remaining text of the line is a comment.

4. Anything following a # character is considered to be a comment. For example,

```
st    r3,0(r4)       # Store total
```

5. Anything following a `;` character is considered to be a comment, except in GNU compatibility mode, where `;` is a statement separator.

Data Alignment

The assembler's default alignment is on a natural boundary for the data size and for the target processor family. To turn off this default alignment, use the `alignment` keyword argument with to the `.option` directive.

NOTE The assembler does not align data automatically in the `.debug` section.

Directives

- Some directives may not be available for the assembler for your target processor.
- The default starting character for most directives is the period (.). However, if you clear the **Directives begin with '.'** checkbox of the Assembler settings panel, you can omit the period.
- You can use the C/C++ preprocessor format to specify several preprocessor directives .

Explanations are in these sections:

- Macro Directives
- Conditional Preprocessor Directives
- Section Control Directives
- Scope Control Directives
- Symbol Definition Directives
- Data Declaration Directives
- Assembler Control Directives
- Debugging Directives

Macro Directives

These directives let you create macros:

- `macro`
- `endm`

- mexit
- #define

For more information on macros, see “Macros” on page 187.

macro

Starts the definition of a macro.

```
label .macro [ parameter ] [ ,parameter ] ...
```

Parameters

label

Name you give the macro.

`parameter`

Optional parameter for the macro.

endm

Ends the definition of a macro.

```
.endm
```

mexit

Stops macro execution before it reaches the .endm directive. Program execution continues with the statement that follows the macro call.

```
.mexit
```

#define

Defines a C pre-processor macro with the specified parameters. Note that the C pre-processor is run on the assembler file before normal assembly. C pre-processor macros should not be confused with normal macros declared using the `MACRO` and `ENDM` directives.

```
#define name [ (parms) ] assembly_statement [ ; ] [ \ ]
assembly_statement [ ; ] [ \ ]
assembly_statement
parms ::= parameter [ ,parameter ]...
```

Parameters

`name`

Name you give the macro.

`parms`

List of parameters, separated by commas.

`assembly_statement`

Any valid assembly statement.

Remarks

To extend an *assembly_statement*, type a backslash (\) and continue the statement on the next line. To specify multiple assembly statements in the macro, type a semicolon and backslash (;\), then type a new assembly statement on the next line. If the assembler is in GNU mode, multiple statements can be on one line of code — separate them with semicolon characters (;).

NOTE For more information, see "Using the #define Directive" on page 193.

Conditional Preprocessor Directives

Conditional directives let you control whether compilation includes a block of code. These directives let you make multiple builds that are slightly different.

You must use conditional directives together to form a complete block. Several conditional directives are variations of `.if` that make it easier to establish blocks that test strings for equality, test whether a symbol is defined, and so on.

NOTE You can use the C/C++ preprocessor format to specify these conditional directives:

```
#if       #ifdef      #ifndef
#else     #elif       #endif
```

With two exceptions, these directives function identically whether their starting character is a pound sign (#) or a period. The exceptions are:

1. You cannot use the pound sign format in a macro.
2. The period form of `#elif` is `.elseif`.

The conditional preprocessor directives are:

- `if`
- `ifdef`
- `ifndef`
- `ifc`
- `ifnc`
- `endif`
- `elseif`
- `else`
- `Compatibility Conditional Directives`

if

Starts a conditional assembly block, making assembly conditional on the truth of a boolean expression.

```
.if bool-expr
```

Parameter

`bool-expr`

Any boolean expression.

Remarks

If `bool-expr` is true, the assembler processes the statements of the block. If `bool-expr` is false, the assembler skips the statements of the block.

Each `.if` directive must have a matching `.endif` directive.

ifdef

Starts a conditional assembly block, making assembly conditional on the definition of a symbol.

```
#ifdef symbol
```

Parameter

`symbol`

Any valid symbol.

Remarks

If previous code includes a definition for `symbol`, the assembler processes the statements of the block. If `symbol` is not defined, the assembler skips the statements of the block.

Each `.ifdef` directive must have a matching `.endif` directive.

ifndef

Starts a conditional assembly block, making assembly conditional on a symbol *not* being defined.

```
.ifndef symbol
```

Parameter

`symbol`

Any valid symbol.

Remarks

If previous code does *not* include a definition for `symbol`, the assembler processes the statements of the block. If there *is* a definition for `symbol`, the assembler skips the statements of the block.

Each `.ifndef` directive must have a matching `.endif` directive.

ifc

Starts a conditional assembly block, making assembly conditional on the equality of two strings.

```
.ifc string1, string2
```

Parameters

`string1`

Any valid string.

`string2`

Any valid string.

Remarks

If `string1` and `string2` are equal, the assembler processes the statements of the block. (The equality comparison is case-sensitive.) If the strings are *not* equal, the assembler skips the statements of the block.

Each `.ifc` directive must have a matching `.endif` directive.

ifnc

Starts a conditional assembly block, making assembly conditional on the *inequality* of two strings.

```
.ifnc string1, string2
```

Parameters

`string1`

Any valid string.

`string2`

Any valid string.

Remarks

If `string1` and `string2` are *not* equal, the assembler processes the statements of the block. (The inequality comparison is case-sensitive.) If the strings *are* equal, the assembler skips the statements of the block.

Each `.ifnc` directive must have a matching `.endif` directive.

endif

Ends a conditional assembly block. A matching `.endif` directive is mandatory for each type of `.if` directive.

```
.endif
```

elseif

Starts an alternative conditional assembly block, making assembly conditional on the truth of a boolean expression.

```
.elseif bool-expr
```

Parameter

`bool-expr`

Any boolean expression.

Remarks

If `bool-expr` is true, the assembler processes the statements of the block. If `bool-expr` is false, the assembler skips the statements of the block.

You can use this directive to create a logical, multilevel *if-then-else* statement, according to this syntax:

```
.if bool-expr statement-group
[ .elseif bool-expr statement-group ]...
[ .else statement-group ]
.endif
```

(In this syntax, `statement-group` is any group of assembly-language statements.)

The `.elseif` directive can be part of more complicated logical structures, such as:

```
.if bool-expr-1
    statement-group-1
.elseif bool-expr-2
    statement-group-2
.elseif bool-expr-3
    statement-group-3
.elseif bool-expr-4
    statement-group-4
.else
    statement-group-5
.endif
```

- If this structure's `bool-expr-1` is true, the assembler executes the `statement-group-1` assembly-language statements, then goes to the `.endif` directive.
- If `bool-expr-1` is false, the assembler skips `statement-group-1`, executing the first `.elseif` directive. If `bool-expr-2` is true, the assembler executes `statement-group-2`, then goes to the `.endif` directive.
- If `bool-expr-2` also is false, the assembler skips `statement-group-2`, *executing* the second `.elseif` directive.
- The assembler continues evaluating the boolean expressions of succeeding `.elseif` directives until it comes to a boolean expression that is true.
- If none of the boolean expressions are true, the assembler processes `statement-group-5`, because this structure includes an `.else` directive. (If none of the boolean values were true and there were no `.else` directive, the assembler would not process any of the statement groups.)

else

Starts an alternative conditional assembly block.

```
.else
```

Remarks

This directive is optional. The assembler processes the statements of the alternative conditional assembly block only if the expressions for an `.if` directive and any associated `.elseif` directives are false.

Compatibility Conditional Directives

For compatibility with other assemblers, the assembler supports these additional conditional directives:

- .ifeq if equal
- .ifne if not equal
- .iflt if less than
- .ifle if less than or equal
- .ifgt if greater than
- .ifge if greater than or equal

.ifeq   if equal

Starts a conditional assembly block, making assembly conditional on a string value being equal to zero.

```
.ifeq string
```

Parameter

`string`

Any valid string.

Remarks

If the `string` value equals 0, the assembler processes the statements of the block. If the `string` value does *not* equal 0, the assembler skips the statements of the block.

.ifne   if not equal

Starts a conditional assembly block, making assembly conditional on a string value *not* being equal to zero.

```
.ifne string
```

Parameter

`string`

Any valid string.

Remarks

If the `string` value is *not* equal to 0, the assembler processes the statements of the block. If the `string` value *does* equal 0, the assembler skips the statements of the block.

.iflt   if less than

Starts a conditional assembly block, making assembly conditional on a string value being less than zero.

```
.iflt string
```

Parameter

`string`

Any valid string.

Remarks

If the `string` value is less than 0, the assembler processes the statements of the block. If the `string` value equals or exceeds 0, the assembler skips the statements of the block.

.ifle if less than or equal

Starts a conditional assembly block, making assembly conditional on a string value being less than or equal to zero.

```
.ifle string
```

Parameter

`string`

Any valid string.

Remarks

If the `string` value is less than or equal to 0, the assembler processes the statements of the block. If the `string` value is *greater* than 0, the assembler skips the statements of the block.

.ifgt if greater than

Starts a conditional assembly block, making assembly conditional on a string value being greater than zero.

```
.ifgt string
```

Parameter

`string`

Any valid string.

Remarks

If the `string` value is greater than 0, the assembler processes the statements of the block. If the string value is less than or equal to 0, the assembler skips the statements of the block.

.ifge if greater than or equal

Starts a conditional assembly block, making assembly conditional on a the string value being greater than or equal to zero.

```
.ifge string
```

Parameter

`string`

Any valid string.

Remarks

If the `string` value is greater than or equal to 0, the assembler processes the statements of the block. If the `string` value is less than 0, the assembler skips the statements of the block.

Section Control Directives

These directives identify the different sections of an assembly file:

- `text`
- `data`
- `rodata`
- `bss`
- `sdata`
- `sdata2`
- `sbss`
- `text_vle`
- `debug`
- `previous`
- `offset`
- `section`

text

Specifies an executable code section; must be in front of the actual code in a file.

```
.text
```

data

Specifies an initialized read-write data section.

```
.data
```

rodata

Specifies an initialized read-only data section.

```
.rodata
```

bss

Specifies an uninitialized read-write data section.

```
.bss
```

sdata

Specifies a small data section as initialized and read-write.

```
.sdata
```

sdata2

Specifies a small data section as initialized and read-only.

```
.sdata2
```

sbss

Specifies a small data section as uninitialized and read-write.

```
.sbss
```

text_vle

Specifies a Variable length encoded section as read/execute.

```
.text_vle
```

debug

Specifies a debug section.

```
.debug
```

Remarks

If you enable the debugger, the assembler automatically generates some debug information for your project. However, you can use special directives in the debug section that provide the debugger with more detailed information. For more information on the debug directives, see "Debugging Directives" on page 184.

previous

Reverts to the previous section; toggles between the current section and the previous section.

```
.previous
```

offset

Starts a record definition, which extends to the start of the next section.

```
.offset [expression]
```

Parameter

`expression`

Optional initial location-counter value.

Remarks

Table 13.6 lists the only directives your can use inside a record.

Table 13.6 Directives within a Record

`.align`	`.double`	`.org`	`.textequ`
`.ascii`	`.equ`	`.set`	
`.asciz`	`.float`	`.short`	
`.byte`	`.long`	`.space`	

Data declaration directives such as `.byte` and `.short` update the location counter, but do not allocate any storage.

Example

Listing 13.5 shows a sample record definition.

Listing 13.5 Record Definition with Offset Directive

```
            .offset
top:        .short   0
left:       .short   0
bottom:     .short   0
right:      .short   0
rectSize    .equ     *
```

section

Defines a section of an ELF (Executable and Linkable Format) object file.

`.section name [,alignment [,type [,flags]]]`

Parameters

`name`

Name of the section.

`alignment`

Alignment boundary.

`type`

Numeric value for the ELF section type, per Table 13.7. The default `type` value is `1`: (SHT_PROGBITS).

`flags`

Numeric value for the ELF section flags, per Table 13.8. The default `flags` value is `0x00000002, 0x00000001`: (SHF_ALLOC+SHF_WRITE).

Table 13.7 ELF Section Header Types (SHT)

Type	Name	Meaning
0	NULL	Section header is inactive.
1	PROGBITS	Section contains information that the program defines.
2	SYMTAB	Section contains a symbol table.
3	STRTAB	Section contains a string table.
4	RELA	Section contains relocation entries with explicit addends.
5	HASH	Section contains a symbol hash table.
6	DYNAMIC	Section contains information used for dynamic linking.
7	NOTE	Section contains information that marks the file, often for compatibility purposes between programs.
8	NOBITS	Section occupies no space in the object file.
9	REL	Section contains relocation entries without explicit addends.

Table 13.7 ELF Section Header Types (SHT) (*continued*)

Type	Name	Meaning
10	SHLIB	Section has unspecified semantics, so does not conform to the Application Binary Interface (ABI) standard.
11	DYNSYM	Section contains a minimal set of symbols for dynamic linking.

Table 13.8 ELF Section Header Flags (SHF)

Flag	Name	Meaning
0x00000001	WRITE	Section contains data that is writable during execution.
0x00000002	ALLOC	Section occupies memory during execution.
0x00000004	EXECINSTR	Section contains executable machine instructions.
0xF0000000	MASKPROC	Bits this mask specifies are reserved for processor-specific purposes.

Remark

Use this directive to create arbitrary relocatable sections, including sections to be loaded at an absolute address.

Possible syntax forms

The section directive accepts a number of different syntax forms, partly for convenience and partly for compatibility with other assemblers. A section declaration requires four pieces of information: a section name, alignment, ELF section type (for example, SHT_PROGBITS) and ELF section flags (for example, SHF_ALLOC+SHF_EXECINSTR).

The possible syntax forms are as follows:

- Specify built-in section name.

  ```
  .section text
  ```

  This example specifies a built-in section name `text`. Equivalently, `.text` is also a valid syntax form.

Table 13.9 provides a list of all the possible values, together with their ELF types and ELF Section Header Flags.

Table 13.9 Built-in Section names with their ELF Types and Flags

Name	ELF Type	ELF Flag
.text	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
.data	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.rodata	SHT_PROGBITS	SHF_ALLOC
.bss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.sdata	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.sdata0	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.sdata2	SHT_PROGBITS	SHF_ALLOC
.sbss	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.sbss0	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
.sbss2	SHT_PROGBITS	SHF_ALLOC
.debug	SHT_PROGBITS	0
.text_vle	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR+ SHF_PE_EXECINSTR
.PPC.EMB.sdata0	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
.PPC.EMB.sbss0	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE

In general, `.text` is for instructions, `.data` for initialised data, `.rodata` for read-only data (constants) and `.bss` for uninitialised data. The additional forms like `.sdata` are for small data areas. The built-in section names are architecture-specific, and are intended to make access to data more efficient. The alignment used for these sections is architecture-specific and is usually `4`.

- Specify your own section name but get the attributes of the built-in section type.

  ```
  .section mySection,text
  ```

  This example is equivalent to writing `.text` except that the section will be called `mySection`.

- Specify the alignment, and optionally the ELF type and flags.

  ```
  .section name [,alignment [,type [,flags]]]
  ```

In the syntax above, if the `alignment` is not specified it defaults to `16`. If the `type` or `flags` are not specified, the defaults are as follows:

- If the `name` parameter is a built-in section name, the type and the flags are taken as specified in the Table 13.9 .
  For example, in the syntax form `.section text,8`
  the type is `SHT_PROGBITS` and the flags value is
  `SHF_ALLOC+SHF_EXECINSTR`.
- In all other cases, the default type is `SHT_PROGBITS` and the default flags value is `SHF_ALLOC+SHF_WRITE`, corresponding to a writeable data section.

• Specify the type and flags parameters in pre-defined characters, optionally in double quotes.

`.section mySection,4,"rx"` or `.section mySection,4,rx`

The values are additive. For example, `rx` is equivalent to
`SHF_ALLOC+SHF_WRITE+SHF_EXECINSTR`

NOTE If the syntax doesn't specify a type it defaults to `SHT_PROGBITS`

Table 13.10 provides a list of all the possible characters and their corresponding ELF Type and ELF Flags.

Table 13.10 Characters and their corresponding ELF Type and ELF Flags

Character	ELF Type	ELF Flag
b	SHT_NOBITS	SHF_ALLOC+SHF_WRITE
c	SHT_PROGBITS	SHF_ALLOC+SHF_EXECINSTR
d	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE
m	SHT_PROGBITS	SHF_ALLOC+SHF_WRITE+ SHF_EXECINSTR
r	0	SHF_ALLOC
w	0	SHF_ALLOC+SHF_WRITE
x	0	SHF_ALLOC+SHF_EXECINSTR

Scope Control Directives

These directives let you import and export labels:

- `global`

- extern
- public

For more information on labels, see “Labels” on page 145.

NOTE You cannot import or export equates or local labels.

global

Tells the assembler to *export* the specified labels, that is, make them available to other files.

```
.global label [ ,label ]…
```

Parameter

`label`

Any valid label.

extern

Tells the assembler to *import* the specified labels, that is, find the definitions in another file.

```
.extern label [ ,label ]…
```

Parameter

`label`

Any valid label.

public

Declares specified labels to be public.

```
.public label [ ,label ]…
```

Parameter

`label`

Any valid label.

Remarks

If the labels already are defined in the same file, the assembler exports them (makes them available to other files). If the labels are *not* already defined, the assembler imports them (finds their definitions in another file).

Symbol Definition Directives

These directives let you create equates:

- `set`
- `equal sign (=)`
- `equ`
- `textequ`

set

Defines an equate, assigning an initial value. You can change this value at a later time.

```
equate .set expression
```

Parameters

`equate`

Name of the equate.

`expression`

Temporary initial value for the equate.

equal sign (=)

Defines an equate, assigning an initial value. You can change this value at a later time.

```
equate = expression
```

Parameters

`equate`

Name of the equate.

`expression`

Temporary initial value for the equate.

Remarks

This directive is equivalent to `.set`. It is available only for compatibility with assemblers provided by other companies.

equ

Defines an equate, assigning a permanent value. You cannot change this value at a later time.

```
equate .equ expression
```

Parameters

`equate`

Name of the equate.

`expression`

Permanent value for the equate.

textequ

Defines a text equate, assigning a string value.

```
equate .textequ "string"
```

Parameters

`equate`

Name of the equate.

`string`

String value for the equate, in double quotes.

Remarks

This directive helps port existing code. You can use it to give new names to machine instructions, directives, and operands.

Upon finding a text equate, the assembler replaces it with the string value before performing any other processing on that source line.

Examples

```
dc.b     .textequ    ".byte"
endc     .textequ    ".endif"
```

Data Declaration Directives

These directive types initialize data:

- Integer Directives
- String Directives
- Floating-Point Directives

Integer Directives

These directives let you initialize blocks of integer data:

- `byte`
- `fill`
- incbin
- `long`
- `short`
- `space`

byte

Declares an initialized block of bytes.

```
[ label ]  .byte   expression [ ,expression ]…
```

Parameters

`label`

Name of the block of bytes.

`expression`

Value for one byte of the block; must fit into one byte.

fill

Declares a block of bytes, initializing each byte to zero.

```
[ label ]  .fill expression
```

Parameters

`label`

Name of the block of bytes.

`expression`

Number of bytes in the block.

incbin

Tells the assembler to include the binary content of the specified file.

```
. incbin filename[,start[,length]]
```

Parameters

`filename`

Name of a binary file.

`start`

Offset from start of file at which to begin including data.

`length`

Number of bytes of data to include.

Remarks

This directive is useful when you wish to include arbitrary data directly in the program being assembled, e.g.

```
logoPicture: .incbin "CompanyLogo.jpg"
```

long

Declares an initialized block of 32-bit short integers.

```
[ label ]  .long   expression [ ,expression ]…
```

Parameters

`label`

Name of the block of integers.

`expression`

Value for 32 bits of the block; must fit into 32 bits.

short

Declares an initialized block of 16-bit short integers.

```
[ label ]  .short expression [ ,expression ]…
```

Parameters

`label`

Name of the block of integers.

`expression`

Value for 16 bits of the block; must fit into 16 bits.

space

Declares a block of bytes, initializing each byte to zero.

```
[ label ]  .space expression
```

Parameters

`label`

Name of the block of bytes.

`expression`

Number of bytes in the block.

String Directives

These directives initialize blocks of character data:

- `ascii`
- `asciz`

A string can contain any of the escape sequences Table 13.11 lists.

Table 13.11 Escape sequences

Sequence	Description
`\b`	Backspace
`\n`	Line feed (ASCII character 10)
`\r`	Return (ASCII character 13)
`\t`	Tab
`\"`	Double quote
`\\`	Backslash
`\nnn`	Octal value of \nnn

ascii

Declares a block of storage for a string; the assembler allocates a byte for each character.

```
[ label ]  .ascii "string"
```

Parameters

`label`

Name of the storage block.

`string`

String value to be stored, in double quotes.

asciz

Declares a zero-terminated block of storage for a string.

```
[ label ]  .asciz "string"
```

Parameters

`label`

Name of the storage block.

`string`

String value to be stored, in double quotes.

Remarks

The assembler allocates a byte for each `string` character. The assembler then allocates an extra byte at the end, initializing this extra byte to zero.

Floating-Point Directives

These directives initialize blocks of floating-point data:

- `float`
- `double`

float

Declares an initialized block of 32-bit, floating-point numbers; the assembler allocates 32 bits for each value.

```
[ label ] .float  value [ ,value ]…
```

Parameters

`label`

Name of the storage block.

`value`

Floating-point value; must fit into 32 bits.

double

Declares an initialized block of 64-bit, floating-point numbers; the assembler allocates 64 bits for each value.

```
[ label ] .double  value [ ,value ]…
```

Parameters

`label`

Name of the storage block.

`value`

Floating-point value; must fit into 64 bits.

Assembler Control Directives

These directives let you control code assembly:

- `align`
- `endian`
- `error`
- `include`
- `pragma`
- `org`
- `option`

align

Aligns the location counter to the next multiple of the specified power of 2.

```
.align expression
```

Parameter

`expression`

Power of 2, such as 2, 4, 8, 16, or 32.

endian

Specifies byte ordering for the target processor; valid only for processors that permit change of endianness.

```
.endian big | little
```

Parameters

`big`

Big-endian specifier.

`little`

Little-endian specifier.

error

Prints the specified error message to the IDE Errors and Warnings window.

```
.error "error"
```

Parameter

`error`

Error message, in double quotes.

include

Tells the assembler to take input from the specified file.

```
.include filename
```

Parameter

`filename`

Name of an input file.

Remarks

When the assembler reaches the end of the specified file, it takes input from the assembly statement line that follows the `.include` directive. The specified file can itself contain an `.include` directive that specifies yet another input file.

pragma

Tells the assembler to use a particular pragma setting as it assembles code.

```
.pragma pragma-type setting
```

Parameters

`pragma-type`

Type of pragma.

`setting`

Setting value.

org

Changes the location-counter value, relative to the base of the current section.

```
.org expression
```

Parameter

`expression`

New value for the location counter; must be greater than the current location-counter value.

Remarks

Addresses of subsequent assembly statements begin at the new expression value for the location counter, but *this value is relative to the base of the current section*.

Example

In Listing 13.6, the label `Alpha` reflects the value of `.text + 0x1000`. If the linker places the `.text` section at `0x10000000`, the runtime `Alpha` value is `0x10001000`.

Listing 13.6 Address-Change Example

```
     .text
     .org 0x1000
Alpha:
     ...
     blr
```

NOTE You must use the CodeWarrior IDE and linker to place code at an absolute address.

option

Sets an assembler control option as Table 13.12 describes.

```
.option keyword setting
```

Parameters

`keyword`

Control option.

`setting`

Setting value appropriate for the option: OFF, ON, RESET, or a particular number value. RESET returns the option to its previous setting.

Table 13.12 Option Keywords

Keyword	Description
alignment off | on | reset	Controls data alignment on a natural boundary. Does not correspond to any option of the Assembler settings panel.
branchsize 8 | 16 | 32	Specifies the size of forward branch displacement. Applies only to x86 and 68K assemblers. Does not correspond to any option of the Assembler settings panel.
case off | on | reset	Specifies case sensitivity for identifiers. Corresponds to the **Case-sensitive identifiers** checkbox of the Assembler settings panel.
colon off | on | reset	Specifies whether labels must end with a colon (:). The OFF setting means that you can omit the ending colon from label names that start in the first column. Corresponds to the **Labels must end with ':'** checkbox of the Assembler settings panel.
no_at_macros off | on	Controls $AT use in macros. The OFF setting means that the assembler issues a warning if a macro uses $AT. Applies only to the MIPS Assembler.

Table 13.12 Option Keywords (*continued*)

Keyword	Description
period off | on | reset	Controls period usage for directives. The ON setting means that each directive must start with a period. Corresponds to the **Directives begin with '.'** checkbox of the Assembler settings panel.
reorder off | on | reset	Controls NOP instructions after jumps and branches. The ON setting means that the assembler inserts a NOP instruction, possibly preventing pipeline problems. The OFF setting means that the assembler does not insert a NOP instruction, so that you can specify a different instruction after jumps and branches. Applies only to the MIPS Assembler.
space off | on | reset	Controls spaces in operand fields. The OFF setting means that a space in an operand field starts a comment. Corresponds to the **Allow space in operand field** checkbox of the Assembler settings panel.

Debugging Directives

When you enable the debugger, the assembler automatically generates some debug information for your project. However, you can use these directives in the debug section to provide additional information to the debugger:

- `file`
- `function`
- `line`
- `size`
- `type`

NOTE These debugging directives are valid *only* in the `.debug` and `.text` sections of an assembly file.
Additionally, you must enable debugging for the file that contains the debugging directives; you use the Project window to enable this debugging.

file

Specifies the source-code file; enables correlation of generated assembly code and source code.

```
.file "filename"
```

Parameter

`filename`

Name of source-code file, in double quotes.

Remarks

Must precede other debugging directives in the assembly file. If you write your own DWARF code, you must use the `.function` and `.line` directives as well as this `.file` directive.

Example

Listing 13.7 shows how to use the `.file` directive for your own DWARF code.

Listing 13.7 DWARF Code Example

```
     .file  “MyFile.c”
     .text
     .globl _MyFunction
     .function "MyFunction",_MyFunction, _MyFunctionEnd -_MyFunction
_MyFunction:
     .line 1
     lwz r3, 0(r3)
     .line 2
     blr
_MyFunctionEnd:
```

function

Tells the assembler to generate debugging data for the specified subroutine.

```
.function "func", label, length
```

Parameters

`func`

Subroutine name, in double quotes.

`label`

Starting label of the subroutine.

`length`

Number of bytes in the subroutine.

line

Specifies the absolute line number (of the current source file) for which the assembler generates subsequent code or data.

```
.line number
```

Parameter

`number`

Line number of the file; the file's first line is number 1.

size

Specifies a length for a symbol.

```
.size  symbol, expression
```

Parameters

`symbol`

Symbol name.

`expression`

Number of bytes.

type

Specifies the type of a symbol.

```
.type  symbol, @function | @object
```

Parameters

`symbol`

Symbol name.

`@function`

Function type specifier.

`@object`

Variable specifier.

Macros

This chapter explains how to define and use macros. You can use the same macro language regardless of your target processor.

This chapter includes these topics:

- Defining Macros
- Invoking Macros

Defining Macros

This section explains how to define macros.

- Macro Definition Syntax
- Using Macro Arguments
- Creating Unique Labels and Equates
- Referring to the Number of Arguments

Macro Definition Syntax

A *macro definition* is one or more assembly statements that define:

- the name of a macro
- the format of the macro call

- the assembly statements of the macro

To define a macro, use either the `,macro` or the `#define` directive.

NOTE If you use a local label in a macro, the scope of the label is limited to the expansion of the macro. (Local labels begin with the @ character.)

Using the .macro Directive

The `.macro` directive is part of the first line of a macro definition. Every macro definition ends with the `.endm` directive .Listing 13.8 shows the full syntax, and Table 13.13 explains the syntax elements.

Listing 13.8 Macro Definition Syntax: .macro Directive

```
name: .macro  [ parameter ] [ ,parameter ] ...
macro_body
.endm
```

Table 13.13 Syntax Elements: .macro Directive

Element	Description
`name`	Label that invokes the macro.
`parameter`	Operand the assembler passes to the macro for us in the macro body.
`macro_body`	One or more assembly language statements. Invoking the macro tell the assembler to substitutes these statements.

The body of a simple macro consists of just one or two statements for the assembler to execute. Then, in response to the `.endm` directive, the assembler resumes program execution at the statement immediately after the macro call.

But not all macros are so simple. For example, a macro can contain a conditional assembly block, The conditional test could lead to the `.mexit` directive stopping execution early, before it reaches the `.endm` directive.

Listing 13.9 is the definition of macro `addto`, which includes an `.mexit` directive. Listing 13.10 shows the assembly-language code that calls the `addto` macro. Listing 13.11 shows the expanded `addto` macro calls.

Listing 13.9 Conditional Macro Definition

```
//define a macro
addto:   .macro dest,val
         .if val==0
         nop
         .elseif val >= -32768 && val <= 32767
         addi dest,dest,val         // use compact instruction
         .else
         addi dest,dest,val@l       // use 32-bit add
         addis dest,dest,val@ha
         .endif
// end macro definition
         .endm
```

Listing 13.10 Assembly Code that Calls addto Macro

```
// specify an executable code section
.text
li       r3,0
// call the addto macro
addto    r3,0
addto    r3,1
addto    r3,2
addto    r3,0x12345678
```

Listing 13.11 Expanded addto Macro Calls

```
li       r3,0
nop
addi     r3,r3,1
addi     r3,r3,2
addi     r3,r3,0x12345678@l
addis    r3,r3,0x12345678@ha
```

Using Macro Arguments

You can refer to parameters directly by name. Listing 13.12 shows the `setup` macro, which moves an integer into a register and branches to the label `_final_setup`. Listing 13.13 shows a way to invoke the `setup` macro., and Listing 13.14 shows how the assembler expands the `setup` macro.

Listing 13.12 Setup Macro Definition

```
setup:   .macro name
          li r3,name
          bl _final_setup
          .endm
```

Listing 13.13 Calling Setup Macro

```
VECT:   .equ 0
        setup       VECT
```

Listing 13.14 Expanding Setup Macro

```
            li    r3,VECT
            bl    _final_setup
```

If you refer to named macro parameters in the macro body, you can precede or follow the macro parameter with `&&`. This lets you embed the parameter in a string. For example, Listing 13.15 shows the `smallnum` macro, which creates a small float by appending the string `E-20` to the macro argument. Listing 13.16 shows a way to invoke the `smallnum` macro, and Listing 13.17 shows how the assembler expands the `smallnum` macro.

Listing 13.15 Smallnum Macro Definition

```
smallnum:     .macro    mantissa
              .float    mantissa&&E-20
               endm
```

Listing 13.16 Invoking Smallnum Macro

```
smallnum  10
```

Listing 13.17 Expanding Smallnum Macro

```
.float      10E-20
```

Creating Unique Labels and Equates

Use the backslash and at characters (\@) to have the assembler generate unique labels and equates within a macro. Each time you invoke the macro, the assembler generates a unique symbol of the form `??`*nnnn*, such as `??0001` or `??0002`.

In your code, you refer to such unique labels and equates just as you do for regular labels and equates. But each time you invoke the macro, the assembler replaces the `\@` sequence with a unique numeric string and increments the string value.

Listing 13.18 shows a macro that uses unique labels and equates. Listing 13.19 shows two calls to the `putstr` macro. Listing 13.20 shows the expanded code after the two calls.

Listing 13.18 Unique Label Macro Definition

```
putstr:     .macro    string
            lis       r3,(str\@)@h
            oris      r3,r3,(str\@)@l
            bl        put_string
            b         skip\@
str\@:      .asciz    string
            .align    4
skip\@:
            .endm
```

Listing 13.19 Invoking putstr Macro

```
putstr 'SuperSoft Version 1.3'
putstr 'Initializing...'
```

Listing 13.20 Expanding putstr Calls

```
            lis       r3,(str??0000)@h
            oris      r3,r3,(str??0000)@l
            bl        put_string
            b         skip??0000
str??0000:  .asciz    'SuperSoft Version
            .align    4
skip??0000:
            lis       r3,(str??0001)@h
            oris      r3,r3,(str??0001)@l
            bl        put_string
            b         skip??0001
str??0001:  .asciz    'Initializing...'
            .align    4
```

```
skip??0001:
```

Referring to the Number of Arguments

To refer to the number of non-null arguments passed to a macro, use the special symbol `narg`. You can use this symbol during macro expansion.

Invoking Macros

To invoke a macro, use its name in your assembler listing, separating parameters with commas. To pass a parameter that includes a comma, enclose the parameter in angle brackets.

For example, Listing 13.21 shows macro `pattern`, which repeats a pattern of bytes passed to it the number of times specified in the macro call. Listing 13.22 shows a statement that calls `pattern`, passing a parameter that includes a comma. Listing 13.23 is another example calling statement; the assembler generates the same code in response to the calling statement of either Listing 13.22 or Listing 13.23.

Listing 13.21 Pattern Macro Definition

```
pattern:     .macro times,bytes
                 .rept times
                 .byte bytes
                 .endr
             .endm
```

Listing 13.22 Macro Argument with Commas

```
             .data

halfgrey:    pattern 4,<0xAA,0x55>
```

Listing 13.23 Alternate Byte-Pattern Method

```
halfgrey:    .byte 0xAA,0x55,0xAA,0x55,0xAA,0x55,0xAA,0x55
```

Using the #define Directive

Another way to define a macro is to use the #define directive. This will define a pre-processor style macro using a syntax that will be familiar to C programmers. Note that C pre-processor macros are complementary to the assembler's main `MACRO...ENDM` macro language.

Listing 13.24 shows the full syntax, and explains the syntax elements.

Listing 13.24 Macro Definition Syntax: #define Directive

```
#define name [ (parms) ] assembly_statement [ ; ] [ \ ]
assembly_statement [ ; ] [ \ ]
assembly_statement

parms ::= parameter [ ,parameter ]...
```

NOTE If you specify parameters for a macro, you must enclose them in parentheses.

Table 13.14 Syntax Elements: #define Directive

Element	Description
name	Label that invokes the macro.
parameter	Operand the assembler passes to the macro.
assembly_statement	An assembly language statement. To extend the statement beyond the length of one physical line, type a backslash (\) at the end of a line, then continue the statement on the next line. To specify multiple statements on the same line, separate then with semicolon and backslash characters (;\).

GNU Compatibility

The Codewarrior Assembler supports several GNU-format assembly language extensions.

- GNU Compatible Syntax option
- Supported Extensions
- Unsupported Extensions

GNU Compatible Syntax option

Only in cases where GNU's assembler format conflicts with that of the CodeWarrior assembler does the **GNU Compatible Syntax** option have any effect. Specifically:

- Defining Equates

  Whether defined using .equ or .set, all equates can be re-defined.
- Ignored directives

  The .type directive ignored.
- Undefined Symbols

  Undefined symbols are automatically treated as imported
- Arithmetic Operators

  < and > mean left-shift and right-shift instead of less than and greater than.

  ! means bitwise-or-not instead of logical not.
- Precedence Rules

  Precedence rules for operators are changed to be compatible with GNU rather than with C.
- Local Labels

  Local labels with multi-number characters are supported (example: "1000:"). There is no limit on the number of digits in the label name. Multiple instances of the label are allowed. When referenced, you get the nearest one - forwards or backwards depending on whether you append 'f' or 'b' to the number.
- Numeric Constants

  Numeric constants beginning with 0 are treated as octal.
- Semicolon Use

  Semicolons can be used as a statement separator.
- Unbalanced Quotes

  A single unbalanced quote can be used for character constants. For example: .byte 'a

Supported Extensions

Some GNU extensions are always available, regardless whether you enable **GNU compatible syntax**. Specifically:

- Lines beginning with # * or ; are always treated as comment, even if the comment symbol for that assembler is someting different.
- Escape characters in strings extended to include `\xNN` for hex digits and `\NNN` for octal.

- Binary constants may begin with `0b`.
- Supports the GNU macro language, with macros defined by:

```
.macro     name,arg1[=default1],arg2...s1
...
.endm
```

  Arguments may have default values as shown, and when called may be specified by value or position. See the GNU documentation for details.
- New or enhanced directives (see GNU documentation for details)

Table 13.15 Supported GNU Assembler Directives

Directive	Description	Comment
.abort	End assembly	Supported
.align N,[pad]	Align	Now accepts optional padding byte
.app-file name	Source name	Synonym for .file
.balign[wl] N,[pad]	Align	Align to N (with optional padding value)
.comm name,length	Common data	Reserve space in BSS for global symbol
.def	Debugging	Accepted but ignored
.desc	Debugging	Accepted but ignored
.dim	Debugging	Accepted but ignored
.eject	Eject page	Accepted but ignored
.endr	End repeat	See .irp, .irpc
.endef	Debugging	Accepted but ignored
.fill N,[size],[val]	Repeat data	Emit N copies of width 'size', value 'val'
.hword val..	Half-word	Synonym for .short
.ident	Tags	Accepted but ignored
.ifnotdef name	Conditional	Synonym for .ifndef
.include name	Include file	Now accepts single, double or no quotes
.int val..	Word	Synonm for .long
.irp name,values	Repeat	Repeat up to .endr substituting values for name

Table 13.15 Supported GNU Assembler Directives

Directive	Description	Comment
.irpc name,chars	Repeat	Repeat up to .endr substituting chars for name
.lcomm name,length	Local common	Reserve length bytes in bss
.lflags	Ignored	Accepted but ignored
.ln lineno	Line number	Synonym for .line
.list	Listing on	Switch on listing
.local name	Local macro var	Declare name as local to macro
.macro name, args..	Macros	Supports Gnu syntax, default values, etc
.nolist	Listing off	Disable listing
.org pos,fill	Origin	Now allows fill value ot be specified
.p2align[wl] N[,pad]	Align	Align to 2**N, using pad value 'pad'
.psize	Page size	Accepted but ignored
.rept N	Repeat	Repeat block up to .endr N times
.sbttl	Subtitle	Accepted but ignored
.scl	Debugging	Accepted but ignored
.size name,N	Set size	Set size of name to N
.skip N[,pad]	Space	Skip N bytes, pad with 'pad'
.space N[,pad]	Space	Skip N bytes, pad with 'pad'
.stabd	Debugging	Accepted but ignored
.stabn	Debugging	Accepted but ignored
.stabs	Debugging	Accepted but ignored
.str “string”	Constant string	Synonym for .asciz
.string “string”	Constant string	Synonym for .asciz
.tag	Debugging	Accepted but ignored
.title	Title	Accepted but ignored

Table 13.15 Supported GNU Assembler Directives

Directive	Description	Comment
.type	Debugging	Ignored in Gnu mode
.val	Debugging	Accepted but ignored
.word	Word	Synonym for .long

Unsupported Extensions

Among the GNU extensions that the CodeWarrior Assembler does not support are:

- Sub-sections (such as "`.text 2`"). The sub-section number will be ignored.

  As a workaround, you can create your own sections with the `.section <name>` directive. You may have an arbitrary number of text subsections with the names `.text1`, `.text2`, etc.

- Assignment to location counter (such as "`. = .+4`")

  As a workaround, you can advance the location counter with `.space <expr>`

- Empty expressions defaulting to 0. Example:
  "`.byte ,`" equivalent to "`.byte 0,0`")

  There is no workaround for this. You must always supply the arguments.

- `.linkonce` directive

  The linker automatically detects logically-identical sections, and uses the following factors to determine whether to keep only one or both in the final image:

  – the binding of the symbols associated with each section

  – the location of these two sections. For example, are the sections in the same overlay or overlay group? Is one in main, and the other in an overlay group?

- `.octa`

  We do not support 16-byte numbers directly. As a workaround, you may use consecutive `.long` directives to build a large number in memory.

- `.quad`

  We do not support eight-byte numbers directly. As a workaround, you may use consecutive `.long` directives to build a large number in memory.

14

Linker

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

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

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

Specifying Link Order in the IDE

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

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

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

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

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

Dead-Stripping

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

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

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

Use the `FORCEACTIVE` directive in a linker command file to specify to the linker which objects must not be dead-stripped.

Listing 14.1 shows an example from a linker command file that tells the linker not to dead-strip an object named `InterruptVectorTable`.

Listing 14.1 FORCEACTIVE example

```
FORCEACTIVE { InterruptVectorTable }
```

Use `FORCEFILES` directive to prevent deadstripping entire files. Listing 14.1 shows an example from a linker command file that prevents the linker dead-stripping entire files.

Listing 14.2 FORCEFILES example

```
FORCEFILES { segfault.o }
```

Defining the Target's Memory Map

Use the linker command file's `MEMORY` directive to delineate areas in the target platform's memory map and associate a name for each of these areas. Names defined in a `MEMORY` directive may be used later in the linker command file to specify where object code should be stored. Listing 14.3 shows an example.

Listing 14.3 MEMORY directive example

```
MEMORY
{
   ISR_table : org = 0x00000000, len = 0x400
   data : org = 0x00000400, len = 0x10000
   flash: org = 0x10000000, len = 0x10000
   text : org = 0x80000000
}
```

This example defines 4 memory areas named `ISR_table`, `data`, `flash`, and `text`. The `org` argument specifies the beginning byte address of a memory area. The `len` argument is optional, It specifies how many bytes of data or executable code the linker may store in an area. The linker issues a warning message if an attempt to store object code in an area exceeds its length.

Defining Sections in the Output File

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

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

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

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

Listing 14.4 shows an example.

Listing 14.4 SECTIONS and GROUP example

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

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

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

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

Associating Input Sections With Output Sections

Normally the linker stores sections from input object code in the sections of the linker's output file that have the same name. The linker command file's `SECTIONS` and `GROUP` directives allow you to specify other ways to associate input object code with sections in linker output. Listing 14.5 shows an example.

Listing 14.5 Associating object code with sections in linker output

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

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

The `SECTIONS` and `GROUP` directives also allow you to filter what kinds of object code from input files will be stored in a section in the output file. Table 14.1 shows the kinds of data that may be filtered.

Table 14.1 Filter types for object code in input files

This filter	allows input objects that have these permissions	and contain this kind of object code
`TEXT`	readable, executable	initialized
`CODE`	readable, executable	initialized
`DATA`	readable, writable	initialized
`BSS`	readable, writable	uninitialized
`CONST`	readable	initialized
`MIXED`	readable, writable, executable	initialized
`VLECODE`	readable, executable	initialized

Listing 14.6 shows an example.

Listing 14.6 Filtering objects from input files

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

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

Controlling Alignment

Use the ALIGN argument in a SECTIONS or GROUP directive to specify an alignment relative to the start of the physical address.

Listing 14.7 shows an example.

Listing 14.7 Example of the ALIGN directive

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

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

Specifying Memory Area Locations and Sizes

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

Listing 14.8 shows an example.

Listing 14.8 BIND, ADDR, and SIZEOF example

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

This example defines a section in the output file named .text. This section will be loaded at address 0x00010000 on the target platform at runtime. The next section, .rodata, will be loaded at the address immediately proceeding the last byte in the

`.text` section. The last section, `.data`, will be loaded at the address that is the sum of the beginning of the `.rodata` section's address and the size of the `.rodata` section. This last section will be aligned at the next address that is evenly divisible by `0x10`.

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

Listing 14.9 shows an example.

Listing 14.9 Skipping areas of memory

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

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

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

Creating Memory Gaps

You can create gaps in memory by performing alignment calculations such as

```
. = (. + 0x20) & ~0x20;
```

This kind of calculation can occur between `output_specs`, between `input_specs`, or even in `address_modifiers`. A "." refers to the current address. You may assign the `.` to a specific unallocated address or just do alignment as the example shows. The gap is filled with zeroes, in the case of an alignment (but not with `ALIGN()`).

You can specify an alternate fill pattern with `= <short_value>`, as in

```
.text : { . = (. + 0x20) & ~0x20; *(.text) } = 0xAB > text
```

`short_value` is 2 bytes long. Note that the fill pattern comes before the `memory_spec`. You can add a fill to a `GROUP` or to an individual `output_spec` section. Fills cannot be added between `.bss` type sections. All calculations must end in a “`;`”.

Creating Symbols

You can create symbols that you can use in your program by assigning a symbol to some value in your linker command file.

```
.text : { _red_start = .; *(.text) _red_end = .;} > text
```

In the example above, the linker generates the symbols `_red_start` and `_red_end` as 32 bit values that you can access in your source files. `_red_start` is the address of the first byte of the `.text` section and `_red_end` is the byte that follows the last byte of the `.text` section.

You can use any of the pseudo functions in the `address_modifiers` in a calculation.

The CodeWarrior linker automatically generates symbols for the start address, the end address, and the start address for the section if it is to be burned into ROM. For a section `.red`, we create `_f_red`, `_e_red`, and `_f_red_rom`. In all cases, any “`.`” in the name is replaced with a “`_`”. Addresses begin with an “`_f`”, addresses after the last byte in section begin with an “`_e`”, and ROM addresses end in a “`_rom`”. See the header file `__ppc_eabi_linker.h` for further details.

All user defined sections follow the preceding pattern. However, you can override one or more of the symbols that the linker generates by defining the symbol in the linker command file.

NOTE BSS sections do not have a ROM symbol.

Linker Command File Syntax

Linker command file syntax is a notation and implies what an LCF file includes. Listing 14.10 shows the syntax for linker command files.

Listing 14.10 Linker Command File Syntax

```
linker-command-file =
   command* memory? command* sections? command*
```

This syntax implies that an LCF file can contain:

Zero or more command directives followed by
Zero or at most one memory directive followed by
Zero or more command directives followed by

Zero or at most one sections directive followed by
Zero or more command directives.

Table 14.2 lists the notations used in the linker command file syntax.

Table 14.2 Linker Command File Notations

Notation	Description
*	Implies zero or any number of directives
?	Implies zero or at most one directive

Listing 14.11 shows the syntax of all valid linker command file keywords, directives, and commands.

Listing 14.11 Linker Command File Syntax (Commands, Directives, and Keywords)

```
command =
   exclude-files |
   force-active |
   force-files |
   include-dwarf |
   keep |
   ref-include |
   shorten-names-for-tornado-101 |
   cats-bss-mod |
   cats-header-mod |
   data-type-converts |
   entry |
   init |
   term |
   external-symbol |
   internal-symbol |
   memory-gaps

exclude-files =
   "EXCLUDEFILES" "{" file-name+ "}"

force-active =
   "FORCEACTIVE" "{" symbol+ "}"

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

```
file-name =
   (letter |"_") (letter |digit |"_")*
   (".")?(letter |digit |"_")*

section-name =
   (letter |"_") (letter |digit |"_")*

object-file =
   (letter |"_") (letter |digit |"_")* (".") ("o"|"O")

archive-file =
   (letter |"_") (letter | digit |"_")* (".") ("a"|"A")

include-dwarf =
   "INCLUDEDWARF" "{" file-name "}"

keep =
   "KEEP""(" *( section-name )")"

ref-include =
   "REF_INCLUDE" "{" section-name+ "}"

shorten-names-for-tornado-101=
   "SHORTEN_NAMES_FOR_TOR_101"

cats-bss-mod =
   "CATS_BSS_MOD"

cats-header-mod =
   "CATS_HEADER_MOD"

data-type-converts =
   "DATA_TYPE_CONVERTS"

entry =
   "ENTRY" "(" symbol ")"

init =
   "INIT" "(" symbol ")"

term =
   "TERM" "(" symbol ")"

external-symbol =
   "EXTERNAL_SYMBOL" "{" symbol [","  symbol] "}"

internal-symbol =
   "INTERNAL_SYMBOL" "{" symbol [","  symbol] "}"
```

```
group=
   "GROUP" address-modifiers ":"
   "{" (section-spec )* "}" ["=" fill-shortnumber ]
   [ "> " mem-area-symbol ]

hexadigit =
   '0'|'1'|'2'|'3'|'4'|'5'|'6'|'7'|'8'|'9'|
   'A'|'B'|'C'|'D'|'E'|'a'|'b'|'c'|'d'|'e'

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

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

decimal-number =
   digit+

number =
   hexadecimal-number | decimal-number

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

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

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

symbol-declaration =
   ( symbol "=" address-spec ) |
   ( "PROVIDE" "(" identifier "=" address=spec ")" ) |
   ( "EXPORTSYMTAB”) |
   ( "EXPORTSTRTAB”) |
   ( "BYTE” "(" address-spec ")” |
   ( "SHORT” "(" address-spec ")” |
   ( "LONG” "(" address-spec ")”

symbol =
   (letter |"_") (letter|"_"|digit )*

operand =
   number |
   ("ADDR" "(" output-section-spec | address-expr ")" ) |
   ("ROMADDR" "(" output-section-spec | address-expr ")" |
   ("SIZEOF" "(" output-section-spec | address-expr ")" ) |
   ("SIZEOF_ROM" "(" output-section-spec | address-expr ")" )

address-spec =
   number |
   "." |
   operand |
   (address-spec binary-op operand ) |
   (unary-op address-spec ) |
   (address-spec postfix-unary-op )

memory-spec =
   memory-area-symbol ":" "origin" |
   "org" |
   "o" "=" number "," "length"|
   "len" |
   "l" "=" number
   [“>” file-name]

memory-gaps =
   "." "=" address-spec

memory =
    "MEMORY" "{" memory-spec + "}"

sections =
   "SECTIONS" "{"
   (section-spec | memory-gaps | symbol-declaration | group )*
   "}"

section-spec =
   output-section-name ":"
   ["(" input-type ")"]
   [address-modifiers ] "{"
      [( input-section-spec )*] "}"
   [= fill-shortnumber] [(">"|">>") memory-area-symbol ]

output-section-name =
   section-name

input-type =
   [ "TEXT" | "DATA" | "BSS" | "CONST" | "MIXED" | "ZTEXT" | "ZCODE"|
"VLECODE" ]

address-modifiers =
   ["BIND" "(" address-spec ")" ]
   ["ALIGN" "(" address-spec ")" ]
   ["NEXT" "(" address-spec ")"]
```

```
    ["LOAD" "(" address-spec ")"]
    ["INTERNAL_LOAD" "(" address-spec ")"]

input-section-spec =
    ( file-name |
       file-name "(" section-name ")" |
       "*(" section-name ")" |
       symbol-declaration |
       data-write )+

data-write =
    ("LONG" | "SHORT" | "BYTE" ) "(" number ")"

fill-shortnumber =
   number
```

Commands, Directives, and Keywords

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

. (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 14.12 example does.

Example

The code of Listing 14.12 moves the location counter to a position 0x10000 bytes past the symbol `__start`.

Listing 14.12 Moving the Location Counter

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

ADDR

Returns the address of the named section or memory segment.

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

Parameters

`sectionName`

Identifier for a file section.

`segmentName`

Identifier for a memory segment

Example

The code of Listing 14.13 uses the `ADDR` function to assign the address of `ROOT` to the symbol `__rootbasecode`.

Listing 14.13 ADDR() Function

```
MEMORY{
    ROOT : origin = 0x80000400, length = 0
}

SECTIONS{
  .code :
  {
  __rootbasecode = ADDR(ROOT);
     *.(text);
  } > ROOT
}
```

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
```

BIND

Specifies a section modifier for setting an address.

```
BIND(address)
```

Parameter

`address`

A memory address, such as 0x80000400.

BIN_FILE_TYPE

Controls the generation of a binary file. By default, the linker does not generate any binary file, even if the s-record generation is enabled.

`BIN_FILE_TYPE (`*`NO_BIN_FILE`* `|` *`ONE_BIN_FILE`*
    `|`*`MULTIPLE_BIN_FILES`*`)`

Parameters

`NO_BIN_FILE`

No binary file will be generated even if s-record generation is `on`.

`ONE_BIN_FILE`

Generates a single binary file with all the loadable code and data, even if s-record generation is off.

`MULTIPLE_BIN_FILES`

A separate binary file is generated for each MEMORY directive. When selected, the hexidecimal address of the location, the file should be loaded is inserted between the filename and extension.

Example

Listing 14.14 is an example of a binary file generated using the *`MULTIPLE_BIN_FILES`* parameter for a non-ROM Image build.

Listing 14.14 Binary File for a non-ROM Image Build Example

```
Memory map:
        Starting Size     File     S-Record Bin File Bin File
        address           Offset   Line     Offset   Name
  .init 00002000 00000318 000001c0        2 00000000 Test.00002000.bin
  .text 00002320 00042250 000004e0         42  00000000  Test.00002320.bin
```

Listing 14.15 is an example of a binary file generated using the *`MULTIPLE_BIN_FILES`* parameter for a ROM Image build.

Listing 14.15 Binary File for a ROM Image Build Example

```
Memory map:
        Starting Size     File     ROM      RAM Buffer S-Record Bin File Bin File
        address           Offset   Address  Address    Line     Offset   Name
  .init ffe00000 00000318 00000300 ffe00000 ffe00000          2 00000000 Test.ffe00000.bin
  .text  00002000 0004f9d0 00086500 ffe00318 ffe00318        42 00000000 Test.ffe00318.bin
```

The code of Listing 14.16 uses the *`MULTIPLE_BIN_FILES`* parameter to change the name of a binary file in the `MEMORY` directive.

Listing 14.16 Change Binary File Name in the MEMORY Directive

```
MEMORY {
        code : org = 0x00002000 > my_binary_file.xyz
```

Any `MEMORY` directive without a custom name will be given a name like `Test.ffe00000.bin`, where the binary file reflects the ROM address of the segement.

The code of Listing 14.17 uses the *`MULTIPLE_BIN_FILES`* parameter to merge some binary files together, assuming that there are no segments between them.

Listing 14.17 Merge Binary Files Together Example

```
MEMORY {
        code : org = 0x00002000 > my_binary_file.xyz
        special : org = 0x00004000 > my_binary_file.xyz
```

BYTE

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

`BYTE (`*`expression`*`);`

Parameter

`expression`

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

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 14.18 shows. (At runtime, the system knows the values of the two symbols.)

Example

Listing 14.18 shows the code for creating an exception table.

Listing 14.18 Creating an Exception Table

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

EXCLUDEFILES

Ignores object code in files.

Syntax

`EXCLUDEFILES { *file-name* }`

Remarks

This directive is for partial link projects only. It makes your partial link file smaller. `EXCLUDEFILES` can be used independently of INCLUDEDWARF. Unlike `INCLUDEDWARF`, `EXCLUDEFILES` can take any number of executable files.

In this example

```
EXCLUDEFILES { kernel.elf }
```

`kernel.elf` is added to your project but the linker does not add any section from `kernel.elf` to your project. However, it does delete any weak symbol from your partial link that also exists in `kernel.elf`. Weak symbols can come from templates or out-of-line inline functions.

EXPORTSTRTAB

Creates a string table from the names of exported symbols.

```
EXPORTSTRTAB
```

Remarks

Table 14.3 shows the structure of the export string table. As with an ELF string table, the system zero-terminates the library and symbol names.

Table 14.3 Export String Table Structure

`0x00`	1 byte
library name	varies

Table 14.3 Export String Table Structure (*continued*)

symbol1 name	varies
symbol2 name	varies

Example

Listing 14.19 shows the code for creating an export string table.

Listing 14.19 Creating an Export String Table

```
.expstr:
{
 EXPORTSTRTAB
} > EXPSTR
```

EXPORTSYMTAB

Creates a jump table of the exported symbols.

```
EXPORTSYMTAB
```

Remarks

Table 14.4 shows the structure of the export symbol table. The start of the export symbol table must be aligned on at least a four-byte boundary.

Table 14.4 Export Symbol Table Structure

Size (in bytes) of export table	4 bytes
Index to *library* name in export symbol table	4 bytes
Index to *symbol1* name in export symbol table	4 bytes
Address of *symbol1*	4 bytes
A5 value for *symbol1*	4 bytes
Index to *symbol2* name in export symbol table	4 bytes
Address of *symbolf2*	4 bytes
A5 value for *symbol2*	4 bytes

Example

Listing 14.20 shows the code for creating an export symbol table.

Listing 14.20 Creating an Export Symbol Table

```
.expsym:
{
 EXPORTSYMTAB
} > EXPSYM
```

EXTERNAL_SYMBOL

Specifies objects that may be referred to by modules outside of an object code file.

Syntax

`EXTERNAL_SYMBOL { `*`symbol`*` [, `*`symbol`*`]* }`

Remarks

The symbols must be the link time symbol names. In the case of C++ object code, these names must be the mangled.

FORCEACTIVE

Specifies objects that must not be deadstripped.

Syntax

`FORCEACTIVE { `*`symbol`*` [, `*`symbol`*`]* }`

GROUP

Organizes objects listed in a linker command file.

Syntax

`GROUP `*`address-modifiers`*` : { `*`section-spec`*` [, `*`section-spec`*`]* }`

`[ > `*`memory-area-name`*` ]`

Remarks

The GROUP directive lets you organize the linker command file.

Listing 14.21 shows that each group starts at a specified address. If no *address_modifiers* are present, it would start following the previous section or group. Although you normally do not have an *address_modifier* for an output_spec within a group; all sections in a group follow contiguously unless there is an address_modifier for that output_spec.

The optional *memory-area-name* clause specifies a name defined with the MEMORY directive. Using this clause specifies the memory space in the target platform where the group's object code should be placed.

Listing 14.21 Example of linker GROUP directive

```
SECTIONS {
 GROUP BIND(0x00010000) : {
  .text : {}
  .rodata : {*(.rodata) *(extab) *(extabindex)}
  }

 GROUP BIND(0x2000) : {
  .data : {}
  .bss : {}
  .sdata BIND(0x3500) : {}
  .sbss   : {}
  .sdata2 : {}
  .sbss2  : {}
  }

 GROUP BIND(0xffff8000) : {
  .PPC.EMB.sdata0   : {}
  .PPC.EMB.sbss0 : {}
  }
}
```

IMPORTSTRTAB

Creates a string table from the names of imported symbols.

IMPORTSTRTAB

Remarks

Table 14.5 shows the structure of the import string table. As with an ELF string table, the system zero-terminates the library and symbol names.

Table 14.5 Import String Table Structure

`0x00`	1 byte
library name	varies
symbol1 name	varies
symbol2 name	varies

Example

Listing 14.22 shows the code for creating an import string table.

Listing 14.22 Creating an Import String Table

```
.impstr:
{
 IMPORTSTRTAB
} > IMPSTR
```

IMPORTSYMTAB

Creates a jump table of the imported symbols.

`IMPORTSYMTAB`

Remarks

Table 14.6 shows the structure of the import symbol table. The start of the import symbol table must be aligned on at least a four-byte boundary.

Table 14.6 Import Symbol Table Structure

Size (in bytes) of import table	4 bytes
Index to *library1* name in import string table	4 bytes
Number of entries in *library1*	4 bytes

Table 14.6 Import Symbol Table Structure (*continued*)

Index to *symbol1* name in import string table	4 bytes
Address of *symbol1* vector in export string table	4 bytes
Index to *symbol2* name in import string table	4 bytes
Address of *symbol2* vector in export string table	4 *bytes*
Index to *library2* name in import string table	4 bytes
Number of entries in *library2*	4 bytes

Example

Listing 14.23 shows the code for creating an import symbol table.

Listing 14.23 Creating an Import Symbol Table

```
.expsym:
{
 IMPORTSYMTAB
} > EXPSYM
```

INCLUDEDWARF

Allows source-level kernel debugging.

Syntax

`INCLUDEDWARF { file-name }`

Remarks

In this example

`INCLUDEDDWARF { kernel.elf }`

the linker inserts the `.debug` and `.line` sections of `kernel.elf` to your application. These sections let you debug source level code in the kernel while debugging your application.

You are limited to one executable file when using this directive. If you need to process more than one executable, add this directive to another file.

INTERNAL_LOAD

Loads one or several segments at an address not specified at link time.

Syntax

```
INTERNAL_LOAD(address)
```

Parameter

`address`

A memory address, such as 0x80000400.

Remarks

Use `INTERNAL_LOAD` directive to specify an internal ROM addr_mode.

INTERNAL_SYMBOL

Symbols created in a linker command file are considered external unless they are redefined using INTERNAL_SYMBOL

Syntax

`INTERNAL_SYMBOL { `*`symbol`*` [, `*`symbol`*`]* }`

KEEP

Forces the linker to *not* dead strip the unused symbols in the specified section.

Syntax

```
KEEP(*(sectionType))
```

Parameter

`sectionType`

Identifier for any user-defined or predefined section.

Example

Listing 14.24 shows the sample usage.

Listing 14.24 KEEP Directive Usage

```
GROUP : {
          .text (TEXT) : {}
          .mycode (TEXT) : {KEEP(*(.mycode))}
...}
```

LOAD

Loads one or several segments at a specific address.

Syntax

`LOAD(address)`

Parameter

`address`

A memory address, such as 0x80000400.

Remarks

- Use `LOAD` directive to specify an external ROM addr_mode.
- The `LOAD` directive can be used on all the sections going into ROM.
- The directive is ignored in links if **Generate ROM image** is not checked in the linker preference panel.
- Inconsistent initialized data sections copied from ROM to RAM may need a reference to a ROM address. For example, to store the `.data` and `.sdata` sections in ROM such that they are immediately after the `.text` section, try the following:

```
.text LOAD(ADDR(.text)) : {} > rom
.data LOAD(ROMADDR(.text) + SIZEOF(.text)): {} > code
.sdata LOAD(ROMADDR(.data) + SIZEOF(.data)): {} > code
```

LONG

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

`LONG (`*`expression`*`);`

Parameter

`expression`

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

MEMORY

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

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

Parameters

`memory_spec`

```
segmentName: origin = address,
 length = length [> fileName]
```

`segmentName`

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

`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.

MAX_BIN_GAP

Controls the maximum gap size value between two segments.

```
MAX_BIN_GAP(nnnnn)
```

Parameters

`nnnnn`

Size of the maximum gap allowed between segments.

Remarks

The directive can be placed in the LCF anyway except within the `MEMORY` and `SECTIONS` directives.

This directive can only be used if you are generating a single binary file.

NEXT

Specifies an expression for setting an address.

```
NEXT(address)
```

Parameter

`address`

A memory address, such as 0x80000400.

NO_TRAILING_BSS_IN_BIN_FILES

Removes uninitiallized data contained in a binary file.

Syntax

`NO_TRAILING_BSS_IN_BIN_FILES`

Remarks

This directive can only be used if the last section or sections of a binary file contains uninitialized data.

Example

Listing 14.25 is an example use of the `NO_TRAILING_BSS_IN_BIN_FILES` directive.

Listing 14.25 NO_TRAILING_BSS_IN_BIN_FILES Directive Example

```
SECTIONS
        {
           GROUP {
           .text:{}
           }>code
         }
NO_TRAILING_BSS_IN_BIN_FILES
```

OBJECT

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

`OBJECT (`*`function, sourcefile.c`*`)`

Parameters

`function`

Name of a function.

`sourcefile.c`

Name of the C file that contains the function.

Remarks

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

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.

REGISTER

Use the `REGISTER` directive to assign one of the EPPC processor's non-volatile registers to a user-defined small data section.

```
REGISTER(nn [ , limit])
```

Parameter

`nn`

Specifies one of the predefined small data base registers, a non-volative EPPC register, or any of the following values:

- `0, 2, 13`

  These registers are for the predefined small data sections:

  ```
  0 - .PPC.EMB.sdata0/.PPC.EMB.sbss0
  2 - .sdata2/sbss2
  13 - .sdata/sbss
  ```

  You do not have to define these sections using REGISTER because they are predefined.

- `14 - 31`

  Match any value in this range with the register reserved by your global register variable declaration.

- `-1`

This "register" value instructs the linker to treat relocations that refer to objects in your small data section as non-small data area relocations. These objects are converted to near absolute relocations, which means that the objects referenced must reside within the first 32 KB of memory. If they do not, the linker emits a "relocation out of range" error. To fix this problem, rewrite your code such that the offending objects use large data relocations.

`limit`

Specifies the maximum size of the small data section to which register `nn` is bound. This value is the size of the initialized and uninitialized sections of the small data section combined. If `limit` is not specified, `0x00008000` is used.

NOTE Each small data section you create makes one less register available to the compiler; it is possible to starve the compiler of registers. As a result, create only the number of small data sections you need.

ROMADDR

Equivalent to `ADDR`. Returns `ROM` address.

`ROMADDR (`*`sectionName`* `|` *`segmentName`*`)`

Parameters

`sectionName`

Identifier for a file section.

`segmentName`

Identifier for a memory segment

Example

The code of Listing 14.26 uses the `ROMADDR` function to assign the address of `ROM` to the symbol `__rootbasecode`.

Listing 14.26 ROMADDR() Function

```
MEMORY{
    ROM : origin = 0x80000400, length = 0
}

SECTIONS{
  .code :
  {
    __rootbasecode = ROMADDR(ROM);
        *.(text);
    } > ROM
}
```

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`* `: [LOAD (`*`loadAddress`*`)] {contents}`
> `  > `*`segmentName`*

`sectionName`

Name for the output section. Must start with a period.

`LOAD`

Loads one or several segments at a specific address.

`contents`

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

`segmentName`

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

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

Example

Listing 14.27 is an example sections-segment definition.

Listing 14.27 SECTIONS Directive Example

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

SHORT

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

`SHORT (`*`expression`*`);`

Parameter

`expression`

Any expression that returns a value `0x0000` to `0xFFFF`

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.

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 `BYTE`, `SHORT`, or `LONG` 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`.

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"
```

15

Linker for Power Architecture Processors

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

- Predefined Sections
- Additional Small Data Sections
- Linker Map File
- Deadstripping
- Linker Command Files

Predefined Sections

Table 15.1 describes the sections that the compiler creates.

NOTE The Compiler-defined section names are case sensitive. For example, using .binary instead of .BINARY will not give expected results.

Table 15.1 Compiler-defined sections

Name	Description
`.bss`	uninitialized global data
`.BINARY`	Binary files.
`.ctors`	C++ constructors and Altivec vector constructors
`.dtors`	C++ destructors
`.data`	initialized global data
`extab`	C++ exception tables
`extabindex`	C++ exception tables

Table 15.1 Compiler-defined sections

Name	Description
`.init`	initialization executable code from the runtime library
`.init_vle`	Initialization executable code for VLE compilers
`.PPC.EMB.sdata0`	Initialized data with addressing relative to address 0
`.PPC.EMB.sbss0`	Uninitialized data with addressing relative to address 0
`.rodata`	literal values and initialization values in the application's source code
`.sdata`	initialized small global data
`.sdata2`	initialized global small data defined with the `const` keyword
`.sbss`	uninitialized global small data
`.sbss2`	uninitialized global constant small data defined with the `const` keyword
`.text`	application code
`.text_vle`	application code for VLE compilers

Linking Binary Files

You can link external binary files/data (tables, Bitmap graphics, sound records) into the project image. The following sections explain how to link binary files using CodeWarrior IDE and Command line:

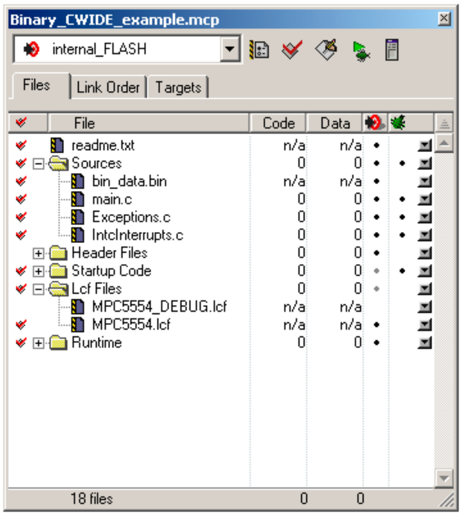
- Using CodeWarrior IDE
- Using Command-Line

Using CodeWarrior IDE

To link a binary file using CodeWarrior IDE, perform the following steps:

1. Launch CodeWarrior and open the desired project to add the binary file.
2. Add a binary file (`bin_data.bin`) to project (Figure 15.1).

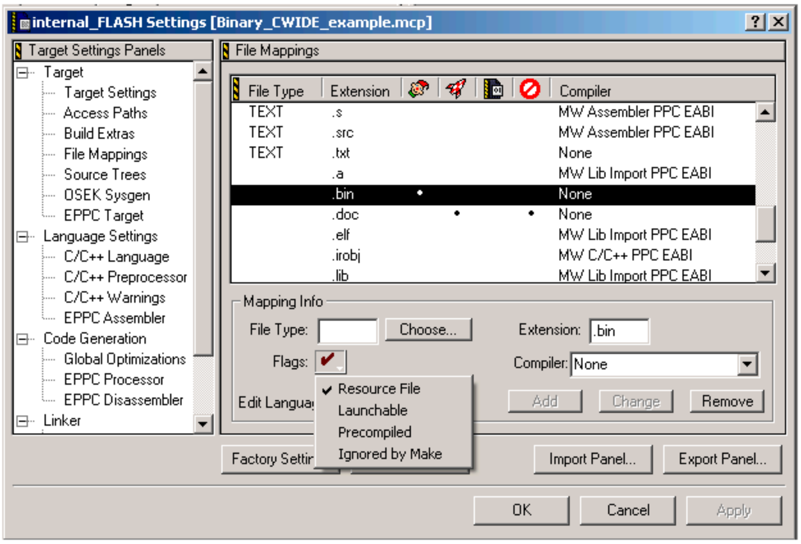
Figure 15.1 Add a Binary File



	File	Code	Data		
✓	readme.txt	n/a	n/a	•	
✓	Sources	0	0	•	•
✓	bin_data.bin	n/a	n/a	•	
✓	main.c	0	0	•	•
✓	Exceptions.c	0	0	•	•
✓	IntcInterrupts.c	0	0	•	•
	Header Files	0	0	•	
✓	Startup Code	0	0	•	•
✓	Lcf Files	0	0	•	
	MPC5554_DEBUG.lcf	n/a	n/a		
✓	MPC5554.lcf	n/a	n/a	•	
✓	Runtime	0	0	•	
	18 files	0	0		

3. Add binary file extension into project's **File Mappings** preference panel (if not existing) and give it **Flag** of **Resource File** (Figure 15.2).

Figure 15.2 File Mappings Preference Panel



File Type	Extension					Compiler
TEXT	.s					MW Assembler PPC EABI
TEXT	.src					MW Assembler PPC EABI
TEXT	.txt					None
	.a					MW Lib Import PPC EABI
	.bin	•				None
	.doc		•		•	None
	.elf					MW Lib Import PPC EABI
	.irobj					MW C/C++ PPC EABI
	.lib					MW Lib Import PPC EABI

File Type:
Extension: .bin
Flags: ✓ Resource File
Compiler: None

4. Update linker command file (.lcf) and place .BINARY section into memory. Listing 15.1 shows a sample linker command file with .BINARY section.

Listing 15.1 Linker Command File with .BINARY section

```
MEMORY
{
    resetvector:          org = 0x00000000,   len = 0x00000008
    init:                 org = 0x00000020,   len = 0x00000FE0
    exception_handlers: org = 0x00001000,   len = 0x00001000
    internal_flash:       org = 0x00002000,   len = 0x001FD000
    my_binary_data:       org = 0x001FE000,   len = 0x00001000
...
}

SECTIONS
{
.__bam_bootarea LOAD (0x00000000): {} > resetvector
...
  .binary1_area:
{
        binary1Start = .;
        bin_data1.bin
        binary1End = .;

} > my_binary_data
}
.binary2_area:
  {
      binary2Start = .;
      bin_data2.bin
      binary2End = .;

 } > my_binary_data
}

}
```

Using Command-Line

To link a binary file using Command line, perform the following steps:

1. Linker recognizes `.bin` extension as a binary data input file. If binary file has another extension it may not be recognized correctly by the command line linker.
2. Update linker command file (`.lcf`) and place `.BINARY` section into memory. Listing 15.1 shows a sample linker command file with `.BINARY` section.

3. Add a binary file (`.bin`) as an input file for linker (MWLDEPPC.exe)

   ```
   mwldeppc main.o msl.lib bin_data.bin -o myapp.elf -lcf
   commandfile.lcf
   ```

Additional Small Data Sections

The PowerPC EABI specification mandates that compliant build tools predefine three small data sections. The EPPC Linker target settings panel lets you specify the address at which the CodeWarrior linker puts two of these sections (if the default locations are unsatisfactory).

CodeWarrior Development Studio, MPC55xx Edition lets you create small data sections in addition to those mandated by the PowerPC EABI specification. The CodeWarrior tools let you specify that the contents of a given user-defined section will be accessed by the small data base register selected from the available non-volatile registers. To do this, you use a combination of source code statements and linker command file directives.

To create one additional small data area, follow these steps:

1. Open the CodeWarrior project in which you want to create an additional small data section.
2. Select the build target in which you want to create an additional small data section.
3. Press **ALT-F7**

   The IDE displays the **Target Settings** window.
4. In the left pane of the **Target Settings** window, select **C/C++ Preprocessor** .

   The **C/C++ Preprocessor** target settings panel appears in the right side of the **Target Settings** window.
5. Open the prefix file whose name appears in the Prefix File text box in an editor window.
6. Add the statements that define a small data section to the top of the prefix file:

   a. Add a statement that creates a global register variable.

      For example, to create a global register variable for register 14, add this statement to the prefix file:

      ```
      // _dummy does not have to be defined
      extern int _dummy asm(“r14”);
      ```

   b. Create a user-defined section using the section pragma; include the clause `data_mode = sda_rel` so the section can use small data area addressing.

      For example:

      ```
      // you do not have to use the names in this example
      // .red is the initialized part of the section
      ```

```
// .blue is the uninitialized part of the section
#pragma section RW ".red" ".blue" data_mode = sda_rel
```

NOTE If you want your small data area to be the default section for all small data, use the following form of the `section` pragma instead of the one above:
`#pragma section sdata_type ".red" "blue" data_mode = sda_rel`

7. Save the prefix file and close the editor window.
8. In each header or source file that declares or defines a global variable that you want to put in a small data section, put the storage-class modifier `__declspec(section "initialized_small_sect_nm")` in front of the definition or declaration.

   For example, the statement:

   `__declspec(section ".red") int x = 5;`

   instructs the compiler to put the global variable `x` into the small data section named `.red`

CAUTION The `section name` specified in the `__declspec(section <section_name>)` statement must be the name of an initialized data section. It is an error to use the uninitialized data section name.

NOTE The semantics of `__declspec(section ".sdata") int x;` is to use the section pair `.sdata` and `.sbss` to store `x`. The location where `x` is stored is determined by whether or not `x` is explicitly initialized.

NOTE If you want your small data section to be the default section for all small data, use
`#pragma section sdata_type ".foo" ".bar" data_mode = sda_rel`
Use `__declspec(section ".foo")` only when the object is greater than the size threshold for small data.

9. In the left pane of the **Target Settings** window, select **EPPC Linker** .

   The **EPPC Linker** target settings panel appears.
10. In the Segment Addresses group box, check the Use Linker Command File checkbox.

    The other checkboxes and text boxes in the group become disabled.
11. In the left pane of the **Target Settings** window , select **EPPC Target** .

    The **EPPC Target** settings panel appears.

12. From the **Code Model** listbox, select **Absolute Addressing.**
13. From the **ABI** listbox, select **EABI**.
14. Click **OK.**

    The IDE saves your settings and closes the **Target Settings** window.
15. Modify the project's linker command file such that it instructs the linker to use the global register declared above as the base register for your new small data section.

    To do this, follow these steps:

    a. In the linker command file, add two REGISTER directives, one for the initialized part of the small data section and one for uninitialized part.

       For example, to make register 14 the base register, add statements like these:

       ```
       .red REGISTER(14) : {} > ram
       .blue REGISTER(14) : {} > ram
       ```

    b. Add the linker command file to each build target in which you want to use the new small data section.
16. Open the CodeWarrior project for the runtime library used by your project. The runtime library project is here:

    ```
    InstallDir\PowerPC_EABI_Support\
    Runtime\Project\Runtime.PPCEABI.mcp
    ```
17. In the build target listbox of the runtime library project window, select the build target of the runtime library that your main project uses.
18. Open this build target's prefix file in a CodeWarrior editor window.
19. Add the same statements to this prefix file that you added to the prefix file of the main project.
20. Save the prefix file and close the editor window.
21. Open `__start.c` in a CodeWarrior editor window.
22. Find the string `__init_registers(void)` and add statements that initialize the small data section base register you are using near the end of this function (immediately above the terminating `blr` instruction).

    For example, to initialize register 14, add these statements:

    ```
    lis r14, _SDA14_BASE_@ha
    addi r14, r14, _SDA14_BASE_@l
    ```
23. Save `__start.c` and close the editor window.
24. Open `__ppc_eabi_linker.h` in a CodeWarrior editor window.

25. Find the string `_SDA_BASE_[]` in this file and add this statement after the block of statements that follow this string:

    ```
    // SDAnn_BASE is defined by the linker if
    // the REGISTER(nn) directive appears in the .lcf file
    __declspec(section ".init") extern char _SDA14_BASE_[];
    ```

26. Save `__ppc_eabi_linker.h` and close the editor window.
27. Press **F7**.

    The IDE builds a new runtime library.

28. Close the runtime library project.
29. Return to your main project.
30. Press **F7** .

    The IDE builds your project.

    You can now use the new small data section in this project.

NOTE You can create more small data segments by following the procedure above. Remember, however, that for each small data section created, the compiler loses one non-volatile register to use for other purposes.

Linker Map File

A linker map file is a text file containing information about a program's global symbols, source file and source line numbers. The linker names the map file with the base name of the program and the extension `.map`. The linker map consists of the following sections:

- Closure
- Section Layout
- Memory Map
- Linker Generated Symbols

Closure

The linker lists all the required objects under the closure section with the following details:

- `Level of closure:` Object B is in Object A's closure if and only if, the level of B is higher than the level of A and one of the following conditions is true:

  Condition 1: There is no object in between B and A.

  Condition 2: There are objects between B and A, and the level of A is lower than the levels of all the objects between B and A.

- `Object name:` specifies the name of an object.
- `Object characterstics:` specifies the characteristics of an object. They can be one of the following:
  - function, local | global | weak
  - section, local | global | weak
  - object, local | global | weak
  - notype, local | global | weak
- `Object locations:` specifies an object location.

Listing 15.2 shows a sample closure section.

Listing 15.2 Sample closure section

```
1] reset (func,global) found in reset.o
 2] __reset (func,global) found in 8568mds_init.o
  3] __start (func,global) found in Runtime.PPCEABI.E2.UC.a __start.o
   4] __init_registers (func,weak) found in Runtime.PPCEABI.E2.UC.a __start.o
    5] _stack_addr found as linker generated symbol
    5] _SDA2_BASE_ found as linker generated symbol
    5] _SDA_BASE_ found as linker generated symbol
   4] __init_hardware (func,global) found in __ppc_eabi_init.o
    5] usr_init (func,global) found in 8568mds_init.o
     6] gInterruptVectorTable (notype,global) found in eppc_exception.o
      7] gInterruptVectorTableEnd (notype,global) found in eppc_exception.o
      7] .intvec (section,local) found in eppc_exception.o
       8] InterruptHandler (func,global) found in interrupt.o
        9] @21 (object,local) found in interrupt.o
        9] printf (func,global) found in MSL_C.PPCEABI.bare.E2.UC.a printf.o
         9] __msl_count_trailing_zero64 (func,weak) found in MSL_C.PPCEABI.bare.E.a
math_double.o
        9] >>> UNREFERENCED DUPLICATE __msl_count_trailing_zero64
        9] >>> (func,weak) found in MSL_C.PPCEABI.bare.E.a math_float.o
        9] >>> (func,weak) found in MSL_C.PPCEABI.bare.E.a math_longdouble.o
         9] >>> (func,weak) found in MSL_C.PPCEABI.bare.E.a math_ppc.o
```

In the sample above:

- `__reset` is in the closure of `reset` because:
  - `__reset` is of level 2, `reset` is of level 1 and
  - there is no object in between `__reset` and `reset`
- `_SDA_BASE_` is in the closure of `__init_registers` because:
  - `_SDA_BASE_` is of level 5, `__init_registers` is of level 4; and
  - the objects between `__init_registers` and `_SDA_BASE_` are all of level 5 and are higher than the level of `__init_registers`
- `InterruptHandler` is in the closure of `__init_hardware` because:
  - `InterruptHandler` is of level 8, `__init_hardware` is of level 4; and

  - the objects between `__init_hardware` and `InterruptHandler` are of level 5, 6, 7 respectively and are all higher than the level of `__init_hardware`
- `__init_hardware` is NOT in the closure of `_init_registers` because:
  - they both are of level 4
- `gInterruptVectorTableEnd` is NOT in the closure of `__init_registers` because:
  - the objects between `gInterruptVectorTableEnd` and `__init_registers` are not all of a higher level than `__init_registers`
  - `__init_hardware` is of the same level as `__init_registers`.

Weak symbols are allowed by the ABI and are global. They can have the same name as another symbol. The line before the `UNREFERENCED DUPLICATE` lists the first weak symbol found by the linker, that appears in the executable.

The line after the `UNREFERENCED DUPLICATE` lists other versions of a same object found by the linker. Linker will not copy the duplicate objects to the executable.

Section Layout

The linker lists information of all the objects within a section in a section layout. Listing 15.3 shows a sample `.text` section layout.

Listing 15.3 Sample `.text` section layout

```
.text section layout
     Starting          Virtual   File
     address   Size    address   offset
     ---------------------------------
     00000084  000030  fffc1964  00001ce4  1 .text
     00000084  00000c  fffc1964  00001ce4  4 __init_user                    __ppc_eabi_init.o
     00000090  000020  fffc1970  00001cf0  4 exit                           __ppc_eabi_init.o
     000000b0  000004  fffc1990  00001d10  4 _ExitProcess                   __ppc_eabi_init.o
     ...
     UNUSED   000030 ........ ........    __copy Runtime.PPCEABI.E2.UC.a CPlusLibPPC.o
     UNUSED   000084 ........ ........    __init_arr Runtime.PPCEABI.E2.UC.a CPlusLibPPC.o
      ...
```

The first line of a section layout specifies the name of a section. Starting from the 5th line (the line after the dotted line separator), objects within section are listed with the following information:

- `Starting address`: specifies the starting address of an object. The object is listed as `UNUSED` if it is dead-stripped.
- `Size`: specifies the size of each object in a section.
- `Virtual address:` specifies the virtual address of the object.

- `File offset:` specifies the offset of an object in the section.
- `Alignment:` specifies the alignment of an objects. For legacy reasons, the alignment of all section symbols is 1. In reality, a section symbol is the highest alignment of all symbols in its section which in the above listing is 4.
- `Object name:` specifies the name of an object. The names are the C or mangled C++, depending on the language. The name of an object is similar to the one in a disassembled file.
- `Object location:` specifies the location an object. This is usually a name of the object file (.o), when no other column exists. In presence of an other column, the library file information is listed here.

In the sample above, note that the 5th line has a section name and that its Starting address, Virtual address and File offset values are similar to `__init_user` values. The 5th line is the section symbol for the objects that follow it. Its Size column is the total size of the section in the executable file (after dead stripping) and its alignment (column 5) is 1.

Although Listing 15.3 is from a normal `.o` file, if this project had an input file which was a partially linked file, then you would see a section symbol between each merged section from each input file used to create the partially linked file. For example, if `plf.o` is a partially linked file, and it is composed of `a.o` and `b.o` and each of those files had a `.text` section, then `plf.o` would have one `.text` section but two `.text` section symbols. The `.text` content from `a.o` would follow the first section symbol and the content from `b.o` would follow the second section symbol.

Memory Map

You can verify segment allocation from the Memory map section in a linker map file. Listing 15.4 shows a sample Memory map section.

Listing 15.4 Verifying segment allocation in a .MAP file

```
Memory map:
        Starting Size     File     ROM      RAM Buffer S-Record Bin File Bin File
        address           Offset   Address  Address    Line     Offset   Name
  .init ffe00000 00000318 00000300 ffe00000 ffe00000          2 00000000 Test.ffe00000.bin
  .text 00002000 0004f9d0 00086500 ffe00318 ffe00318         42 00000000 Test.ffe00318.bin
```

- `Starting address`: specifies the starting address for each section. Constant and executable object code are allocated in ROM space and data object code is allocated in RAM space.
- `Size`: specifies the size of each sections.
- `File offset:` specifies the offset of a section in the file.

- `ROM Address`: specifies the address of the section in the ROM image. For executable code and constants sections, `Starting address` is equal to `ROM Address`. For data sections, `ROM Address` specifies the address in ROM where the initialization values are stored.
- `RAM Buffer Address`: specifies the address in RAM that is to be used as a buffer for the flash image programmer. It is important to note that the RAM buffer is not used when the RAM address equals to the ROM address.
- `S-Record Line`: specifies the line number of a section in the S-Record file in decimal format.
- `Bin File Offset`: specifies the offset of a section in the binary file.
- `Bin File Name`: specifies the binary file name of the section. The file name also reflects the ROM address of the section.

Linker Generated Symbols

You can find a complete list of the linker generated symbols and user-defined symbols in either the C include file `__ppc_eabi_linker.h` or the assembly include file `__ppc_eabi_linker.i`. The CodeWarrior linker automatically generates symbols for the start address, the end address (the first byte after the last byte of the section), and the start address for the section if it will be burned into ROM. With a few exceptions, all CodeWarrior linker-generated symbols are immediate 32 bit values. Listing 15.5 shows a sample list of linker-generated symbols.

Listing 15.5 Sample list of linker-generated symbols

```
        _f_init       000034d8
    _f_init_rom       000034d8
        _e_init       000035b0
    _f_init_vle       000035b0
_f_init_vle_rom       000035b0
    _e_init_vle       00003864
        _f_text       00003864
    _f_text_rom       00003864
        _e_text       00003864
    _f_text_vle       00003870
_f_text_vle_rom       00003870
    _e_text_vle       00003ad4
```

If addresses are declared in your source file as `unsigned char _f_text[];` you can treat `_f_text` just as a C variable even though it is a 32-bit immediate value.

`unsigned int textsize = _e_text - _f_text;`

If you do need linker symbols that are not addresses, you can access them from C.

```
unsigned int size = (unsigned int)&_text_size;
```

The linker generates four symbols:

- `__ctors` — an array of static constructors
- `__dtors` — an array of destructors
- `__rom_copy_info` — an array of a structure that contains all of the necessary information about all initialized sections to copy them from ROM to RAM
- `__bss_init_info` — a similar array that contains all of the information necessary to initialize all of the bss-type sections. Please see `__init_data` in `__start.c`.

These four symbols are actually not 32-bit immediate values but are variables with storage. You access them just as C variables. The startup code now automatically handles initializing all bss type sections and moves all necessary sections from ROM to RAM, even for user defined sections.

Deadstripping

If the **Pool Data** checkbox is checked in the CodeWarrior IDE's **EPPC Processor** panel, the pooled data is not stripped. However, all small data and code is still subject to deadstripping.

Linker Command Files

Linker command files are an alternative way of specifying segment addresses. The other method of specifying segment addresses is by entering values manually in the Segment Addresses area of the **EPPC Linker** settings panel.

Only one linker command file is supported per target in a project. The linker command filename must end in the `.lcf` extension.

Setting up CodeWarrior IDE to accept LCF files

Projects created with the CodeWarrior IDE version 3 or earlier may not recognize the `.lcf` extension. Therefore, you may not be able to add a filename with the `.lcf` extension to the project. You need to create a file mapping to avoid this.

To add the `.lcf` file mapping to your project:

1. Select **Edit >** ***Target*** **Settings**, where *Target* is the name of the current build target**.**
2. Select the **File Mappings** panel.

3. In the File Type text box, type `TEXT`
4. In the Extension text box, type `.lcf`
5. From the Compiler listbox, select None
6. Click **Add** to save your settings.

Now, when you add an `.lcf` file to your project, the compiler recognizes the file as a linker command file.

Linker Command File Commands

The CodeWarrior Power Architecture linker supports these additional commands listed below:

- AGGRESSIVE_MERGE
- AGGRESSIVE_MERGE_FILES
- AGGRESSIVE_MERGE_SECTIONS
- DO_NOT_MERGE
- DO_NOT_MERGE_FILES
- DO_NOT_MERGE_SECTIONS
- INIT
- FORCEFILES
- SHORTEN_NAMES_FOR_TOR_101

AGGRESSIVE_MERGE

Specifies functions that should be considered for aggressive merging when applying the code merging optimization.

Syntax

```
DO_NOT_MERGE { symbol [, symbol]* }
```

AGGRESSIVE_MERGE_FILES

Specifies that all functions in object code files should be considered for aggressive merging when applying the code merging optimization.

Syntax

DO_NOT_MERGE_FILES { *file-name* [, *file-name*]* }

AGGRESSIVE_MERGE_SECTIONS

Specifies that all functions in object code sections should be considered for aggressive merging when applying the code merging optimization.

Syntax

AGGRESSIVE_MERGE_SECTIONS { *section-name* [, *section-name*]* }

DO_NOT_MERGE

Specifies functions that should not be removed when applying the code merging optimization.

Syntax

DO_NOT_MERGE { *symbol* [, *symbol*]* }

Remarks

This directive specifies functions that the linker should keep in the output file when applying the code merging optimization even if other functions with identical object code exist.

DO_NOT_MERGE_FILES

Specifies that all functions in a file should not be removed when applying the code merging optimization.

Syntax

DO_NOT_MERGE_FILES { *file-name* [, *file-name*]* }

DO_NOT_MERGE_SECTIONS

Specifies that all functions in an object code section should not be removed when applying the code merging optimization.

Syntax

```
DO_NOT_MERGE_SECTIONS { section-name [, section-name]* }
```

INIT

Defines the initialization entry point for the application.

Syntax

```
INIT (FunctionName)
```

Remarks

This command is mandatory for assembly application and optional otherwise. It cannot be specified more than once in the `prm` file. When you specify the `INIT` command in the `prm` file, the linker uses the specified function as application entry point. This is either the main routine or a startup routine calling the main routine.

When `INIT` is not specified in the `prm` file, the linker looks for a function named `__start` and uses it as the application entry point.

Example

```
INIT (MyGlobStart) /* Specify a global variable as
application entry point.*/
```

FORCEFILES

Specifies that the contents of object code files must not be deadstripped.

Syntax

```
FORCEFILES { file-name [, file-name]* }
```

Remarks

Use `FORCEFILES` to list source files, archives, or archive members that you do not want dead-stripped. All objects in each of the files are included in the linker's output file even if the linker has determined that those objects are not referenced by other objects.

If you only have a few symbols that you do not want deadstripped, use `FORCEACTIVE`.

SHORTEN_NAMES_FOR_TOR_101

The directive `SHORTEN_NAMES_FOR_TOR_101` instructs the linker to shorten long template names for the benefit of the WindRiver® Systems Target Server. To use this directive, simply add it to the linker command file on a line by itself.

```
SHORTEN_NAMES_FOR_TOR_101
```

WindRiver Systems Tornado Version 1.0.1 (and earlier) does not support long template names as generated for the MSL C++ library. Therefore, the template names must be shortened if you want to use them with these versions of the WindRiver Systems Target Server.

16

C Compiler

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

- Extensions to Standard C
- C99 Extensions
- GCC Extensions

Extensions to Standard C

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

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

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.

Table 16.1 shows how to control the compiler's features for ISO conformance.

Table 16.1 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++ Language Settings** panel
source code	`#pragma ANSI_strict` `#pragma only_std_keywords`
command line	`-ansi`

C++-style Comments

When ANSI strictness is off, the C compiler allows C++-style comments. Listing 16.1 shows an example.

Listing 16.1 C++ Comments

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

Unnamed Arguments

When ANSI strictness is off, the C compiler allows unnamed arguments in function definitions. Listing 16.2 shows an example.

Listing 16.2 Unnamed Arguments

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

Extensions to the Preprocessor

When ANSI strictness is off, the C compiler allows a # to prefix an item that is not a macro argument. It also allows an identifier after an `#endif` directive. Listing 16.3 and Listing 16.4 show examples.

Listing 16.3 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 16.4 Identifiers After #endif

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

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

Non-Standard Keywords

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

Declaring Variables by Address

The C compiler lets you explicitly specify the address that contains the value of a variable. For example, the following definition states that the variable `MemErr` contains the contents of the address `0x220`:

```
short MemErr:0x220;
```

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

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

Controlling C99 Extensions

Table 16.2 shows how to control C99 extensions.

Table 16.2 Controlling C99 extensions to the C language

To control this option from here...	use this setting
CodeWarrior IDE	**Enable C99 Extensions** in the **C/C++ Language Settings** panel
source code	`#pragma c99`
command line	`-c99`

Trailing Commas in Enumerations

When the C99 extensions setting is on, the compiler allows a comma after the final item in a list of enumerations. Listing 16.5 shows an example.

Listing 16.5 Trailing comma in enumeration example

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

Compound Literal Values

When the C99 extensions setting is on, the compiler allows literal values of structures and arrays. Listing 16.6 shows an example.

Listing 16.6 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});
```

Designated Initializers

When the C99 extensions setting is on, the compiler allows an extended syntax for specifying which structure or array members to initialize. Listing 16.7 shows an example.

Listing 16.7 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. */
```

Predefined Symbol __func__

When the C99 extensions setting is on, the compiler offers the `__func__` predefined variable. Listing 16.8 shows an example.

Listing 16.8 Predefined symbol __func__

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

Implicit Return From main()

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

```
return 0;
```

Non-constant Static Data Initialization

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

Variable Argument Macros

When the C99 extensions setting is on, the compiler allows macros to have a variable number of arguments. Listing 16.9 shows an example.

Listing 16.9 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;
}
```

Extra C99 Keywords

When the C99 extensions setting is on, the compiler recognizes extra keywords and the language features they represent. Table 16.3 lists these keywords.

Table 16.3 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

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 until the end of a source code line.

A C-style comment begins with

```
/*
```

ends with

```
*/
```

and may span more than one line.

C++-Style Digraphs

When the C99 extensions setting is on, the compiler recognizes C++-style two-character combinations that represent single-character punctuation. Table 16.4 lists these digraphs.

Table 16.4 C++-Style Digraphs

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

Empty Arrays in Structures

When the C99 extensions setting is on, the compiler allows an empty array to be the last member in a structure definition. Listing 16.10 shows an example.

Listing 16.10 Example of an Empty Array as the Last struct Member

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

Hexadecimal Floating-Point Constants

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

Examples:

`0x2f.3a2p3`

`0xEp1f`

`0x1.8p0L`

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

Variable-Length Arrays

Variable length arrays are supported within local or function prototype scope, as required by the ISO/IEC 9899-1999 (“C99”) standard. Listing 16.11 shows an example.

Listing 16.11 Example of C99 Variable Length Array usage

```
#pragma c99 on

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

While the example shown in Listing 16.12 generates an error message.

Listing 16.12 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 Listing 16.13.

Listing 16.13 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
```

Unsuffixed Decimal Literal Values

Listing 16.14 shows an example of specifying decimal literal values without a suffix to specify the literal's type.

Listing 16.14 Examples of C99 Unsuffixed Constants

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

#pragma c99 off

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

C99 Complex Data Types

The compiler supports the C99 complex and imaginary data types when the `C99 extensions` option is enabled. Listing 16.15 shows an example.

Listing 16.15 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.

GCC Extensions

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

- Controlling GCC Extensions
- Initializing Automatic Arrays and Structures
- The sizeof() Operator
- Statements in Expressions

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

Controlling GCC Extensions

Table 16.5 shows how to turn GCC extensions on or off.

Table 16.5 Controlling GCC extensions to the C language

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

Initializing Automatic Arrays and Structures

When the GCC extensions setting is on, array and structure variables that are local to a function and have the automatic storage class may be initialized with values that do not need to be constant. Listing 16.16 shows an example.

Listing 16.16 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 };
}
```

The sizeof() Operator

When the GCC extensions setting is on, the `sizeof()` operator computes the size of function and void types. In both cases, the `sizeof()` operator evaluates to 1. The ISO/IEC 9899-1990 C Standard ("C90") does not specify the size of the `void` type and functions. Listing 16.17 shows an example.

Listing 16.17 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 */
}
```

Statements in Expressions

When the GCC extensions setting is on, expressions in function bodies may contain statements and definitions. To use a statement or declaration in an expression, enclose it within braces. The last item in the brace-enclosed expression gives the expression its value. Listing 16.18 shows an example.

Listing 16.18 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);
}
```

Redefining Macros

When the GCC extensions setting is on, macros may be redefined with the `#define` directive without first undefining them with the `#undef` directive. Listing 16.19 shows an example.

Listing 16.19 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
```

The typeof() Operator

When the GCC extensions setting is on, the compiler recognizes the `typeof()` operator. This compile-time operator returns the type of an expression. You may use the value returned by this operator in any statement or expression where the compiler expects you to specify a type. The compiler evaluates this operator at compile time. The `__typeof()__` operator is the same as this operator. Listing 16.20 shows an example.

Listing 16.20 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;
```

Void and Function Pointer Arithmetic

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

The __builtin_constant_p() Operator

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

Forward Declarations of Static Arrays

When the GCC extensions setting is on, the compiler will not issue an error when you declare a static array without specifying the number of elements in the array if you later declare the array completely. Listing 16.21 shows an example.

Listing 16.21 Forward declaration of an empty array

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

Omitted Operands in Conditional Expressions

When the GCC extensions setting is on, you may skip the second expression in a conditional expression. The default value for this expression is the first expression. Listing 16.22 shows an example.

Listing 16.22 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. */
}
```

The __builtin_expect() Operator

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

This compile-time operator takes two arguments:

- the first argument must be an integral expression

- the second argument must be a literal value

The second argument is the most likely result of the first argument. Listing 16.23 shows an example.

Listing 16.23 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);
         }
    }
}
```

Void Return Statements

When the GCC extensions setting is on, the compiler allows you to place expressions of type `void` in a `return` statement. Listing 16.24 shows an example.

Listing 16.24 Returning void

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

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

Minimum and Maximum Operators

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

Listing 16.25 Example of minimum and maximum operators

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

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. Listing 16.26 shows an example.

Listing 16.26 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. */
  }
}
```

17

C++ Compiler

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

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

C++ Compiler Performance

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

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

Precompiling C++ Source Code

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

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

Using the Instance Manager

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

- 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.

NOTE

Table 17.1 shows how to control the C++ instance manager.

Table 17.1 Controlling the C++ instance manager

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

Extensions to Standard C++

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

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

`__PRETTY_FUNCTION__` Identifier

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

Standard and Non-Standard Template Parsing

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

A qualified name that refers to a type and that depends on a template parameter must begin with `typename` (ISO/IEC 14882-1998 C++, §14.6). Listing 17.1 shows an example.

Listing 17.1 Using the `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. Listing 17.2 shows an example.

Listing 17.2 Using the `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. Listing 17.3 shows an example.

Listing 17.3 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 14882-1998 C++, §14.6.2). See Listing 17.4.

Listing 17.4 Qualifying template arguments in base classes

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

When a template contains a function call in which at least one of the function's arguments is type-dependent, the compiler uses the name of the function in the context of the template definition (ISO/IEC 14882-1998 C++, §14.6.2.2) and the context of its instantiation (ISO/IEC 14882-1998 C++, §14.6.4.2). Listing 17.5 shows an example.

Listing 17.5 Function call with type-dependent argument

```
void f(char);

template <typename T> void type_dep_func()
{
  f(1); // Uses f(char), above; f(int) is not declared yet.
  f(T()); // f() called with a type-dependent argument.
}

void f(int);
struct A{};
void f(A);

int main()
{
  type_dep_func<int>(); // Calls f(char) twice.
  type_dep_func<A>(); // Calls f(char) and f(A);
  return 0;
}
```

The compiler only uses external names to look up type-dependent arguments in function calls. See Listing 17.6.

Listing 17.6 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 Listing 17.7.

Listing 17.7 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 Listing 17.8.

Listing 17.8 Address of Template-id Supported

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

Implementation-Defined Behavior

Annex A of the ISO/IEC 14882-1998 C++ Standard lists compiler behaviors that are beyond the scope of the standard, but which must be documented for a compiler implementation. This annex also lists minimum guidelines for these behaviors, although a conforming compiler is not required to meet these minimums.

The CodeWarrior C++ compiler has these implementation quantities listed in Table 17.2, based on the ISO/IEC 14882-1998 C++ Standard, Annex A.

NOTE The term *unlimited* in Table 17.2 means that a behavior is limited only by the processing speed or memory capacity of the computer on which the CodeWarrior C++ compiler is running.

Table 17.2 Implementation Quantities for C/C++ Compiler (ISO/IEC 14882-1998 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

Table 17.2 Implementation Quantities for C/C++ Compiler (ISO/IEC 14882-1998 C++, §A)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
Characters in a character string literal or wide string literal (after concatenation)	65536	Unlimited
Size of an object	262144	2 GB
Nesting levels for `#include` files	256	256
`Case` labels for a `switch` statement (excluding those for any nested `switch` statements)	16384	Unlimited
Data members in a single class, structure, or union	16384	Unlimited
Enumeration constants in a single enumeration	4096	Unlimited
Levels of nested class, structure, or union definitions in a single struct-declaration-list	256	Unlimited
Functions registered by `atexit()`	32	64
Direct and indirect base classes	16384	Unlimited
Direct base classes for a single class	1024	Unlimited
Members declared in a single class	4096	Unlimited
Final overriding virtual functions in a class, accessible or not	16384	Unlimited
Direct and indirect virtual bases of a class	1024	Unlimited
Static members of a class	1024	Unlimited
Friend declarations in a class	4096	Unlimited
Access control declarations in a class	4096	Unlimited
Member initializers in a constructor definition	6144	Unlimited
Scope qualifications of one identifier	256	Unlimited
Nested external specifications	1024	Unlimited

Table 17.2 Implementation Quantities for C/C++ Compiler (ISO/IEC 14882-1998 C++, §A)

Behavior	Standard Minimum Guideline	CodeWarrior Limit
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

GCC Extensions

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

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

Listing 17.9 Using the :: operator in class declarations

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

18

Precompiling

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

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

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

- What Can be Precompiled
- Using a Precompiled File
- Creating a Precompiled File

What Can be Precompiled

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

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

  You cannot precompile libraries or other binary files.
-
- 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.

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, variables or types before including a precompiled file.
- Typically, a source code file includes a precompiled file before anything else (except comments).

Listing 18.1 shows an example.

Listing 18.1 Using a precompiled file

```
/* sock_main.c */

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

/* ... */
```

Creating a Precompiled File

This section shows how to create and manage precompiled files:

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

Precompiling a File in the CodeWarrior IDE

To precompile a file in the CodeWarrior IDE, follow these steps:

1. Start the CodeWarrior IDE.
2. Open or create a project.
3. Choose or create a build target in the project.

   The IDE will use the settings in the project's active build target when preprocessing and precompiling the file.

4. Open the source code file to precompile.

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

5. Choose a location and enter a name for the new precompiled file.
6. Click **Save**.

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

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

Precompiling a File on the 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 -precompile p_file`

   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.

Updating a Precompiled File Automatically

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

- The text file name ends with `.pch`.
- The file is in a project's build target.
- The file uses the `precompile_target` pragma.

- The file, or files it depends on, have been modified.

  See the *CodeWarrior IDE User Guide* for information on how the IDE determines a file's dependencies.

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

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 Listing 18.2 specifies that the variable `xxx` is a `far` variable.

Listing 18.2 Pragma Settings in a Precompiled Header

```
/* my_pch.pch */

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

#pragma far_data on
extern int xxx;
```

The source code in Listing 18.3 includes the precompiled version of Listing 18.2.

Listing 18.3 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.

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 explains these optimizations and how to apply them:

- Interprocedural Analysis
- Intermediate Optimizations
- Inlining

Interprocedural Analysis

Most compiler optimizations are applied only within a function. The compiler analyzes a function's flow of execution and how the function uses variables. It uses this information to find shortcuts in execution and reduce the number of registers and memory that the function uses. These optimizations are useful and effective but are limited to the scope of a function.

The CodeWarrior compiler has a special optimization that it applies at a greater scope. Widening the scope of an optimization offers the potential to greatly improve performance and reduce memory use. *Interprocedural analysis* examines the flow of execution and data within entire files and programs to improve performance and reduce size.

- Invoking Interprocedural Analysis
- File-Level Optimization
-

Invoking Interprocedural Analysis

Table 19.1 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
source code	`#pragma ipa file | on | function | off`
command line	`-ipa file | function | off`

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.

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.

Intermediate Optimizations

After it translates a function into its intermediate representation, the compiler may optionally apply some optimizations. The result of these optimizations on the intermediate representation will either reduce the size of the executable code, improve the executable code's execution speed, or both.

- Dead Code Elimination
- Expression Simplification
- Common Subexpression Elimination
- Copy Propagation
- Dead Store Elimination
- Live Range Splitting
- Loop-Invariant Code Motion
- Strength Reduction
- Loop Unrolling

Dead Code Elimination

The dead code elimination optimization removes expressions that are not accessible or are not referred to. This optimization reduces size and increases execution speed.

Table 19.2 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 **Global Optimizations** settings panel.
source code	`#pragma opt_dead_code on | off | reset`
command line	`-opt [no]deadcode`

In Listing 19.1, the call to `func1()` will never execute because the `if` statement that it is associated with will never be true. Consequently, the compiler can safely eliminate the call to `func1()`, as shown in Listing 19.2.

Listing 19.1 Before dead code elimination

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

Listing 19.2 After dead code elimination

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

Expression Simplification

The expression simplification optimization attempts to replace arithmetic expressions with simpler expressions. Additionally, the compiler also looks for operations in expressions that can be avoided completely without affecting the final outcome of the expression. This optimization reduces size and increases speed.

Table 19.3 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 **Global Optimizations** 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, Listing 19.3 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 19.3 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;
}
```

Listing 19.4 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 19.4 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;
}
```

Common Subexpression Elimination

Common subexpression elimination replaces multiple instances of the same expression with a single instance. This optimization reduces size and increases execution speed.

Table 19.4 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 **Global Optimizations** settings panel.
source code	`#pragma opt_common_subs on | off | reset`
command line	`-opt [no]cse`

For example, in Listing 19.5, the subexpression `x * y` occurs twice.

Listing 19.5 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;
    }
}
```

Listing 19.6 shows equivalent source code after the compiler applies common subexpression elimination. The compiler generates instructions to compute `x * y` and store it in a hidden, temporary variable. The compiler then replaces each instance of the subexpression with this variable.

Listing 19.6 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;
    }
}
```

Copy Propagation

Copy propagation replaces variables with their original values if the variables do not change. This optimization reduces runtime stack size and improves execution speed.

Table 19.5 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 Listing 19.7, the variable `j` is assigned the value of `x`. But `j`'s value is never changed, so the compiler replaces later instances of `j` with `x`, as shown in Listing 19.8.

By propagating `x`, the compiler is able to reduce the number of registers it uses to hold variable values, allowing more variables to be stored in registers instead of slower memory. Also, this optimization reduces the amount of stack memory used during function calls.

Listing 19.7 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;
    }
}
```

Listing 19.8 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;
    }
}
```

Dead Store Elimination

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

Table 19.6 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 **Global Optimizations** settings panel.
source code	`#pragma opt_dead_assignments on | off | reset`
command line	`-opt [no]deadstore`

For example, in Listing 19.9 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()`.

In Listing 19.10 the compiler can safely remove the first assignment to `x` since the result of this assignment is never used.

Listing 19.9 Before dead store elimination

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

Listing 19.10 After dead store elimination

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

Live Range Splitting

Live range splitting attempts to reduce the number of variables used in a function. This optimization reduces a function’s runtime stack size, requiring fewer instructions to invoke the function. This optimization potentially improves execution speed.

Table 19.7 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 **Global Optimizations** settings panel.
source code	There is no pragma to control this optimization.
command line	`-opt level=3`, `-opt level=4`

For example, in Listing 19.11 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`.

In Listing 19.12, 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 19.11 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);
}
```

Listing 19.12 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);
}
```

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.

Table 19.8 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 **Global Optimizations** settings panel.
source code	`#pragma opt_loop_invariants on | off | reset`
command line	`-opt [no]loop[invariants]`

For example, in Listing 19.13, 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 19.14 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 19.13 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;
    }
}
```

Listing 19.14 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;
    }
}
```

Strength Reduction

Strength reduction attempts to replace slower multiplication operations with faster addition operations. This optimization improves execution speed but increases code size.

Table 19.9 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 **Global Optimizations** settings panel.
source code	`#pragma opt_strength_reduction on | off | reset`
command line	`-opt [no]strength`

For example, in Listing 19.15, the assignment to elements of the `vec` array use a multiplication operation that refers to the `for` loop’s counter variable, `i`.

In Listing 19.16, 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 19.15 Before strength reduction

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

Listing 19.16 After strength reduction

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

Loop Unrolling

Loop unrolling inserts extra copies of a loop's body in a loop to reduce processor time executing a loop's overhead instructions for each iteration of the loop body. In other words, this optimization attempts to reduce the ratio of time that the processor executes a loop's completion test and branching instructions compared to the time the processor executes the loop's body. This optimization improves execution speed but increases code size.

Table 19.10 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 **Global Optimizations** settings panel.
source code	`#pragma opt_unroll_loops on | off | reset`
command line	`-opt level=3, -opt level=4`

For example, in Listing 19.17, the `for` loop's body is a single call to a function, `otherfunc()`. For each time the loop's completion test executes

`for (i = 0; i < MAX; ++i)`

the function executes the loop body only once.

In Listing 19.18, 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 19.17 Before loop unrolling

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

Listing 19.18 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;
    }
}
```

Inlining

Inlining replaces instructions that call a function and return from it with the actual instructions of the function being called. Inlining functions makes your program faster because it executes the function code immediately without the overhead of a function call and return. However, inlining can also make your program larger because the compiler may insert the function's instructions many times throughout your program.

The rest of this section explains how to specify which functions to inline and how the compiler performs the inlining:

- Choosing Which Functions to Inline
- Inlining Techniques

Choosing Which Functions to Inline

The compiler offers several methods to specify which functions are eligible for inlining.

To specify that a function is eligible to be inlined, precede its definition with the `inline`, `__inline__`, or `__inline` keyword. To allow these keywords in C source code, turn off **ANSI Keywords Only** in the CodeWarrior IDE's **C/C++ Language** settings **C/C++ Language** panel or turn off the `only_std_keywords` pragma in your source code.

To verify that an eligible function has been inlined or not, use the **Non-Inlined Functions** option in the IDE's **C/C++ Warnings** panel or the `warn_notinlined` pragma. Listing 19.19 shows an example.

Listing 19.19 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))`. Listing 19.20 shows an example.

Listing 19.20 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. Listing 19.21 shows an example.

Listing 19.21 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;
}
```

Some kinds of functions are never inlined:

- 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 **Global Optimizations** panel
- functions which have their addresses stored in variables

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

- functions that return class objects that need destruction
- functions with class arguments that need destruction

The compiler will inline functions that need destruction, without any dependency on the ISO C++ templates, if the class has a trivial empty constructor. Listing 19.22 shows an example.

Listing 19.22 Inlining function with an empty destructor

```
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;
}
```

Inlining Techniques

The depth of inlining explains how many levels of function calls the compiler will inline. The **Inline Depth** setting in the IDE's **C/C++ Language** settings panel and the `inline_depth` pragma control inlining depth.

Normally, the compiler only inlines an eligible function if it has already translated the function's definition. In other words, if an eligible function has not yet been compiled, the compiler has no object code to insert. To overcome this limitation, the compiler can perform interprocedural analysis (IPA). 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++ Language** settings panel.

The compiler normally inlines functions from the first function in a chain of function calls to the last function called. Alternately, the compiler may inline functions from the last function called to the first function in a chain of function calls. The **Bottom-up Inlining** option in the IDE's **C/C++ Language** settings panel and the `inline_bottom_up` and `inline_bottom_up_once` pragmas control this reverse method of inlining.

Some functions that have not been defined with the `inline`, `__inline__`, or `__inline` keywords may still be good candidates to be inlined. Automatic inlining allows the compiler to inline these functions in addition to the functions that you explicitly specify as eligible for inlining. The **Auto-Inline** option in the IDE's **C/C++ Language** panel and the `auto_inline` pragma control this capability.

When inlining, the compiler calculates the complexity of a function by counting the number of statements, operands, and operations in a function to determine whether or not to inline an eligible function. The compiler does not inline functions that exceed a 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.

20

Power Architecture Optimizations

This chapter describes optimizations specific to Power Architecture platforms that the CodeWarrior compiler applies to your object code:

- Code Merging

Code Merging

Code merging reduces the size of object code by removing identical functions. Two or more functions are identical when their executable code is identical.

The CodeWarrior build tools can only apply this optimization to object files generated by the CodeWarrior compilers. The CodeWarrior build tools can only apply this optimization to object code translated from C and C++ source code; the tools cannot apply this optimization to object code generated from assembly files.

TIP For example, the C++ compiler often generates several copies of the same function when it instantiates template functions. These functions have different names, and these names are considered *weak*. Under normal circumstances, the linker will issue an error message if it encounters duplicate names. But the linker ignores duplicate names that are marked as weak.

The code merging optimization removes all but one of a group of identical functions. Table 20.1 shows how to invoke this optimization for all functions. Table 20.2 shows how to invoke this optimization for weak functions.

Table 20.1 Controlling code merging for all identical functions

Control this option from here...	use this setting
CodeWarrior IDE	Choose **All** from the **Code Merging** option of the **Linker Optimizations** settings panel.
command line	`-code_merging all`

Table 20.2 Controlling code merging for weak functions only

Control this option from here...	use this setting
CodeWarrior IDE	Choose **Safe** from the **Code Merging** option of the **Linker Optimizations** settings panel.
command line	`-code_merging safe`

The code merging optimization will not remove an identical copy of a function if your program refers to its address. In this case, the compiler keeps this copied function but replaces its executable code with a branch instruction to the original function.

To ignore references to function addresses, use aggressive code merging. Table 20.3 shows how to invoke aggressive code merging.

Table 20.3 Controlling aggressive code merging

Control this option from here...	use this setting
CodeWarrior IDE	Choose **Aggressive Merge** in the **Linker Optimizations** settings pane.l
command line	`-code_merging all,aggressive,` or `-code_merging safe,aggressive`

To specify that the compiler and linker must not apply code merging to a function, use this directive in your source code:

`__declspec(do_not_merge)` *`fname;`*

where *fname* is the name of a function.

To specify to the linker how to apply code merging to functions, object files, or sections, use these directives in linker command file:

```
DO_NOT_MERGE
DO_NOT_MERGE_FILES
DO_NOT_MERGE_SECTIONS
AGGRESSIVE_MERGE
AGGRESSIVE_MERGE_FILES
AGGRESSIVE_MERGE_SECTIONS
```

21 Inline-Assembly for Power Architecture Build Tools

This chapter explains how to use the inline assembler built into the CodeWarrior™ C and C++ compilers for Power Architecture processors. The compiler's inline assembler allows you to embed assembly language statements in C and C++ functions.

The chapter does *not* describe the standalone CodeWarrior assembler. For information about this tool, refer to the chapter titled Assembler.

This chapter does not document all the instructions in the Power Architecture instruction set. For complete documentation of this instruction set, see *Programming Environments Manual for 32-Bit Implementations of the PowerPC™ Architecture,* published by Freescale.

Finally, refer to this web page for documentation of Freescale's entire Power Architecture product line:

http://www.freescale.com/powerarchitecture

The sections in this chapter are:

- Assembly Syntax
- Referring to Assembly, C, and C++ Variables
- Assembler Directives
- Intrinsic Functions

Assembly Syntax

The compiler's inline assembler allows a variety of ways to insert assembly language statements in your C or C++ source code:

- Specifying Inline Assembly Statements
- Function-Level Inline Assembly
- Statement-Level Inline Assembly
- GCC-Style Inline Assembly
- Branch Prediction
- PC-Relative Addressing

- Normal, Record, and Overflow Forms
- Creating Statement Labels
- Using Comments
- Using the Preprocessor

Specifying Inline Assembly Statements

To specify that a block of C or C++ source code should be interpreted as assembly language, use the `asm` keyword.

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

As an alternative, the compiler also recognizes the keyword `__asm` even if the **ANSI Keywords Only** checkbox is checked.

There are a few ways to use assembly language with the CodeWarrior compilers.

- Function-level assembly language: an entire function is in assembly language.
- Statement-level assembly language: mix assembly language with regular C or C++ statements.
- Intrinsic functions: the compiler makes some assembly instructions available as functions that your program calls as regular C or C++ functions.

Keep these tips in mind as you write inline assembly statements:

- All statements must follow this syntax:

  ```
  [label:] (instruction | directive) [operands]
  ```

- Each inline assembly statement must end with a newline or a semicolon (`;`).
- Hexadecimal constants must be in C-style.

  For example: `li r3, 0xABCDEF`

- Assembler directives, instructions, and registers are case-sensitive and must be in lowercase.

Function-Level Inline Assembly

The compiler accepts function definitions that are composed entirely of assembly statements. Function-level assembly code uses this syntax:

```
asm function-definition
```

A function that uses function-level assembly must end with a `blr` instruction.

Listing 21.1 Example Assembly Language Function

```
asm void mystrcpy(char *tostr, char *fromstr)

{
    addi  tostr,tostr,-1
    addi  fromstr,fromstr,-1
@1  lbzu  r5,1(fromstr)
    cmpwi r5,0
    stbu  r5,1(tostr)
    bne   @1
  blr
}
```

Statement-Level Inline Assembly

The compiler accepts functions that mix regular C/C++ statements with inline assembly. Statement-level assembly language acts as a block of assembly language that may appear anywhere that the compiler allows a regular C or C++ statement. It has this syntax:

`asm {` *`one or more instructions`* `}`

Listing 21.2 Example of statement-level inline assembly

```
void g(void)
{
  asm { add r2,r3,r4 ; }
}
```

NOTE If you check the **Inlined Assembler is Volatile** checkbox in the **EPPC Processor** panel, functions that *contain* an `asm` block are only partially optimized. The optimizer optimizes the function, but skips any `asm` blocks of code. If the **Inlined Assembler is Volatile** checkbox is clear, the compiler also optimizes `asm` statements.

GCC-Style Inline Assembly

The CodeWarrior compiler accepts GCC (Gnu Compiler Collection) syntax for inline assembly statements:

`asm("`*assembly-statements*`")`

where *assembly-statements* represents inline assembly statements that follow the syntax recognized by the GCC C/C++ compiler.

Listing 21.3 Example of GCC-style inline assembly

```
void g(void)
{
  asm ("add r2,r3,r4\n\t");
}
```

NOTE Refer to this web page for details on extensions priovided by the GNU Compiler Collection (GCC): http://gcc.gnu.org/onlinedocs/gcc/C-Extensions.html#C-Extensions

Branch Prediction

To set the branch prediction (`y`) bit for those branch instructions that can use it, use plus (+) or minus (-). For example:

```
@1 bne+ @2
@2 bne- @1
```

PC-Relative Addressing

The compiler does not accept references to addresses that are relative to the program counter. For example, the following is not supported:

```
asm(b *+8);
```

Instead, use one of the following:

1. Use labels to specify an address in executable code.

Listing 21.4 Using a label instead if PC-relative addressing

```
asm(b next);
asm(next:);

/* OR */

asm{
    b next1
    next1:
}
```

2. Use relative branch in the function-level assembly instead of statement level.

Listing 21.5 Using relative branching in the function-level assembly

```
asm void functionLevel();
asm void functionLevel(){
  b *+8
  nop
  blr
}
```

Normal, Record, and Overflow Forms

Most integer instructions have four forms:

- normal form — `add r3,r4,r5`
- record form — `add. r3,r4,r5`

  This form ends in a period. This form sets register `cr0` to whether the result is less, than, equal to, or greater than zero.
- overflow — `addo r3,r4,r5`

  This form ends in the letter (`o`). This form sets the `SO` and `OV` bits in the `XER` if the result overflows.
- overflow and record — `addo. r3,r4,r5`

  This form ends in (`o.`). This form sets both registers.

Some instructions only have a record form (with a period). Always make sure to include the period. For example:

```
andi.  r3,r4,7
andis. r3,r4,7
stwcx. r3,r4,r5
```

Creating Statement Labels

The name of an inline assembly language statement label must follow these rules:

- A label name cannot be the same as the identifier of any local variables of the function in which the label name appears.
- A label name does not have to start in the first column of the function in which it appears; a label name can be preceded by white space.
- A label name can begin with an "at-sign" character (`@`) unless the label immediately follows a local variable declaration.

- A label name must end with a colon character (`:`) unless it begins with an at-sign character (`@`).

  For example, `red:` and `@red` are valid, but `red` is *not* valid.

- A label name *can* be the same as an assembly language statement mnemonic.

  For example, this statement is valid:

  ```
  add: add r3, r4, r5
  ```

Examples:

```
asm void func1(){
int i;
  @x: li r0,1 //Invalid !!!
}
```

```
asm void func2(){
int i;
  x:  li r0,1 //OK
  @y: add r3, r4, r5 //OK
}
```

This is an example of a complete inline assembly language function:

```
asm void red(void){
  x1:  add r3,r4,r5
  @x2: add r6,r7,r8
}
```

Using Comments

You cannot begin comments with a pound sign (#) because the preprocessor uses the pound sign. For example, this format is invalid:

```
add   r3,r4,r5 # Comment
```

Use C and C++ comments in this format:

```
add   r3,r4,r5 // Comment
add   r3,r4,r5 /* Comment */
```

Using the Preprocessor

You can use all preprocessor features, such as comments and macros, in the assembler. In multi-line macros, you must end each assembly statement with a semicolon (`;`) because the (\) operator removes newlines. For example:

```
#define remainder(x,y,z) \
divw z,x,y; \
```

```
mullw z,z,y; \
subf z,z,x

asm void newPointlessMath(void)
{
  remainder(r3,r4,r5)
  blr
}
```

Referring to Assembly, C, and C++ Variables

The compiler's inline assembler has access to the local and global variables declared in C or C++ source code. The assembler also allows access to members of array, `struct`, and `class` objects:

- Using Local Variables and Arguments
- Creating a Stack Frame
- Referring to Variables in Instruction Operands
- Referring to Variables in Memory
- Referring to Registers
- Referring to Labels
- Using Variable Names as Memory Locations
- Using Immediate Operands

Using Local Variables and Arguments

To refer to a memory location, you can use the name of a local variable or argument.

The rule for assigning arguments to registers or memory depends on whether the function has a stack frame.

If function has a stack frame, the inline assembler assigns:

- scalar arguments declared as `register` to general purpose registers `r14` to `r31`
- floating-point arguments declared as `register` to floating point `fp14` to `fp31`
- other arguments to memory locations

If a function has no stack frame, the inline assembler assigns arguments that are declared `register` and kept in registers. If you have variable or non-register arguments, the compiler will warn you that you should use `frfree`

NOTE Some op-codes require registers, and others require objects. For example, if you use `nofralloc` with function arguments, you may run into difficulties.

Creating a Stack Frame

You need to create a stack frame for a function if the function:

- calls other functions.
- declares non-register arguments or local variables.

To create a stack frame, use the `fralloc` directive at the beginning of your function and the `frfree` directive just before the `blr` statement. The directive `fralloc` automatically allocates (while `ffree` automatically de-allocates) memory for local variables, and saves and restores the register contents.

Listing 21.6 Example of creating a stack frame

```
asm void red ()
{
  fralloc
  // Your code here
  frfree
  blr
}
```

The `fralloc` directive has an optional argument, *number*, that lets you specify the size, in bytes, of the parameter area of the stack frame. The stack frame is an area for storing parameters used by the assembly code. The compiler creates a 0-byte parameter area for you to pass variables into your assembly language functions.

Function arguments are passed using registers. If your assembly-language routine calls any function that requires more parameters than will fit into registers `r3` to `r10` and `fp1` to `fp8`, you need to pass that size to `fralloc`. In the case of integer values, registers `r3` — `r10` are used. For floating-point values, registers `fp1` — `fp8` are used.

As an example, if you pass 12 values of type `long integer` to your assembly function, this would consume 16 bytes of the parameter area. Registers `r3` — `r10` will hold eight integers, leaving 4 byte integers in the parameter area.

Referring to Variables in Instruction Operands

For instructions that require register operands, (such as the `add` instruction), global variables, function parameters, and local variables must be declared with the keyword `register`.

Listing 21.7 shows inline assembly language statements that correctly use C-language variables as operands in instructions that require register operands.

Listing 21.7 Using C Variables with Instructions that Require Register Operands

```
register int my_global = 25; /* global variable */

asm void red(register int *my_param)
{
register int my_loc = 1; /* my_loc is in register, not the stack */
register int result;

   fralloc

   add result, 1,          my_global   /* line 10 */
   add result, my_global, my_param    /* line 11 */
   add result, my_param,  my_loc,     /* line 12 */

   frfree
   blr
}
```

In Listing 21.7, the statement on line 10, 11, and 12 are all correct because their operands are all declared with the `register` keyword.

Referring to Variables in Memory

For instructions that take a memory operand (such as the `lwz` instruction), follow these rules when using a C-language variable as an operand:

- Global variables and function parameters must:

  Be declared with the `register` keyword.

  – Adhere to the syntax below when used as operands, so they are treated as an offset from zero.

    `instrName regName, 0(globalVarName)`

    or

    `instrName regName, 0(parameterName)`

- Local variable declarations must *not* use the `register` keyword.

Listing 21.8 shows inline assembly language statements that correctly use C-language variables as operands in instructions that take a memory operand.

Listing 21.8 Using C Variables with Instructions that Take a Memory Operand

```
register int my_global = 25; /* global variable */

asm void red(register int *my_param)
{
int my_loc = 1; /* my_loc is on the stack, not in a register */

   fralloc

   lwz r4, 0(my_global)   /* line 9 */
   lwz r4, 0(my_param)    /* line 10 */
   lwz r4, my_loc         /* line 11 */
   lwz r4, my_loc(SP)     /* line 12 - equivalent to statement 11 */

   frfree
   blr
}
```

In Listing 21.8:

- The statement on line 9 is correct.
  - The operand is fully expressed (because it is an offset from zero).
  - The argument `my_global` is in a register.
- The statement on line 10 is correct for the same reasons as stated above.
- The statement on line 11 is correct.

  The CodeWarrior inline assembler automatically adds the contents of the `SP` register to local variable `my_loc`.
- The statement on line 12 is correct.

  Note that statements 11 and 12 are equivalent.

  As mentioned above, the inline assembler automatically adds the `SP` register to local variable `my_loc`, so explicitly including `(SP)` is redundant.

Referring to Registers

For a register operand, you must use one of the register names of the appropriate kind for the instruction. The register names are case-sensitive. You also can use a symbolic name for an argument or local variable that was assigned to a register.

The general registers are `SP`, `r0` to `r31`, and `gpr0` to `gpr31`. The floating-point registers are `fp0` to `fp31` and `f0` to `f31`. The condition registers are `cr0` to `cr7`.

Referring to Labels

For a label operand, you can use the name of a label. For long branches (such as `b` and `bl` instructions) you can also use function names. For `bla` and `la` instructions, use absolute addresses.

For other branches, you must use the name of a label. For example,

- `b @3` — correct syntax for branching to a local label
- `b red` — correct syntax for branching to external function `red`
- `bl @3` — correct syntax for calling a local label
- `bl red` — correct syntax for calling external function `red`
- `bne red` — incorrect syntax; short branch outside function `red`

NOTE You cannot use local labels that have already been declared in other functions.

Using Variable Names as Memory Locations

Whenever an instruction, such as a load instruction, a store instruction, or `la`, requires a memory location, you can use a local or global variable name. You can modify local variable names with struct member references, class member references, array subscripts, or constant displacements. For example, all the local variable references in Listing 21.9 are valid.

Listing 21.9 Example of referring to variables stored in memory locations

```
asm void red(void){
  long myVar;
  long myArray[1];
  Rect myRectArray[3];
  fralloc
  lwz r3,myVar(SP)
  la  r3,myVar(SP)
  lwz r3,myRect.top
  lwz r3,myArray[2](SP)
  lwz r3,myRectArray[2].top
  lbz r3,myRectArray[2].top+1(SP)
  frfree
  blr
}
```

You can also use a register variable that is a pointer to a `struct` or `class` to access a member of the object, shown in Listing 21.10.

Listing 21.10 Example of referring to a struct or class member

```
void red(void){
  Rect q;
  register Rect *p = &q;
  asm {
  lwz r3,p->top;
  }
}
```

You can use the `@hiword` and `@loword` directives to access the high and low four bytes of 8 byte long longs and software floating-point doubles (Listing 21.11).

Listing 21.11 Example of referring to high and low words

```
long long gTheLongLong = 5;
asm void Red(void);
asm void Red(void)
{
  fralloc
  lwz r5, gTheLongLong@hiword
  lwz r6, gTheLongLong@loword
  frfree
  blr
}
```

Using Immediate Operands

For an immediate operand, you can use an integer or enum constant, `sizeof` expression, and any constant expression using any of the C dyadic and monadic arithmetic operators.

These expressions follow the same precedence and associativity rules as normal C expressions. The inline assembler carries out all arithmetic with 32-bit signed integers.

An immediate operand can also be a reference to a member of a struct or class type. You can use any struct or class name from a `typedef` statement, followed by any number of member references. This evaluates to the offset of the member from the start of the struct. For example:

```
lwz   r4,Rect.top(r3)
addi  r6,r6,Rect.left
```

As a side note, `la rD,d(rA)` is the same as `addi rD,rA,d`.

You also can use the top or bottom half-word of an immediate word value as an immediate operand by using one of the @ modifiers (Listing 21.12).

Listing 21.12 Example of referring to immediate operands

```
long gTheLong;
asm void red(void)
{
  fralloc
  lis r6, gTheLong@ha
  addi r6, r6, gTheLong@h
  lis r7, gTheLong@h
  ori r7, br7, gTheLong@l
  frfree
  blr
}
```

The access patterns are:

```
lis x,var@ha
la  x,var@l(x)
```

or

```
lis x,var@h
ori x,x,var@l
```

In this example, `la` is the simplified form of `addi` to load an address. The instruction `las` is similar to `la` but shifted. Refer to the Freescale Power Architecture manuals for more information.

Using `@ha` is preferred since you can write:

```
lis x,var@ha
lwz v,var@l(x)
```

You cannot do this with `@h` because it requires that you use the `ori` instruction.

Assembler Directives

This section describes some special assembler directives that the PowerPC built-in assembler accepts. These directives are:

- entry
- fralloc

- frfree
- machine
- nofralloc
- opword

entry

Defines an entry point into the current function.

`entry [ extern | static ]` *name*

Use the `extern` qualifier to declare a global entry point; use the `static` qualifier to declare a local entry point. If you leave out the qualifier, `extern` is assumed.

NOTE Inline-assembly directive `entry` can be used only with Function-level assembly code.

Listing 21.13 shows how to use the `entry` directive.

Listing 21.13 Using the entry directive

```
void __save_fpr_15(void);
void __save_fpr_16(void);
asm void __save_fpr_14(void)
{
  stfd   fp14,-144(SP)
  entry  __save_fpr_15
  stfd   fp15,-136(SP)
  entry  __save_fpr_16
  stfd  fp16,-128(SP)
  // ...
  blr
}
```

fralloc

Creates a stack frame for a function and reserves registers for local register variables.

`fralloc [` *number* `]`

You need to create a stack frame for a function if the function:

- calls other functions.

- uses more arguments than will fit in the designated parameters (`r3` — `r10`, `fp1` — `fp8`).
- declares local registers.
- declares non-registered parameters.

The `fralloc` directive has an optional argument *number* that lets you specify the size in bytes of the parameter area of the stack frame. The compiler creates a 0-byte parameter area. If your assembly language routine calls any function that requires more parameters than will fit in `r3` — `r10` and `fp1` — `fp8`, you must specify a larger amount.

frfree

Frees a function's stack frame and restores local register variables.

`frfree`

This directive frees the stack frame and restores the registers that `fralloc` reserved.

The `frfree` directive does not generate a `blr` instruction. If your function uses function-level inline assembly, you must explicitly terminate it with this instruction.

machine

Specifies the processor that the assembly language targets.

`machine` *number*

The value of *number* must be one of those listed in Table 21.1.

Table 21.1 CPU Identifiers

`505`	`509`	`555`	`56x`
`all`	`generic`		

If you use `generic`, the compiler supports the core instructions for the 603, 604, 740, and 750 processors. In addition, the compiler supports all optional instructions.

If you use `all`, the compiler recognizes assembly instructions for all core and optional instructions for all Power Architecture processors.

If you do not use the `machine` directive, the compiler uses the settings you selected from the **Processor** listbox of the **EPPC Processor** settings panel.

nofralloc

Specifies that the function will build a stack frame explicitly.

`nofralloc`

Use the `nofralloc` directive so that an inline assembly function does not build a stack frame. When you use `nofralloc`, if you have local variables, parameters or make function calls, you are responsible for creating and deleting your own stack frame. For an example of `nofralloc`, see the file `__start.c` in the directory:

InstallDir`\PowerPC_EABI_Support\Runtime\Src`

where *InstallDir* is the name of the directory on your host computer where you installed your CodeWarrior development tools.

opword

Inserts raw bytes into the object code.

`opword` *`value`*

This directive inserts *value* into the object code. For example

```
opword 0x7C0802A6
```

is equivalent to

```
mflr r0
```

The compiler does not check the validity of *value*; the compiler simply copies it into the object code that it generates.

Intrinsic Functions

Intrinsic functions are a mechanism you can use to get assembly language into your source code without using the `asm` keyword. Intrinsic functions are not part of the ISO/IEC C or C++ standards. They are an extension provided by the CodeWarrior compilers.

There is an intrinsic function for several common processor op-codes (instructions). Rather than using inline assembly syntax and specifying the op-code in an `asm` block, you call the intrinsic function that matches the op-code.

When the compiler encounters the intrinsic function call in your source code, it does not actually make a function call. The compiler substitutes the assembly instruction that matches your function call. As a result, no function call occurs in the final object code. The final code is the assembly language instructions that correspond to the intrinsic functions.

Low-Level Processor Synchronization

These functions perform low-level processor synchronization.

- `void __eieio(void)` — Enforce in-order execution of I/O
- `void __sync(void)` — Synchronize
- `void __isync(void)` — Instruction synchronize

For more information on these functions, see the instructions `eieio`, `sync`, and `isync` in *PowerPC Microprocessor Family: The Programming Environments* by Freescale.

Absolute Value Functions

These functions generate inline instructions that take the absolute value of a number.

- `int __abs(int)` — Absolute value of an integer
- `float __fabs(float)` — Absolute value of a float
- `float __fnabs(float)` — Negative absolute value of a float
- `long __labs(long)` — Absolute value of a long int

`__fabs(float)` and `__fnabs(float)` are not available if the **Hardware** option button is cleared in the **EPPC Processor** settings panel.

Byte-Reversing Functions

These functions generate inline instructions that can dramatically speed up certain code sequences, especially byte-reversal operations.

- `int __lhbrx(const void *, int)` — Load halfword byte; reverse indexed
- `int __lwbrx(const void *, int)` — Load word byte; reverse indexed
- `void __sthbrx(unsigned short, const void *, int)` — Store halfword byte; reverse indexed
- `void __stwbrx(unsigned int, const void *, int)` — Store word byte; reverse indexed

Setting the Floating-Point Environment

This function lets you change the Floating Point Status and Control Register (FPSCR). It sets the FPSCR to its argument and returns the original value of the FPSCR.

This function is not available if you select the **None** option button in the **EPPC Processor** settings panel.

```
float __setflm(float);
```

shows how to set and restore the FPSCR.

Listing 21.14 Example of setting the FPSCR

```
double old_fpscr;

/* Clear flag/exception/mode bits, save original settings */
oldfpscr = __setflm(0.0);
/* Peform some floating-point operations */

__setflm(old_fpscr); /* Restores the FPSCR */
```

Manipulating the Contents of a Variable or Register

These functions rotate the contents of a variable to the left:

- `int __rlwinm(int, int, int, int)` — Rotate left word (immediate), then AND with mask
- `int __rlwnm(int, int, int, int)` — Rotate left word, then AND with mask
- `int __rlwimi(int, int, int, int, int)` — Rotate Left word (immediate), then mask insert

The first argument to `__rlwimi` is overwritten. However, if the first parameter is a local variable allocated to a register, it is both an input and output parameter. For this reason, this intrinsic should always be written to put the result in the same variable as the first parameter as shown here:

```
ra = __rlwimi( ra, rs, sh, mb, me );
```

You can count the leading zeros in a register using this intrinsic:

```
int __cntlzw(int);
```

You can use inline assembly for a complete assembly language function, as well as individual assembly language statements.

Data Cache Manipulation

The intrinsics shown in Table 21.2 map directly to Power Architecture assembly instructions

Table 21.2 Data Cache Intrinsics

Intrinsic Prototype	Power Architecture Instruction
`void __dcbf(const void *, int);`	`dcbf`
`void __dcbt(const void *, int);`	`dcbt`
`void __dcbst(const void *, int);`	`dcbst`
`void __dcbtst(const void *, int);`	`dcbtst`
`void __dcbz(const void *, int);`	`dcbz`
`void __dcba(const void *, int);`	`dcba`

Math Functions

Table 21.3 lists intrinsic functions for mathematical operations.

Table 21.3 Math Intrinsics

Intrinsic Prototype	Power Architecture Instruction
`int __mulhw(int, int);`	`mulhw`
`uint __mulhwu(uint, uint);`	`mulhwu`
`double __fmadd(double, double, double);`	`fmadd`
`double __fmsub(double, double, double);`	`fmsub`
`double __fnmadd(double, double, double);`	`fnmadd`
`double __fnmsub(double, double, double);`	`fnmsub`
`float __fmadds(float, float, float);`	`fmadds`
`float __fmsubs(float, float, float);`	`fmsubs`
`float __fnmadds(float, float, float);`	`fnmadds`
`float __fnmsubs(float, float, float);`	`fnmsubs`
`double __mffs(void);`	`mffs`

Table 21.3 Math Intrinsics (*continued*)

`float __fabsf(float);`	`fabsf`
`float __fnabsf(float);`	`fnabsf`

Buffer Manipulation

Some intrinsics allow control over areas of memory, so you can manipulate memory blocks.

`void *__alloca(ulong);`

`__alloca` implements `alloca()` in the compiler.
`char *__strcpy(char *, const char *);`

`__strcpy()` detects copies of constant size and calls `__memcpy()`. This intrinsic requires that a `__strcpy` function be implemented because if the string is not a constant it will call `__strcpy` to do the copy.

`void *__memcpy(void *, const void *, size_t);`

`__memcpy()` provides access to the block move in the code generator to do the block move inline.

22 Power Architecture Code Generation

This chapter describes the conventions that the C/C++ compiler and linker follow to generate object code for Power Architecture processors, the data types that the compiler recognizes, and how to specify to the compiler the byte-alignment of data in object code.

- ABI Conformance
- Data Representation
- Data Addressing
- Aligning Data
- Small Data Area PIC/PID Support
- Variable Length Encoding
- Building a ROM Image
- Specifying Jump Table Location

ABI Conformance

The CodeWarrior compiler for Power Architecture processors follows the application binary interface (ABI) specified by *PowerPC Embedded Binary Interface, 32-Bit Implementation*.

Data Representation

The compiler recognizes ISO standard data types and some Power Architecture-specific types:

- Boolean Type
- Character Types
- Integer Types
- Floating-Point
- AltiVec™ Data Types

Boolean Type

Table 22.1 lists the name, size, and range of the boolean data type. The compiler recognizes this data type when compiling C99 (ISO/IEC 9899-1999) source code.

Table 22.1 C99 boolean data type

This type	has this size	and holds this range of values
`_Bool`	8 bits when pragma `uchar_bool` is `on`, 32 bits when pragma `uchar_bool` is `off`	`0` (“false”) and `1` (“true”)

Table 22.2 lists the name, size, and range of the C++ boolean data type. The C++ compiler does not recognize the C99 `_Bool` type.

Table 22.2 Boolean data type

This type	has this size	and holds this range of values
`bool`	8 bits when pragma `uchar_bool` is `on`, 32 bits when pragma `uchar_bool` is `off`	`true`, `false`

Character Types

Table 22.3 lists the name, size, and range of the character data types.

Table 22.3 Character data types

This type	has this size	and holds this range of values
`char`	8 bits	either `-128` to `127` or `0` to `255`
`unsigned char`	8 bits	`0` to `255`
`signed char`	8 bits	`-128` to `127`

Table 22.4 lists the name, size, and range of the C++ `wchar_t` data types.

Table 22.4 Character data types

This type	has this size	and holds this range of values
`wchar_t`	16 bits	either -32768 to 32767 or 0 to 65535
`unsigned wchar_t`	16 bits	0 to 65535
`signed wchar_t`	16 bits	-32768 to 32767

The pragma `unsigned_char` controls whether or not the compiler treats the `wchar_t` and `char` types as signed or unsigned.

Integer Types

Table 22.5 lists the name, size, and range of the integer data types.

Table 22.5 Integer data type

This type	has this size	and holds this range of values
`short`	16 bits	-32,768 to 32,767
`unsigned short`	16 bits	0 to 65,535
`int`	32 bits	-2,147,483,648 to 2,147,483,647
`unsigned int`	32 bits	0 to 4,294,967,295
`long`	32 bits	-2,147,483,648 to 2,147,483,647
`unsigned long`	32 bits	0 to 4,294,967,295
`long long`	64 bits	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807
`unsigned long long`	64 bits	0 to 18,446,744,073,709,551,615

The compiler recognizes the `long long` data type when pragma `longlong` is on or when compiling C99 source code (ISO/IEC 9899-1999 standard).

Floating-Point

Table 22.6 lists the floating point data types

Table 22.6 Power Architecture Floating Point Types

Type	Size	Range
`float`	32 bits	1.17549e-38 to 3.40282e+38
`double`	64 bits	2.22507e-308 to 1.79769e+308
`long double`	64 bits	2.22507e-308 to 1.79769e+308

AltiVec™ Data Types

There are `vector` data types for use in writing AltiVec-specific code. (See Table 22.7). All the types are a constant size, 16 bytes (128 bits). This is due to the AltiVec programming model which is optimized for quantities of this size.

Table 22.7 AltiVec Vector Data Types

Vector Data Type	Contents	Possible Values
`vector unsigned char`	16 unsigned char	0 to 255
`vector signed char`	16 signed char	-128 to 127
`vector bool char`	16 unsigned char	0 (“false”), 1 (“true”)
`vector unsigned short [int]`	8 unsigned short	0 to 65535
`vector signed short [int]`	8 signed short	-32768 to 32767
`vector bool short [int]`	8 unsigned short	0 (“false”), 1 (“true”)
`vector unsigned long [int]`	4 unsigned int	0 to 2^{32} - 1
`vector signed long [int]`	4 signed int	-2^{31} to 2^{31}-1
`vector bool long [int]`	4 unsigned int	0 (“false”), 1 (“true”)
`vector float`	4 float	any IEEE-754 value
`vector pixel`	8 unsigned short	1/5/5/5 pixel

In Table 22.7, the `[int]` portion of the Vector Data Type is optional.

There are two additional keywords besides `pixel` and `vector`, `__pixel` and `__vector`.

The `bool` keyword is not a reserved word in C unless it is used as an AltiVec vector data type.

Data Addressing

In absolute addressing, the compiler generates two instructions to fetch the address of a variable. For example the compiler translates Listing 22.1 into the instructions in Listing 22.2.

Listing 22.1 Source Code

```
int red;
int redsky;
void sky()
{
  red = 1;
  redsky = 2;
}
```

Listing 22.2 Generated Code

```
li   r3,1
lis  r4,red@ha
addi r4,r4,red@l
stw  r3,0(r4)
li   r5,2
lis  r6,redsky@ha
addi r6,r6,redsky@l
stw  r5,0(r6)
```

Each variable access takes two instructions and a total of four bytes to make a simple assignment. If you set the small data threshold to be at least the size of an `int` data type, the compiler generates instructions to fetch variables with one instruction (Listing 22.3).

Listing 22.3 Fetching variables with one instruction

```
li   r3,1
stw  r3,red
li   r4,2
stw  r4,redsky
```

Because small data sections are limited in size you might not be able to put all of your application data into the small data and small data2 sections. We recommend that you make the threshold as high as possible until the linker reports that you have exceeded the size of the section.

If you do exceed the available small data space, consider using pooled data.

Because the linker can not deadstrip unused pooled data, you should:

1. Check the **Generate Link Map** and **List Unused Objects** checkboxes in the CodeWarrior IDE's **EPPC Linker** panel.
2. Link and examine the map for data objects that are reported unused.
3. Delete or comment out those used definitions in your source.
4. Check the **Pool Data** checkbox.

The code in Listing 22.4 has a zero small data threshold.

Listing 22.4 Zero Small Data Threshold

```
lis    r3,...bss.0@ha
addi   r3,r3,...bss.0@l
li     r0,1
stw    r0,0(r3)
li     r0,2
stw    r0,4(r3)
```

When pooled data is implemented, the first used variable of either the `.data`, `.bss` or `.rodata` section gets a two-instruction fetch of the first variable in that section. Subsequent fetches in that function use the register containing the already-loaded section address with a calculated offset.

NOTE You can access small data in assembly files with the two-instruction fetch used with large data, because any data on your board can be accessed as if it were large data. The opposite is not true; large data can never be accessed with small data relocations (the linker issues an error if you try to do so). External declarations of empty arrays (for example, `extern int red [];`) are always treated as if they were large data. If you know that the size of the array fits into a small data section, specify the size in the brackets.

Aligning Data

This section contains these topics:

- Alignment Attribute Syntax

- Aligning a Variable Declaration
- Alignment in a Structure Definition
- Typedef Declaration
- Structure Member
- Bitfields

Alignment Attribute Syntax

Use `__attribute__ ((aligned(...)))` directive to specify to the compiler on what memory boundary to store data objects. This directive specifies which multiple of bytes to store an object.

The format of this directive is

```
__attribute__ ((aligned(x))
```

where *x* is a decimal number of a power of 2 from 1 to 4096.

Aligning a Variable Declaration

Use the alignment attribute to specify a variable's alignment. For example, the following variable declaration aligns `V1` on a 16-byte boundary.

```
int V1[4] __attribute__ ((aligned (16)));
```

The following variable declaration aligns `V2` on a 2-byte boundary.

```
int V2[4] __attribute__ ((aligned (2)));
```

Alignment in a Structure Definition

Use the alignment attribute to specify how instances of a structure should be aligned. You must specify a minimum alignment of at least 4 bytes for structures. Specifying a lower number might cause alignment exceptions at runtime.

For example, this definition aligns all definitions of `struct S1` on an 8-byte boundary.

```
struct S1 { short f[3]; }
    __attribute__ ((aligned (8)));
struct S1 s1;
```

The following definition aligns all definitions of `struct S2` on a 4-byte boundary.

```
struct S2 { short f[3]; }
    __attribute__ ((aligned (1)));
struct S2 s2;
```

Structure Member

Use the alignment attribute to specify how to align a member in a structure.

For example, the following structure member definition aligns all definitions of `struct S3` on an 8-byte boundary, where `a` is at offset 0 and `b` is at offset 8.

```
struct S3 {
   char a;
   int b __attribute__ ((aligned (8)));
};
struct S3 s3;
```

The following struct member definition aligns all definitions of `struct S4` on a 4-byte boundary, where `a` is at offset 0 and `b` is at offset 4.

```
struct S4 {
   char a;
   int b __attribute__ ((aligned (2)));
};
struct S4 s4;
```

NOTE Specifying `__attribute__ ((aligned (2)))` does not affect the alignment of `S4` because 2 is less than the natural alignment of `int`.

Typedef Declaration

Use the alignment attribute to specify how objects of a specific type should be aligned.

For example, the following typedef declaration aligns all definitions of `T1` on an 8-byte boundary.

```
typedef int T1 __attribute__ ((aligned (8)));
T1 t1;
```

The following typedef declaration aligns all definitions of `T2` on an 1-byte boundary.

```
typedef int T2 __attribute__ ((aligned (1)));
T2 t2;
```

Bitfields

If your program's structure has bitfields and the Power Architecture alignment does not give you as small a structure as you desire, double-check that you are specifying the smallest integer size for your bitfields.

For example, Listing 22.5 would be smaller if it were written as shown in Listing 22.6.

Listing 22.5 Before

```
typedef struct red {
   unsigned a: 1;
   unsigned b: 1;
   unsigned c: 1;
} red;
```

Listing 22.6 After

```
typedef struct red {
   unsigned char a: 1;
   unsigned char b: 1;
   unsigned char c: 1;
} red;
```

Small Data Area PIC/PID Support

The basic requirement for position independent code and data in the small data area is, at runtime, maintaining the link time address relationships between the startup code (`.init`) and the `.sdata` and `.sdata2` segments. For example, if the link time addresses are:

`.init = 0x00002000`

`.sdata2 = 0x00003000`

`.sdata = 0x00004000`

but `.init` somehow is executed at 0x00002500, then those link time addresses must all increment by 0x00000500 for their runtime addresses.

Any segment that does not maintain the address relationship at runtime is considered external and must be addressed with absolute addresses. Segments that do maintain their link time address relationship at runtime are considered internal and must be addressed with PC-relative and SDA-relative addressing.

- Internal and External Segments and References
- PIC/PID Linker Command File Directives
- Linker-defined Symbols
- Uses for SDA PIC/PID
- Building an SDA PIC/PID Application
- Internal and External Addressing Modes

Internal and External Segments and References

The linker determines at link time whether code and data segments are external or internal. Internal segments reference their data as far or near offsets of the small data registers `r2` and `r13`. Their code references are normally PC-relative, but if far code references are needed, they also use offsets of the small data registers.

Internal segments can also reference code and data in other internal segments with the same addressing that they would use for their own code and data.

By default, the linker considers all segments in your application to be internal with the exception of segments that are at absolute addresses. Segments with names such as `.abs.xxxxxxxx`, where `xxxxxxxx` is a hex address, are considered external.

External segments reference their data with absolute addressing and code references within the segment may be either PC-relative or absolute. Any other segment must use absolute references to reference code or data in external segments. External segments must reference an internal segment with small data registers for code and data.

Related to external segments are external symbol references. These are symbols, usually linker-generated, that are determined not to be within any segment in your application. They are referenced with absolute addressing. All symbols in an external segment are considered to be external symbol references.

PIC/PID Linker Command File Directives

A few linker command file directives override PIC/PID related linker default settings:

- `MEMORY`
- `INTERNAL_SYMBOL`
- `EXTERNAL_SYMBOL`

Linker-defined Symbols

The linker-generated start and end symbols that are automatically generated for loadable segments are internal if they are addresses into internal segments, and external if they are for external segments. All other linker defined symbols you create in a LCF are considered external unless you redefine them with `INTERNAL_SYMBOL`. The linker also defines some linker defined symbols for its own use (Table 22.8).

Table 22.8 Linker-defined Symbols

Symbol Name	Value	Description
`_stack_addr`	top of the stack -	External. Comes from settings panel settings.
`_stack_end`	bottom of the stack	External. Comes from settings panel settings.
`_heap_addr`	bottom of the heap	External. Comes from settings panel settings.
`_heap_end`	top of the heap	External. Comes from settings panel settings.
`_SDA_BASE_`	`.sdata + 0x00008000`	Internal per EABI requirement. May not be redefined.
`_SDA2_BASE_`	`.sdata2 + 0x00008000`	Internal per EABI requirement. May not be redefined.
`_ABS_SDA_BASE_`	`.sdata + 0x00008000`	External version of _SDA_BASE_ that can be used as an absolute. May not be redefined.
`_ABS_SDA2_BASE_`	`.sdata2 + 0x00008000`	External version of _SDA2_BASE_ that can be used as an absolute. May not be redefined.
`_nbfunctions`	number of functions in program	Deprecated. External. This is a number, not an address. May not be redefined.
`SIZEOF_HEADERS`	size of the segment headers	External. This is a number, not an address. May not be redefined.

NOTE The symbols `_SDA_BASE_` and `_SDA2_BASE` are not accessible until the small data registers are properly initialized before being accessible. The symbols `_ABS_SDA_BASE` and `_ABS_SDA2_BASE` allow you to access those pointers as absolutes addresses, as it is difficult to initialize those pointers without accessing them as absolute addresses.

NOTE The stack and heap linker generated symbols are external. It may be more practical in a SDA PIC/PID application to make the heap and stack be contiguous with an internal segment and define them as internal.

Uses for SDA PIC/PID

The PIC/PID runtime can be used for different scenarios:

1. All code and data segments are internal. The simplest case would be for all segments to use the same `MEMORY` directive and to have all of the `.bss` type segments at the end. In such a simple case, the application could be converted to a binary file and linked into another application which could copy it to RAM and jump to its entry point.
2. All of the essential segments are internal and therefore moveable. But, there may be some external segments which are absolute. This situation is probably difficult to test but we can download the entire application to the chip and at least debug it at its link time addresses.
3. There are internal and external segments, but the application is linked as a ROM image (the application does not need to be flashed to ROM, however). It is possible to change the ROM Image Address to be an address into RAM and have the debugger download the image to the RAM address. Alternatively, we could have the ROM image converted to a binary file and linked into another application as in 1, above. The structures used in `__init_data()`, `_rom_copy_info` and `__bss_init_info`, have been modified for SDA PIC/PID to have an extra field which tells the runtime where the segment is internal or external so that the internal segments are copied to position-relative addresses and the external segments copied to absolute addresses.

Building an SDA PIC/PID Application

To build a SDA PIC/PID application, select **SDA PIC/PID** in the **ABI** list box in the CodeWarrior IDE's **EPPC Target** target preferences panel. The compiler defines a simple variable that we can use to guard PIC/PID source.

```
#if __option(sda_pic_pid) // is true if we have chosen SDA
PIC/PID ABI
```

At link-time, the linker generates a table used for the runtime files `__ppc_eabi_init.cpp` and `__ppc_eabi_init.c`.

If our application contains absolute addressing relocations, we will receive linker warnings telling us that those relocations may cause a problem. To resolve these warnings, either:

- change the **Code Model** listbox in the CodeWarrior IDE's **EPPC Target** target preferences panel to be **SDA Based PIC/PID Addressing** for all of our sources and libraries
- check the **Tune Relocations** checkbox in the **EPPC Target** target preferences panel. This new option is only available for the EABI and SDA PIC/PID ABIs. For EABI, it changes 14-bit branch relocations to 24-bit branch relocations, but only if they can not reach the calling site from the original relocation.

  For SDA PIC/PID, this option changes absolute-addressed references of data from code to use a small data register instead of `r0` and changes absolute code-to-code references to use the PC-relative relocations.

Linking Assembly Files

It is always possible to link in an assembly file that does not behave in a standard way. For example, taking the address of a variable with:

```
addis   rx,r0,object@h
ori     rx,rx,objec@l
```

generally can not be converted by the linker to SDA PIC/PID Addressing and the linker will warn us if it finds an occurrence.

The following will work with Absolute Addressing as well as allow the linker to convert the instructions to SDA PIC/PID Addressing:

```
addis   rx,r0,object@ha
addi    rx,rx,objec@l
```

Another possible problem may arise if we put constant initialized pointers into a read-only section, thereby not letting the runtime convert the addresses.

Modifications to the Section Pragma

The pragma `#pragma section` has been modified to accept `far_sda_rel` for the `data_mode` and `code_mode` options, even if we are not using Code Model SDA Based PIC/PID Addressing. If we omit these options, the compiler uses the Code Model to determine the appropriate modes.

- Absolute Addressing

  ```
  data_mode = far_abs
  code_mode = pc_rel
  ```

- SDA Based PIC/PID Addressing

  ```
  data_mode = far_sda_rel
  code_mode = pc_rel
  ```

Internal and External Addressing Modes

An address mode is applied to a memory segment as a part of the ROM image or at the executing (or logical) address of the segment. Following address modes can be applied to a memory segment:

- Internal—the segment executes from an address not specified at link time.
- External— the segment must execute from the address specified at the link time.

Consider an example where the segment `.foo` is a part of ROM Image and will be copied to a RAM location. The link time addresses are:

- ROM = 0x00100000
- RAM = 0x00002000

NOTE Both the link time addresses can be external or internal.

Also assume that the real time (physical) ROM address is 0x00200000 instead of the link time specified address 0x00100000. Table 22.9 lists the possible address mode scenarios.

Table 22.9 Possible addr_mode Scenarios

Scenario	ROM addr_mode	RAM addr_mode	Description
A	internal	external	Runtime correctly figures out that the ROM address is 0x00200000 and copies it to 0x00002000
B	internal	internal	Runtime correctly figures out that the ROM address is 0x00200000 and copies it to 0x00102000
C	external	external	Runtime incorrectly assumes that the ROM address is 0x00100000 and copies it to 0x00002000
D	external	internal	Runtime incorrectly assumes that the ROM address is 0x00100000 and copies it to 0x00102000

In the above possible scenarios only A and B are correct. The difference between scenario A and B is that in A, the executing (logical) address of `.foo` is absolute and that in B, the executing (logical) address of `.foo` is relative.

Scenario C and D are possible if `.foo` is flashed to ROM at its correct ROM address and all other segments are at an offset from their link time ROM addresses.

NOTE `.init` segment determines the correct address of an application. If `.init` is at its link time ROM address, then all the segments in the application will be treated as external.

Specifying ROM addr_mode

Use the following directives to specify ROM addr_mode:

- `LOAD`— To specify an external ROM addr_mode.
- `INTERNAL_LOAD`—To specify an internal ROM addr_mode.

By default the ROM addresses are external.

Specifying RAM addr_mode

Use `MEMORY` directive and any of the following parameters to specify the RAM addr_mode.

`addr_mode = external`—To specify an external RAM addr_mode.

`addr_mode = internal`—To specify an internal RAM addr_mode.

By default the RAM addresses are internal.

For example, `RAM : org = 0x000e0000, addr_mode = external` will make sections defined in the RAM external.

NOTE `addr_mode` is ignored if **SDA PIC/PID** in the **ABI** list box in the CodeWarrior IDE's **EPPC Target** target preferences panel is not selected.

Variable Length Encoding

The Variable Length Encoding (VLE) instruction set architecture is an extension to the instruction set specified in Freescale Semiconductor's Book E Implementation Standard (EIS) for Power Architecture processors. This instruction set adds a few identically operating counterparts to the regular EIS instruction set. But where regular EIS instructions occupy 32 bits and must be aligned to 32-bit boundaries, VLE instructions are either 16 or 32 bits long and can be aligned to 16-bit boundaries. This extra flexibility in instruction encoding and alignment allows the compiler and linker to greatly compress the size of runtime object code with only a small penalty in execution performance.

These topics describe how and when to configure the build tools to generate VLE object code:

- Processors With VLE Capability
- Compiling VLE Instructions
- Assembling VLE Instructions
- Linking VLE Object Code

Processors With VLE Capability

The VLE (Variable Length Encoding) instruction set is an extension to the instruction set specified in the Freescale Book E Implementation Standard (EIS). Not all Power Architecture processors have VLE capability. Refer to the manufacturer’s documentation for the processor you are targeting. For information on the Book E and VLE programming models, see *EREF: A Programmer’s Reference Manual for Freescale Book E Processors*, published by Freescale Semiconductor.

Compiling VLE Instructions

Table 22.10 shows how to control VLE (Variable Length Encoding) code generation.

Table 22.10 Controlling VLE code generation

To control this option from here...	use this setting
CodeWarrior IDE	Select **Zen** in the **Processor** drop-down list box of the **EPPC Processor** settings panel, then check **Generate VLE Instructions** in **e500/Zen Options** panel.

C/C++ source code	`__declspec(vle_on)`
	`__declspec(vle_off)`

command line	`-vle`

Assembling VLE Instructions

The mnemonics for VLE (Variable Length Encoding) instructions begin with “se_” or “e_”. The compiler’s inline assembler recognizes these mnemonics when the compiler is configured to generate VLE object code.

Only a subset of EIS instructions have equivalent VLE instructions. To save you time and effort, the inline assembler can convert regular EIS instructions to equivalent VLE instructions automatically. In other words, the inline assembler can generate VLE object code from inline assembly statements that use only regular mnemonics. Table 22.11 shows how to control VLE code generation for inline assembly statements.

Table 22.11 Controlling VLE inline assembly

To control this option from here...	use this setting
CodeWarrior IDE	Select **Zen** in the **Processor** drop-down list box of the **EPPC Processor** settings panel, then check **Translate PPC Asm to VLE ASM** in **e500/Zen Options** panel.
command line	`-ppc_asm_to_vle`

The stand-alone assembler also recognizes and generates VLE instructions. Table 22.12 shows how to control VLE code generation with the standalone assembler.

Table 22.12 Controlling VLE code generation for the standalone assembler

To control this option from here...	use this setting
CodeWarrior IDE	Select **Zen** in the **Processor** drop-down list box of the **EPPC Processor** settings panel, then check **Generate VLE Instructions** in **e500/Zen Options** panel.
command line	`-vle`

To specify that a section containing executable code should be executed in the processor's VLE mode, use the `text_vle` identifier with the `.section` directive. Listing 22.7 shows examples.

Listing 22.7 Examples of specifying VLE sections in standalone assembly

```
.section .text_vle # Section name is .text_vle
.section .text,text_vle # Section name is .text
.section .littletext,text_vle # Section name is .littletext
```

Unlike the inline assembler, the standalone assembler does not offer the option to convert regular instruction mnemonics to VLE instructions. To perform this conversion automatically, copy and paste standalone assembly source code into a C or C++ source file, shown in Listing 22.8.

Listing 22.8 Using automatic VLE instruction conversion

```
extern asm void my_func(void)
{
    nofralloc /* No stack frame. */
    /* Paste standalone assembly source code here. */
}
```

Linking VLE Object Code

A processor capable of executing VLE (Variable Length Encoding) instructions must use separate memory pages for VLE and regular instructions. The compiler and linker ensure this separation by placing executable code that uses VLE instructions and regular instructions in separate object code sections.

To maintain this separation in your own linker command file, specify output sections for VLE and regular instructions. Listing 22.9 shows an example. This linker control file specifies that output sections named `.init_vle` and `.text_vle` should only contain object code that the compiler has tagged with `VLECODE`.

Listing 22.9 Separating VLE and regular object code in the linker's output file

```
.init : { } > code
.init_vle (VLECODE) : {
    *(.init)
    *(.init_vle)
} > code

.text : { } > code
.text_vle (VLECODE) : {
*(.text)
*(.text_vle)
} > code
```

To save memory space, the linker compresses VLE object code by shortening the gaps between functions. A VLE function must meet these criteria to be re-aligned:

- The VLE function is referred to only by other VLE functions.

  The linker will not re-align a function if it is referred to by a non-VLE function.
- The VLE function's alignment is 4 bytes.

The linker will not re-align a function if the compiler's function alignment settings specify an explicit alignment value.

- The object code was generated by the CodeWarrior compiler.

Building a ROM Image

The CodeWarrior compiler and linker can generate a program image that may be stored in and started from ROM (read-only memory). This section uses the term *ROM* to mean any kind of persistent main storage, including ROM and flash memory.

To create an image for read-only memory, you must configure the compiler and linker:

- Linking a ROM Image
- ROM Image Addresses
- Specifying A Single ROM Block
- Specifying Several ROM Blocks
- Specifying Jump Table Location
- Specifying Constant Data Location

Linking a ROM Image

Table 22.13 compares the differences between the linker's default RAM image layout and how you should configure your program for loading into and running from ROM.

Table 22.13 Comparing RAM and ROM images

RAM image properties	ROM image properties
The S record file contains executable code, constants, and initialization values for data.	One or more memory areas defined in the linker's `.lcf` file specifies where store the program image in the target system's memory map.
Executable code, constants, and data are loaded for execution by the debugger or the program loader.	Initialization values for data are copied from the ROM image to RAM at program startup. Executable code and constant data may also be copied to RAM to improve performance (while requiring more RAM space).

The linker's output ELF file for a ROM image contains a segment named `.PPC.EMB.seginfo`. This segment describes which segments in the image will be copied from ROM to RAM at runtime. The linker uses this non-loadable segment to generate a data structure named `_rom_copy_info`. At startup, the program uses the `_rom_copy_info` structure to determine which segments to move from ROM to RAM.

Listing 22.10 shows the part of an example disassembly that lists the contents of segment `.PPC.EMB.seginfo`. When `is_rom_copy` is set to 1, the corresponding segment is copied from ROM to its final destination during startup. In this example, these sections will be copied from ROM to RAM at startup:

`.bss, .data, .sdata, .sbss, .sdata2.`

Listing 22.10 Example of segments to copy to RAM at startup

```
entry  is_rom_copy  name            ram index
[   0] 0            .abs.00010000   0
[   1] 0            .reset          0
[   2] 0            .init           0
[   3] 0            .text           0
[   4] 0            .rodata         0
[   5] 0            .dtors          0
[   6] 1            .bss            7
[   7] 0            .bss            0
[   8] 1            .data           9
[   9] 0            .data           0
[  10] 1            .sdata         11
[  11] 0            .sdata          0
[  12] 1            .sbss          13
[  13] 0            .sbss           0
[  14] 1            .sdata2        15
[  15] 0            .sdata2         0
```

ROM Image Addresses

The program ROM image is usually the whole image of the program. The ROM image allocates RAM space for its variables, and optionally for its executable code, at application startup. A ROM image is defined by these addresses:

- ROM image address: is the address where you want the ROM image to be allocated.

  Usually it is the start address of one of the memory blocks defined in the linker `.lcf` file.
- RAM buffer address: specifies the address in RAM that is to be used as a buffer for the flash image programmer.

To specify these addresses in the CodeWarrior IDE, turn on **Generate ROM Image** in the **EPPC Linker** settings panel, then enter addresses in the **RAM Buffer Address** and **ROM Image Address** fields.

Specifying A Single ROM Block

When specifying a single ROM memory block in a `.lcf` file, the start address of this memory block can be used as ROM image address. All executable code and constant sections will be allocated in ROM and all variables initialization values will be copied from ROM to RAM during startup.

Listing 22.11 shows an example `.lcf` file for a single ROM block.

Listing 22.11 Configuring a linker file for a ROM image

```
MEMORY {
    ram : org = 0x00c02000
    rom : org = 0x00000000 // desired ROM address (boot
                           // address for 555)
}

SECTIONS {
    .reset : {} > rom
    .init  : {} > rom
    GROUP : {
        .text (TEXT) ALIGN(0x1000) : {}
        .rodata (CONST) : {
            *(.rdata)
            *(.rodata)
        }
        .ctors : {}
        .dtors : {}
        extab : {}
        extabindex : {}
    } > rom // for ROM images, this can be 'rom' if you want
            // to execute in ROM or 'ram' if you want to
            // execute in RAM
    GROUP : {
        .data : {}
        .sdata : {}
        .sbss : {}
        .sdata2 : {}
        .sbss2 : {}
        .bss : {}
        .PPC.EMB.sdata0 : {}
        .PPC.EMB.sbss0 : {}
    } > ram
}
```

Specifying Several ROM Blocks

To specify several ROM blocks in a `.lcf` file, the start address of the main memory block must be the ROM image address.

To prevent all executable code or constants allocated in other ROM blocks to be copied during startup, use the `LOAD` linker directive. To prevent a specific executable code or constant section from being copied to its runtime RAM destination, specify the final destination address in the `LOAD` directive.

Listing 22.12 shows an example `.lcf` file that specifies several ROM blocks.

Listing 22.12 Configuring linker file for an image with several ROM blocks

```
MEMORY
 {
  APPL_INT_VECT       : org= 0x00010000, len= 0x000000FF
      //  If org is changed, make sure to adjust start address in
      //  .applexctbl LOAD (0x00010000): {} > APPL_INT_VECT
      //  accordingly
  CST_DATA            : org= 0x00010100, len= 0x000000FF
  APPL_CODE_FLASH     : org= 0x00010200, len= 0x000EFE00
      //  APPL_CODE_FLASH= int. flash area for application
      //  external RAM
  EXT_RAM_A           : org= 0x00800000, len= 0x00100000
}

SECTIONS {
       .applexctbl LOAD (0x0001000): {} > APPL_INT_VECT
       .syscall: {} > APPL_CODE_FLASH
       .reset : {} > APPL_CODE_FLASH
       .init: {} > APPL_CODE_FLASH
       GROUP : {
              .text (TEXT) : {}
              .rodata (CONST) : {
                     *(.rdata)
                     *(.rodata)
              }
              .ctors : {}
              .dtors   : {}
              extab    : {}
              extabindex : {}

       } > APPL_CODE_FLASH
       GROUP : {
                 .bss    : {}
                 .data   : {}
                 .sdata  : {}
                 .sbss   : {}
                 .sdata2  : {}
                 .sbss2   : {}
                 .PPC.EMB.sdata0: {}
                 .PPC.EMB.sbss0 : {}
             } > EXT_RAM_A   //DPTRAM_AB
       GROUP:{
                .CstData LOAD (0x00010100): {}
             } > CST_DATA
}
```

If several sections must be allocated in one of the secondary memory areas, use the linker's ROMADDR directive to evaluate the final destination address of the sections. Listing 22.13 shows an example.

Listing 22.13 Placing several sections in a secondary memory area

```
  .applexctbl LOAD (0x0010000): {} > APPL_INT_VECT
  .syscall LOAD (ROMADDR(.applexctbl)+SIZEOF(.applexctbl)):{}
      > APPL_INT_VECT
```

If the program contains an absolute code section, a section which contains object code that must not be copied at startup, the section must also be specified in the `.lcf` file with the LOAD directive. Listing 22.14 shows example C source code that generates an interrupt service routine that must be placed at a specific address at runtime. Listing 22.15 shows the linker directives that ensure that this routine's object code will be loaded at a specific address at runtime.

Listing 22.14 Absolute code example

```
#pragma push
#pragma section code_type ".abs.00010000" code_mode=pc_rel
asm void  _ISRVectorTable(void)
{
      b InterruptHandler
      nop
      nop
      b InterruptHandler
}
#pragma pop
```

Listing 22.15 Linker commands for absolute code in ROM

```
MEMORY
 {
  //internal Flash
  APPL_INT_VECT      : org= 0x00010000, len= 0x000000FF;
      //  If org is changed, make sure to adjust start
      //  address in .abs.00010000 LOAD (0x00010000): {} >
      //  APPL_INT_VECT accordingly

  // ...

}

SECTIONS {

    .abs.00010000 LOAD (0x00010000): {} > APPL_INT_VECT
    <…>
}
```

Specifying Jump Table Location

By default the CodeWarrior compiler and linker allocate jump tables for `switch` statements in RAM. When the application executes from ROM, it is sometimes better to have the switch table allocated in ROM to reduce RAM requirements.

To tell the compiler and linker to place jump tables in an object code section that will be placed in ROM, use this directive in your C or C++ source code:

```
#pragma read_only_switch_tables on
```

Alternately, to tell the compiler to generate a branch tree in executable code instead of a jump table in a data section, use this directive in C or C++:

```
#pragma switch_tables off
```

Specifying Constant Data Location

By default, the CodeWarrior compiler allocates all constant values of a size greater than 8 bytes in the `.rodata` section. There are two solutions for storing constants with sizes smaller than 8 bytes to be allocated in this section:

Solution 1: Define the variable in section `.rodata` using the `__declspec(section)` directive in C or C++ source code. Listing 22.16 shows an example.

Listing 22.16 Using __declspec(section) to store small constants in .rodata

```
#define SMALL_ROM_CONST __declspec(section ".rodata")

SMALL_ROM_CONST const unsigned int MyInt2 = 0x4534
```

Solution 2: Enter 0 in the **Small Data2** option in the CodeWarrior IDE's **EPPC Target** settings panel.

Embedded C++

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

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

Activating EC++

Table 22.14 shows how to control Embedded C++ conformance.

Table 22.14 Controlling Embedded C++ conformance

To control this option from here...	use this setting
CodeWarrior IDE	**EC++ Compatibility Mode** in the **C/C++ Language Settings** panel
source code	`#pragma ecplusplus`
command line	`-dialect ec++`

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

Differences Between ISO C++ and EC++

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

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

Templates

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

Libraries

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

File Operations

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

Localization

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

Exception Handling

The EC++ proposal does not support exception handling.

Unsupported Language Features

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

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

EC++ Specifications

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

- Language Related Issues
- Library-Related Issues

Language Related Issues

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

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

Library-Related Issues

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

23

Libraries and Support Code

CodeWarrior software includes libraries and support files you can add to your project. This chapter describes these libraries and how to choose among them:

- Main Standard Libraries (MSL): ISO/IEC-standard C and C++ libraries
- runtime libraries: support for higher-level C and C++ language features
- board initialization: low-level startup routines

The sections of this chapter are:

- Main Standard Libraries
- Third Party Standard Libraries
- Embedded Warrior Library
- Runtime Libraries
- Board Initialization Code

Main Standard Libraries

This section explains how to use the Power Architecture version of the Main Standard Libraries (MSL).

- Using the Main Standard Libraries
- Choosing an MSL Library
- Using Console I/O
- Allocating Additional Heap Space

For more information refer to the *MSL C Reference* and the *MSL C++ Reference*.

Using the Main Standard Libraries

The Main Standard Libraries (MSL) are a complete, configurable set of C and C++ standard libraries. These libraries also include MSL Extras, which extends the standard library and adds compatibility with common UNIX libraries. All of the source files required to build MSL are included in your CodeWarrior product, along with project files for different MSL configurations.

To use the MSL library, you must also use a runtime library. To support custom hardware capabilities, such as a new memory configuration, make changes to the runtime libraries instead of the MSL library's source files. Then, if necessary, reconfigure and recompile the MSL library. Refer to the *MSL C Reference* or *MSL C++ Reference* for more information.

Choosing an MSL Library

If your program uses features in the MSL libraries, you must choose a configuration that matches your software and hardware requirements.

The filenames of the configurations of MSL libraries follow a naming convention to describe each library's capabilities and features. Table 23.1 lists the types of MSL configurations.

Table 23.1 MSL Library Naming Conventions

These characters in a filename...	Applicable?	mean that the library has these features....
`fdlibm`	No	High-level math functions, including as the trigonometric functions.
`MSL_C`	Yes	C standard library.
`MSL_C++`	No	C++ standard library.
`MSL_EC++`	No	Embedded C++ standard library.
`MSL_SUPP_TRK`	No	CodeWarrior TRK support.
`MSL_SUPP_UART`	No	UART (serial communications).
`.bare`	No	Boards with no operating system.
`PPCEABI`	Yes	Conforms to the PowerPC Embedded Application Binary Interface (EABI) standard.
`SZ`	`No`	Optimized for size.
`SP`	`Yes`	Single Precision Floating Point only.
`A`	`No`	AltiVec™ support.
`C`	No	Code compression.
`E`	`Yes`	e500 and e200z (formerly Zen) targets.

Table 23.1 MSL Library Naming Conventions (*continued*)

These characters in a filename...	Applicable?	mean that the library has these features....
`E2`	No	e500v2 targets, with double-precision floating-point operations.
`H`	`No`	Hardware floating-point operations.
`HC`	`No`	Hardware floating-point operations and code compression.
`S`	`No`	Software emulation of floating-point operations.
`N`	`Yes`	No floating-point support.
`NC`	`No`	No floating-point support, but with code compression.
`LE`	`No`	Little-endian mode.
`UC`	Yes	Function parameters declared `char` are treated as if they were declared `unsigned char`. Use a `UC` library in build targets for which the **Use Unsigned Chars** option is enabled. Use a non-`UC` library in build targets for which this option is disabled. If the option used by the build target is different from the option used to generate the build target's runtime library, the linker issue a warning.
`V`	`No`	Uses VLE instructions SPFP/SPE floating point operations in software routines. Use only with processors that have an e200z (formerly Zen) core.
`VS`	`No`	Uses VLE instructions, single-precision floating point operations using native processor instructions, and double-precision floating point operations using software routines.
`SC`	`Yes`	Function parameters declared `char` are treated as if they were declared `signed char`.

Using Console I/O

The default MSL configuration for Power Architecture processors provides the `stdout`, `stderr`, and `stdin` file streams through serial I/O on most evaluation boards. The C++ standard library assigns `cin`, `cout`, and `cerr` to the target board's serial port. Also, I/O functions that refer to the standard streams implicitly, such as `printf()`, are available.

This configuration does not provide disk I/O, so functions such as `fprintf()` are not available.

To use the MSL console I/O functions, you must include a special serial I/O library in your project. Your hardware must be initialized properly to work with this library.

Allocating Additional Heap Space

The heap you define using the **Heap Address** option of the **EPPC Linker** panel is the default heap. The default heap needs no initialization. The CodeWarrior linker will only link the object code for memory management if your program calls `malloc()` or `new()`.

You may find that you do not have enough contiguous memory available for your needs. In this case, you can initialize multiple memory pools to form a large heap.

You create each memory pool by calling `init_alloc()`. You can find an example of this call in `__ppc_eabi_init.c` and `__ppc_eabi_init.cpp`. You do not need to initialize the memory pool for the default heap.

Third Party Standard Libraries

You might be able to use a third-party C standard library with your CodeWarrior tools. To determine if the CodeWarrior tools will generate object code that is compatible with a third-party library, compare the file `stdarg.h` from the third-party library with `stdarg.h` from the MSL library. The CodeWarrior C/C++ compiler for Power Architecture processors uses the MSL files `stdarg.h` and runtime library file `__va_arg.c` to generate variable-length parameter functions. Your third-party library must be compatible with these files.

You cannot use a third-party standard C++ library with your CodeWarrior product.

Embedded Warrior Library

This section explains how to use the Power Architecture version of the Embedded Warrior Library (EWL) .

- Using the Embedded Warrior Libraries

- EWL Naming Convention
- How to Rebuild the EWL Libraries

For more information refer to the *EWL C Reference* and the *EWL C++ Reference.*

Using the Embedded Warrior Libraries

Embedded Warrior Library (EWL) is the next generation of MSL. With this release, EWL will be an alternative library. The sources are based on MSL and are more MISRA compliant. Existing standard prefix file name, library (archive) names have been modified. some of the legacy libraries have been depricated and a greater number of processor core specific libraries are introduced.

NOTE EWL is not supported by all products. All of your existing projects and makefile access paths will not use the EWL unless you specifically select it.

EWL Naming Convention

Each archive name has 3 pieces: prefix, core and flags. Following are the details of each piece:

- The prefix is one of the following:
  - `libm_` - mean that the library has math features
  - `librt_` - mean that the library has runtime features
  - `libc_` - mean that the library has reduced code size C features
  - `libc99_` - mean that the library has faster and increased C99 conformant C features
  - `libstdc++__` - mean that the library has latest C++ features
  - `libc++_` - mean that the library has reduced code size C++ features
- The core starts with the processor family (like e200 or e500) and optionally ends with the core name (such as z750).
- Current flags are VLE, Soft (software floating point) and SPFP_Only (math library only has single precision sources and source file doubles are treated as if they are single precision). SPFP_Only is only used with e200 and e500 which have single precision floating point instructions but no double precision instructions.

NOTE EWL can only build unsigned char libraries. CodeWarrior no longer provides signed char library in EWL as it is not compliant with EABI. If required, users must build their own signed char libraries.

The Prefix Name used in EWL also differs from that of the MSL Prefix Name. For example, the prefix name `ansi_prefix.PPCEABI.bare.h` in MSL is referred to as `ansi_prefix.PA_EABI.bare.h` in EWL.

Table 23.2 lists the EWL Library Core And Flag Name and its equivalent MSL Suffix Name.

NOTE Some of the libraries listed in Table 23.2 may not be available in the current release.

Table 23.2 EWL Library Core and Flag Name and its equivalent MSL Suffix Name

EWL Library Core and Flag Name	Equivalent MSL Suffix Name
`Generic_N`	`PPCEABI.N.UC`
`82xx_soft`	`PPCEABI.S.UC`
`E200z0_VLE_Soft`	`PPCEABI.VS.UC`
`E200z150_VLE_Soft`	`PPCEABI.VS.UC`
`E200z335_VLE`	`PPCEABI.V.UC`
`E200z335_VLE_SPFP_Only`	`PPCEABI.V.SP.UC`
`E200z336_VLE`	`PPCEABI.V.UC`
`E200z336_VLE_SPFP_Only`	`PPCEABI.V.SP.UC`
`E200z446_VLE`	`PPCEABI.V.UC`
`E200z446_VLE_SPFP_Only`	`PPCEABI.V.SP.UC`
`E200z448_VLE`	`PPCEABI.V.UC`
`E200z448_VLE_SPFP_Only`	`PPCEABI.V.SP.UC`
`E200z650`	`PPCEABI.E.UC`
`E200z650_SPFP_Only`	`PPCEABI.E.SP.UC`
`E200z650_VLE`	`PPCEABI.V.UC`
`E200z650_VLE_SPFP_Only`	`PPCEABI.V.SP.UC`
`E200z652`	`PPCEABI.E.UC`
`E200z652_SPFP_Only`	`PPCEABI.E.SP.UC`

Table 23.2 EWL Library Core and Flag Name and its equivalent MSL Suffix Name

EWL Library Core and Flag Name	Equivalent MSL Suffix Name
`E200z652_VLE`	`PPCEABI.V.UC`
`E200z652_VLE_SPFP_Only`	`PPCEABI.V.SP.UC`
`E200z750_VLE`	`PPCEABI.V.UC`
`E200z750_VLE_SPFP_Only`	`PPCEABI.V.SP.UC`
`E200z760_VLE`	`PPCEABI.V.UC`
`E200z760_VLE_SPFP_Only`	`PPCEABI.V.SP.UC`
`E300c1`	`PPCEABI.H.UC`
`E300c2`	`PPCEABI.H.UC`
`E300c3`	`PPCEABI.H.UC`
`E300c4`	`PPCEABI.H.UC`
`E500V1`	`PPCEABI.E.UC`
`E500V1_SPFP_Only`	`PPCEABI.E.SP.UC`
`E500V2`	`PPCEABI.E2.UC`
`E600`	`PPCEABI.A.UC`

How to Rebuild the EWL Libraries

The EWL library files are present in the `ewl\lib` folder. To rebuild the EWL library files, perform the following steps:

NOTE Ensure that you have access to a `make` utility within DOS, before rebuilding the EWL libraries.

1. Open a DOS command prompt.
2. Define the CWINSTALL environment variable.

   For example, if your PA product layout is in the folder `C:\Program Files\Freescale\CW for MPC55xx and MPC56xx 2.10` then you can define CWINSTALL as follows:

   ```
   set CWINSTALL='C:\Program Files\Freescale\CW for MPC55xx
   and MPC56xx 2.10'
   ```

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:\Program Files\Freescale\CW for MPC55xx and MPC56xx
   2.10\PA_Support\ewl
   ```

4. Modify '`TOOLS_ROOT = $(CWFOLDER)/PA_Tools`' to '`TOOLS_ROOT = $(CWFOLDER)/PowerPC_EABI_Tools`' in the `EWL_C.PA.mak`, `EWL_C++.PA.mak` and `EWL_Runtime.PA.mak` ewl makefiles.

5. Clean the existing library files using the following command:

   ```
   <path_to_make_utility>\make -f makefile clean PLATFORM=PA
   TARGETS="libm_XXX libc_XXX libc99_XXX"
   ```

NOTE You could skip the `<path_to_make_utility>` if you have `make` in your `PATH` variable.

   For example, the following command will delete only the `libm_E200z650.a`, `libc_E200z650.a`, `libc99_E200z650.a` library files.

   ```
   make -f makefile clean PLATFORM=PA TARGETS="libm_E200z650
   libc_E200z650 libc99_E200z650"
   ```

6. Rebuild a `C` or `math` or `C99` library fileusing the following command:

   ```
   make -f EWL_C.PA.mak -C EWL_C TARGETS="libm_XXX libc_XXX
   libc99_XXX"
   ```

   For example:

   ```
   make -f EWL_C.PA.mak -C EWL_C TARGETS="libm_E200z650
   libc_E200z650 libc99_E200z650" (for building
   libm_E200z650.a, libc_E200z650.a libc99_E200z650.a)
   ```

NOTE Re-building any particular `C` and `C99` library, requires `math` library of the same target. It is suggested that the `math` library is built prior to building the `C99` or `C` libraries.

7. Rebuild a `C++` or `libc++` library file using the following command:

   ```
   make -f EWL_C++.PA.mak -C EWL_C++ TARGETS="libstdc++_XXX
   libc++_XXX"
   ```

   For example:

   ```
   make -f EWL_C++.PA.mak -C EWL_C++
   TARGETS="libstdc++_E200z0_VLE_Soft
   ```

```
libc++_E200z0_VLE_Soft" (for building
libstdc++_E200z0_VLE_Soft.a, libc++_E200z0_VLE_Soft.a)
```

8. Rebuild a `Runtime` library file using the following command:

   ```
   make -f EWL_Runtime.PA.mak -C EWL_Runtime "librt_XXX"
   ```

   For example:

   ```
   make -f EWL_Runtime.PA.mak -C EWL_Runtime "librt_E300c1"
   (for building librt_E300c1.a)
   ```

9. Upon successful execution of the `make` command, check the lib folder for PA EWL libraries.

Runtime Libraries

A runtime library provides low-level functions that support high-level C and C++ language features, such as memory management and file system access. Conceptually, a runtime library acts as an interface between a target system's hardware or operating system and the CodeWarrior C or C++ runtime environment.

This CodeWarrior product includes many runtime libraries and support code files. These files are here:

`*installDir*\PowerPC_EABI_Support\Runtime`

where *`installDir`* is a placeholder for the path in which you installed your product.

For your projects to build and run, you must include the correct runtime library and startup code. These sections explain how to pick the correct files:

- Required Libraries and Source Code Files
- Allocating Additional Heap Space
- Choosing a Runtime Library

Required Libraries and Source Code Files

Every CodeWarrior project *must* include a runtime library.

Select the library appropriate for your project, given the language your are using (C or C++), the processor on your target board, and your target setting choices. Use the information in Table 23.3 to help you pick the correct library.

The runtime libraries are in this directory:

`*installDir*\PowerPC_EABI_Support\Runtime\Lib\`

Along with the pre-built runtime libraries, this CodeWarrior product includes the source code and project files required to build the runtime libraries. As a result, you can modify them as necessary.

All runtime library source code files are in this directory:

`installDir\PowerPC_EABI_Support\Runtime\Src`

The runtime library project files are in this directory:

`installDir\PowerPC_EABI_Support\Runtime\Project`

The project names are `Runtime.PPCEABI.mcp` and `Run_EC++.PPCEABI.mcp`. Each project has a different build target for each configuration of the runtime library.

For more information about customizing the runtime libraries, read the comments in the source code files as well as the runtime library release notes.

NOTE The C and C++ runtime libraries do *not* initialize hardware. The CodeWarrior tools assume that you load and run the programs linked with these libraries with the CodeWarrior debugger. When your program is ready to run as a standalone application, you must add the required hardware initialization code.

Finally, in addition to a runtime library, every C and C++ project must include one of the startup files listed below. These files contain functions called by the runtime code that you can customize if necessary. One kind of customization is board-specific initialization. For other customization examples, see either of these files:

- `__ppc_eabi_init.c` (for C language projects)
- `__ppc_eabi_init.cpp` (for C++ projects)

Allocating Additional Heap Space

If you specify a heap size in the EPPC Target settings panel, the linker creates a default heap of this size. The default heap needs no initialization.

You can create additional heaps by:

- Defining `ALLOC_ADDITIONAL_HEAPS` equal to `1` in either `__ppc_eabi_init.c` or `__ppc_eabi_init.cpp`.

  Doing so causes the stub implementation of `AllocMoreHeaps()` to be called by the runtime initialization code.
- Implementing the `AllocMoreHeaps()` stub by calling `init_alloc()` as many times as desired.

  Each time `init_alloc()` is called, the heap is expanded by the amount specified.

Choosing a Runtime Library

Substrings embedded in the name of a runtime library indicate the type of support the library provides. Use these substrings to pick the runtime library appropriate for your project. Table 23.3 lists and defines the meaning of each library filename substring.

Table 23.3 Runtime Library Naming Conventions

Substring	Meaning
`Runtime`	The library is a C language library.
`Run_EC++`	The library is an embedded C++ library.
`PPCEABI`	The library conforms to the PowerPC Embedded Application Binary Interface (EABI) standard.
`A`	The library provides AltiVec™ support.
`E`	The library is for e500 and e200z (formerly, Zen) targets.
`E.fast`	The library is for e500 and e200z (formerly, Zen) targets. Further, this library's floating-point operations are faster than those of a `.E` library, but they do not strictly conform to the IEEE floating-point standard.
`E`2	The library is for e500v2 targets and supports double-precision floating-point operations.
`H`	The library supports hardware floating-point operations.
`HC`	The library supports hardware floating-point operations and code compression.
`S`	The library provides software emulation of floating-point operations.
`SP`	Single Precision Floating Point only.
`N`	The library provides no floating-point support.
`NC`	The library provides no floating-point support, but supports code compression.
`LE`	The library is for a processor running in little-endian mode.
`UC`	The library was built with the Use Unsigned Chars option of the C++ Language target settings panel enabled. As a result, all library function parameters declared `char` are treated as if the were declared `unsigned char`. Use a `UC` library in build targets for which the Use Unsigned Chars option is enabled. Use a non-`UC` library in build targets for which this option is disabled. If the option used by the build target is different from the option used to generate the build target's runtime library, the linker issue a warning.

Table 23.3 Runtime Library Naming Conventions (*continued*)

Substring	Meaning
`V`	The library's functions: • Contain VLE instructions. • Perform single-precision floating point operations using the core's SPE auxiliary processing unit (APU). • Perform double-precision floating using software routines. Use only with processors that have an SPE APU.
`VS`	The library's functions: • Contain VLE instructions. • Perform all floating-point operations using softare routines Use only with processors that have an e200z (formerly Zen) core.

Board Initialization Code

For each supported board, CodeWarrior for Power Architecture Processors includes a hardware initialization routine. Each routine is in a source code file whose name reflects the board for which the routine is designed. These files are in this directory:

`*installDir*\PowerPC_EABI_Support\Runtime\Src`

The initialization routines are in source code form, so you can modify them to work with different configurations of a board or with a different board.

If you run your program under control of the CodeWarrior debugger, the program must *not* perform hardware initialization because the debugger performs the required board initialization.

However, if your program is running standalone (that is, without the debugger), the program may need to execute hardware initialization code. The easiest way to include this code in your program is to add the appropriate board initialization file to your project.

Each board initialization file includes a function named `usr_init()`. This function performs the required hardware initialization for your board. `usr_init()` is *conditionally* called by the `__init_hardware()` function in `ppc_eabi_init.c` (or in `ppc_eabi_init.cpp`, if you are using C++) . The startup code always calls `__init_hardware()`.

The default implementation of the `__init_hardware()` function calls `usr_init()` if either the `ROM_VERSION` or `CACHE_VERSION` preprocessor constant is defined. (See Listing 23.1.) However, you can change the implementation of `__init_hardware()` to suit your project's requirements.

Listing 23.1 Code Showing call of usr_init() in __init_hardware()

```
asm void __init_hardware(void)
{
   /*
   *   If not running with the CodeWarrior debugger, initialize the
   *   board. Define the preprocessor symbols for the initialization
   *   your program requires. You may need to perform other
   *   initializations.
   */
   nofralloc

/* ... code omitted */

#if defined(ROM_VERSION) || defined(CACHE_VERSION)
   mflr   r31        /* save off return address in NV reg */
   bl     usr_init   /* init board hardware */
   mtlr   r31        /* get saved return address */
#endif

   blr
}
```

To get you program to perform hardware initialization when run outside the CodeWarrior debugger, follow these steps:

1. Add the appropriate board initialization file to your project.
2. Change the preprocessor symbol that conditionalizes the `usr_init()` call in `__init_hardware()` to a symbol that makes sense for your project.
3. Define this symbol in the prefix file for each build target for which you want to run the hardware initialization code.

24 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.

- Syntax for Declaration Specifications
- Declaration Specifications

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.

Declaration Specifications

__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.

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 delcaration and definition of a function, function parameter, or variable. Listing 24.1 shows an example.

Listing 24.1 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;
}
```

Attribute Specifications

__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 24.2 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 */
}
```

__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.

__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.

__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 Listing 24.3 will not return a correct result if `ip` is pointed to `a`.

Listing 24.3 Example of the noalias attribute

```
extern int a __attribute__((noalias));
int f(int *ip)
{
  a = 1;
  *ip = 0;
  return a;   // optimized to return 1;
}
```

__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.

__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 24.4 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;
}
```

__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.

25 Declaration Specifications for Power Architecture Code

Declaration specifications describe special attributes to associate with a function at compile time. For example, these attributes might change how the compiler translates the function, describe what properties it has, or pass other information to the compiler.

- Syntax for Declaration Specifications
- Declaration Specifications

Syntax for Declaration Specifications

The syntax for a declaration specification is

```
__declspec(spec [ options ]) function-prototype;
```

where *spec* is the declaration specification, *options* represents possible arguments for the declaration specification, and *function-prototype* represents the declaration of the function. Unless otherwise specified in the declaration specification's description, a function's definition does not require a matching declaration specification.

Declaration Specifications

- __declspec(do_not_merge)
- __declspec(final)
- __declspec(force_export)
- __declspec(interrupt)
- __declspec(no_linker_opts)
- __declspec(section name)
- __declspec(vle_off)
- __declspec(vle_on)

__declspec(do_not_merge)

Specifies that a function must not be removed from object code during code merging optimization.

Syntax

```
__declspec(do_not_merge) function-declaration;
```

Remarks

When you declare or define a function with this declaration specification the linker does not consider this function when applying the code merging optimization. This declaration specification ensures that the linker will not remove a function from object code during code merging if another function with identical object code exists.

Use this declaration specification for functions that your program refers to with function pointers.

__declspec(final)

Provides Java-style function override checking.

Syntax

```
__declspec(final) declaration
```

Example

```
struct A {
virtual __declspec(final) void vf1();
};
struct B : A {
     void vf1();  /* Error : the final function A::vf()
                     is overridden by B::vf1() */
};
```

__declspec(force_export)

Specifies that a function or variable must not be dead-stripped.

Syntax

```
__declspec(force_export) function-declaration;
__declspec(force_export) variable-declaration;
```

Remarks

When the linker determines that a function or static variable is not referred to by the rest of the program, the linker removes the object from the final executable image. The linker does not remove a static variable or function declared with this specification even if this object is not referred to.

__declspec(interrupt)

Controls the compilation of object code for interrupt service routines.

Syntax

```
__declspec (interrupt [option [ optionN]]) void
    __InterruptHandler__(void);

__declspec (interrupt [option [ optionN]]) void
    __InterruptHandler__(void)
    {
        /* ... */
    }
```

where *option* and *optionN* are zero or more of the following choices:

NOTE If no choice is specified, `save_spe` is enabled.

`enable`

Enables interrupts while this interrupt handler is executing.

`SRR`

Saves the appropriate save/restore register (SRR0 or SRR1) on the stack.

`DAR`

Saves the data address register on the stack.

`DSISR`

Saves the DSI status register on the stack.

`save_fprs`

Saves the floating-point registers on the stack.

`save_altivec`

Saves the Altivec® registers on the stack.

`save_spe`

Saves the special-purpose embedded registers on the stack. This is the default.

`nowarn`

Does not issue a warning message if the function being compiled as an interrupt service routine is larger than the processor's interrupt vector area (256 bytes/64 instructions).

`noncritical | critical | machine | debug`

Specify the type of interrupt service routine to generate. Specifically, the compiler choose a return instruction based on this choice. The default is noncritical.

`vle_multiple`

Enables generation of new VLE instructions in the interrupt prolog/epilog in C-functions by the compiler. This option is only valid for e200 (Zen) core. Compile error will be returned if this option is used for any other cores.

When enabled, the compiler:

- emits e_lmvgprw and e_stmvgprw if any of [r3,r12] needs to be saved to stack
- emits e_lmvsprw and e_stmvsprw if any of the CR, LR, CTR and XER needs to be saved to stack
- emits e_lmvsrrw and e_stmvsrr if any of the SRR0 and SRR1 needs to be saved to stack
- emits e_lmvcsrrw and e_stmvcsrrw if any of the CSRR0 and CSRR1 needs to be saved to stack
- emits e_lmvdsrrw and e_stmvdsrrw if any of the DSRR0 and DSRR1 needs to be saved to stack.

NOTE Enabling `vle_multiple` could increase the space being taken up by the stack, especially when saving the volatile gprs.

Remarks

When you declare or define a function with this declaration specification the compiler generates a special prologue and epilogue for the function so that it can respond to a processor interrupt. For convenience, the compiler also marks interrupt functions so that the linker does not dead-strip them.

__declspec(no_linker_opts)

Specifies that the linker must not apply its optimizations to a function.

Syntax

`__declspec(no_linker_opts)` *`function-declaration`*

__declspec(section name)

Specifies where to store a variable or function in a section that has been predefined or defined with the `#pragma section` directive.

Syntax

```
__declspec(section <section_name>) declaration
```

```
__declspec(section <section_name>) definition
```

Parameters

`section_name`

Specifies the name of an initialized data section.

NOTE The name of a section must be enclosed in double-quotes (""). To use a user defined section, create the section using `#pragma section` directive before using `__declspec(section <section_name>)`.

Remarks

- When you specify an uninitialized section name while declaring or defining a function or variable with this declaration specification, the compiler generates an error.

  For example, if you use `__declspec (section ".bss") extern int my_var;`, where `.bss` is an uninitialized section you will get a descriptive error. In this case, use `__declspec (section ".data") extern int my_var;` as `.data` is normally paired with `.bss` and `.data` is an initialized section. Assuming the variable you are attaching this `__declspec` to is an uninitialized object (which is the case with `my_var`), the object will go into `.bss`.

- When you use `__declspec` on the definition, `__declspec` on a declaration is ignored. Listing 25.1 shows an example of `__declspec` used to put data and code objects into specific sections.

Listing 25.1 Example of a declaration specification

```
__declspec (section ".init") extern void cache_init1(void);
__declspec (section ".text") extern void cache_init2(void);
extern void cache_init3(void);

void cache_init1(){} // goes into .init if the prototype is visible for
this definition

__declspec (section ".init") void cache_init2(){} // ignores previous
section .text and goes into .init

__declspec (section ".init") void cache_init3(){} // by default the
declaration  implies .text but the definition forces it to .init
```

Predefined sections and default sections

The predefined sections set with an object type become the default section for that type. The compiler predefines the sections in Table 25.1.

Table 25.1 Predefined sections

Type	Name	Data mode	Code mode
`code_type`	`".text"`	`data_mode=far_abs`	`code_mode=pc_rel`
`data_type`	`".data”`	`data_mode=far_abs`	`code_mode=pc_rel`
`const_type`	`".rodata”`	`data_mode=far_abs`	`code_mode=pc_rel`
`sdata_type`	`".sdata”`	`data_mode=sda_rel`	`code_mode=pc_rel`
`sconst_type`	`".sdata2” ".sbss2"`	`data_mode=sda_rel`	`code_mode=pc_rel`
	`".PPC.EMB.sdata0”` `".PPC.EMB.sbss0"`	`data_mode=sda_rel`	`code_mode=pc_rel`

NOTE The “`.PPC.EMB.sdata0`” and “`.PPC.EMB.sbss0`” sections are predefined as an alternative to the `sdata_type` object type.

__declspec(vle_off)

Forces the compiler to use the regular instruction set instead of the Variable Length Encoded (VLE) instruction set.

Syntax

`__declspec (vle_off) `*`function_prototype`*`;`

Remarks

Declaring a function's prototype with this declaration specification tells the compiler to use the regular instruction set defined by the Freescale Book E Implementation Standard (EIS) for Power Architecture processors. The compiler and linker must arrange such instructions more rigidly than VLE instructions, resulting in larger object code.

For information on the availability of VLE instructions, refer to your processor's documentation.

This declaration specification overrides other compiler settings for VLE code generation.

__declspec(vle_on)

Forces the compiler to use the VLE (Variable Length Encoded) instruction set for a function.

Syntax

`__declspec (vle_on) `*`function`*`_prototype;`

Remarks

Declaring a function's prototype with this declaration specification tells the compiler to use the VLE instruction set when generating object code for the function. The compiler and linker can arrange VLE instructions more compactly than regular instructions.

The VLE instruction set extends the regular instruction set defined by the Freescale Book E Implementation Standard (EIS). For information on the availability of VLE instructions, refer to your processor's documentation.

This declaration specification overrides other compiler settings for VLE code generation.

26 Predefined Macros

The compiler preprocessor has predefined macros (some refer to these as predefined symbols). The compiler simulates variable definitions that describe the compile-time environment and properties of the target processor.

This chapter lists the predefined macros that all CodeWarrior compilers make available.

__COUNTER__

Preprocessor macro that expands to an integer.

Syntax

```
__COUNTER__
```

Remarks

The compiler defines this macro as an integer that has an initial 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.

__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.

__CWBUILD__

Preprocessor macro defined as the build number of the CodeWarrior compiler.

Syntax

```
__CWBUILD__
```

Remarks

For example, in `Help->About Freescale CodeWarrior`, click `Installed Product`, and expand `Plugins->Compiler->ppc_eabi.dll`. If the value in the `Version` column is `4.3 180`, the corresponding value of `__CWBUILD__` is `180`.

The ISO standards do not specify this symbol.

__CWCC__

Preprocessor macro defined as the version of the CodeWarrior compiler frontend.

Syntax

```
__CWCC__
```

Remarks

CodeWarrior compilers issued after 2006 define this macro with the compiler's frontend version. For example, if the compiler frontend 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.

__DATE__

Preprocessor macro defined as the date of compilation.

Syntax

```
__DATE__
```

Remarks

The compiler defines this macro as a character string representation of the date of compilation. The format of this string is

`"`*`Mmm dd yyyy`*`"`

where *Mmm* is the a three-letter abbreviation of the month, *dd* is the day of the month, and *yyyy* is the year.

__embedded_cplusplus

Defined as 1 when compiling embedded C++ source code, undefined otherwise.

Syntax

```
__embedded_cplusplus
```

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the Embedded C++ proposed standard. The compiler does not define this macro otherwise.

__FILE__

Preprocessor macro of the name of the source code file being compiled.

Syntax

```
__FILE__
```

Remarks

The compiler defines this macro as a character string literal value of the name of the file being compiled, or the name specified in the last instance of a `#line` directive.

__func__

Predefined variable of the name of the function being compiled.

Prototype

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

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__func__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body. This variable is also undefined when C99 (ISO/IEC 9899-1999) or GCC (GNU Compiler Collection) extension settings are off.

__FUNCTION__

Predefined variable of the name of the function being compiled.

Prototype

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

Remarks

The compiler implicitly defines this variable at the beginning of each function if the function refers to `__FUNCTION__`. The character string contained by this array, *function-name*, is the name of the function being compiled.

This implicit variable is undefined outside of a function body.

__ide_target()

Preprocessor operator for querying the IDE about the active build target.

Syntax

`__ide_target("`*`target_name`*`")`

`target-name`

The name of a build target in the active project in the CodeWarrior IDE.

Remarks

Expands to `1` if *target_name* is the same as the active build target in the CodeWarrior IDE's active project. Expands to `0` otherwise. The ISO standards do not specify this symbol.

__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 a integer value of the number of the line of the source code file that the compiler is translating. The `#line` directive also affects the value that this macro expands to.

__MWERKS__

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.

__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 14882-1998 C++ standard does not specify this symbol.

__profile__

Preprocessor macro that specifies whether or not the compiler is generating object code for a profiler.

Syntax

`__profile__`

Remarks

Defined as 1 when generating object code that works with a profiler. Undefined otherwise. The ISO standards does not specify this symbol.

__STDC__

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

Syntax

`__STDC__`

Remarks

The compiler defines this macro as 1 when the compiler's settings are configured to restrict the compiler to translate source code that conforms to the ISO/IEC 9899-1990 and ISO/IEC 9899-1999 standards. The compiler does not define this macro otherwise.

__TIME__

Preprocessor macro defined as a character string representation of the time of compilation.

Syntax

`__TIME__`

Remarks

The compiler defines this macro as a character string representation of the time of compilation. The format of this string is

`"`*`hh`*`:`*`mm`*`:`*`ss`*`"`

where *hh* is a 2-digit hour of the day, *mm* is a 2-digit minute of the hour, and *ss* is a 2-digit second of the minute.

27

Predefined Macros for Power Architecture Compiler

As listed in the previous chapter, the compiler preprocessor has predefined macros. The macros in the previous chapter are available to all architecture. This chapter describes the predefined macros available specifically for the Power Architecture compiler.

- __ALTIVEC__
- __PPCBROADWAY__
- __PPCGECKO__

__ALTIVEC__

Preprocessor macro defined if language extensions for using Power Architecture AltiVec™ instructions are available.

Syntax

```
__ALTIVEC__
```

Remarks

The compiler defines this macro when pragma `altivec_model` is `on`. This macro is undefined otherwise.

__PPCBROADWAY__

Preprocessor macro defined if the compiler is generating object code for the Power Architecture "Broadway" processor.

Syntax

`__PPCBROADWAY__`

Remarks

The compiler defines this macro as `1` when generating object code for the "Broadway" processor. The `__PPCGECKO__` macro is also defined.

This macro is undefined otherwise.

__PPCGECKO__

Preprocessor macro defined if the compiler is generating object code for the Power Architecture "Gecko" processor.

Syntax

`__PPCGECKO__`

Remarks

The compiler defines this macro as `1` when generating object code for the "Gecko" processor. This macro is undefined otherwise.

28

Using Pragmas

The #pragma preprocessor directive specifies option settings to the compiler to control the compiler and linker's code generation.

- Checking Pragma Settings
- Saving and Restoring Pragma Settings
- Determining Which Settings Are Saved and Restored
- Invalid Pragmas

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.

Listing 28.1 shows an example.

Listing 28.1 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
```

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. Listing 28.2 shows an example.

Listing 28.2 Using push and pop to save and restore pragma settings

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

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

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

Pragmas that accept the `reset` argument perform the same actions as pragmas `push` and `pop`, but apply to a single pragma. A pragma's `on` and `off` arguments save the pragma's current setting before changing it to the new setting. A pragma's `reset` argument restores the pragma's setting. The `on`, `off`, and `reset` arguments may be nested to an unlimited depth. Listing 28.3 shows an example.

Listing 28.3 Using the reset option to save and restore a pragma setting

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

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

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

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

Determining Which Settings Are Saved and Restored

Not all pragma settings are saved and restored by pragmas `push` and `pop`. Pragmas that do not change compiler settings are not affected by `push` and `pop`. For example, pragma `message` cannot be saved and restored.

Listing 28.4 shows an example that checks if the `ANSI_strict` pragma setting is saved and restored by pragmas `push` and `pop`.

Listing 28.4 Testing if pragmas push and pop save and restore a setting

```
/* Preprocess this source code. */
#pragma ANSI_strict on
```

```
#pragma push
#pragma ANSI_strict off
#pragma pop
#if __option(ANSI_strict)
#error "Saved and restored by push and pop."
#else
#error "Not affected by push and pop."
#endif
```

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 Listing 28.5 generate warnings with the invalid pragmas setting enabled.

Listing 28.5 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
```

Table 28.1 shows how to control the recognition of invalid pragmas.

Table 28.1 Controlling invalid pragmas

To control this option from here...	use this setting
CodeWarrior IDE	**Illegal Pragmas** in the **C/C++ Warnings** panel
source code	`#pragma warn_illpragma`
command line	`-warnings illpragmas`

Pragma Scope

The scope of a pragma setting is limited to a compilation unit (a source code file and the files that it includes).

At the beginning of compilation unit, the compiler uses its default settings. The compiler then uses the settings specified by the CodeWarrior IDE's build target or in command-line options.

The compiler uses the setting in a pragma beginning at the pragma's location in the compilation unit. The compilers continues using this setting:

- until another instance of the same pragma appears later in the source code
- until an instance of pragma `pop` appears later in the source code
- until the compiler finishes translating the compilation unit

29 Pragmas for Standard C Conformance

ANSI_strict

Controls the use of non-standard language features.

Syntax

```
#pragma ANSI_strict on | off | reset
```

Remarks

If you enable the pragma `ANSI_strict`, the compiler generates an error message if it encounters some CodeWarrior extensions to the C language defined by the ISO/IEC 9899-1990 ("C90") standard:

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

This pragma corresponds to the **ANSI Strict** setting in the CodeWarrior IDE's **C/ C++ Language** settings panel. By default, this pragma is `off`.

c99

Controls the use of a subset of ISO/IEC 9899-1999 ("C99") language features.

Syntax

```
#pragma c99 on | off | reset
```

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`.

This pragma corresponds to the **Enable C99 Extensions** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is disabled.

c9x

Equivalent to `#pragma c99`.

ignore_oldstyle

Controls the recognition of function declarations that follow the syntax conventions used before ISO/IEC standard C (in other words, “K&R” style).

Syntax

```
#pragma ignore_oldstyle on | off | reset
```

Remarks

If you enable this pragma, the compiler ignores old-style function declarations and lets you prototype a function any way you want. In old-style declarations, you specify the types of arguments on separate lines instead of the function’s argument list. For example, the code in Listing 29.1 defines a prototype for a function with an old-style definition.

Listing 29.1 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.

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`.

This pragma corresponds to the **ANSI Keywords Only** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is disabled.

require_prototypes

Controls whether or not the compiler should expect function prototypes.

Syntax

```
#pragma require_prototypes on | off | reset
```

Remarks

This pragma only affects non-static functions.

If you enable this pragma, the compiler generates an error message if you use a function that does not have a preceding prototype. Use this pragma to prevent error messages caused by referring to a function before you define it. For example, without a function prototype, you might pass data of the wrong type. As a result, your code might not work as you expect even though it compiles without error.

In Listing 29.2, function `main()` calls `PrintNum()` with an integer argument even though `PrintNum()` takes an argument of type `float`.

Listing 29.2 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 intended. Since `PrintNum()` does not have a prototype, the compiler does not know to generate instructions to convert the integer to a floating-point number before calling `PrintNum()`. Consequently, the function interprets the bits it received as a floating-point number and prints nonsense.

A prototype for `PrintNum()`, as in Listing 29.3, gives the compiler sufficient information about the function to generate instructions to properly convert its argument to a floating-point number. The function prints what you expected.

Listing 29.3 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.

This pragma corresponds to the **Require Function Prototypes** setting in the CodeWarrior IDE's **C/C++ Language** settings panel.

Pragmas for C++

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`.

always_inline

Controls the use of inlined functions.

Syntax

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

Remarks

This pragma is deprecated. We recommend that you use the `inline_depth()` pragma instead.

arg_dep_lookup

Controls C++ argument-dependent name lookup.

Syntax

```
#pragma arg_dep_lookup on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler uses argument-dependent name lookup.

This pragma does not correspond to any IDE panel setting. By default, this setting is `on`.

ARM_conform

This pragma is no longer available. Use `ARM_scoping` instead.

ARM_scoping

Controls the scope of variables declared in the expression parts of `if`, `while`, `do`, and `for` statements.

Syntax

```
#pragma ARM_scoping on | off | reset
```

Remarks

If you enable this pragma, any variables you define in the conditional expression of an `if`, `while`, `do`, or `for` statement remain in scope until the end of the block that contains the statement. Otherwise, the variables only remain in scope until the end of that statement. Listing 30.1 shows an example.

This pragma corresponds to the **Legacy for-scoping** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is `off`.

Listing 30.1 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.
```

array_new_delete

Enables the operator `new[]` and `delete[]` in array allocation and deallocation operations, respectively.

Syntax

```
#pragma array_new_delete on | off | reset
```

Remarks

By default, this pragma is `on`.

auto_inline

Controls which functions to inline.

Syntax

```
#pragma auto_inline on | off | reset
```

Remarks

If you enable this pragma, the compiler automatically chooses functions to inline for you, in addition to functions declared with the `inline` keyword.

Note that if you enable either the **Do not Inline** setting or the `dont_inline` pragma, the compiler ignores the setting of the `auto_inline` pragma and does not inline any functions.

This pragma corresponds to the **Auto-Inline** setting in the CodeWarrior IDE’s **C/ C++ Language** settings panel. By default, this pragma is disabled.

bool

Determines whether or not `bool`, `true`, and `false` are treated as keywords in C++ source code.

Syntax

```
#pragma bool on | off | reset
```

Remarks

If you enable this pragma, you can use the standard C++ `bool` type to represent `true` and `false`. Disable this pragma if `bool`, `true`, or `false` are defined in your source code.

Enabling the `bool` data type and its `true` and `false` values is not equivalent to defining them in source code with `typedef`, `enum`, or `#define`. The C++ `bool` type is a distinct type defined by the ISO/IEC 14882-1998 C++ Standard. Source code that does not treat `bool` as a distinct type might not compile properly.

This pragma corresponds to the **Enable bool Support** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this setting is `on`.

cplusplus

Controls whether or not to translate subsequent source code as C or C++ source code.

Syntax

```
#pragma cplusplus on | off | reset
```

Remarks

If you enable this pragma, the compiler translates the source code that follows as C++ code. Otherwise, the compiler uses the suffix of the filename to determine how to compile it. If a file name ends in `.c`, `.h`, or `.pch`, the compiler automatically compiles it as C code, otherwise as C++. Use this pragma only if a file contains both C and C++ code.

NOTE The CodeWarrior C/C++ compilers do not distinguish between uppercase and lowercase letters in file names and file name extensions except on UNIX-based systems.

This pragma corresponds to the **Force C++ Compilation** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is disabled.

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.

cpp_extensions

Controls language extensions to ISO/IEC 14882-1998 C++.

Syntax

`#pragma cpp_extensions on | off | reset`

Remarks

If you enable this pragma, you can use the following extensions to the ISO/IEC 14882-1998 C++ standard that would otherwise be invalid:

- Anonymous `struct` & `union` objects. Listing 30.2 shows an example.

Listing 30.2 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. Listing 30.3 shows an example.

Listing 30.3 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.

By default, this pragma is disabled.

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`.

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 14882-1998 C++ standard does not support the `inherited` symbol. Only the CodeWarrior C++ language implements the `inherited` symbol for single inheritance.

By default, this pragma is `off`.

defer_codegen

Obsolete pragma. Replaced by interprocedural analysis options. See "Interprocedural Analysis" on page 281.

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, Listing 30.4 results in an error message.

Listing 30.4 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>());
};
```

Listing 30.5 does not generate an error message.

Listing 30.5 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`.

direct_destruction

This pragma is obsolete. It is no longer available.

direct_to_som

This pragma is obsolete. It is no longer available.

dont_inline

Controls the generation of inline functions.

Syntax

```
#pragma dont_inline on | off | reset
```

Remarks

If you enable this pragma, the compiler does not inline any function calls, even those declared with the `inline` keyword or within a class declaration. Also, it does not automatically inline functions, regardless of the setting of the `auto_inline` pragma, described in "auto_inline" on page 401. If you disable this pragma, the compiler expands all inline function calls, within the limits you set through other inlining-related pragmas.

This pragma corresponds to the **Do not Inline** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is `off`.

ecplusplus

Controls the use of embedded C++ features.

Syntax

```
#pragma ecplusplus on | off | reset
```

Remarks

If you enable this pragma, the C++ compiler disables the non-EC++ features of ISO/IEC 14882-1998 C++ such as templates, multiple inheritance, and so on.

This pragma corresponds to the **EC++ Compatibility Mode** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is `off`.

exceptions

Controls the availability of C++ exception handling.

Syntax

```
#pragma exceptions on | off | reset
```

Remarks

If you enable this pragma, you can use the `try` and `catch` statements in C++ to perform exception handling. If your program does not use exception handling, disable this setting to make your program smaller.

You can throw exceptions across any code compiled by the CodeWarrior C/C++ compiler with `#pragma exceptions on`.

You cannot throw exceptions across libraries compiled with `#pragma exceptions off`. If you throw an exception across such a library, the code calls `terminate()` and exits.

This pragma corresponds to the **Enable C++ Exceptions** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is `on`.

inline_bottom_up

Controls the bottom-up function inlining method.

Syntax

```
#pragma inline_bottom_up on | off | reset
```

Remarks

Bottom-up function inlining tries to expand up to eight levels of inline leaf functions. The maximum size of an expanded inline function and the caller of an inline function can be controlled by the pragmas shown in Listing 30.6 and Listing 30.7.

Listing 30.6 Maximum Complexity of an Inlined Function

```
// Maximum complexity of an inlined function
#pragma inline_max_size( max )          // default max == 256
```

Listing 30.7 Maximum Complexity of a Function that Calls Inlined Functions

```
// Maximum complexity of a function that calls inlined functions
#pragma inline_max_total_size( max )    // default max == 10000
```

where *max* loosely corresponds to the number of instructions in a function.

If you enable this pragma, the compiler calculates inline depth from the last function in the call chain up to the first function that starts the call chain. The number of functions the compiler inlines from the bottom depends on the values of `inline_depth`, `inline_max_size`, and `inline_max_total_size`. This method generates faster and smaller source code for some (but not all) programs with many nested inline function calls.

If you disable this pragma, top-down inlining is selected, and the `inline_depth` setting determines the limits for top-down inlining. The `inline_max_size` and `inline_max_total_size` pragmas do not affect the compiler in top-down mode.

This pragma corresponds to the **Bottom-up** setting of the **Inline Depth** menu in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is disabled.

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`.

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.

Remarks

The pragmas `dont_inline` and `always_inline` override this pragma. This pragma corresponds to the **Inline Depth** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. Setting the **Inline Depth** option to "Do not Inline" in the settings panel will also override this pragma. By default, this pragma is disabled.

inline_max_auto_size

Determines the maximum complexity for an auto-inlined function.

Syntax

```
#pragma inline_max_auto_size ( complex )
```

Parameters

`complex`

The `complex` value is an approximation of the number of statements in a function, the current default value is 15. Selecting a higher value will inline more functions, but can lead to excessive code bloat.

Remarks

This pragma does not correspond to any panel setting.

inline_max_size

Sets the maximum number of statements, operands, and operators used to consider the function for inlining.

Syntax

```
#pragma inline_max_size ( size )
```

Parameters

`size`

The maximum number of statements, operands, and operators in the function to consider it for inlining, up to a maximum of 256.

Remarks

This pragma does not correspond to any panel setting.

inline_max_total_size

Sets the maximum total size a function can grow to when the function it calls is inlined.

Syntax

```
#pragma inline_max_total_size ( max_size )
```

Parameters

`max_size`

The maximum number of statements, operands, and operators the inlined function calls that are also inlined, up to a maximum of 7000.

Remarks

This pragma does not correspond to any panel setting.

internal

Controls the internalization of data or functions.

Syntax

```
#pragma internal on | off | reset
```

`#pragma internal list` *name1* `[,` *name2* `]*`

Remarks

When using the `#pragma internal on` format, all data and functions are automatically internalized.

Use the `#pragma internal list` format to tag specific data or functions for internalization. It applies to all names if it is used on an overloaded function. You cannot use this pragma for C++ member functions or static class members.

Listing 30.8 shows an example:

Listing 30.8 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.

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`.

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`.

no_conststringconv

Disables the deprecated implicit const string literal conversion (ISO/IEC 14882-1998 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 30.9 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`.

no_static_dtors

Controls the generation of static destructors in C++.

Syntax

```
#pragma no_static_dtors on | off | reset
```

Remarks

If you enable this pragma, the compiler does not generate destructor calls for static data objects. Use this pragma to generate smaller object code for C++ programs that never exit (and consequently never need to call destructors for static objects).

This pragma does not correspond to any panel setting. By default, this setting is disabled.

nosyminline

Controls whether debug information is gathered for inline/template functions.

Syntax

```
#pragma nosyminline on | off | reset
```

Remarks

When on, debug information is not gathered for inline/template functions.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

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 30.10 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
}
```

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.

old_vtable

This pragma is no longer available.

opt_classresults

Controls the omission of the copy constructor call for class return types if all return statements in a function return the same local class object.

Syntax

```
#pragma opt_classresults on | off | reset
```

Remarks

Listing 30.11 shows an example.

Listing 30.11 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`.

parse_func_templ

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler.

Syntax

```
#pragma parse_func_templ on | off | reset
```

Remarks

If you enable this pragma, your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions.

This option actually corresponds to the **ISO C++ Template Parser** option (together with pragmas parse_func_templ and warn_no_typename). By default, this pragma is disabled.

parse_mfunc_templ

Controls whether or not to use the new parser supported by the CodeWarrior 2.5 C++ compiler for member function bodies.

Syntax

```
#pragma parse_mfunc_templ on | off | reset
```

Remarks

If you enable this pragma, member function bodies within your C++ source code is compiled using the newest version of the parser, which is stricter than earlier versions.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

RTTI

Controls the availability of runtime type information.

Syntax

```
#pragma RTTI on | off | reset
```

Remarks

If you enable this pragma, you can use runtime type information (or RTTI) features such as `dynamic_cast` and `typeid`. The other RTTI expressions are available even if you disable the **Enable RTTI** setting. Note that `*type_info::before(const type_info&)` is not implemented.

This pragma corresponds to the **Enable RTTI** setting in the CodeWarrior IDE's **C/C++ Language** settings panel.

suppress_init_code

Controls the suppression of static initialization object code.

Syntax

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

Remarks

If you enable this pragma, the compiler does not generate any code for static data initialization such as C++ constructors.

WARNING! Using this pragma because it can produce erratic or unpredictable behavior in your program.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

template_depth

Controls how many nested or recursive class templates you can instantiate.

```
#pragma template_depth(n)
```

Remarks

This pragma lets you increase the number of nested or recursive class template instantiations allowed. By default, *n* equals 64; it can be set from 1 to 30000. You should always use the default value unless you receive the error message

```
template too complex or recursive
```

This pragma does not correspond to any panel setting.

thread_safe_init

Controls the addition of extra code in the binary to ensure that multiple threads cannot enter a static local initialization at the same time.

Syntax

```
#pragma thread_safe_init on | off | reset
```

Remarks

A C++ program that uses multiple threads and static local initializations introduces the possibility of contention over which thread initializes static local variable first. When the pragma is `on`, the compiler inserts calls to mutex functions around each static local initialization to avoid this problem. The C++ runtime library provides these mutex functions.

Listing 30.12 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.

Listing 30.13 shows another example.

Listing 30.13 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`.

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. Listing 30.14 shows an example.

Listing 30.14 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 14882-1998 C++ Standard does not require this pragma.

NOTE A warning message normally indicates that the pragma name is not recognized, but an error indicates either a syntax problem or that the pragma is not valid in the given context.

This pragma corresponds to the **Hidden Virtual Functions** setting in the CodeWarrior IDE’s **C/C++ Language** settings panel.

warn_no_explicit_virtual

Controls the issuing of warning messages if an overriding function is not declared with a virtual keyword.

Syntax

```
#pragma warn_no_explicit_virtual on | off | reset
```

Remarks

Listing 30.15 shows an example.

Listing 30.15 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 14882-1998 C++ standard, but can help you track down unwanted overrides.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

warn_no_typename

Controls the issuing of warning messages for missing `typenames`.

Syntax

```
#pragma warn_no_typename on | off | reset
```

Remarks

The compiler issues a warning message if a `typenames` required by the C++ standard is missing but can still be determined by the compiler based on the context of the surrounding C++ syntax.

This pragma does not correspond to any panel setting. This pragma is enabled by the ISO/IEC 14882-1998 C++ template parser.

warn_notinlined

Controls the issuing of warning messages for functions the compiler cannot inline.

Syntax

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

Remarks

The compiler issues a warning message for non-inlined inline (i.e., on those indicated by the `inline` keyword or in line in a class declaration) function calls.

This pragma corresponds to the **Non-Inlined Functions** setting in the CodeWarrior IDE's **C/C++ Warnings** settings panel. By default, this pragma is disabled.

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 30.16 Inconsistent use of `class` and `struct`

```
class X;
struct X { int a; }; // WARNING
```

Use this warning when using static or dynamic libraries to link with object code produced by another C++ compiler that distinguishes between class and structure variables in its name "mangling."

This pragma corresponds to the **Inconsistent 'class' / 'struct' Usage** setting in the CodeWarrior IDE's **C/C++ Warnings** settings panel. By default, this pragma is disabled.

wchar_type

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

Syntax

```
#pragma wchar_type on | off | reset
```

Remarks

If you enable this pragma, `wchar_t` is treated as a built-in type. Otherwise, the compiler does not recognize this type.

This pragma corresponds to the **Enable wchar_t Support** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is enabled.

31

Pragmas for Language Translation

asmpoundcomment

Controls whether the “#” symbol is treated as a comment character in inline assembly.

Syntax

```
#pragma asmpoundcomment on | off | reset
```

Remarks

Some targets may have additional comment characters, and may treat these characters as comments even when

```
#pragma asmpoundcomment off
```

is used.

Using this pragma may interfere with the function-level inline assembly language.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

asmsemicolcomment

Controls whether the “`;`” symbol is treated as a comment character in inline assembly.

Syntax

```
#pragma asmsemicolcomment on | off | reset
```

Remarks

Some targets may have additional comment characters, and may treat these characters as comments even when

`#pragma asmsemicolcomment off`

is used.

Using this pragma may interfere with the assembly language of a specific target.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

const_strings

Controls the `const`-ness of character string literals.

Syntax

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

Remarks

If you enable this pragma, the type of string literals is an array `const char[`*n*`]`, or `const wchar_t[`*n*`]` for wide strings, where *n* is the length of the string literal plus 1 for a terminating `NUL` character. Otherwise, the type `char[`*n*`]` or `wchar_t[`*n*`]` is used.

By default, this pragma is `on` when compiling C++ source code and `off` when compiling C source code.

dollar_identifiers

Controls use of dollar signs ($) in identifiers.

Syntax

```
#pragma dollar_identifiers on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts dollar signs (`$`) in identifiers. Otherwise, the compiler issues an error if it encounters anything but underscores, 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`.

gcc_extensions

Controls the acceptance of GNU C language extensions.

Syntax

```
#pragma gcc_extensions on | off | reset
```

Remarks

If you enable this pragma, the compiler accepts GNU C extensions in C source code. This includes the following non-ANSI C extensions:

- Initialization of automatic `struct` or `array` variables with non-`const` values.
- Illegal pointer conversions
- `sizeof( void ) == 1`
- `sizeof( function-type ) == 1`
- Limited support for GCC statements and declarations within expressions.
- Macro redefinitions without a previous `#undef`.
- The GCC keyword `typeof`
- Function pointer arithmetic supported
- `void*` arithmetic supported
- Void expressions in return statements of void
- `__builtin_constant_p (`*`expr`*`)` supported
- Forward declarations of arrays of incomplete type
- Forward declarations of empty static arrays
- Pre-C99 designated initializer syntax (deprecated)
- shortened conditional expression (`c ?: y`)
- `long __builtin_expect (long exp, long c)` now accepted

This pragma corresponds to the **Enable GCC Extensions** setting in the CodeWarrior IDE’s **C/C++ Language** settings panel. By default, this pragma is disabled.

mark

Adds an item to the **Function** pop-up menu in the IDE editor.

Syntax

`#pragma mark` *itemName*

Remarks

This pragma adds *itemName* to the source file's **Function** pop-up menu. If you open the file in the CodeWarrior Editor and select the item from the **Function** pop-up menu, the editor brings you to the pragma. Note that if the pragma is inside a function definition, the item does not appear in the **Function** pop-up menu.

If *itemName* begins with "`--`", a menu separator appears in the IDE's **Function** pop-up menu:

```
#pragma mark --
```

This pragma does not correspond to any panel setting.

mpwc_newline

Controls the use of newline character convention.

Syntax

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

Remarks

If you enable this pragma, the compiler translates `'\n'` as a Carriage Return (0x0D) and `'\r'` as a Line Feed (0x0A). Otherwise, the compiler uses the ISO standard conventions for these characters.

If you enable this pragma, use ISO standard libraries that were compiled when this pragma was enabled.

If you enable this pragma and use the standard ISO standard libraries, your program will not read and write `'\n'` and `'\r'` properly. For example, printing `'\n'` brings your program's output to the beginning of the current line instead of inserting a newline.

This pragma does not correspond to any IDE panel setting. By default, this pragma is disabled.

mpwc_relax

Controls the compatibility of the `char*` and `unsigned char*` types.

Syntax

```
#pragma mpwc_relax on | off | reset
```

Remarks

If you enable this pragma, the compiler treats `char*` and `unsigned char*` as the same type. Use this setting to compile source code written before the ISO C standards. Old source code frequently uses these types interchangeably.

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

NOTE Turning this option on may prevent the compiler from detecting some programming errors. We recommend not turning on this option.

Listing 31.1 shows how to use this pragma to relax function pointer checking.

Listing 31.1 Relaxing function pointer checking

```
#pragma mpwc_relax on
extern void f(char *);

/* Normally an error, but allowed. */
extern void(*fp1)(void *) = &f;

/* Normally an error, but allowed. */
extern void(*fp2)(unsigned char *) = &f;
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

multibyteaware

Controls how the **Source encoding** option in the IDE is treated

Syntax

```
#pragma multibyteaware on | off | reset
```

Remarks

This pragma is deprecated. See `#pragma text_encoding` for more details.

This pragma does not correspond to any panel setting, but the replacement option **Source encoding** appears in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

multibyteaware_preserve_literals

Controls the treatment of multibyte character sequences in narrow character string literals.

Syntax

```
#pragma multibyteaware_preserve_literals on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

text_encoding

Identifies the character encoding of source files.

Syntax

```
#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, `#pragma text_encoding` is only effective through the end of file. To affect the default text encoding assumed for the current and all subsequent files, supply the "global" modifier.

This pragma corresponds to the **Source Encoding** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this setting is `ASCII`.

trigraphs

Controls the use trigraph sequences specified in the ISO standards.

Syntax

```
#pragma trigraphs on | off | reset
```

Remarks

If you are writing code that must strictly adhere to the ANSI standard, enable this pragma.

Table 31.1 Trigraph table

Trigraph	Character
`??=`	`#`
`??/`	`\`
`??'`	`^`
`??(`	`[`
`??)`	`]`
`??!`	`|`
`??<`	`{`

Table 31.1 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 31.2 Example of Pragma trigraphs

```
char c = '????'; /* ERROR: Trigraph sequence expands to '??^ */
char d = '\?\?\?\?'; /* OK */
```

This pragma corresponds to the **Expand Trigraphs** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is disabled.

unsigned_char

Controls whether or not declarations of type `char` are treated as `unsigned char`.

Syntax

```
#pragma unsigned_char on | off | reset
```

Remarks

If you enable this pragma, the compiler treats a `char` declaration as if it were an `unsigned char` declaration.

NOTE If you enable this pragma, your code might not be compatible with libraries that were compiled when the pragma was disabled. In particular, your code might not work with the ISO standard libraries included with CodeWarrior.

This pragma corresponds to the **Use unsigned chars** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this setting is disabled.

32

Pragmas for Diagnostic Messages

extended_errorcheck

Controls the issuing of warning messages for possible unintended logical errors.

Syntax

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

Remarks

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

It also issues a warning message when it encounters a delete operator for a class or structure that has not been defined yet. Listing 32.1 shows an example.

Listing 32.1 Attempting to delete an undefined structure

```
#pragma extended_errorcheck on
struct X;
int func(X *xp)
{
    delete xp;    // Warning: deleting incomplete type X
}
```

- An integer or floating-point value assigned to an `enum` type. Listing 32.2 shows an example.

Listing 32.2 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, Listing 32.3 results in a warning message.

Listing 32.3 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 */
}
```

Listing 32.4 shows how to prevent this warning message.

Listing 32.4 A non-void function with a proper return statement

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err != -1)
  {
    err = GetMoreResources();
  }
  return err; /* OK */
}
```

This pragma corresponds to the **Extended Error Checking** setting in the CodeWarrior IDE's **C/C++ Warnings** settings panel. By default, this setting is `off`.

maxerrorcount

Limits the number of error messages emitted while compiling a single file.

Syntax

`#pragma maxerrorcount(` *`num`* `| off )`

Parameters

num

Specifies the maximum number of error messages issued per source file.

`off`

Does not limit the number of error messages issued per source file.

Remarks

The total number of error messages emitted may include one final message:

`Too many errors emitted`

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

message

Tells the compiler to issue a text message to the user.

Syntax

```
#pragma message( msg )
```

Parameter

msg

Actual message to issue. Does not have to be a string literal.

Remarks

In the CodeWarrior IDE, the message appears in the **Errors & Warnings** window . On the command line, the message is sent to the standard error stream.

This pragma does not correspond to any panel setting.

showmessagenumber

Controls the appearance of warning or error numbers in displayed messages.

Syntax

```
#pragma showmessagenumber on | off | reset
```

Remarks

When enabled, this pragma causes messages to appear with their numbers visible. You can then use the `warning` pragma with a warning number to suppress the appearance of specific warning messages.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

show_error_filestack

Controls the appearance of the current `#include` file stack within error messages occurring inside deeply-included files.

Syntax

```
#pragma show_error_filestack on | off | reset
```

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

suppress_warnings

Controls the issuing of warning messages.

Syntax

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

Remarks

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

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

sym

Controls the generation of debugger symbol information for subsequent functions.

Syntax

```
#pragma sym on | off | reset
```

Remarks

The compiler pays attention to this pragma only if you enable the debug marker for a file in the IDE project window. If you disable this pragma, the compiler does not put debugging information into the source file debugger symbol file (SYM or DWARF) for the functions that follow.

The compiler always generates a debugger symbol file for a source file that has a debug diamond next to it in the IDE project window. This pragma changes only which functions have information in that symbol file.

This pragma does not correspond to any panel setting. By default, this pragma is enabled.

unused

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

Syntax

`#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 32.5 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 32.6 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.

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`.

warning_errors

Controls whether or not warnings are treated as errors.

Syntax

```
#pragma warning_errors on | off | reset
```

Remarks

If you enable this pragma, the compiler treats all warning messages as though they were errors and does not translate your file until you resolve them.

This pragma corresponds to the **Treat All Warnings as Errors** setting in the CodeWarrior IDE's **C/C++ Warnings** settings panel.

warn_any_ptr_int_conv

Controls if the compiler generates a warning message when an integral type is explicitly converted to a pointer type or vice versa.

Syntax

```
#pragma warn_any_ptr_int_conv on | off | reset
```

Remarks

This pragma is useful to identify potential 64-bit pointer portability issues. An example is shown in.

Listing 32.7 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. */
}
```

Remarks

This pragma corresponds to the **Pointer/Integral Conversions** setting in the CodeWarrior IDE's **C/C++ Warnings** settings panel. By default, this pragma is `off`.

warn_emptydecl

Controls the recognition of declarations without variables.

Syntax

`#pragma warn_emptydecl on | off | reset`

Remarks

If you enable this pragma, the compiler displays a warning message when it encounters a declaration with no variables.

Listing 32.8 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 32.9 Example of empty declaration in C++

```
#pragma warn_emptydecl on
extern "C" {
}; /* WARNING */
```

This pragma corresponds to the **Empty Declarations** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is disabled.

warn_extracomma

Controls the recognition of superfluous commas in enumerations.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a trailing comma in enumerations. For example, Listing 32.10 is acceptable source code but generates a warning message when you enable this setting.

Listing 32.10 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** panel. By default, this pragma is disabled.

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`.

This pragma corresponds to the **Include File Capitalization** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is `off`.

warn_filenamecaps_system

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

Syntax

```
#pragma warn_filenamecaps_system on | off | reset
```

Remarks

If you enable this pragma along with `warn_filenamecaps`, the compiler issues a warning message when an `#include` directive capitalizes a filename within a system include differently from the way the filename appears on a disk. It also detects use of "8.3" DOS filenames in Windows® 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.

This pragma corresponds to the **Check System Includes** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is off.

NOTE Some SDKs (Software Developer Kits) use "colorful" capitalization, so this pragma may issue a lot of unwanted messages.

warn_hiddenlocals

Controls the recognition of a local variable that hides another local variable.

Syntax

```
#pragma warn_hiddenlocals on | off | reset
```

Remarks

When `on`, the compiler issues a warning message when it encounters a local variable that hides another local variable. An example appears in Listing 32.11.

Listing 32.11 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`.

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.

This pragma corresponds to the **Illegal Pragmas** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this setting is `off`.

warn_illtokenpasting

Controls whether or not to issue a warning message for improper preprocessor token pasting.

Syntax

```
#pragma warn_illtokenpasting on | off | reset
```

Remarks

An example of this is shown below:

```
#define PTR(x) x##* / PTR(y)
```

Token pasting is used to create a single token. In this example, `y` and `x` cannot be combined. Often the warning message indicates the macros uses "`##`" unnecessarily.

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

warn_illunionmembers

Controls whether or not to issue a warning message 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`.

warn_impl_f2i_conv

Controls the issuing of warning messages for implicit `float`-to-`int` conversions.

Syntax

`#pragma warn_impl_f2i_conv on | off | reset`

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting floating-point values to integral values. Listing 32.12 provides an example.

Listing 32.12 Example of Implicit `float`-to-`int` Conversion

```
#pragma warn_impl_f2i_conv on

float f;
signed int si;

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

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

This pragma corresponds to the **Float to Integer** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is `on`.

warn_impl_i2f_conv

Controls the issuing of warning messages for implicit `int`-to-`float` conversions.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting integral values to floating-point values. Listing 32.13 shows an example.

Listing 32.13 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 */

}
```

This pragma corresponds to the **Integer to Float** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is `off`.

warn_impl_s2u_conv

Controls the issuing of warning messages for implicit conversions between the `signed int` and `unsigned int` data types.

Syntax

```
#pragma warn_impl_s2u_conv on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message for implicitly converting either from `signed int` to `unsigned int` or vice versa. Listing 32.14 provides an example.

Listing 32.14 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 */
}
```

This pragma corresponds to the **Signed / Unsigned** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is enabled.

warn_implicitconv

Controls the issuing of warning messages for all implicit arithmetic conversions.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message for all implicit arithmetic conversions when the destination type might not represent the source value. Listing 32.15 provides an example.

Listing 32.15 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.

This pragma corresponds to the **Implicit Arithmetic Conversions** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is `off`.

warn_largeargs

Controls the issuing of warning messages for passing non-"int" numeric values to unprototyped functions.

Syntax

`#pragma warn_largeargs on | off | reset`

Remarks

If you enable this pragma, the compiler issues a warning message if you attempt to pass a non-integer numeric value, such as a `float` or `long long`, to an unprototyped function when the `require_prototypes` pragma is disabled.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

warn_missingreturn

Issues a warning message when a function that returns a value is missing a `return` statement.

Syntax

```
#pragma warn_missingreturn on | off | reset
```

Remarks

An example is shown in Listing 32.16.

Listing 32.16 Example of warn_missingreturn pragma

```
#pragma warn_missingreturn on

int func()
{
  /* WARNING: no return statement. */
}
```

This pragma corresponds to the **Missing 'return' Statements** setting in the CodeWarrior IDE's **C/C++ Warnings** panel.

warn_no_side_effect

Controls the issuing of warning messages for redundant statements.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a statement that produces no side effect. To suppress this warning message, cast the statement with `(void)`. Listing 32.17 provides an example.

Listing 32.17 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. */
}
```

This pragma corresponds to the **Expression Has No Side Effect** panel setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is `off`.

warn_padding

Controls the issuing of warning messages for data structure padding.

Syntax

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

Remarks

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

This pragma corresponds to the **Pad Bytes Added** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this setting is `off`.

warn_pch_portability

Controls whether or not to issue a warning message when `#pragma once on` 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`.

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 `==`. Listing 32.18 shows an example.

Listing 32.18 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 `=`. Listing 32.19 shows an example.

Listing 32.19 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, Listing 32.20 generates a warning message.

Listing 32.20 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 Listing 32.21 suppress the above error or warning messages.

Listing 32.21 Intentional empty statements

```
while (i++) ; /* OK: White space separation. */
while (i++) /* OK: Comment separation */ ;
```

This pragma corresponds to the **Possible Errors** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is `off`.

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 32.22 Example for #pragma warn_ptr_int_conv

```
#pragma warn_ptr_int_conv on

char *my_ptr;
char too_small = (char)my_ptr;   /* WARNING: char is too small. */
```

See also "warn_any_ptr_int_conv" on page 439.

This pragma corresponds to the **Pointer / Integral Conversions** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is off.

warn_resultnotused

Controls the issuing of warning messages when function results are ignored.

Syntax

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

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a statement that calls a function without using its result. To prevent this, cast the statement with `(void)`. Listing 32.23 provides an example.

Listing 32.23 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.

warn_undefmacro

Controls the detection of undefined macros in `#if` and `#elif` directives.

Syntax

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

Remarks

Listing 32.24 provides an example.

Listing 32.24 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 defined first.

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

This pragma corresponds to the **Undefined Macro in #if** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is off.

warn_uninitializedvar

Controls the compiler to perform some dataflow analysis and emits warning messages whenever local variables are initialized before being used.

Syntax

```
#pragma warn_uninitializedvar on | off | reset
```

Remarks

This pragma has no corresponding setting in the CodeWarrior IDE. By default, this pragma is on.

warn_unusedarg

Controls the recognition of unreferenced arguments.

Syntax

```
#pragma warn_unusedarg on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters an argument you declare but do not use.

This check helps you find arguments that you either misspelled or did not use in your program. Listing 32.25 shows an example.

Listing 32.25 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 Listing 32.26.

Listing 32.26 Using pragma unused() to prevent unused argument messages

```
void func(int temp, int error)
{
  #pragma unused (temp)
  /* Compiler does not warn that temp is not used. */

  error=do_something();
}
```

- Do not give the unused argument a name. Listing 32.27 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 32.27 Unused, Unnamed Arguments

```
void func(int /* temp */, int error)
{
  /* Compiler does not warn that "temp" is not used. */

  error=do_something();
}
```

This pragma corresponds to the **Unused Arguments** setting in the **C/C++ Warnings Panel**. By default, this pragma is `off`.

warn_unusedvar

Controls the recognition of unreferenced variables.

Syntax

```
#pragma warn_unusedvar on | off | reset
```

Remarks

If you enable this pragma, the compiler issues a warning message when it encounters a variable you declare but do not use.

This check helps you find variables that you either misspelled or did not use in your program. Listing 32.28 shows an example.

Listing 32.28 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 Listing 32.29.

Listing 32.29 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 */
}
```

This pragma corresponds to the **Unused Variables** setting in the CodeWarrior IDE's **C/C++ Warnings** panel. By default, this pragma is off.

33

Pragmas for Preprocessing

check_header_flags

Controls whether or not to ensure that a precompiled header's data matches a project's target settings.

Syntax

```
#pragma check_header_flags on | off | reset
```

Remarks

This pragma affects precompiled headers only.

If you enable this pragma, the compiler verifies that the precompiled header's preferences for `double` size, `int` size, and floating point math correspond to the build target's settings. If they do not match, the compiler generates an error message.

If your precompiled header file depends on these settings, enable this pragma. Otherwise, disable it.

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `off`.

faster_pch_gen

Controls the performance of precompiled header generation.

Syntax

```
#pragma faster_pch_gen on | off | reset
```

Remarks

If you enable this pragma, generating a precompiled header can be much faster, depending on the header structure. However, the precompiled file can also be slightly larger.

This pragma does not correspond to any panel setting. By default, this setting is `off`.

flat_include

Controls whether or not to ignore relative path names in `#include` directives.

Syntax

```
#pragma flat_include on | off | reset
```

Remarks

For example, when `on`, the compiler converts this directive

```
#include <sys/stat.h>
```

to

```
#include <stat.h>
```

Use this pragma when porting source code from a different operating system, or when a CodeWarrior IDE project's access paths cannot reach a given file.

By default, this pragma is `off`.

fullpath_file

Controls if `__FILE__` macro expands to a full path or the base file name.

Syntax

```
#pragma fullpath_file on | off | reset
```

Remarks

When this pragma `on`, the `__FILE__` macro returns a full path to the file being compiled, otherwise it returns the base file name.

fullpath_prepdump

Shows the full path of included files in preprocessor output.

Syntax

```
#pragma fullpath_prepdump on | off | reset
```

Remarks

If you enable this pragma, the compiler shows the full paths of files specified by the `#include` directive as comments in the preprocessor output. Otherwise, only the file name portion of the path appears.

This pragma corresponds to the **Show full paths** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

keepcomments

Controls whether comments are emitted in the preprocessor output.

Syntax

```
#pragma keepcomments on | off | reset
```

Remarks

This pragma corresponds to the **Keep comments** option CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

line_prepdump

Shows `#line` directives in preprocessor output.

Syntax

```
#pragma line_prepdump on | off | reset
```

Remarks

If you enable this pragma, `#line` directives appear in preprocessing output. The compiler also adjusts line spacing by inserting empty lines.

Use this pragma with the command-line compiler's `-E` option to make sure that `#line` directives are inserted in the preprocessor output.

This pragma corresponds to the **Use #line** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

macro_prepdump

Controls whether the compiler emits `#define` and `#undef` directives in preprocessing output.

Syntax

```
#pragma macro_prepdump on | off | reset
```

Remarks

Use this pragma to help unravel confusing problems like macros that are aliasing identifiers or where headers are redefining macros unexpectedly.

msg_show_lineref

Controls diagnostic output involving `#line` directives to show line numbers specified by the `#line` directives in error and warning messages.

Syntax

```
#pragma msg_show_lineref on | off | reset
```

Remarks

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `on`.

msg_show_realref

Controls diagnostic output involving `#line` directives to show actual line numbers in error and warning messages.

Syntax

```
#pragma msg_show_realref on | off | reset
```

Remarks

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `on`.

notonce

Controls whether or not the compiler lets included files be repeatedly included, even with `#pragma once on`.

Syntax

```
#pragma notonce
```

Remarks

If you enable this pragma, files can be repeatedly `#included`, even if you have enabled `#pragma once on`. For more information, see "once" on page 461.

This pragma does not correspond to any CodeWarrior IDE panel setting.

old_pragma_once

This pragma is no longer available.

once

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

Syntax

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

Remarks

Use this pragma to ensure that the compiler includes header files only once in a source file. This pragma is especially useful in precompiled header files.

There are two versions of this pragma:

```
#pragma once
```

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. When a `once` option or pragma is used, a header file of same name in another directory is not included.

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`.

pop, push

Saves and restores pragma settings.

Syntax

```
#pragma push
```

```
#pragma pop
```

Remarks

The pragma `push` saves all the current pragma settings. The pragma `pop` restores all the pragma settings that resulted from the last `push` pragma. For example, see Listing 33.1.

Listing 33.1 push and pop example

```
#pragma ANSI_strict on
#pragma push /* Saves all compiler settings. */
#pragma ANSI_strict off
#pragma pop /* Restores ANSI_strict to on. */
```

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

TIP Pragmas directives that accept `on | off | reset` already form a stack of previous option values. It is not necessary to use `#pragma pop` or `#pragma push` with such pragmas.

pragma_prepdump

Controls whether pragma directives in the source text appear in the preprocessing output.

Syntax

```
#pragma pragma_prepdump on | off | reset
```

Remarks

This pragma corresponds to the **Emit #pragmas** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

TIP When submitting bug reports with a preprocessor dump, be sure this option is enabled.

precompile_target

Specifies the file name for a precompiled header file.

Syntax

```
#pragma precompile_target filename
```

Parameters

`filename`

A simple file name or an absolute path name. If *filename* is a simple file name, the compiler saves the file in the same folder as the source file. If *filename* is a path name, the compiler saves the file in the specified folder.

Remarks

If you do not specify the file name, the compiler gives the precompiled header file the same name as its source file.

Listing 33.2 shows sample source code from a precompiled header source file. By using the predefined symbols `__cplusplus` and the pragma `precompile_target`, the compiler can use the same source code to create different precompiled header files for C and C++.

Listing 33.2 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.

simple_prepdump

Controls the suppression of comments in preprocessing output.

Syntax

`#pragma simple_prepdump on | off | reset`

Remarks

By default, the compiler adds comments about the current include file being in preprocessing output. Enabling this pragma disables these comments.

This pragma corresponds to the **Emit file changes** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

space_prepdump

Controls whether or not the compiler removes or preserves whitespace in the preprocessor's output.

Syntax

`#pragma space_prepdump on | off | reset`

Remarks

This pragma is useful for keeping the starting column aligned with the original source code, though the compiler attempts to preserve space within the line. This pragma does not apply to expanded macros.

This pragma corresponds to the **Keep whitespace** option in the CodeWarrior IDE's **C/C++ Preprocessor** settings panel. By default, this pragma is `off`.

srcrelincludes

Controls the lookup of `#include` files.

Syntax

```
#pragma srcrelincludes on | off | reset
```

Remarks

When `on`, the compiler looks for `#include` files relative to the previously included file (not just the source file). When `off`, the compiler uses the CodeWarrior IDE's access paths or the access paths specified with the `-ir` option.

Use this pragma when multiple files use the same file name and are intended to be included by another header file in that directory. This is a common practice in UNIX programming.

This pragma corresponds to the **Source-relative includes** option in the **Access Paths** panel. By default, this pragma is `off`.

syspath_once

Controls how included files are treated when `#pragma once` is enabled.

Syntax

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

Remarks

When this pragma and pragma `once` are set to `on`, the compiler distinguishes between identically-named header files referred to in

`#include <`*`file-name`*`>`

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 33.3 Pragma syspath_once example

```
#pragma syspath_once off
#pragma once on /* Include all subsequent files only once. */
#include "sock.h"
#include <sock.h> /* Skipped because syspath_once is off. */
```

34

Pragmas for Library and Linking

always_import

Controls whether or not `#include` directives are treated as `#pragma import` directives.

Syntax

```
#pragma always_import on | off | reset
```

Remarks

If you enable this pragma, the compiler treats all `#include` statements as `#pragma import` statements.

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `off`.

export

Controls the exporting of data and functions to be accessible from outside a program or library.

Syntax

```
#pragma export on | off | reset
#pragma export list name1 [, name2, ...]
```

`name1, name2`

Names of functions or global variables to export.

Remarks

When using the `#pragma export on` format, all functions in the source file being compiled will be accessible from outside the program or library that the compiler and linker are building.

Use the `#pragma export list` format to specify global variables and functions for exporting. In C++, this form of the pragma applies to all variants of an overloaded function. You cannot use this pragma for C++ member functions or static class members. Listing 34.1 shows an example:

Listing 34.1 Example of an Exported List

```
extern int f(),g;
#pragma export list f,g
```

import

Controls the importing of global data or functions.

Syntax

`#pragma import on | off | reset`

`#pragma import list` *`name1`* `[,` *`name2`*`, ...]`

`name1, name2`

Names of functions or global variables to import.

Remarks

When using the `#pragma import on` format, all functions are automatically imported.

Use the `#pragma import list` format to specify data or functions for importing. In C++, this form of the pragma applies to all variants of an overloaded function. You cannot use this pragma for C++ member functions or static class members.

Listing 34.2 shows an example:

Listing 34.2 Example of an Imported List

```
extern int f(),g;
#pragma import list f,g
```

This pragma does not correspond to any CodeWarrior IDE panel setting. By default, this pragma is `off`.

lib_export

Controls the exporting of data or functions.

Syntax

`#pragma lib_export on | off | reset`

`#pragma lib_export list` *name1* `[,` *name2* `]*`

Remarks

When using the `#pragma lib_export on` format, the linker marks all data and functions that are within the pragma's scope for export.

Use the `#pragma lib_export list` format to tag specific data or functions for exporting. In C++, this form of the pragma applies to all variants of an overloaded function. You cannot use this pragma for C++ member functions or static class members.

Listing 34.3 shows an example:

Listing 34.3 Example of a `lib_export` List

```
extern int f(),g;
#pragma lib_export list f,g
```

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

35 Pragmas for Code Generation

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.

dont_reuse_strings

Controls whether or not to store identical character string literals separately in object code.

Syntax

```
#pragma dont_reuse_strings on | off | reset
```

Remarks

Normally, C and C++ programs should not modify character string literals. Enable this pragma if your source code follows the unconventional practice of modifying them.

If you enable this pragma, the compiler separately stores identical occurrences of character string literals in a source file.

If this pragma is disabled, the compiler stores a single instance of identical string literals in a source file. The compiler reduces the size of the object code it generates for a file if the source file has identical string literals.

The compiler always stores a separate instance of a string literal that is used to initialize a character array. Listing 35.1 shows an example.

Although the source code contains 3 identical string literals, `"cat"`, the compiler will generate 2 instances of the string in object code. The compiler will initialize `str1` and `str2` to point to the first instance of the string and will initialize `str3` to contain the second instance of the string.

Using `str2` to modify the string it points to also modifies the string that `str1` points to. The array `str3` may be safely used to modify the string it points to without inadvertently changing any other strings.

This pragma corresponds to the **Reuse Strings** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is `off`.

Listing 35.1 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.
}
```

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`.

Listing 35.2 shows an example.

Listing 35.2 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. */
```

This pragma corresponds to the **Enums Always Int** setting in the CodeWarrior IDE's **C/C++ Language** settings panel. By default, this pragma is `off`.

errno_name

Tells the optimizer how to find the `errno` identifier.

Syntax

```
#pragma errno_name id | ...
```

Remarks

When this pragma is used, the optimizer can use the identifier `errno` (either a macro or a function call) to optimize standard C library functions better. If not used, the optimizer makes worst-case assumptions about the effects of calls to the standard C library.

NOTE The MSL C library already includes a use of this pragma, so you would only need to use it for third-party C libraries.

If `errno` resolves to a variable name, specify it like this:

```
#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).

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`.

float_constants

Controls how floating pointing constants are treated.

Syntax

```
#pragma float_constants on | off | reset
```

Remarks

If you enable this pragma, the compiler assumes that all unqualified floating point constant values are of type `float`, not `double`. This pragma is useful when porting source code for programs optimized for the “`float`” rather than the “`double`” type.

When you enable this pragma, you can still explicitly declare a constant value as double by appending a “D” suffix.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

instmgr_file

Controls where the instance manager database is written, to the target data directory or to a separate file.

Syntax

```
#pragma instmgr_file "name"
```

Remarks

When the **Use Instance Manager** option is on, the IDE writes the instance manager database to the project's data directory. If the `#pragma instmgr_file` is used, the database is written to a separate file.

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.

NOTE

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.

longlong_enums

Controls whether or not enumerated types may have the size of the `long long` type.

Syntax

```
#pragma longlong_enums on | off | reset
```

Remarks

This pragma lets you use enumerators that are large enough to be `long long` integers. It is ignored if you enable the `enumsalwaysint` pragma (described in "enumsalwaysint" on page 472).

This pragma does not correspond to any panel setting. By default, this setting is enabled.

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.

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.

This pragma corresponds to the **Pool Strings** setting in the CodeWarrior IDE's **C/ C++ Language** settings panel.

readonly_strings

Controls whether string objects are placed in a read-write or a read-only data section.

Syntax

```
#pragma readonly_strings on | off | reset
```

Remarks

If you enable this pragma, 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 `const char *`, even though their type is really `char *`.

This pragma does not correspond to any IDE panel setting.

reverse_bitfields

Controls whether or not the compiler reverses the bitfield allocation.

Syntax

```
#pragma reverse_bitfields on | off | reset
```

Remarks

This pragma reverses the bitfield allocation, so that bitfields are arranged from the opposite side of the storage unit from that ordinarily used on the target. The compiler still orders the bits within a single bitfield such that the lowest-valued bit is in the right-most position.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

NOTE 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

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`.

36

Pragmas for Optimization

global_optimizer

Controls whether the Frontend IR Optimizer is invoked by the compiler.

Syntax

```
#pragma global_optimizer on | off | reset
```

Remarks

In most compilers, this #pragma determines whether the Frontend IR Optimizer is invoked. If disabled, only simple optimizations and back-end optimizations are performed.

NOTE This is not the same as `#pragma optimization_level`. The Frontend IR Optimizer is invoked even at `optimization_level 0` if `#pragma global_optimizer` is enabled.

This pragma does not correspond to any panel setting. By default, this setting is `on`.

ipa

Specifies how to apply interprocedural analysis optimizations.

Syntax

```
#pragma ipa  file | on | function | off
```

Remarks

See "Interprocedural Analysis" on page 281.

Place this pragma at the beginning of a source file, before any functions or data have been defined. There arelevels of interprocedural analysis:

-
- 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.

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.

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.

opt_dead_assignments

Controls the use of dead store optimization.

Syntax

```
#pragma opt_dead_assignments on | off | reset
```

Remarks

If you enable this pragma, the compiler removes assignments to unused variables before reassigning them.

This pragma does not correspond to any panel setting. By default, this settings is related to the "global_optimizer" on page 479 level.

opt_dead_code

Controls the use of dead code optimization.

Syntax

```
#pragma opt_dead_code on | off | reset
```

Remarks

If you enable this pragma, the compiler removes a statement that other statements never execute or call.

This pragma does not correspond to any panel setting. By default, this settings is related to the "global_optimizer" on page 479 level.

opt_lifetimes

Controls the use of lifetime analysis optimization.

Syntax

```
#pragma opt_lifetimes on | off | reset
```

Remarks

If you enable this pragma, the compiler uses the same processor register for different variables that exist in the same routine but not in the same statement.

This pragma does not correspond to any panel setting. By default, this settings is related to the "global_optimizer" on page 479 level.

opt_loop_invariants

Controls the use of loop invariant optimization.

Syntax

```
#pragma opt_loop_invariants on | off | reset
```

Remarks

If you enable this pragma, the compiler moves all computations that do not change inside a loop outside the loop, which then runs faster.

This pragma does not correspond to any panel setting.

opt_propagation

Controls the use of copy and constant propagation optimization.

Syntax

```
#pragma opt_propagation on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces multiple occurrences of one variable with a single occurrence.

This pragma does not correspond to any panel setting. By default, this settings is related to the "global_optimizer" on page 479 level.

opt_strength_reduction

Controls the use of strength reduction optimization.

Syntax

```
#pragma opt_strength_reduction on | off | reset
```

Remarks

If you enable this pragma, the compiler replaces array element arithmetic instructions with pointer arithmetic instructions to make loops faster.

This pragma does not correspond to any panel setting. By default, this settings is related to the "global_optimizer" on page 479 level.

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.

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" on page 479 level.

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`.

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.

These pragmas correspond to the settings in the **Global Optimizations** panel. By default, this pragma is disabled.

optimize_for_size

Controls optimization to reduce the size of object code.

```
#pragma optimize_for_size on | off | reset
```

Remarks

This setting lets you choose what the compiler does when it must decide between creating small code or fast code. If you enable this pragma, the compiler creates smaller object code at the expense of speed. It also ignores the `inline` directive and generates function calls to call any function declared `inline`. If you disable this pragma, the compiler creates faster object code at the expense of size.

The pragma corresponds to the **Optimize for Size** setting on the **Global Optimizations** panel.

optimizewithasm

Controls optimization of assembly language.

Syntax

```
#pragma optimizewithasm on | off | reset
```

Remarks

If you enable this pragma, the compiler also optimizes assembly language statements in C/C++ source code.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

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.

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`.

37

Pragmas for Power Architecture Compiler

This chapter describes the pragmas that control how the compiler translates source code into instructions and data for Power Architecture processors.

Diagnostic Pragmas

incompatible_return_small_structs

Warns when returning structures using the R3 and R4 registers.

Syntax

```
#pragma incompatible_return_small_structs on | off | reset
```

Remarks

This pragma makes CodeWarrior-built object files more compatible with those created using a GNU compiler.

The PowerPC EABI specifies that structures that are up to 8 bytes in size should be in registers `R3` and `R4`, while larger structures are returned by accessing a hidden argument in `R3`. GCC compilers always uses the hidden argument method regardless of size.

The CodeWarrior linker checks to see if you are including objects in your project that have incompatible EABI settings. If you do, a warning message is issued.

NOTE Different versions of GCC compilers may fix these incompatibilities, so you should check your version if you will be mixing GCC and CodeWarrior objects.

incompatible_sfpe_double_params

Warns when skipping registers for `double` data types.

Syntax

```
#pragma incompatible_sfpe_double_params on | off | reset
```

Remarks

This pragma makes CodeWarrior-built object files more compatible with those created with a GCC compiler.

The PowerPC EABI states that software floating-point parameters of type `double` always begin on an odd register. In other words, in the function

```
void red(long a, double b)
```

`a` is passed in register `R3` and `b` is passed in registers `R5` and `R6` (effectively skipping `R4`). GCC compilers do not skip registers if objects of type `double` are passed (although it does skip them for values of type `long long`).

The CodeWarrior linker checks to see if you are including objects in your project that have incompatible EABI settings. If you do, a warning message is issued.

NOTE Different versions of GCC compilers may fix these incompatibilities, so you should check your version if you will be mixing GCC and CodeWarrior object code.

Debugging Information Pragmas

dwarf2typedefchains

Generates DWARF2 debugging information `typedef` statements.

Syntax

```
#pragma dwarf2typedefchains on | off | reset
```

Remarks

When this pragma is `on`, the compiler generates DWARF2 debugging information for type definitions.

By default, this option is `off`.

dwarf2lexblockcodeaddrcalc

Calculates the first and the last instruction in a lexical block by inspecting the entire lexical block start and end lines..

Syntax

```
#pragma dwarf2lexblockcodeaddrcalc on | off | reset
```

Remarks

When this pragma is off, the compiler looks for the instruction following the lexical block start and end but, does not emit the correct address range for unparenthesiszed loop expressions.

By default, this option is off.

Library and Linking Pragmas

force_active

Deprecated.

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 `FORCEACTIVE` command.

prepare_compress

Ensures that generated code is suitable for compression by a post-link tool.

Syntax

```
#pragma prepare_compress on | off | reset
```

Remarks

This pragma prepares object code to be compressed for Power Architecture processors that have on-chip decompression features.

section

This sophisticated and powerful pragma lets you arrange compiled object code into predefined sections and sections you define.

NOTE Deprecated only when used without an associated `__declspec(section)`. To avoid C++ parsing ambiguities and other possible inadvertent errors, use `__declspec(section)` instead.

Syntax

```
#pragma section [ objecttype | permission ][iname][uname]
      [data_mode=datamode][code_mode=codemode]
```

Parameter

`objecttype`

specifies where types of object data are stored. It may be one or more of these values:

- `code_type` — executable object code
- `data_type` — non-constant data of a size greater than the size specified in the small data threshold option in the EPPC Target settings panel
- `sdata_type` — non-constant data of a size less than or equal to the size specified in the small data threshold option in the EPPC Target settings panel
- `const_type` — constant data of a size greater than the size specified in the small const data threshold option in the EPPC Target settings panel
- `sconst_type` — constant data of a size less than or equal to the size specified in the small const data threshold option in the EPPC Target settings panel
- `all_types` — all code and data

Specify one or more of these object types without quotes separated by spaces.

The CodeWarrior C/C++ compiler generates some of its own data, such as exception and static initializer objects, which are not affected by `#pragma section`.

NOTE To classify character strings, the CodeWarrior C/C++ compiler uses the setting of the Make Strings Read Only checkbox in the EPPC Processor settings panel. If the checkbox is checked, character strings are stored in the same section as data of type const_type. If the checkbox is clear, strings are stored in the same section as data for data_type.

`permission`

specifies access permission. It may be one or more of these values:

- `R` — read only permission
- `W` — write permission
- `X` — execute permission

Specify one or more of these object types without quotes separated by spaces.

For more information on access permission, see "Section access permissions" on page 495.

`iname`

specifies the name of the section where the compiler stores initialized objects. Variables that are initialized at the time they are defined, functions, and character strings are examples of initialized objects.

The `iname` parameter may be of the form `.abs.xxxxxxxx` where `xxxxxxxx` is an 8-digit hexadecimal number specifying the address of the section.

`uname`

specifies the name of the section where the compiler stores uninitialized objects. This parameter is required for sections that have data objects. The `uname` parameter value may be a unique name or it may be the name of any previous `iname` or `uname` section. If the `uname` section is also an `iname` section then uninitialized data is stored in the same section as initialized objects.

The special `uname` `COMM` specifies that uninitialized data will be stored in the common section. The linker will put all common section data into the "`.bss`" section. When the **Use Common Section** checkbox is checked in the EPPC Processor panel, `COMM` is the default uname for the `.data` section. If the **Use Common Section** checkbox is clear, `.bss` is the default name of `.data` section.

The `uname` parameter value may be changed. For example, you may want most uninitialized data to go into the `.bss` section while specific variables be stored in the `COMM` section.

Listing 37.1 shows an example where specific uninitialized variables are stored in the `COMM` section.

Listing 37.1 Storing Uninitialized Data in the COMM Section

```
#pragma push // save the current state
#pragma section ".data" "COMM"
int red;
int sky;

#pragma pop // restore the previous state
```

NOTE You may not use any of the object types, data modes, or code modes as the names of sections. Also, you may not use pre-defined section names in the PowerPC EABI for your own section names.

`data_mode=datamode`

specifies the compiler for the kind of addressing mode to be used for referring to data objects for a section.

The permissible addressing modes for `datamode` are:

- `near_abs` — objects must be within the range -65,536 bytes to 65,536 bytes (16 bits on each side)
- `far_abs` — objects must be within the first 32 bits of RAM
- `sda_rel` — objects must be within a 32K range of the linker-defined small data base address

  The `sda_rel` addressing mode may only be used with the "`.sdata`", "`.sbss`", "`.sdata2`", "`.sbss2`", "`.PPC.EMB.sdata0`", and "`.PPC.EMB.sbss0`" sections.

The default addressing mode for large data sections is `far_abs`. The default addressing mode for the predefined small data sections is `sda_rel`.

Specify one of these addressing modes without quotes.

`code_mode=codemode`

specifies the compiler for the kind of addressing mode to be used for referring to executable routines of a section.

The permissible addressing modes for `codemode` are:

- `pc_rel` — routines must be within plus or minus 24 bits of where pc_rel is called from
- `near_abs` — routines must be within the first 24 bits of memory address space
- `far_abs` — routines must be within the first 32 bits of memory address space

The default addressing mode for executable code sections is `pc_rel`.

Specify one of these addressing modes without quotes.

NOTE All sections have a data addressing mode (data_mode=datamode) and a code addressing mode (code_mode=codemode). Although the CodeWarrior C/C++ compiler for PowerPC embedded allows you to store executable code in data sections and data in executable code sections, this practice is not encouraged.

Remarks

CodeWarrior compilers generate their own data, such as exception and static initializer objects, which the `#pragma section` statement does not affect.

Section access permissions

When you define a section by using `#pragma section`, its default access permission is read only. Changing the definition of the section by associating an object type with it sets the appropriate access permissions for you. The compiler adjusts the access permission to allow the storage of newly-associated object types while continuing to allow objects of previously-allowed object types. For example, associating `code_type` with a section adds execute permission to that section. Associating `data_type`, `sdata_type`, or `sconst_type` with a section adds write permission to that section.

Occasionally you might create a section without associating it with an object type. You might do so to force an object into a section with the `__declspec` keyword. In this case, the compiler automatically updates the access permission for that section to allow the object to be stored in the section, then issue a warning. To avoid such a warning, make sure to give the section the proper access permissions before storing object code or data into it. As with associating an object type to a section, passing a specific permission adds to the permissions that a section already has.

Predefined sections and default sections

When an object type is associated with the predefined sections, the sections are set as default sections for that object type. After assigning an object type to a non-standard section, you may revert to the default section with one of the forms in "Forms for #pragma section" on page 497.

The compiler predefines the sections in Listing 37.2.

Listing 37.2 Predefined sections

```
#pragma section code_type ".text" data_mode=far_abs code_mode=pc_rel

#pragma section data_type ".data" ".bss" data_mode=far_abs
code_mode=pc_rel

#pragma section const_type ".rodata" ".rodata" data_mode=far_abs
code_mode=pc_rel

#pragma section sdata_type ".sdata" ".sbss" data_mode=sda_rel
code_mode=pc_rel

#pragma section sconst_type ".sdata2" ".sbss2" data_mode=sda_rel
code_mode=pc_rel

#pragma section ".PPC.EMB.sdata0" ".PPC.EMB.sbss0" data_mode=sda_rel
code_mode=pc_rel

#pragma section RX ".init" ".init" data_mode=far_abs code_mode=pc_rel
```

NOTE The `.PPC.EMB.sdata0` and `.PPC.EMB.sbss0` sections are predefined as an alternative to the `sdata_type` object type. The `.init` section is also predefined, but it is not a default section. The `.init` section is used for startup code.

Forms for #pragma section

`#pragma section ".name1"`

This form simply creates a section called `.name1` if it does not already exist. With this form, the compiler does not store objects in the section without an appropriate, subsequent `#pragma section` statement or an item defined with the `__declspec` keyword. If only one section name is specified, it is considered the name of the initialized object section, iname. If the section is already declared, you may also optionally specify the uninitialized object section, uname. If you know that the section must have read and write permission, use `#pragma section RW .name1` instead, especially if you use the `__declspec` keyword.

`#pragma section objecttype ".name2"`

With the addition of one or more object types, the compiler stores objects of the types specified in the section `.name2`. If `.name2` does not exist, the compiler creates it with the appropriate access permissions. If only one section name is specified, it is considered the name of the initialized object section, `iname`. If the section is already declared, you may also optionally specify the uninitialized object section, `uname.`

`#pragma section objecttype`

When there is no `iname` parameter, the compiler resets the section for the object types specified to the default section. Resetting the section for an object type does not reset its addressing modes. You must reset them.

When declaring or setting sections, you also can add an uninitialized section to a section that did not have one originally by specifying a `uname` parameter. The corresponding uninitialized section of an initialized section may be the same.

Forcing individual objects into specific sections

You may store a specific object of an object type into a section other than the current section for that type without changing the current section. Use the `__declspec` keyword with the name of the target section and put it next to the extern declaration or static definition of the item you want to store in the section.

Listing 37.3 shows examples.

Listing 37.3 Using __declspec to Force Objects into Specific Sections

```
__declspec(section ".data") extern int myVar;
#pragma section "constants"
__declspec(section "constants") const int myConst = 0x12345678
```

Using #pragma section with #pragma push and #pragma pop

You can use this pragma with `#pragma push` and `#pragma pop` to ease complex or frequent changes to sections settings.

See Listing 37.1 for an example.

NOTE The `#pragma pop` does not restore any changes to the access permissions of sections that exist before or after the corresponding `#pragma push`.

Code Generation Pragmas

alignment_metrowerks

Determines which structure alignment policy to use.

Syntax

```
#pragma alignment_metrowerks on | off | reset
```

Remarks

This pragmas specifies which policy that

```
#pragma options align=power
```

will use for aligning data in structures. When this pragma is on, then this directive

```
#pragma options align=power
```

is equivalent to this directive

```
#pragma options align=power_mw
```

When this pragma is off, then this directive

```
#pragma options align=power
```

is equivalent to this directive

```
#pragma options align=power_gcc
```

altivec_codegen

Controls the use Power Architecture AltiVec™ instructions during optimization.

Syntax

```
#pragma altivec_codegen on | off | reset
```

Remarks

If this pragma is `on`, the compiler uses Power Architecture AltiVec instructions, if possible, during optimization. When this pragma is `off`, the pragma `altivec_model` is also set to `off`.

altivec_model

Controls the use Power Architecture AltiVec™ language extensions.

Syntax

```
#pragma altivec_model on | off | reset
```

Remarks

If you enable this pragma, the compiler allows language extensions to take advantage of the AltiVec instructions available on some Power Architecture processors. When this pragma is `on` the compiler sets `#pragma altivec_codegen` to `on` and defines the `__VEC__` preprocessor directive.

altivec_pim_warnings

Controls how the compiler translates literal values for AltiVec vectors.

Syntax

```
#pragma altivec_pim_warnings on | off | reset
```

Remarks

When this pragma is `on`, the compiler follows the syntax rules described in *AltiVec™ Technology Programming Interface Manual* ("PIM") to specify literal values for vector objects. This syntax specifies these requirements:

- vector values must be enclosed in parentheses
- vector values must be preceded by a type specifier

When this pragma is `off`, the compiler expects a style more consistent with C source code conventions:

- vector values must be enclosed in braces
- vector values do not need to be preceded by a type specifier

Listing 37.4 shows an example.

Listing 37.4 Example of using altivec_pim_warnings

```
#pragma altivec_pim_warnings on
vector signed int vsi1 = (__vector signed int)(1, 2, 3, 4);

#pragma altivec_pim_warnings off
vector signed int vsi2 = {1, 2, 3, 4};
```

altivec_vrsave

Controls which AltiVec™ registers to save to the stack between function calls.

Syntax

```
#pragma altivec_vrsave on | off | reset | allon
```

Parameter

`allon`

Tells the compiler to set all bits in the `VRSAVE` register.

Remarks

When generating instructions to call a function, the compiler uses the `VRSAVE` register's contents to determine which AltiVec registers should be saved to and restored from the stack.

When this pragma is `on`, the compiler determines which AltiVec registers a function uses. It generates instructions to record this information in the `VRSAVE` register.

When this pragma is `off`, the compiler clears the `VRSAVE` register and consequently does not save AltiVec registers on the stack.

When this pragma is set to `allon`, the compiler sets the `VRSAVE` register to specify that all AltiVec registers should be saved and restored on the stack.

b_range

Tests that all branch instructions branch no further than *value*.

Syntax

```
#pragma b_range value | off | default
```

Parameter

`value`

Branch value. Default value is (`0x04000000 - 1`).

Remarks

Use this pragma for executable code that runs on Power Architecture processors with on-chip decompression capabilities.

bc_range

Ensures that all branch conditional instructions branch no further than *value*.

Syntax

```
#pragma bc_range value | off | default
```

Parameter

`value`

Branch value.

If `prepare_compress` is `off`, default value is:

(`0x00010000 - 1`)

If `prepare_compress` is `on`, deafult value is:

(0x`00001000 - 1`)

Remarks

Use this pragma for executable code that runs on Power Architecture processors with on-chip decompression capabilities.

cats

Controls the generation of relative `.rela.*` sections in the `.elf` file

Syntax

```
#pragma cats on | off | reset
```

Remarks

The default for this pragma is `off` for Freescale Power Architecture processors.

NOTE Standard libraries should be rebuild using `#pragma cats off` in order to get rid of all `.rela.*` symbols in the `internal_FLASH.elf` file.

disable_registers

Controls compatibility for the ISO/IEC standard library function `setjmp()`.

Syntax

```
#pragma disable_registers on | off | reset
```

Remarks

If this pragma is `on`, the compiler disables certain optimizations for any function that calls `setjmp()`. It also disables global optimization and does not store local variables and arguments in registers. These changes ensure that all local variables have correct values when the `setjmp()` function saves the processor state.

Use this pragma only if you are porting code that relies on this feature because it makes your program much larger and slower. In new code, declare a variable to be `volatile` if you expect its value to persist across setjmp() calls.

e500_floatingpoint

Generates single-precision floating point instructions for the Power Architecture e500 SPE (Signal Processing Unit) APU (Auxiliary Processing Unit).

Syntax

```
#pragma e500_floatingpoint on | off | reset
```

e500v2_floatingpoint

Generates double-precision floating point instructions for the Power Architecture e500v2 SPE (Signal Processing Unit) APU (Auxiliary Processing Unit).

Syntax

```
#pragma e500v2_floatingpoint on | off | reset
```

function_align

Aligns the executable object code of functions on a specified byte boundary.

Syntax

```
#pragma function_align 4 | 8 | 16 | 32 | 64 | 128 | reset
```

gen_fsel

Controls the use of the floating-point select instruction, `fsel`.

Syntax

`#pragma gen_fsel on | off |` *`number`* `| always`

where *number* is a value from 1 to 255.

Remarks

The compiler uses this pragma to determine how often it generates the `fsel` instruction. The *number* argument specifies how aggressively the compiler should use this instruction, 1 is equivalent to “rarely” and 255 is equivalent to `always`. The `on` choice is equivalent to a value of 10.

gen_isel

Controls the use of the integer select instruction, `isel`.

Syntax

```
#pragma gen_isel on | off | number | always
```

where *number* is a value from 1 to 255.

Remarks

The compiler uses this pragma to determine how often it generates the `isel` instruction. The *number* argument specifies how aggressively the compiler should use this instruction, 1 is equivalent to “rarely” and 255 is equivalent to `always`. The `on` choice is equivalent to a value of 10.

gprfloatcopy

Takes advantage of simpler alignment restrictions for copying floating point data.

Syntax

```
#pragma gprfloatcopy on | off | reset
```

Remarks

When this pragma is `on`, the compiler uses integer load and store instructions for memory-to-memory assignments for objects of type `double` and `float`, which improves the speed of memory-to-memory assignments of unaligned floating-point data. When this pragma is `off`, the compiler issues floating-point load and store instructions instead.

has8bytebitfields

Controls the use of bitfields that fit in the `long long` data type.

Syntax

```
#pragma has8bytebitfields on | off | reset
```

Remarks

When this pragma is `on`, the compiler allows bitfields in the `long long` data type. Such bitfields may occupy up to 64 bits (8 bytes). When this pragma is `off`, the compiler allows bitfields only in integer types of the same size or smaller than the `long` type.

Listing 37.5 Example for pragma has8bytebitfields

```
#pragma has8bytebitfields on
struct X {
    long long fielda : 12;
    long long fieldb : 18;
    long long fieldc : 32;
    long long fieldd : 2;
};
```

interrupt

Deprecated. To avoid C++ parsing ambiguities and other possible inadvertent errors, use `__declspec(interrupt)` instead.

legacy_struct_alignment

Avoids the possibility of misaligned load or store instructions caused by promoting the alignment of global and local data objects to a minimum of 4 bytes.

Syntax

```
#pragma legacy_struct_alignment on | off | reset
```

Remarks

The default for this pragma is `off` for Freescale Power Architecture processors as the big endian systems do not crash and misalignment is rare.

merge_float_consts

Each floating point constant is placed in a unique variable such that the linker will merge floating point constants which have the same value. (The variable names are not legal C/

C++ and are not accessible by the user). This option works with either small data in TOC on or off. This option minimizes TOC entry usage for programs which frequently use the same floating point constant in many different source files.

Syntax

```
#pragma merge_float_consts on|off
```

Remarks

The default for this pragma is `off` for Freescale Power Architecture processors.

min_struct_align

Increases aggregate alignments for better memory access.

Syntax

```
#pragma min_struct_align 4 | 8 | 16 | 32 | 64 | 128 | on | off
      | reset
```

Remarks

When this pragma is `off`, the compiler aligns objects to their regular alignments. The default alignment is 4.

NOTE This pragma only applies if the optimization level is greater than 0.

misaligned_mem_access

Controls how the compiler copies structures that are not aligned to 4-byte boundaries.

Syntax

```
#pragma misaligned_mem_access on | off | reset
```

Remarks

When this pragma is `on`, the compiler uses 4-byte load and store instructions to copy structures that are not aligned to 4-byte boundaries. By using these misaligned load and store instructions, the compiler improves runtime performance and reduces code size.

When this pragma is `off`, the compiler uses 1-, 2-, and 4-byte load and store instructions to copy structures that are aligned on corresponding boundaries.

However, misaligned load and store instructions on some Power Architecture processors give poor performance or even generate processor exceptions. For these processors, turn this pragma off. Desktop variants of the Power Architecture processor family do not have this limitation.

Consult the processor manufacturer's documentation for information on the processor's behavior when loading and storing 4-byte values that are not aligned to 4-byte boundaries.

The default for this pragma is `on` for processors that allow misaligned memory access. The default is `off` for processors that have limited misaligned memory access performance or generate an exception.

no_register_save_helpers

Controls the save and restore registers without calling helper functions

Syntax

```
#pragma no_register_save_helpers on | off | reset
```

options

Specifies how to align structure and class data.

Syntax

```
#pragma options align= alignment
```

Parameter

`alignment`

Specifies the boundary on which structure and class data is aligned in memory. Values for *alignment* range from 1 to 16, or use one of the following preset values:

Table 37.1 Structs and Classes Alignment

If *alignment* is …	The compiler …
`mac68k`	Aligns every field on a 2-byte boundaries, unless a field is only 1 byte long. This is the standard alignment for 68K Mac OS alignment.
`mac68k4byte`	Aligns every field on 4-byte boundaries.
`power`	Aligns every field on its natural boundary. For example, it aligns a character on a 1-byte boundary and a 16-bit integer on a 2-byte boundary. The compiler applies this alignment recursively to structured data and arrays containing structured data. So, for example, it aligns an array of structured types containing an 4-byte floating point member on an 4-byte boundary.
`native`	Aligns every field using the standard alignment.
`packed`	Aligns every field on a 1-byte boundary. It is not available in any panel. This alignment causes your code to crash or run slowly on many platforms. ***Use it with caution.***
`reset`	Resets to the value in the previous `#pragma options align` statement.

NOTE There is a space between `options` and `align`.

Overload

pool_data

Controls whether data larger than the small data threshold is grouped into a single data structure.

Syntax

```
#pragma pool_data on | off | reset
```

Remarks

When this pragma is `on` the compiler optimizes pooled data. You must use this pragma before the function to which you apply it.

NOTE Even if this pragma is `on`, the compiler will only pool the data if there is a performance improvement.

This pragma corresponds to the CodeWarrior IDE's **Pool Data** setting in the **PowerPC Processor** panel.

ppc_lvxl_stvxl_errata

Controls the instruction encoding for the `lvxl` and `stvxl` instructions on the Power Architecture 745*x* processors to correct a bug in the processors.

Syntax

```
#pragma ppc_lvxl_stvxl_errata on | off | reset
```

profile

Controls the generation of extra object code for use with the CodeWarrior profiler.

Syntax

```
#pragma profile on | off | reset
```

Remarks

If you enable this pragma, the compiler generates code for each function that lets the CodeWarrior Profiler collect information on it.

This pragma corresponds to the CodeWarrior IDE's **Profiler Information** setting in the **PPC Processor** panel.

read_only_switch_tables

Controls where tables for `switch` statements are placed in object code.

Syntax

```
#pragma read_only_switch_tables on | off | reset
```

Remarks

This option specifies where the compiler places executable code addresses for `switch` statements. When this option is `on`, the compiler places these tables in a read-only section (`.rodata`), allowing the linker to place this object code in a ROM area instead of RAM.

When this option is `off`, the compiler places these `switch` tables in an object code section that is readable and writable (`.data`). Putting these tables in a read/write section allows relocation at runtime. The System V ABI, SuSE, YellowDog, and SDA PIC/PID application binary interfaces (ABIs) allow relocatable object code at runtime.

strict_ieee_fp

Controls generation of executable code that conforms to the IEEE floating point standard.

Syntax

```
#pragma strict_ieee_fp on | off | reset
```

Remarks

Disabling this option may improve performance but may change the results generated.

- Use Fused Mult-Add/Sub

  Uses a single instruction to do a multiply accumulate. This runs faster and generates slightly more accurate results than specified by IEEE, as it has an extra rounding bit between the multiply and the add/subtract).

- Generate `fsel` instruction

  The `fsel` instruction is not accurate for denormalized numbers, and may have issues related to unordered compares, but generally runs faster.

- Assume Ordered Compares

  Ignore the unordered issues when comparing floating point which allows converting:

```
if (a <= b)
```

into

```
if (!(a > b))
```

switch_tables

Controls the generation of switch tables.

Syntax

```
#pragma switch_tables on | off | reset
```

Remarks

When `on`, the compiler translates `switch` statements into tables of addresses where each address in the list corresponds to a `case` statement. Using tables improves the performance of `switch` statements but may increase the size of the executable code if there are many `case` statements or if the `case` label values are not contiguous.

When `off`, the compiler translates `switch` statements into a series of comparisons, one comparison for each `case` statement.

uchar_bool

Controls the size of the `_Bool` and `bool` data types.

Syntax

```
#pragma uchar_bool on | off | reset
```

Remarks

When `on`, the compiler translates the `_Bool` data type in C99 (ISO/IEC 9899-1999) source code and the `bool` data type in C++ source code to 8 bit values. When `off`, these data types are 32 bits. Use this pragma only before any declarations.

When this pragma is off (boolean values are 32-bits), use bitfields to ensure that a boolean value is 8 bits. Listing 37.6 shows an example.

Listing 37.6 Example of overriding uchar_bool in a structure in C++

```
#pragma uchar_bool off /* Boolean values are 32 bits */

typedef struct
{
  bool sockclean:8 /* This value will only occupy 8 bits. */
} sockrec;
```

use_lmw_stmw

Controls the use of `lmw` and `stmw` instructions.

Syntax

```
#pragma use_lmw_stmw on | off | reset
```

Remarks

Use of `lmw` and `stmw` may be slower on some processors.

ushort_wchar_t

Controls the size of `wchar_t`.

Syntax

```
#pragma ushort_wchar_t on | off | reset
```

Remarks

When this pragma is `on`, `wchar_t` changes from 4-bytes to 2-bytes.

vec2x32float_align_4

Controls the alignment of type `__vec2x32float__`.

Syntax

```
#pragma vec2x32float_align_4 on | off | reset
```

Remarks

When the pragma is `on`, type `__vec2x32float__` is aligned on `4byte` boundary.

The default value for this pragma is `off`.

z4_16bit_cond_branch_errata_5116

Controls the use of `16-bit` conditional instructions.

Syntax

```
#pragma z4_16bit_cond_branch_errata_5116 on | off | reset
```

Remarks

When the pragma is `on`, `32-bit` conditional instructions are used instead of `16-bit`.

The default value for this pragma is `off`.

z4_mtlr_se_rfi_errata_26553

Ensures that there are at least three instructions between the `mtlr` and the `se_rfi`.

Syntax

```
#pragma z4_mtlr_se_rfi_errata_26553 on | off | reset
```

Remarks

When the pragma is `on` and you are using either `__declspec(interrupt)` or `#pragma interrupt`, the compiler ensures that there are at least three instructions between the `mtlr` and the `se_rfi`.

If your interrupt handler is written in function level assembler, compiler support only happens if you do not use the `nofralloc` directive. Standalone assembler does not include this support.

The default value for this pragma is `off`.

NOTE Other forms of the `se_rfi` such as `se_rfci` and `se_rfdi` are also supported.

Optimization Pragmas

aggressive_hoisting

Improves the number of variables that get hoisted out of a loop.

Syntax

```
#pragma aggressive_hoisting on | off | reset
```

Remarks

This pragma produces faster code and causes a slight increase in code size, especially when optimizing for size. In some cases, hoisting variables out of a loop when the loop does not have a lot of iterations can make your code slower.

The default value for this pragma is `off`.

c9x_alias_by_type

Allows back-end optimizations to use alias type information.

Syntax

```
#pragma c9x_alias_by_type on | off | reset
```

Remarks

When this pragma is on, the compiler’s back-end optimizations take advantage of type information gathered during alias analysis. Turn this pragma on if your source code follows the type rules specified by the ISO/IEC 9899-1999 C standard (“C99”), section 6.5.

Turn this pragma off if your source code violates type rules. The information collected from source code that violates these rules might lead the compiler to apply its optimizations incorrectly.

This pragma does not have a corresponding IDE panel setting.

epilogue_helper_functions

Controls size optimization for function termination instructions.

Syntax

```
#pragma epilogue_helper_functions on | off | reset
```

Remarks

When this pragma is on, the compiler reduces the size of object code in function terminations. It performs this optimization by replacing several instructions for function termination with fewer calls to special functions that perform the same tasks. This optimization reduces executable code size but also slows the program's performance.

When this pragma is on, the compiler generates instructions that may appear as inconsistent information in a symbolic debugger when stepping through the end of a function.

This pragma does not correspond to any panel setting. By default, this pragma is off. The compiler turns this optimization on implicitly when size optimization is on and optimization level is equal to or greater than 2. For example, these pragma settings will also turn on epilogue helper function generation:

```
#pragma optimization_level 2
#pragma optimize_for_size on
```

fp_contract

Controls the SPE additional fused multiply-add instructions codegen, when `-spe2_vector` is selected.

Syntax

```
#pragma fp_contract on|off
```

Remarks

Floating point accuracy is maintained by turning off this optimization.

fp_contract_aggressive

Enables the peephole pattern to detect and convert the seperate instructions.

Syntax

```
#pragma fp_contract_aggressive on|off|reset
```

Remarks

`#pragma fp_contract_aggressive on` can be used to further optimize multiply-add opportunities.

NOTE Precision could be lost due to rounding issues.

ipa_rescopes_globals

Rescopes the application global variables, that are only used in one function, to local static. The change to static enables other optimizations that improve alias analysis and load/store optimizations.

Syntax

```
#pragma ipa_rescopes_globals on | off
```

Remarks

Ensure that the following requirements are met to rescope the application global variables to local static:

- Program IPA is enabled in all application source files
- use of `#pragma ipa_rescopes_globals on` in all application source files (prefix file or with `-flag ipa_rescopes_globals` on the commandline)
- `main()` is defined in one of the application files.
- It is not necessary, or even desirable, to have standard library, runtime or startup code compiled with program IPA and `ipa_rescopes_globals on`. However, it is important to have as many of your application sources as possible compiled with those options enabled.

NOTE As the third party libraries generally do not access the application variables, these libraries can be kept in archive form.

For a simple example, compile/assemble your startup code without program IPA. Compile all of the application code with program IPA, `#pragma ipa_rescopes_globals on` and link the startup objects, your application objects and the library archives (For more details on Program IPA linking procedures, refer "Interprocedural Analysis").

For a complex example where the application sources are put into groups, compiled and then pre-built into several archives or partially linked objects and the build procedure cannot be matched with the simple example, following changes to the build procedure are suggested:

- Try to make the build setup as similar to the simple example as possible. This will help you identify if the code will benefit from `ipa_rescopes_globals` or you will need to modify your source files to get a successful link. For more details, refer "Generating a successful link".
- All of your functions are not visible to the compiler at once during program IPA. It is possible that a defined global variable in your core files may be used by only one core file but might also be used in one of your application archives that you were unable to build the simple way. If this is true, `ipa_rescopes_globals` will rescope the variable and at link time, your application archive will not be able to find the variable and you will get an undefined symbol link error.

NOTE If you get a successful link you do not have to make any further changes to the build or source.

Generating a successful link

Optimization prevents an improper build. If you do not get a successful link or you only get a few such link errors, identify the source file that defines the "undefined" symbol and try one of the following (in decreasing order of general preference):

- Move the definition of the symbol into the application archive. Symbols that are undefined do not get rescoped.
- Force the export of the symbol with `__declspec`(force_export). Symbols that are exported do not get rescoped.
- Change the symbols to weak with `__declspec`(weak) by inserting before definition. Weak symbols do not get rescoped.
- Change the symbols to volatile. Volatile symbols do not get rescoped.

with smaller, more efficient groups of instructions.

peephole

Controls the use peephole optimization.

Syntax

```
#pragma peephole on | off | reset
```

Remarks

If you enable this pragma, the compiler performs *peephole optimizations*. These optimizations replace redundant or commonly occurring groups of instructions with smaller, more efficient groups of instructions.

peephole_enable_16bit_load_store_inst

Enables use of 16-bit load/store instructions instead of 32-bit load/store instructions

Syntax

```
#pragma peephole_enable_16bit_load_store_inst on | off |
    reset
```

Remarks

This peephole optimization will replace the 32 bit load/store instructions with 16 bit load/store instructions.

This pragma is on by default under size optimization and off under speed optimization.

Please note that this optimization is applicable only when VLE instruction set is enabled.

Example

Converts the below pattern

```
e_stb   r0,28(r3)
e_stb   r0,32(r3)
e_stb   r4,36(r3)
e_stb   r4,40(r3)
e_stb   r4,41(r3)
e_stb   r4,42(r3)
e_stb   r4,43(r3)
```

to

```
e_add16i r3,r6,28
se_stb   r0,0(r3)
se_stb   r0,4(r3)
se_stb   r4,8(r3)
se_stb   r4,12(r3)
se_stb   r4,13(r3)
se_stb   r4,14(r3)
se_stb   r4,15(r3)
```

ppc_opt_defuse_mem_limit

Controls memory consumed by compiler optimizations on the host computer.

Syntax

`#pragma ppc_opt_defuse_mem_limit on | off | reset | *limit*`

Parameter

`limit`

Number of megabytes to use on the host computer when optimizing object code. The default value is `150`, which specifies 150 megabytes.

Remarks

Some optimizations need a lot of memory on the host computer, especially when optimizing large functions that make many function calls or refer to many variables. This pragma controls how much memory these optimizations consume.

If *limit* is set too low, the compiler will not be able to complete some optimizations and will issue an error message.

NOTE This pragma is to be used when users see the compiler error or warning that the compiler needs more memory to be allocated for usedef/defuse chain computation.

ppc_unroll_instructions_limit

Limits number of instructions in an unrolled loop to *value*.

Syntax

```
#pragma ppc_unroll_instructions_limit value | on | off
```

Parameter

`value`

Count limit of instructions. The default is 70.

Remarks

Use this pragma to specify the maximum number of instructions to place in an unrolled loop. The `opt_unroll_loops` pragma controls loop unrolling optimization.

When this pragma is `on`, the compiler uses the default value.

ppc_unroll_speculative

Controls speculative unrolling of counting loops which do not have fixed counts.

Syntax

```
#pragma ppc_unroll_speculative on | off
```

Remarks

The compiler uses the value specified with the `ppc_unroll_factor_limit` pragma to compute how many times to unroll eligible loops. The compiler adjusts the value specified with `ppc_unroll_factor_limit` so that it is equal to or less than the closest power of 2.

This optimization is only applied when:

- loop unrolling is turned on with the `opt_unroll_loops` pragma
- the loop iterator is a 32-bit value (`int`, `long`, `unsigned int`, `unsigned long`)
- the loop's body has no conditional statements

If you enable this pragma, the loop unrolling factor is a power of 2, less than or equal to the value specified by the `ppc_unroll_factor_limit` pragma.

The `opt_unroll_loops` pragma controls all loop unrolling optimization. To check this setting, use `__option (ppc_unroll_speculative)`. By default, this pragma is `on` when loop unrolling is enabled.

processor

Specifies the scheduling model used for instruction scheduling optimization.

Syntax

```
#pragma processor model
```

```
model
```

This argument is one of these choices:

```
401 | 403 | 405 | 505 | 509 | 5100 | 5200 | 555 | 56x |
601 | 602 | 603 | 603e | 604 | 604e | 74x | 75x | 801 |
821 | 823 | 85x | 86x | 87x | 88x | 7400 | 744x | 745x |
82xx| 85xx | e300v1 | e500v1 | e500v2 | e600 | Zen |
generic
```

prologue_helper_functions

Controls size optimization for function initialization instructions.

Syntax

```
#pragma prologue_helper_functions on | off | reset
```

Remarks

When this pragma is on, the compiler reduces the size of object code in function initialization. It performs this optimization by replacing several instructions for function initialization with fewer calls to special functions that perform the same tasks. This optimization reduces executable code size but also reduces the program's performance.

This pragma does not correspond to any panel setting. By default, this pragma is off. The compiler turns this optimization on implicitly when size optimization is on and optimization level is equal to or greater than 2. For example, these pragma settings will also turn on prologue helper function generation:

```
#pragma optimization_level 2
#pragma optimize_for_size on
```

remove_frsp_aggressive

Improves code optimization by transforming Power Architecture LFS and FRSP instructions into the equivalent FMR instructions.

Syntax

```
#pragma remove_frsp_aggressive on | off | reset
```

Remarks

This pragma allows the compiler more opportunity to use copy propogation optimizations to improve the generated code.

The default value for this pragma is `off`.

schedule

Specifies the use of instruction scheduling optimization.

Syntax

```
#pragma schedule once | twice | altivec | off
```

Remarks

This pragma lets you choose how many times the compiler passes object code through its instruction scheduler.

On highly optimized C code where loops were manually unrolled, running the scheduler once seems to give better results than running it twice, especially in functions that use the `register` specifier.

When the scheduler is run twice, it is run both before and after register colorizing. If it is only run once, it is only run after register colorizing.

This pragma does not correspond to any panel setting. The default value for this pragma is `twice`.

scheduling

Specifies the scheduling model used for instruction scheduling optimization.

Syntax

```
#pragma scheduling model | off
```

`model`

This argument is one of these choices:

```
401 | 403 | 405 | 505 | 509 | 5100 | 5200 | 555 | 56x |
601 | 602 | 603 | 603e | 604 | 604e | 74x | 75x | 801 |
821 | 823 | 85x | 86x | 87x | 88x | 7400 | 744x | 745x |
82xx| 85xx | e300v1 | e500v1 | e500v2 | e600 | Zen |
generic
```

spill_to_spe

Controls optimization for `e500` and `e200` cores that support SPE vectors.

Syntax

```
#pragma spill_to_spe on | off | reset
```

Remarks

In complex functions, sometimes the compiler is not able to color all registers without spilling to the stack. The new optimization takes advantage of the unused high half of the gpr vectors as storage and therefore avoids loading and storing to the stack.

The optimization is `off` by default, unless the optimization level is `3` and higher.

volatileasm

Controls the optimization of inline assembly statements.

Syntax

```
#pragma volatileasm on | off | reset
```

Remarks

When this pragma is `off`, the compiler applies peephole and scheduling optimizations to inline assembly statements. If the pragma is `on`, the compiler does not optimize these statements.

switch_op

Minimizes comparisions for “if” and “switch” statements.

Syntax

```
#pragma switch_op on | off
```

Remarks

When this pragma is on, the comparisions for “if” and “switch” statements are minimized. The compiler performs this optimization by using the constants specified within the switch case (or if) statements and based on internal heuristics that determine the cost.

By default, this pragma is `off`. The compiler turns this optimization on implicitly when the optimization level is equal to or greater than 2 and the user specifies `#pragma switch_op on`. For example, these pragma settings will also turn on switch optimization:

```
#pragma optimization_level 2
```

```
#pragma switch_op on
```

Index