CodeWarrior™ Development Studio

C Compilers Reference 3.2

Revised 9-Mar-2005

metrowerks

How to Contact Metrowerks

Corporate Headquarters	Metrowerks Corporation 7700 West Parmer Lane Austin, TX 78729 U.S.A.
World Wide Web	`http://www.metrowerks.com`
Sales	United States Voice: 800-377-5416 United States Fax: 512-996-4910 International Voice: +1-512-996-5300 E-mail: `sales@metrowerks.com`
Technical Support	United States Voice: 800-377-5416 International Voice: +1-512-996-5300 E-mail: `support@metrowerks.com`

Table of Contents

12 PowerPC Pragmas 229

1

Introduction

This reference covers version 3.0 and later of the CodeWarrior™ C/C++ compiler, which implements the C and C++ computer programming languages.

This introduction covers the following topics:

What is in this Reference?

This reference organizes its information in major sections:

- Introduction—this chapter
- Interface—how to interact with the compiler to configure its operation and translate source code
- Language—information on the compilers that apply to CodeWarrior target platforms
- Pragmas and predefined symbols—information on all pragmas for all targets and predefined preprocessor symbols

New Features

This reference has new and updated topics:

- See pragma "gcc_extensions" on page 154 for GCC improvements
- See pragma "c99" on page 133 for C99 improvements
- C++ improvements include:
  - Forward Declarations of Arrays of Incomplete Type now supported
  - Vendor Independent C++ ABI support for specific compiler targets
- C++ template improvements include:

  - Instantiating a Template now supports the explicit instantiation of non-template members
  - Better Template Conformance support for address of template-id rules
  - The "defer_defarg_parsing" on page 142 pragma supports deferred parsing of member functions by default
- "Multibyte and Unicode Support" on page 54
- "Getting Environmental Variables" on page 303 provides support for determining host-specific environmental variables
- New and newly documented #pragmas include:

array_new_delete
asmpoundcomment
asmsemicolcomment
debuginline
defer_defarg_parsing
errno_name
flat_include
fullpath_file
inline_bottom_up_once
inline_max_auto_size
inline_max_size
inline_max_total_size
instmgr_file
keepcomments
macro_prepdump
maxerrorcount
msg_show_lineref
msg_show_realref
multibyteaware_preserve_literals
no_conststringconv
nosyminline
options
pragma_prepdump
showmessagenumber
show_error_filestack
space_prepdump
srcrelincludes
strictheaderchecking
store_object_files
text_encoding
thread_safe_init
warning
warn_any_ptr_int_conv
warn_hiddenlocals
warn_illtokenpasting
warn_illunionmembers
warn_missingreturn
warn_no_explicit_virtual
warn_undefmacro

- this manual now uses references to the ISO C and C++ standards instead of Ellis and Stroustrup's *The Annotated C++ Reference Manual* (ARM) and Kernighan and Richie's *The C Programming Language* (K&R).

Where to Look for Related Information

Your CodeWarrior™ product includes most of the information mentioned here. Some information on the Internet might not be available with the CodeWarrior product you are using.

If you are new to CodeWarrior refer to the Quick Start card included with your product. Attend free, online programming courses at:
www.codewarrioru.com/

NOTE If your product is an update, you can find the Quick Start card (in PDF format) in the `CodeWarrior` installation directory on the CD.

If you are programming for a specific platform, read the *Targeting* manual appropriate for your target platform or processor.

If you are programming in Java or assembly language read the *Targeting* manual for your target platform or processor and the *Assembler Reference*.

NOTE Java might not be available for your target platform or processor.

For general information on using the CodeWarrior IDE and debugger, see the *IDE User Guide*. For information on the standard libraries for CodeWarrior C/C++ compilers, see the *MSL C Reference* and the *MSL C++ Reference*.

Verifying the Compiler Version

To determine what version of the CodeWarrior™ C/C++ compiler you are using, follow the steps below.

From the CodeWarrior IDE

1. Create a new project for your target platform.

   Consult your *Targeting* manual and the *IDE User Guide* for information on creating new projects.

2. Create a new source file named `version.c` that contains the source code in Listing 1.1, and add the file to the new project.

3. Select `version.c` in the project window.
4. From the Project menu, choose Preprocess.

   The CodeWarrior C/C++ compiler's preprocessor reads its directives to produce a preprocessed version of `version.c`.
5. In the preprocessed source code window, look for the value of the variable `version`.

   This represents the version of the compiler. See "Metrowerks Predefined Symbols" on page 112 for information on interpreting this value.

Listing 1.1 Verifying CodeWarrior C/C++ Compiler Version

```
/* version.c */

/* The version of the compiler is */
/* assigned to variable "version." */

long version = __MWERKS__;
```

From the command line

1. Create a new text file named version.c that contains the source code in Listing 1.1.
2. Preprocess the `version.c` file.

   If you are using the command-line version of the CodeWarrior C/C++ compiler on Microsoft Windows or Mac OS X, type:

   ```
   mwcc -EP version.c
   ```

TIP For more information on invoking CodeWarrior tools from the command line, see the "Invoking Command-Line Tools" on page 304.

3. Examine the preprocessor output.

   The CodeWarrior C/C++ compiler's preprocessor reads its directives to produce a preprocessed version of `version.c` in the command line. Check the value assigned to the variable `version`, which represents the version of the compiler. See "Metrowerks Predefined Symbols" on page 112 for information on interpreting this value.

Conventions Used in This Reference

References to a chapter or section number in *The C International Standard* (ISO/IEC 9899:1999) appear as (ISO C, §*number*). References to a chapter or section number in *The C++ International Standard* (ISO/IEC: 14882) appear as (ISO C++, §*number*).

This manual also uses syntax examples that describe the format of C source code statements:

```
#pragma parameter [return-reg] func-name [param-regs]
#pragma optimize_for_size on | off | reset
```

Table 1.1 describes how to interpret these statements.

Table 1.1 Understanding Syntax Examples

If the text looks like…	Then…
`literal`	Include the text in your statement exactly as you see it.
metasymbol	Replace the symbol with an appropriate value. The text after the syntax example describes what the appropriate values are.
`a	b	c`	Use one of the symbols in the statement: either `a`, `b`, or `c`.
[`a`]	Include the symbol, `a`, only if necessary. The text after the syntax example describes when to include it.

2

C/C++ Compiler Settings

This chapter describes the settings panels that control the CodeWarrior™ C/C++ compilers from the CodeWarrior™ IDE (Integrated Development Environment).

The sections in this chapter are:

C/C++ Settings Overview

The CodeWarrior compiler is managed using these settings panels:

- The C/C++ Preprocessor Panel panel controls how the preprocessor interprets source code. By modifying the settings on this panel, you can control how the preprocessor translates source code into preprocessed code.
- The C/C++ Language Panel panel controls how the compiler translates source code. By modifying the settings on this panel, you can control how strictly the compiler must adhere to C/C++ programming standards.
- The C/C++ Warnings Panel panel reports diagnostic messages. By modifying the settings on this panel, you can control when the compiler alerts you to questionable or erroneous programming syntax.

TIP For more information on using the IDE, see the *CodeWarrior IDE User Guide*.

C/C++ Preprocessor Panel

The C/C++ Preprocessor panel (Figure 2.1) provides an editable text field that can be used to #define macros, set #pragmas, or #include prefix files.

Figure 2.1 The C/C++ Preprocessor Panel

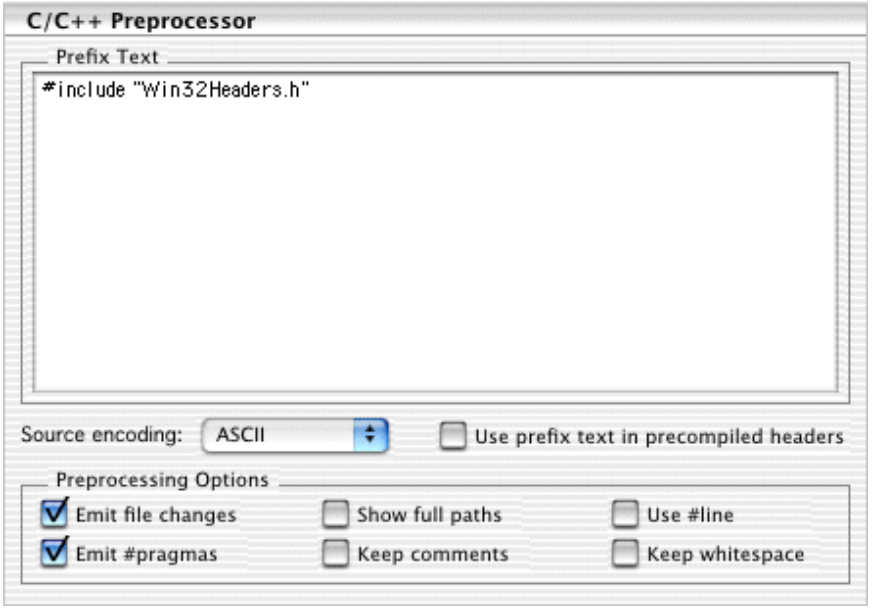


Table 2.1 provides information about the options in this panel.

Table 2.1 C/C++ Preprocessor Panel options

Option	Description
Source encoding	Allows you to specify the default encoding of source files. Multibyte and Unicode source text is supported. To replicate the obsolete option “Multi-Byte Aware”, set this option to System or Autodetect. Additionally, options that affect the "preprocess" request appear in this panel.
Use prefix text in precompiled header	Controls whether a *.pch or *.pch++ file incorporates the prefix text into itself. This option defaults to “off” to correspond with previous versions of the compiler that ignore the prefix file when building precompiled headers. If any #pragmas are imported from old C/C++ Language Panel settings, this option is set to “on”.

Table 2.1 C/C++ Preprocessor Panel options

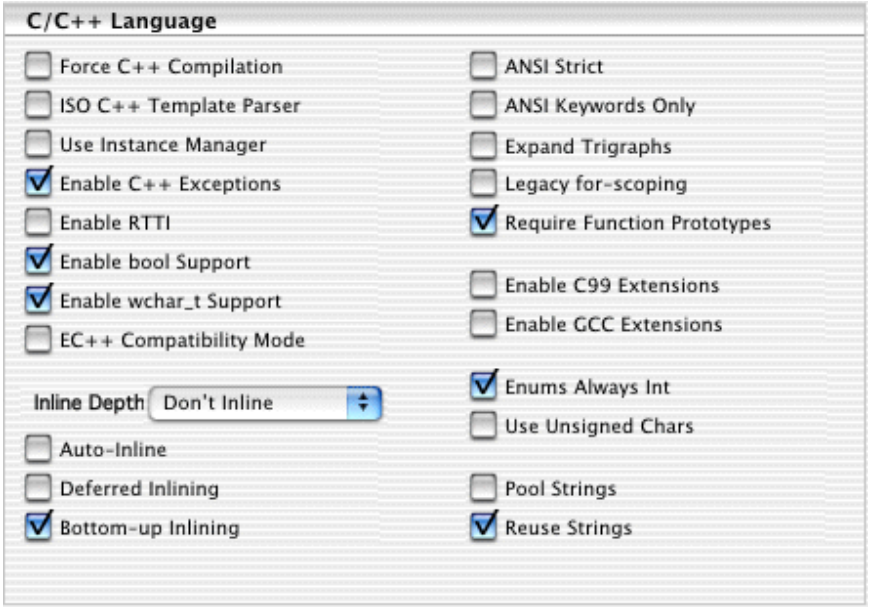
Option	Description
Emit file changes	**C**ontrols whether notification of file changes (or #line changes) appear in the output.
Emit #pragmas	Controls whether #pragmas encountered in the source text appear in the preprocessor output. **NOTE**: This option is essential for producing reproducible test cases for bug reports.
Show full paths	**C**ontrols whether file changes show the full path or the base filename of the file.
Keep comments	**C**ontrols whether comments are emitted in the output.
Use #line	Controls whether file changes appear in comments (as before) or in #line directives.
Keep whitespace	**Co**ntrols whether whitespace is stripped out or copied into the output. This is useful for keeping the starting column aligned with the original source, though the compiler attempts to preserve space within the line. This doesn't apply when macros are expanded.

C/C++ Language Panel

You can configure the C/C++ compiler by specifying a variety of options. Many of these options appear in the C/C++ Language panel, shown in Figure 2.2.

For information on how to see a particular panel for a build target, see the *IDE User Guide*.

Figure 2.2 The C/C++ Language Settings Panel



Settings in the C/C++ Language panel apply to all the source code files compiled by the CodeWarrior C/C++ compiler. You can override a setting by including its corresponding pragma in a source code file.

Each setting in the C/C++ Language panel has a corresponding pragma that you can use in source code to control the setting, regardless of what the C/C++ Language panel says the setting is. Some pragmas do not have a corresponding setting in the C/C++ Compiler panel. See "Common Pragmas" on page 125 for details on all pragmas.

You can also use a special preprocessor directive in your source code to determine the current setting of each option. See "Checking Settings" on page 115 for information on how to use this directive.

Most items in this panel are discussed elsewhere in this manual because they are closely related to how the CodeWarrior compiler implements C and C++.

Table 2.2 lists where to find information about the options in this panel.

Table 2.2 C/C++ Language Panel Options

For information on…	Refer to the section…
Force C++ Compilation	"Using the C++ Compiler Always" on page 65
ISO C++ Template Parser	"Better Template Conformance" on page 76
Use Instance manager	"Use Instance Manager" on page 92
Enable C99 Extensions	See #pragma c99
Enable Objective C	See #pragma objective_c and *Targeting Mac OS* manual
Legacy for-scoping	"Controlling Variable Scope in for Statements" on page 66
Enable C++ Exceptions	"Controlling C++ Extensions" on page 67
Enable RTTI	"Controlling RTTI" on page 66
Enable bool Support	"Using the bool Type" on page 67
Enable wchar_t Support	"Using the wchar_t Type" on page 38
Inline Depth Auto-inline Deferred Inlining Bottom-up Inlining	"Inlining" on page 42
ANSI Strict	"Checking for Standard C and Standard C++ Conformity" on page 37
ANSI Keywords Only	"ANSI Keywords Only" on page 40
Expand Trigraphs	"Expand Trigraphs" on page 41
EC++ Compatibility Mode	"Activating EC++" on page 81
Enums Always Int	"Enumerated Types" on page 34
Use Unsigned Chars	"Use Unsigned Chars" on page 47
Pool Strings	"Pool Strings" on page 43
Reuse Strings	"Reusing Strings" on page 44
Require Function Prototypes	"Require Function Prototypes" on page 45

C/C++ Warnings Panel

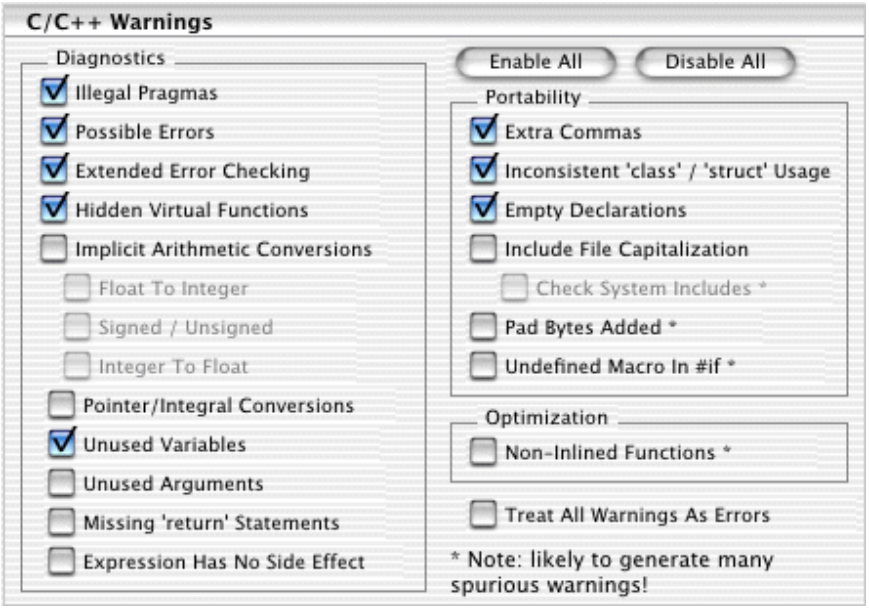
The **C/C++ Warnings** panel contains options that determine which warnings the CodeWarrior C/C++ compiler issues as it translates source code. Figure 2.3 shows this panel.

You can override a warning setting by including its corresponding pragma in a source code file. See "Common Pragmas" on page 125 for details on each available pragma.

Some warnings do not have a corresponding setting in **C/C++ Warnings Panel**. See "CodeWarrior C/C++ Errors and Warnings" on page 93 for more information on warnings.

You can also use a special preprocessor directive in your code to determine the current setting of each option. See "Checking Settings" on page 115 for information on how to use this directive.

Figure 2.3 The C/C++ Warnings Settings Panel



See Table 2.3 for more information on the options shown in Figure 2.3.

Table 2.3 C/C++ Warning Panel options

For information on…	Refer to the section…
Illegal Pragmas	"Illegal Pragmas" on page 94
Possible Errors	"Common Errors" on page 95
Extended Error Checking	"Suspicious Assignments and Incorrect Function Returns" on page 98
Hidden Virtual Functions	"Hidden Virtual Functions" on page 99
Implicit Arithmetic Conversions	"Implicit Arithmetic Conversions" on page 100
Float To Integer	See #pragma warn_impl_f2i_conv
Signed / Unsigned	See #pragma warn_impl_s2u_conv
Integer to Float	See #pragma warn_impl_i2f_conv
Pointer/Integral Conversions	See #pragma warn_any_ptr_int_conv and #pragma warn_ptr_int_conv
Unused Variables	"Unused Variables" on page 96
Unused Arguments	"Unused Arguments" on page 96
Missing Return Statements	See #pragma warn_missingreturn
Expression Has No Side Effect	See #pragma warn_no_side_effect
Extra Commas	"Extra Commas" on page 97
Inconsistent 'class' / 'struct' Keywords	"Mixed Use of 'class' and 'struct' Keywords" on page 101
Empty Declarations	"Empty Declarations" on page 94
Include File Capitalization	See #pragma warn_filenamecaps
Check System Includes	See #pragma warn_filenamecaps_system
Pad Bytes Added	See #pragma warn_padding
Undefined Macro in #if	See #pragma warn_undefmacro

Table 2.3 C/C++ Warning Panel options

For information on…	Refer to the section…
Non-Inlined Functions	"inline Functions That Are Not Inlined" on page 100
Treat All Warnings As Errors	"Warnings as Errors" on page 94

3

C Compiler

This chapter covers the following topics:

The information in this chapter applies to all target platforms for which the CodeWarrior C compiler generates object code.

This chapter does not cover C++ features. For more information on the CodeWarrior C++ language, see "C++ Compiler" on page 59.

The CodeWarrior Implementation of C

This section describes how the CodeWarrior C compiler implements the C programming language:

Identifiers

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

Header Files

(ISO C, §6.10.2) The CodeWarrior C preprocessor lets you nest up to 256 levels of `#include` directives.

You can use full path names in `#include` directives:

Table 3.1 #include directives

Windows	`#include "c:\HD\Tools\my headers\macros.h"`
UNIX and Mac OS X	`#include "/HD/Tools/my headers/macros.h"`
Mac OS	`#include "HD:Tools:my headers:macros.h"`

The CodeWarrior IDE lets you specify where the compiler looks for `#include` files through the Access Paths and Source Tree settings panels. See the *IDE User Guide* for information on using these panels to specify how the CodeWarrior C preprocessor searches for source code files.

See also: "Prefix Files" on page 33.

TIP If you are running the CodeWarrior C compiler from the command line, you can specify where to find `#include` files with a command-line setting. For more information, see "Command-Line Tools" on page 301.

Precompiled Header Files

A precompiled header is an image of the compiler's symbol table. Create a precompiled header file for commonly included files. You can also use a precompiled header file to temporarily change header files that do not normally change otherwise (for example, OS ABI headers or standard ANSI library header files). Then replace the original header files with the precompiled header file to significantly improve compile time.

A precompiled header cannot do any of the following:

- Define non-inline functions
- Define global data
- Instantiate template data
- Instantiate non-inline functions

You must include precompiled headers before defining or declaring other objects. You can only use one precompiled header file in a translation unit.

See also "precompile_target" on page 190.

Prefix Files

In previous CodeWarrior compilers, the prefix file was a distinct setting in the C/C++ Language Panel that told the compiler to include a source code file at the beginning of each source code file in a project's build target, or include a precompiled header file into the project.

With this compiler release, the concept of prefixing files, `#defines`, and `#pragmas` has been extended using the **Prefix Text** field in the C/C++ Language Panel.

To specify a prefix file, add an `#include` directive to this field, for example:

```
#include "Win32Headers.h"
```

To specify `#defines` or `#pragmas`, enter them here as you would in source code:

```
#define DEBUG_BUILD 1
```

```
#pragma warn_illtokenpasting off
```

When building precompiled headers, note the **Include prefix text in precompiled headers** setting. When enabled, the contents of the **Prefix Text** are used to generate the precompiled header. If your project generates a precompiled header in the same target that uses it, follow the form:

```
#if !__option(precompile)
```

```
#include "MyHeaders.mch"
```

```
#endif
```

to exclude the `*.mch` file when generating it.

NOTE `#pragmas` may affect aspects of parsing and type declaration while building a precompiled header, but these settings are not retained in the body of the precompiled header. Thus the prefix text is used for every file in the target.

Sizeof() Operator Data Type

The `sizeof()` operator returns the size of a variable or type in bytes. The data type of this size is `size_t`, which the compiler declares in the file `stddef.h`. If your source code assumes that `sizeof()` returns a number of type `int`, it might not work correctly.

NOTE The compiler evaluates the value returned by `sizeof()` only at compile time, not runtime.

NOTE The `sizeof()` operator is not intended to work in preprocessor `#if/#elif` directives.

Volatile Variables

(ISO C, §6.7.3) When you declare a `volatile` variable, the CodeWarrior C compiler takes the following precautions to respect the value of the variable:

- The compiler stores commonly used variables in processor registers to produce faster object code. However, the compiler never stores the value of a volatile variable in a processor register.
- The compiler uses its common sub-expression optimization to compute the addresses of commonly used variables and the results of often-used expressions once at the beginning of a function to produce faster object code. However, every time an expression uses a volatile variable, the compiler computes both the address of the volatile variable and the results of the expression that uses it.

Listing 3.1 shows an example of volatile variables.

Listing 3.1 Volatile Variables

```
void main(void)
{
    int i[100];
    volatile int a, b; /* a and b are not cached in registers. */

    a = 5;
    b = 20;

    i[a + b] = 15;      /* compiler calculates a + b */
    i[a + b] = 30;      /* compiler recalculates a + b */
}
```

The compiler does not place the value of `a`, `b`, or `a+b` in registers. But it does recalculate `a+b` in both assignment statements.

Enumerated Types

(ISO C, §6.2.5) The CodeWarrior C compiler uses the Enums Always Int and ANSI Strict settings in the C/C++ Language Panel to choose which underlying integer type to use for an enumerated type.

If you enable the Enums Always Int setting, the underlying type for enumerated data types is set to `signed int`. Enumerators cannot be larger than a `signed int`. If an enumerated constant is larger than an `int`, the compiler generates an error.

If you disable the ANSI Strict setting, enumerators that can be represented as an `unsigned int` are implicitly converted to `signed int`.

Listing 3.2 Example of Enumerations as Signed Integers

```
#pragma enumsalwaysint on
#pragma ANSI_strict on
enum foo { a=0xFFFFFFFF }; // ERROR. a is 4,294,967,295:
                           // too big for a signed int
#pragma ANSI_strict off
enum bar { b=0xFFFFFFFF }; // OK: b can be represented as an
                           // unsigned int, but is implicitly
                           // converted to a signed int (-1).
```

If you disable the Enums Always Int setting, the compiler chooses the integral data type that supports the largest enumerated constant. The type can be as small as a `char` or as large as a `long int`. It can even be a 64-bit `long long` value.

If all enumerators are positive, the compiler chooses the smallest unsigned integral base type that is large enough to represent all enumerators. If at least one enumerator is negative, the compiler chooses the smallest signed integral base type large enough to represent all enumerators.

Listing 3.3 Example of Enumeration Base Types

```
#pragma enumsalwaysint off
enum { a=0,b=1 };         // base type: unsigned char
enum { c=0,d=-1 };        // base type: signed char
enum { e=0,f=128,g=-1 };  // base type: signed short
```

The compiler uses `long long` data types only if you disable Enums Always Int and enable the `longlong_enums` pragma. (None of the settings corresponds to the `longlong_enums` pragma.)

Listing 3.4 Example of Enumerations with Type long long

```
#pragma enumsalwaysint off
#pragma longlong_enums off
enum { a=0x7FFFFFFFFFFFFFFF }; // ERROR: a is too large
#pragma longlong_enums on
enum { b=0x7FFFFFFFFFFFFFFF }; // OK: base type: signed long long
enum { c=0x8000000000000000 }; // OK: base type: unsigned long long
enum {bd=-1,e=0x80000000 };    // OK: base type: signed long long
```

When you disable the `longlong_enums` pragma and enable ANSI Strict, you cannot mix unsigned 32-bit enumerators greater than `0x7FFFFFFF` and negative enumerators. If you disable both the `longlong_enums` pragma and the ANSI Strict setting, large unsigned 32-bit enumerators are implicitly converted to signed 32-bit types.

Listing 3.5 Example of Enumerations with Type long

```
#pragma enumsalwaysint off
#pragma longlong_enums off
#pragma ANSI_strict on
enum { a=-1,b=0xFFFFFFFF };     // error
#pragma ANSI_strict off
enum { c=-1,d=0xFFFFFFFF };     // base type: signed int (b==-1)
```

The Enums Always Int setting corresponds to the pragma `enumsalwaysint`. To check this setting, use `__option (enumsalwaysint)`. By default, this setting is disabled.

See also "enumsalwaysint" on page 147, "longlong_enums" on page 166, and "Checking Settings" on page 115.

Extensions to ISO C

The CodeWarrior C language optionally extends ISO C. In most cases, you can control these extensions with settings in the C/C++ Language Panel or using specific pragmas.

- Checking for Standard C and Standard C++ Conformity
- C++ Comments
- Unnamed Arguments in Function Definitions
- A # Not Followed by a Macro Argument
- Using an Identifier After #endif
- Using Typecasted Pointers as lvalues
- Declaring Variables by Address
- ANSI Keywords Only
- Expand Trigraphs
- Character Constants as Integer Values
- Inlining
- Reusing Strings
- Require Function Prototypes
- Map Newlines to CR
- Relaxed Pointer Type Rules
- Use Unsigned Chars
- Using long long Integers
- Converting Pointers to Types of the Same Size

- Getting Alignment and Type Information at Compile Time
- Arrays of Zero Length in Structures
- Intrinsic Functions for Bit Rotation
- The “D” Constant Suffix
- The short double Data Type
- The __typeof__() and typeof() operators
- Initialization of Local Arrays and Structures
- Ranges in case statements
- The __FUNCTION__ Predefined Identifier
- GCC Extension Support
- Multibyte and Unicode Support
- The __declspec Declarator

For information on target-specific extensions, refer to the *Targeting* manual for your particular target.

Checking for Standard C and Standard C++ Conformity

The ANSI Strict setting in the C/C++ Language Panel affects several C language extensions made by the CodeWarrior C compiler:

- C++ Comments
- Unnamed Arguments in Function Definitions
- A # Not Followed by a Macro Argument
- Using an Identifier After #endif
- Using Typecasted Pointers as lvalues
- Converting Pointers to Types of the Same Size
- Arrays of Zero Length in Structures
- The “D” Constant Suffix

If you enable the ANSI Strict setting, the compiler disables all of the above ANSI C language extensions. You cannot enable individual extensions that are controlled by the ANSI Strict setting.

This setting might affect how the compiler handles enumerated constants. See “Enumerated Types” on page 34 for more information. It might also affect the declaration of the `main()` function for C++ programs. See “Implicit Return Statement for main()” on page 61.

The ANSI Strict setting corresponds to the pragma `ANSI_strict`. To check this setting, use `__option (ANSI_strict)`. See also "ANSI_strict" on page 128 and "Checking Settings" on page 115.

Using the wchar_t Type

If you enable the **Enable wchar_t Support** setting, you can use the standard C++ `wchar_t` type to represent wide characters. Disable this setting to use the regular character type, `char`.

C++ Comments

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

Listing 3.6 Example of a C++ Comment

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

To use this feature, disable the ANSI Strict setting in the C/C++ Language Panel.

See also "Checking for Standard C and Standard C++ Conformity" on page 37.

Unnamed Arguments in Function Definitions

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

Listing 3.7 Unnamed Function Arguments

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

To use this feature, disable the ANSI Strict setting in the C/C++ Language Panel.

See also: "Checking for Standard C and Standard C++ Conformity" on page 37.

A # Not Followed by a Macro Argument

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

Listing 3.8 Preprocessor Macros Using # Without an Argument

```
#define add1(x) #x #1  // OK, but probably not what you wanted:
```

```
                        // add1(abc) creates "abc"#1
#define add2(x) #x "2" // OK: add2(abc) creates "abc2"
```

To use this feature, disable the ANSI Strict setting in the C/C++ Language Panel.

See also "Checking for Standard C and Standard C++ Conformity" on page 37.

Using an Identifier After #endif

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

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

To use this feature, disable the ANSI Strict setting in the C/C++ Language Panel.

See also "Checking for Standard C and Standard C++ Conformity" on page 37.

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

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

Using Typecasted Pointers as lvalues

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

Listing 3.9 Example of a Typecasted Pointer as an lvalue

```
char *cp;
```

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

To use this feature, disable the ANSI Strict setting in the C/C++ Language Panel.

See also "Checking for Standard C and Standard C++ Conformity" on page 37.

Declaring Variables by Address

(ISO C, §6.7.8) The C compiler lets you explicitly specify the address that contains the value of a variable. For example, this definition states that the variable `MemErr` contains the contents of the address `0x220`:

```
short MemErr:0x220;
```

NOTE For MacOS Carbon or OS X programming, this extension is no longer allowed. Use the functions defined in `LowMem.h` header file of the Mac OS Universal Header files.

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

ANSI Keywords Only

(ISO C, §6.4.1) The CodeWarrior compiler can recognize several additional reserved keywords. The ANSI Keywords Only setting in the C/C++ Language Panel controls whether the compiler can accept these keywords.

If you enable this setting, the compiler generates an error if it encounters any of the additional keywords that it recognizes. If you must write source code that strictly adheres to the ISO standard, enable the ANSI Strict setting.

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

- `far`—Specifies how the compiler generates addressing modes and operations. It is not available for every target platform.
- `inline`—Lets you declare a C function to be inline. For more information, see "Inlining" on page 42.
- `pascal`—Used in Mac OS programming.

The ANSI Keywords Only setting corresponds to the pragma `only_std_keywords`. To check this setting, use `__option (only_std_keywords)`. By default, this setting is disabled.

See also "only_std_keywords" on page 177 and "Checking Settings" on page 115.

Expand Trigraphs

(ISO C, §5.2.1.1) The C compiler normally ignores trigraph characters. Many common character constants (especially on Mac OS) look like trigraph sequences, and this extension lets you use them without including escape characters.

If you must write source code that strictly adheres to the ISO standard, enable the Expand Trigraphs setting in the C/C++ Language Panel. When you enable this setting, be careful when initializing strings or multi-character constants that contain question marks.

```
char c = '????';       // ERROR:  Trigraph sequence expands to '??^
char d = '\?\?\?\?';   // OK
```

The **Expand Trigraphs** setting corresponds to the pragma `trigraphs`, To check this setting, use `__option (trigraphs)`. By default, this setting is disabled.

See also "trigraphs" on page 202 and "Checking Settings" on page 115.

Character Constants as Integer Values

(ISO C, §6.4.4.4) The C compiler lets you use string literals containing 2 to 8 characters to denote 32-bit or 64-bit integer values. Table 3.2 shows examples.

Table 3.2 Integer Values as Character String Constants

Character constant	Equivalent hexadecimal integer value
`'ABCDEFGH'`	0x4142434445464748 (64-bit value)
`'ABCDE'`	0x0000000041424344 (64-bit value)
`'ABCD'`	0x41424344 (32-bit value)
`'ABC'`	0x00414243 (32-bit value)
`'AB'`	0x00004142 (32-bit value)

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

NOTE This feature differs from using multibyte character sets, where a single character requires a data type larger than 1 byte. See "Multibyte and Unicode Support" on page 54 for information on using character sets with more than 256 characters (such as Kanji).

Inlining

CodeWarrior supports inlining C/C++ functions that you define with the `inline`, `__inline__`, or `__inline` specifier keywords.

The following functions are never inlined:

- Functions that return class objects that need destruction.
- Functions with class arguments that need destruction.
- Functions with variable argument lists.

The compiler determines whether to inline a function based on the ANSI Strict, Inline Depth, Auto-inline, and Deferred Inlining settings in the C/C++ Language Panel.

TIP When you call an inlined function, the compiler inserts the actual instructions of that function rather than a call to that function. Inlining functions makes your programs faster because you execute the function code immediately without the overhead of a function call and return. However, it can also make your program larger because the compiler may have to repeat the function code multiple times throughout your program.

If you disable the **ANSI Keywords Only** setting, you can declare C functions to be `inline`. The inlining items in the C/C++ Language Panel let you choose from the following settings in Table 3.3.

Table 3.3 Settings for the Inline Depth Pop-up Menu

This setting	Does this…
Don't Inline	Inlines no functions, not even C or C++ functions declared `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 Smart and 1 to 8 items in the Inline Depth pop-up menu correspond to the `pragma inline_depth` (inline_depth). To check this setting, use `__option(inline_depth)`, described at "Checking Settings" on page 115.

The Don't Inline item in the Inline Depth pop-up menu corresponds to the pragma `dont_inline`, described at "dont_inline" on page 145. To check this setting, use `__option (dont_inline)`, described at "dont_inline" on page 117. By default, this setting is disabled.

The Auto-Inline setting lets the compiler choose which functions to inline. Also inlines C++ functions declared `inline` and member functions defined within a class declaration. This setting corresponds to the pragma `auto_inline`, described at "auto_inline" on page 132. To check this setting, use `__option (auto_inline)`, described at "auto_inline" on page 116. By default, this setting is disabled.

The Deferred Inlining setting tells the compiler to inline functions that are not yet defined. This setting corresponds to the pragma `defer_codegen`, described at "defer_codegen" on page 141. To check this setting, use `__option (defer_codegen)`, described at "defer_codegen" on page 117.

The Bottom-up Inlining settings tells the compiler to inline functions starting at the last function to the first function in a chain of function calls. This setting corresponds to the pragma `inline_bottom_up`, described at "inline_bottom_up" on page 158. To check this setting, use `__option (inline_bottom_up)`, described at "inline_bottom_up" on page 118.

You can also disable automatic inlining of specific functions within a source file using the `__attribute__((never_inline))`. Listing 3.10 shows an example.

Listing 3.10 Example `__attribute__(never_inline)`

```
inline int f() __attribute__((never_inline))
{
  return 10;
}

int main()
{
  return f();    // f() is never inlined
}
```

NOTE For Intel x86 targets, the `__decspec(noinline)` is a compatible synonym.

Pool Strings

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

NOTE In principle, this setting works for all targets. However, it is useful only for a Table of Contents based linking mechanism such as that used for MacOS Classic on the PowerPC processor.

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

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

TIP You can change the size of the TOC with the Store Static Data in TOC setting in the PPC Processor panel. For more information, see the *Targeting Mac OS* manual.

NOTE This option can be overridden in PPC targets by using the **Linker Pool Strings** or **Linker Merges String Constants** options.

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

NOTE If you enable the Pool Strings setting, the compiler ignores the PC-Relative Strings setting. This is a 68K-only feature.

The Pool Strings setting corresponds to the pragma `pool_strings`. To check this setting, use `__option (pool_strings)`. By default, this setting is disabled.

See also "pool_strings" on page 188 and "Checking Settings" on page 115.

Reusing Strings

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

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

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

This setting helps you save memory if your program contains identical string literals which you do not modify.

If you enable the Reuse Strings setting, the strings are stored separately. After changing the first character, `str1` is still `"Hello"`, but `str2` is `"Yello"`.

If you disable the Reuse Strings setting, the two strings are stored in one memory location because they are identical. After changing the first character, both `str1` and `str2` are `"Yello"`, which is counterintuitive and can create bugs that are difficult to locate.

The Reuse Strings setting corresponds to the pragma `dont_reuse_strings`. To check this setting, use `__option (dont_reuse_strings)`. By default, this setting is enabled, so strings are not reused.

See also "dont_reuse_strings" on page 145. and "Checking Settings" on page 115.

Require Function Prototypes

(ISO C, §6.7.5.3, §6.9.1) The C compiler lets you choose how to enforce function prototypes. The Require Function Prototypes setting in the C/C++ Language Panel controls this behavior.

If you enable the Require Function Prototypes setting, the compiler generates an error if you define a previously referenced function that does not have a prototype. If you define the function before it is referenced but do not give it a prototype, then enabling the Require Function Prototypes setting causes the compiler to issue a warning.

This setting helps you prevent errors that happen when you call a function before you declare or 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 3.11, `PrintNum()` is called with an integer argument but later defined to take a floating-point argument.

Listing 3.11 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 work as you want. Since `PrintNum()` does not have a prototype, the compiler does not know to convert the integer to a floating-point number before calling the function. Instead, the function interprets the bits it received as a floating-point number and prints nonsense.

If you prototype `PrintNum()` first, as in Listing 3.12, the compiler converts its argument to a floating-point number, and the function prints what you wanted.

Listing 3.12 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 the above example, the compiler automatically typecasts the passed value. In other situations where automatic typecasting is not available, the compiler generates an error if an argument does not match the data type required by a function prototype. Such a mismatched data type error is easy to locate at compile time. If you do not use prototypes, you do not get a compiler error. However, at runtime the code might produce an unexpected result whose cause can be extremely difficult to find.

The Require Function Prototypes setting corresponds to the pragma `require_prototypes`. To check this setting, use `__option (require_prototypes)`. By default, this setting is enabled.

See also "require_prototypes" on page 192, and "Checking Settings" on page 115.

Map Newlines to CR

Use the `#pragma mpwc_newline` to set how the C compiler interprets the newline (`'\n'`) and return (`'\r'`) characters.

Most compilers, including the CodeWarrior C/C++ compilers, translate `'\r'` to `0x0D`, the standard value for carriage return, and `'\n'` to `0x0A`, the standard value for linefeed.

However, a few C compilers translate `'\r'` to `0x0A` and `'\n'` to `0x0D`—the opposite of the typical behavior.

If you enable this setting, the compiler uses these non-standard conventions for the `'\n'` and `'\r'` characters. Otherwise, the compiler uses the CodeWarrior C/C++ language's conventions for these characters.

Also if you enable this setting, use ISO C/C++ libraries that were compiled when this setting was enabled. Otherwise, you cannot read and write `'\n'` and `'\r'` properly. For example, printing `'\n'` takes you to the beginning of the current line instead of inserting a new line.

This setting corresponds to the pragma `mpwc_newline`. To check this setting, use `__option (mpwc_newline)`. By default, this setting is disabled.

See also "mpwc_newline" on page 169, and "Checking Settings" on page 115.

Relaxed Pointer Type Rules

Use the `#pragma mpwc_relax` to tell the compiler to treat all pointer types as the same type. While the compiler verifies the parameters of function prototypes for compatible pointer types, it allows direct pointer assignments.

Use this setting if you are using code written before the ISO C standard. Old source code frequently uses these types interchangeably.

This setting has no effect on C++. When compiling C++ source code, the compiler differentiates `char*` and `unsigned char*` data types even if the relaxed pointer setting is enabled.

See also "mpwc_relax" on page 170, and "Checking Settings" on page 115.

Use Unsigned Chars

If you enable the Use Unsigned Chars setting in the C/C++ Language Panel, the C compiler treats a `char` declaration as an `unsigned char` declaration.

NOTE If you enable this setting, your code might not be compatible with libraries that were compiled when this setting was disabled.

The Use Unsigned Chars setting corresponds to the pragma `unsigned_char`. To check this setting, use `__option (unsigned_char)`. By default, this setting is disabled.

See also "unsigned_char" on page 203 and "Checking Settings" on page 115.

Using long long Integers

The C compiler allows the type specifier `long long`. The `longlong` pragma controls this behavior and has no corresponding item in the For more information on using the

IDE, see the CodeWarrior IDE User Guide.. Consult the appropriate *Targeting* manual for information on the size and range of the `long long` data type.

If this setting is disabled, using `long long` causes a syntax error.

In an enumerated type, you can use an enumerator large enough for a `long long`. For more information, see "Enumerated Types" on page 34. However, `long long` bitfields are not supported.

You control the `long long` type with pragma `longlong`. To check this setting, use `__option(longlong)`. By default, this pragma is enabled.

See also "longlong" on page 165 and "Checking Settings" on page 115.

Converting Pointers to Types of the Same Size

The C compiler allows the conversion of pointer types to integral data types of the same size in global initializations. Since this type of conversion does not conform to the ANSI C standard, it is only available if the ANSI Strict setting is disabled in the C/C++ Language Panel. See "Checking for Standard C and Standard C++ Conformity" on page 37 for more information on this setting.

Listing 3.13 Converting a Pointer to a Same-sized Integral Type

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

Getting Alignment and Type Information at Compile Time

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

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

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

Arrays of Zero Length in Structures

If you disable the ANSI Strict setting in the C/C++ Language Panel, the compiler lets you specify an array of no length as the last item in a structure. Listing 3.14 shows an example. You can define arrays with zero as the index value or with no index value.

Listing 3.14 Using Zero-length Arrays

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

Intrinsic Functions for Bit Rotation

The CodeWarrior C language has functions

```
__rol(op, n)
__ror(op, n)
```

that do left- or right-bit rotation, respectively.

The *op* argument represents the item with the rotated bits. The *n* argument represents the number of times to rotate the *op* bits. The *op* argument is not promoted to a larger data type and can be of type `char`, `short`, `int`, `long`, or `long long`.

These functions are intrinsic ("built-in"). That is, you do not have to provide function prototypes or link with special libraries to use these functions.

NOTE Currently, these functions are limited to the Motorola 68K and Intel x86 versions of the CodeWarrior C/C++ compiler.

The "D" Constant Suffix

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

When the float_constants pragma is enabled, floating point constants should end with a "D" so that the compiler does not treat them as values of type `float`.

For related information, see "float_constants" on page 152.

The short double Data Type

The short double data type is no longer supported for most targets.

NOTE The Mac OS compiler knows that this data type provides a unique kind of floating point format used in Mac OS programming. See *Targeting Mac OS* for more information.

The __typeof__() and typeof() operators

With the `__typeof__()` operator, the compiler lets you specify the data type of an expression. Listing 3.15 shows an example.

`__typeof__(`*expression*`)`

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

Like the `sizeof()` operator, `__typeof__()` is only evaluated at compile time, not at runtime. For related information, see "Sizeof() Operator Data Type" on page 33.

If you enable the `gcc_extensions` pragma, the `typeof()` operator is equivalent to the `__typeof__()` operator.

Listing 3.15 Example of `__typeof__()` and `typeof()` Operators

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

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

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

Initialization of Local Arrays and Structures

If you enable the `gcc_extensions` pragma, the compiler allows the initialization of local arrays and `structs` with non-constant values (Listing 3.16).

Listing 3.16 GNU C Extension for Initializing Arrays and Structures

```
void myFunc( int i, double x )
{
  int arr[2] = { i, i + 1 };
```

```
  struct myStruct = { i, x };
}
```

Ranges in case statements

If you disable the ANSI Strict setting, the compiler allows ranges of values in a `switch` statement by using a special form of `case` statement. A `case` statement that uses a range is a shorter way of specifying consecutive `case` statements that span a range of values. Listing 3.17 shows an example.

The range form of a case statement is

`case` *low* `...` *high* `:`

where *low* is a valid `case` expression that is less than *high*, which is also a valid `case` expression. A `case` statement that uses a range is applied when the expression of a `switch` statement is both greater than or equal to the *low* expression and less than or equal to the *high* expression.

NOTE Make sure to separate the ellipsis (`...`) from the *low* and *high* expressions with spaces.

Listing 3.17 Ranges in `case` Statements

```
switch (i)
{
  case 0  ...  2: /* Equivalent to case 0: case 1: case 2: */
    j = i * 2;
    break;
  case 3:
    j = i;
    break;
  default:
    j = 0;
    break;
}
```

The __FUNCTION__ Predefined Identifier

The `__FUNCTION__` predefined identifier contains the name of the function currently being compiled. For related information, see "Predefined Symbols" on page 111.

GCC Extension Support

Use the **Enable GCC Extensions** setting in the "C/C++ Language Panel" on page 25. to enable the compiler to accept some GCC syntax and conventions. This option is the same as the previously-supported `#pragma gcc_extensions`.

The following GCC compatibility enhancements have been added:

- Statements in expressions are allowed

  ({int a; a=myfunc(); a; })
- GCC-style inline assembly supported on some targets only

  ```
  int count_leading_zero(int value)
  {
    asm ( "cntlzw %0, %1" : "=r" (bits) : "r" (value) );
    return bits;
  }
  ```
- GCC-style macro `varargs` supported

  ```
  #define DEBUG(fmt,args...) printf(fmt, ## args)
  DEBUG("test");
  DEBUG("saw %d copies\n", n_copies);
  ```
- The abbreviated "`?:`" operator is allowed

  ```
  x = y ?: z;  // -->  x = y ? y : z;
  ```
- Allow incomplete structs in array declarations

  ```
  struct Incomplete arr[10];
  ```
- Allow Class::Member in a class declaration

  ```
  class MyClass {
    ...
    int MyClass::getval();
    ...
  };
  ```
- Allow empty structs

  ```
  struct empty { } x;
  ```
- Allow empty struct initialization

  ```
  struct empty { } x = {};
  ```
- Struct initializer typecast support

  ```
  typedef struct { int x, y, z; float q; } mystruct;
  ```

void foo(mystruct s);

foo((mystruct) { 1,2,3,6e3 });

- Limited support for "`void *`" and function pointer arithmetic

  void *p;

  p = &data + 10;// point 10 bytes into "p"

  void foo();

  p = foo + 10;// point 10 bytes into "foo"

  At this time, the increment and decrement operators "++" don't work with void/ function pointers.
- `sizeof(function)` and `sizeof(void)` is 1
- Function pointers may be compared to "`void *`"
- Allow null statement (no trailing semicolon) in "`switch`"

  switch(x) {

  label:

  }
- Macro redefinitions with different values allowed

  #define MAC 3

  #define MAC (3)
- "`<?`", "`>?`" MIN/MAX operators supported for some targets
- Arrays may be assigned

  int a[10], b[10];

  a = b;
- Allow trailing comma in enumerations without warning

  enum { A, B, C, };
- Allow empty array as final member of `struct`

```
typedef struct {
   int type;
   char data[];
} Node;
```

- Designated initializer support

```
struct { int x, y, z; float q; } x = { q: 3.0,
         y:1, z:-4, x:-6 };
```

For related information, see the `#pragma` gcc extensions.

Multibyte and Unicode Support

The Codewarrior preprocessor fully supports the use of multibyte and Unicode-formatted source files.

Source and header files may be encoded in any text encoding the operating system recognizes. (Unicode text decoding support is implemented using native OS services in Win32 and Mac OS, and conv() on Un*x systems.)

As per ISO C++98 and C99, universal characters may be used in any string or character literal, identifiers (macros, variables, functions, methods, etc.), and comments. These characters are derived either from multibyte sequences in the source text, by virtue of the source file being encoded in UCS-2 or UCS-4, or by use of the `\uXXXX` or `\UXXXXXXXX` escape sequences.

Wide String Literals

Wide string literals in the form `L"xxx"` and wide characters in the form `L'xxx'` are interpreted in the context of the source text encoding.

```
const wchar_t *str = L"Meine Katze ißt Mäuse nachts.";
```

Multibyte character sequences that appear in strings are internally converted to their Unicode equivalents until the C/C++ token for the string is generated. At that time, if the string literal is a narrow string (i.e. using `char`), the original multibyte character sequence are emitted. If the string is a wide string (using `wchar_t`), the Unicode characters translated from the multibyte sequence are emitted. If `wchar_t` is 16 bits and a character is truncated to 16 bits, a warning is reported. (See ISO C99, §6.4.5.5 for the specification of this behavior.)

In the event that you want translated – and usually truncated – Unicode characters in narrow string literals, enable `#pragma multibyteaware_preserve_literals off`.

Source Encoding

The compiler uses the **Source encoding** option in the "C/C++ Preprocessor Panel" on page 23 to control how it detects the source file encoding. The compiler recognizes the following source encoding options:

- **ASCII**–no detection is done, the high-ASCII characters are not interpreted, and wide character strings are merely zero-extensions of the individual bytes in the string.
- **Autodetect**–the compiler attempts to tell whether byte-encoded source is encoded in UTF-8, Shift-JIS, EUC-JP, ISO-2022-JP, or whichever encoding format the operating system considers the default. This option degrades compile speeds slightly due to the extra scanning.
- **System**–the compiler uses the system locale without scanning the source.
- **UTF-8**, **Shift-JIS**, **EUC-JP**, **ISO-2022-JP**– self-explanatory.

NOTE Currently, the compiler ignores the mapping in some Japanese character sets of `0x5c` (ASCII '\') to Yen (`\u00A5`) because the C++98 and C99 standards say that the ASCII character set must be mapped one-to-one with any multibyte encoding. `0x5C` is always interpreted as '\' (except inside multibyte character sequences).

NOTE The ISO-2022-JP and EUC-JP encoding currently only recognize characters defined by JIS X 0208-1990 (i.e., the escapes `ESC $ @`, `ESC $ B` for ISO-2022-JP and two-byte sequences in EUC-JP). The additional characters in JIS X 0212-1990 are not yet recognized.

No matter what the **Source encoding** setting, the compiler always detects UTF-16{BE,LE} or UCS-4{BE,LE} source through a statistical character scan for NULs.

NOTE Currently, only the command-line tools, not the IDE, can properly handle sources in Unicode format (UTF-16, UCS-2, UCS-4).

Alternately, individual source files can identify which source text encoding is present using `#pragma text_encoding`. The format is:

```
#pragma text_encoding("name" | unknown | reset [, global])
```

Where `name` is an IANA or MIME encoding name or an OS-specific string.

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 U
TF-16BE UTF-16LE UTF16 UTF16BE UTF16LE UCS-4 UCS-4BE
UCS-4LE UCS4 UCS4BE UCS4LE 10646-1:1993 ISO-10646-1
ISO-10646 unicode
```

NOTE UCS-2 is always interpreted as UTF-16; i.e. surrogate character pairs are used to select characters through plane 16.

This `#pragma` may be used several times in one file (probably unlikely usage). The compiler expects the pragma to be encoded in the "current" text encoding, through the end of line.

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.

Other Support

In addition, note the following:

- Specify Universal character names (i.e. Unicode code points) with the `\uXXXX` or `\UXXXXXXXX` escape sequences:

  ```
  wchar_t florette = L'\u273f';

  int \u30AD\u30B3\u30DE\u30A6 = soy_sauce();
  ```

- You can use `\uXXXX` or `\UXXXXXXXX` regardless of the actual size of `wchar_t` (use of the escape does not impose a character width on the literal).
- Preprocessed text is always emitted in ASCII. Wide characters are emitted in the `\uXXXX` or `\UXXXXXXXX` format.

  ```
  extern string *Wörter[];

  -->

  extern string *W\u00F6rter[];
  ```

- Identifiers using characters not representable in ASCII are emitted in object code with the `\uXXXX` or `\UXXXXXXXX` escape sequences. If an object format does not support the '`\\`' character, the encoding may be modified on a target-specific basis.

Also see "Character Constants as Integer Values" on page 41 for information on creating a character constant consisting of more than one character (not to be confused with this topic).

For more information on Unicode, see `www.unicode.org`.

The `__declspec` Declarator

The `__declspec(arg)` declarator is a convention adopted from Win32 compilers to extend the type system. All CodeWarrior compilers recognize this declarator.

The `__declspec` declarator must appear before the name of the object as shown below:

```
__declspec(arg) int foo();

class __declspec(arg) bar { ... };
```

The list of valid arguments are grouped by targets:

- All Targets
- Non-Win32 Targets Only
- Win32 Targets Only

All Targets

`weak`

Specifies that the object may be overridden by another object during linking when applied to global data items or functions. Normally applied to library functions that the user is allowed to override without a warning or error.

`novtable`

Prevents the compiler from emitting a vtable for the class when applied to C++ classes or structs. The class cannot define or call any virtual functions.

Non-Win32 Targets Only

`[dll]export`

When applied to functions or global data, specifies that the symbol will be exported from the DLL or executable being built.

`[dll]import`

When applied to functions or global data, specifies that the symbol is imported from an external DLL. This is required for data items imported from DLLs.

`internal`

Specifies that the data item or function is internally used by the compiler.

`lib_export`

Combines actions of `__declspec(export)` and `__declspec(import)`.

Win32 Targets Only

`align(n)`

When applied to global or local data items, struct/class/union fields, or functions, `align` specifies the minimum alignment of the item in memory. Values from 1 to 8192 are accepted.

`allocate`

Ignored, but included for compatibility and to generate warnings.

`deprecated`

Ignored, but included for compatibility and to generate warnings.

`dllexport`

When applied to functions or global data, `dllexport` specifies that the symbol will be exported from the DLL or executable being built.

`dllimport`

When applied to functions or global data, `dllimport` specifies that the symbol is imported from an external DLL. This is required for data items imported from DLLs.

`nothrow`

When applied to functions, `nothrow` ensures that the function does not directly or indirectly throw a C++ exception.

`naked`

When applied to functions, `naked` prevents the compiler from generating a prologue and epilogue (including a return instruction). Same as the "asm" keyword applied to a function.

`noinline`

When applied to functions, `noinline` prevents the compiler from inlining the function.

`noreturn`

When applied to functions, `noreturn` specifies that the function never returns (i.e. it aborts the program or modifies its return address). This allows calling functions to optimize calls to the function and also suppresses "return value expected" warnings.

`property(...)`

Ignored, but included for compatibility and to generate warnings.

`selectany`

When applied to global data items, `selectany` allows the linker to select any definition, to work around C++'s "one definition" rule.

`thread`

When applied to global data items, `thread` places the item into thread-local storage (TLS) so that one unique copy exists per thread. This only works in executables and directly linked DLLs, not in DLLs loaded via LoadLibrary().

`uuid("string")`

Used for class or struct types, `uuid` specifies the UUID (unique user identifier) for a class or struct type. This is used for COM programming.

The UUID is a string in the form "`hhhhhhhh-hhhh-hhhh-hhhh-hhhhhhhhhhhh`" and is usually generated automatically or by the program `uuidgen`. Use the expression `__uuidof(type)` to retrieve the UUID of a type for comparison against other types.

4

C++ Compiler

This chapter discusses the CodeWarrior C++ compiler as it applies to all CodeWarrior targets. Most information in this chapter applies to any operating system or processor.

Other chapters in this manual discuss other compiler features that apply to specific operating systems and processors. For a complete picture, you need to consider all the information relating to your particular target.

The C compiler is also an integral part of the CodeWarrior C++ compiler. As a result, everything about the C compiler applies equally to C++. This discussion of the C++ compiler does not repeat information on the C compiler. See "C Compiler" on page 31 for information on the C compiler.

This chapter covers compiler features that support C++. This includes advanced C++ features such as RTTI, exceptions, and templates.

This chapter contains the following sections:

For information on using Embedded C++ (EC++) and for strategies on developing smaller C++ programs, see "Overview" on page 81.

CodeWarrior Implementation of C++

This section describes how the CodeWarrior C++ compiler implements certain parts of the C++ standard, as described in *The Annotated C++ Reference Manual* (Addison-Wesley) by Ellis and Stroustrup. The topics discussed in this section are:

- Default Arguments in Member Functions
- Calling an Inherited Member Function
- Forward Declarations of Arrays of Incomplete Type
- Vendor Independent C++ ABI

Namespaces

CodeWarrior supports namespaces, which provide the scope for identifiers. Listing 4.1 provides an example of how you define items in a namespace.

Listing 4.1 Defining Items in a Namespace

```
namespace NS
{
   int foo();
   void bar();
}
```

The above example defines an `int` variable named `NS::foo` and a function named `NS::bar`.

You can nest namespaces. For example, you can define an identifier as `A::B::C::D::E` where `A`, `B`, and `C` are nested namespaces and `D` is either another namespace or a class. You cannot use namespaces within class definitions.

You can rename namespaces for a module. For example:

```
namespace ENA = ExampleNamespaceAlpha;
```

creates a namespace alias called `ENA` for the original namespace `ExampleNamespaceAlpha`.

You can import items from a namespace. For example:

```
using namespace NS;
```

makes anything in NS visible in the current namespace without a qualifier. To limit the scope of an import, specify a single identifier. For example:

```
using NS::bar;
```

only exposes `NS::bar` as `bar` in the current space. This form of `using` is considered a declaration. So, the following statements:

```
using NS::foo;
int foo;
```

are not allowed because `foo` is being redeclared in the current namespace, thereby masking the `foo` imported from `NS`.

Implicit Return Statement for main()

In C++, the compiler adds a

```
return 0;
```

statement to the `main()` function of a program if the function returns an `int` result and does not end with a user `return` statement.

Examples:

```
int main() { } // equivalent to:
               //   int main() { return 0; }
main() { }     // equivalent to:
               //   int main() { return 0; }
```

If you enable the ANSI Strict setting in the C/C++ Language Panel, the compiler enforces an external `int main()` function.

Keyword Ordering

(ISO C++, §7.1.2, §11.4) If you use the `friend` keyword in a declaration, it must be the first word in the declaration. The `virtual` keyword does not have to be the first word in a declaration. Listing 4.2 shows an example.

Listing 4.2 Using the virtual or friend Keywords

```
class foo {
  virtual int f0();  // OK
  int virtual f1();  // OK
  friend int f2();   // OK
  int friend f3();   // ERROR
};
```

Additional Keywords

(ISO C++, §2.8, §2.11) The CodeWarrior C++ language reserves symbols from these two sections as keywords.

Default Arguments in Member Functions

(ISO C++, §8.3.6) The compiler does not bind default arguments in a member function at the end of the class declaration. Before the default argument appears, you must declare any value that you use in the default argument expression. Listing 4.3 shows an example.

Listing 4.3 Using Default Arguments in Member Functions

```
class foo {
  enum A { AA };
  int f(A a = AA); // OK
  int f(B b = BB); // ERROR: BB is not declared yet
  enum B { BB };
};
```

Calling an Inherited Member Function

(ISO C++, §10.3) You can call an inherited virtual member function rather than its local override in two ways. The first method is recommended for referring to member functions defined in a base class or any other parent class. The second method, while more convenient, is not recommended if you are using your source code with other compilers.

The standard method of calling inherited member functions

This method adheres to the ISO C++ Standard and simply qualifies the member function with its base class.

Assume you have two classes, `MyBaseClass` and `MySubClass`, each implementing a function named `MyFunc()`.

From within a function of `MySubClass`, you can call the base class version of `MyFunc()` this way:

```
MyBaseClass::MyFunc();
```

However, if you change the class hierarchy, this call might break. Assume you introduce an intermediate class, and your hierarchy is now `MyBaseClass`, `MyMiddleClass`, and `MySubClass`. Each has a version of `MyFunc()`. The code above still calls the original version of `MyFunc()` in the `MyBaseClass`, bypassing the additional behavior you implemented in `MyMiddleClass`. This kind of subtlety in the code can lead to unexpected results or bugs that are difficult to locate.

Using inheritance to call inherited member functions

The `def_inherited` pragma defines an implicit `inherited` member for a base class. Use this directive before using the `inherited` symbol:

```
#pragma def_inherited on
```

WARNING! The ISO C++ standard does not support the use of `inherited`.

You can call the inherited version of `MyFunc()` this way:

```
inherited::MyFunc();
```

With the `inherited` symbol, the compiler identifies the base class at compile time. This line of code calls the immediate base class in both cases: where the base class is `MyBaseClass`, and where the immediate base class is `MyMiddleClass`.

If your class hierarchy changes at a later date and your subclass inherits from a different base class, the immediate base class is still called, despite the change in hierarchy.

The syntax is as follows:

`inherited::`*func-name*`(`*param-list*`);`

The statement calls the *func-name* in the class's immediate base class. If the class has more than one immediate base class (because of multiple inheritance) and the compiler cannot decide which *func-name* to call, the compiler generates an error.

This example creates a Q class that draws its objects by adding behavior to the O class.

Listing 4.4 Using `inherited` to Call an Inherited Member Function

```
#pragma def_inherited on

struct O { virtual void draw(int,int); };
struct Q : O { void draw(int,int); };

void Q::draw (int x,int y)
{
    inherited::draw(x,y);    // Perform behavior of base class
    ...                      // Perform added behavior
}
```

For related information on this pragma see "def_inherited" on page 141.

Forward Declarations of Arrays of Incomplete Type

The CodeWarrior C++ compiler allows the forward declaration of arrays of incomplete type. Listing 4.5 shows an example.

Listing 4.5 Example of Forward Declaration of Array of Incomplete Type

```
extern struct incomplete arr[10];
```

```
struct incomplete {
  int a, b, c;
};

struct incomplete arr[10];
```

Vendor Independent C++ ABI

The Codewarrior C++ compiler currently supports these parts of the vendor independent C++ application binary interface (ABI):

- name mangling
- class and vtable layout
- constructor/destructor ABI
- virtual and non-virtual member function calls
- array construction/deconstruction ABI
- one-time construction ABI
- RTTI
- pointer to members

The following features work but do not conform to the ABI specification:

- exception handling

NOTE This ABI only works with some compiler targets. Currently, this is only Mac OS Mach-O.

NOTE The pseudo Mac OS base classes `SingleObject`, `SingleInheritance`, and `__comobject` are no longer supported or required in compilers that use the new C++ ABI.

For more information, see `www.codesourcery.com/cxx-abi`.

Extensions to ISO Standard C++

This section describes CodeWarrior extensions to the C standard that apply to all targets. In most cases, you turn the extension on or off with a setting in the C/C++ Language Panel. See "C/C++ Language Panel" on page 25 for information about this panel.

For information on target-specific extensions, you should refer to the *Targeting* manual for your particular target.

The `__PRETTY_FUNCTION__` Predefined Identifier

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

For related information, see "Predefined Symbols" on page 111.

Controlling the C++ Compiler

This section describes how to control compiler behavior by selecting settings in the C/C++ Language Panel.

This section contains the following:

Using the C++ Compiler Always

If you enable the **Force C++ Compilation** setting in the C/C++ Language Panel, the compiler translates all C source files in your project as C++ code. Otherwise, the CodeWarrior IDE uses the suffix of the file name to determine whether to use the C or C++ compiler. The entries in the CodeWarrior IDE's File Mappings panel describes the suffixes that the compiler seeks. See the *IDE User Guide* for more information on configuring these settings.

This setting corresponds to the pragma `cplusplus`. To check this setting, use `__option (cplusplus)`. By default, this setting is disabled.

See also "cplusplus" on page 138 and "Checking Settings" on page 115.

Controlling Variable Scope in for Statements

If you enable the **Legacy for-scoping** setting in the C/C++ Language Panel, the compiler generates an error when it encounters a variable scope issue that the ISO C++ standard disallows, but is allowed in the C++ language specified in *The Annotated C++ Reference Manual*.

With this option off, the compiler allows variables defined in a `for` statement to have scope outside the `for` statement.

Listing 4.6 Example of a Local Variable Outside a for Statement

```
for(int i=1; i<1000; i++) { /* ... */ }
return i;  // OK in ARM, Error in CodeWarrior C++
```

This setting corresponds to the pragma `ARM_conform`. To check this setting, use `__option (ARM_conform)`. By default, this setting is disabled.

See also "ARM_conform" on page 130 and "Checking Settings" on page 115.

Controlling Exception Handling

Enable the **Enable C++ Exceptions** setting in the C/C++ Language Panel if you use the ISO-standard `try` and `catch` statements. Otherwise, disable this setting to generate smaller and faster code.

TIP If you use PowerPlant for Mac OS programming, enable this setting because PowerPlant uses C++ exceptions.

For more information on how CodeWarrior implements the ISO C++ exception handling mechanism, see "Working with C++ Exceptions" on page 68.

This setting corresponds to the pragma `exceptions`. To check this setting, use `__option (exceptions)`. By default, this setting is disabled.

See also "exceptions" on page 148 and "Checking Settings" on page 115.

Controlling RTTI

The CodeWarrior C++ language supports runtime type information (RTTI), including the `dynamic_cast` and `typeid` operators. To use these operators, enable the **Enable RTTI** setting in the C/C++ Language Panel.

For more information on how to use these two operators, see "Working with RTTI" on page 69.

Using the bool Type

Enable the **Enable bool Support** setting to use the standard C++ `bool` type to represent `true` and `false`. Disable this setting if recognizing `bool`, `true`, or `false` as keywords causes problems in your program.

Enabling the `bool` data type and its `true` and `false` values is not equivalent to defining them using `typedef` and `#define`. The C++ `bool` type is a distinct type defined by the ISO C++ Standard. Source code that does not treat it as a distinct type might not compile properly.

For example, some compilers equate the `bool` type with the `unsigned char` data type. If you disable the **Enable bool Support** setting, the CodeWarrior C++ compiler equates the `bool` type with the `unsigned char` data type. Otherwise, using the CodeWarrior C/C++ compiler on source code that involves this behavior might result in errors.

This setting corresponds to the pragma `bool`. To check this setting, use `__option(bool)`. By default, this setting is disabled.

See also "bool" on page 133 and "Checking Settings" on page 115.

Controlling C++ Extensions

The C++ compiler has additional extensions that you can activate using the pragma `cpp_extensions`. The C/C++ Language Panel does not have any items that correspond to any of these extensions.

If you enable this pragma, the compiler lets you use the following extensions to the ISO C++ standard:

- Anonymous `struct` objects (ISO C++, §9).

Listing 4.7 Anonymous `struct` Objects

```
#pragma cpp_extensions on
void foo()
{
  union {
    long       hilo;
    struct { short hi, lo; };
      //  anonymous struct
  };
  hi=0x1234;
  lo=0x5678;
      //  hilo==0x12345678
}
```

- Unqualified pointer to a member function (ISO C++, §8.1).

Listing 4.8 Unqualified Pointer to a Member Function

```
#pragma cpp_extensions on
struct Foo { void f(); }
void Foo::f()
{
  void (Foo::*ptmf1)() = &Foo::f;
    // ALWAYS OK

  void (Foo::*ptmf2)() = f;
    // OK if you enabled cpp_extensions.
}
```

To check this setting, use the `__option (cpp_extensions)`. By default, this setting is disabled.

See also "cpp_extensions" on page 139 and "Checking Settings" on page 115.

Working with C++ Exceptions

If you enable the **Enable C++ Exceptions** setting in the C/C++ Language Panel, you can use the `try` and `catch` statements to perform exception handling. For more information on activating support for C++ exception handling, see "Controlling Exception Handling" on page 66.

Enabling exceptions lets you throw them across any code compiled by the CodeWarrior C/C++ compiler. However, you cannot throw exceptions across the following:

- Mac OS Toolbox function calls
- Libraries compiled with exception support disabled
- Libraries compiled with versions of the CodeWarrior C/C++ compiler earlier than CodeWarrior 8
- Libraries compiled with CodeWarrior Pascal or other compilers

If you throw an exception across one of these, the code calls `terminate()` and exits.

If you throw an exception while allocating a class object or an array of class objects, the code automatically destructs the partially constructed objects and de-allocates the memory for them.

Working with RTTI

This section describes how to work with runtime type information features of C++ supported by the CodeWarrior C++ compiler. RTTI lets you cast an object of one type as another type, get information about objects, and compare their types at runtime.

The topics in this section are:

Using the dynamic_cast Operator

The `dynamic_cast` operator lets you safely convert a pointer of one type to a pointer of another type. Unlike an ordinary cast, `dynamic_cast` returns 0 if the conversion is not possible. An ordinary cast returns an unpredictable value that might crash your program if the conversion is not possible.

The syntax for the `dynamic_cast` operator is as follows:

`dynamic_cast<`*Type*`*>(`*expr*`)`

The *Type* must be either `void` or a class with at least one virtual member function. If the object to which *expr* points (`*`*expr*) is of type *Type* or derived from type *Type*, this expression converts *expr* to a pointer of type *Type*`*` and returns it. Otherwise, it returns `0`, the null pointer.

For example, take these classes:

```
class Person { virtual void func(void) { ; } };
class Athlete : public Person { /* . . . */ };
class Superman : public Athlete { /* . . . */ };
```

And these pointers:

```
Person *lois = new Person;
Person *arnold = new Athlete;
Person *clark = new Superman;
Athlete *a;
```

This is how `dynamic_cast` works with each pointer:

```
a = dynamic_cast<Athlete*>(arnold);
  // a is arnold, since arnold is an Athlete.
a = dynamic_cast<Athlete*>(lois);
  // a is 0, since lois is not an Athelete.
```

```
a = dynamic_cast<Athlete*>(clark);
  // a is clark, since clark is both a Superman and an
Athlete.
```

You can also use the `dynamic_cast` operator with reference types. However, since there is no equivalent to the null pointer for references, `dynamic_cast` throws an exception of type `std::bad_cast` if it cannot perform the conversion.

This is an example of using `dynamic_cast` with a reference:

```
#include <exception>
using namespace std;
Person &superref = *clark;
try {
  Person &ref = dynamic_cast<Person&>(superref);
}
catch(bad_cast) {
  cout << "oops!" << endl;
}
```

Using the typeid Operator

The `typeid` operator lets you determine the type of an object. Like the `sizeof` operator, it takes two kinds of arguments:

- the name of a class
- an expression that evaluates to an object

NOTE Whenever you use `typeid` operator, you must #include the typeinfo header file.

The `typeid` operator returns a reference to a `std::type_info` object that you can compare with the == and `!=` operators. For example, if you have these classes and objects:

```
class Person { /* . . . */ };
class Athlete : public Person { /* . . . */ };

using namespace std;

Person *lois = new Person;
Athlete *arnold = new Athlete;
```

```
Athlete *louganis = new Athlete;
```

All these expressions are true:

```
#include <typeinfo>
// . . .
if (typeid(Athlete) == typeid(*arnold))
   // arnold is an Athlete, result is true
if (typeid(*arnold) == typeid(*louganis))
   // arnold and louganis are both Athletes, result is true
if (typeid(*lois) == typeid(*arnold)) // ...
   // lois and arnold are not the same type, result is false
```

You can access the name of a type with the `name()` member function in the `std::type_info` class. For example, these statements:

```
#include <typeinfo>
// . . .
cout << "Lois is a(n) "
     << typeid(*lois).name() << endl;
cout << "Arnold is a(n) "
     << typeid(*arnold).name() << endl;
```

Print this:

```
Lois is a(n) Person
Arnold is a(n) Athlete
```

Working with Templates

(ISO C++, §14) This section describes how to organize your template declarations and definitions in files. It also describes how to explicitly instantiate templates using a syntax that is not in the ARM but is part of the ISO C++ standard.

This section includes the following topics:

- Declaring and Defining Templates
- Instantiating a Template
- Better Template Conformance

Declaring and Defining Templates

In a header file, declare your class functions and function templates, as shown in Listing 4.9.

Listing 4.9 templ.h: A Template Declaration File

```
template <class T>
class Templ {
  T  member;
public:
  Templ(T x) { member=x; }
  T  Get();
};

template <class T>
T Max(T,T);
```

In a source file, include the header file, then define the function templates and the member functions of the class templates. Listing 4.10 shows you an example.

This source file is a template definition file, which you include in any file that uses your templates. You do not need to add the template definition file to your project. Although this is technically a source file, you work with it as if it were a header file.

The template definition file does *not* generate code. The compiler cannot generate code for a template until you specify what values it should substitute for the template arguments. Specifying these values is called instantiating the template. See "Instantiating a Template" on page 74.

Listing 4.10 templ.cp: A Template Definition File

```
#include "templ.h"

template <class T>
T Templ<T>::Get()
{
  return member;
}

template <class T>
T Max(T x, T y)
{
  return ((x>y)?x:y);
}
```

WARNING! Do *not* include the original template declaration file, which ends in `.h`, in your source file. Otherwise, the compiler generates an error saying that the function or class is undefined.

Providing declarations when declaring the template

CodeWarrior C++ processes any declarations in a template when the template is declared, not when it is instantiated.

Although the C++ compiler currently accepts declarations in templates that are not available when the template is declared, future versions of the compiler will not. Listing 4.11 shows some examples.

Listing 4.11 Declarations in Template Declarations

```
// You must define names in a class template declaration

struct bar;
template<typename T> struct foo {
  bar *member;  // OK
};
struct bar { };
foo<int> fi;


// Names in template argument dependent base classes:

template<typename T> struct foo {
   typedef T *tptr;
};

template<typename T> struct foo {
  typedef T *tptr;
};
template<typename T> struct bar : foo<T> {
  typename foo<T>::tptr member;  // OK
};


// The correct usage of typename in template argument
// dependent qualified names in some contexts:

template<class T> struct X {
   typedef X *xptr;
```

```
   xptr f();
};
template<class T> X<T>::xptr X<T>::f() // 'typename' missing
{
   return 0;
}

// Workaround: Use 'typename':

template<class T> typename X<T>::xptr X<T>::f() // OK
{
  return 0;
}
```

Instantiating a Template

The compiler cannot generate code for a template until you:

- declare the template class
- provide a template definition
- specify the data type(s) for the template

For information on the first two requirements, see "Declaring and Defining Templates" on page 72.

Specifying the data type(s) and other arguments for a template is called instantiating the template. CodeWarrior C++ gives you two ways to instantiate a template. You can let the compiler instantiate it automatically when you first use it, or you can explicitly create all the instantiations you expect to use.

Automatic instantiation

To instantiate templates automatically, include the template definition file in all source files that use the template, then use the template members like any other type or function. The compiler automatically generates code for a template instantiation whenever it sees a new one. Listing 4.12 shows how to automatically instantiate the templates in Listing 4.9 and Listing 4.10, class `Templ` and class `Max`.

Listing 4.12 myprog.cp: A Source File that Uses Templates

```
#include <iostreams.h>
#include "templ.cp" // includes templ.h as well

void main(void) {
  Templ<long> a = 1, b = 2;
```

```
    // The compiler instantiates Templ<long> here.
  cout << Max(a.Get(), b.Get());
    // The compiler instantiates Max<long>() here.
};
```

If you use automatic instantiation, the compiler might take longer to translate your program because the compiler has to determine on its own which instantiations you need. It also scatters the object code for the template instantiations throughout your program.

Explicit instantiation

To instantiate templates explicitly, include the template definition file in a source file, and write a template instantiation statement for every instantiation. The syntax for a class template instantiation is as follows:

`template class` *class-name<templ-specs>*`;`

The syntax for a function template instantiation is as follows:

`template` *return-type func-name<templ-specs>*`(`*arg-specs*`);`

Listing 4.13 shows how to explicitly instantiate the templates in Listing 4.9 and Listing 4.10.

Listing 4.13 myinst.cp: Explicitly Instantiating Templates

```
#include "templ.cp"

template class Templ<long>;          // class instantiation
template long Max<long>(long,long); // function instantiation
```

When you explicitly instantiate a function, you do not need to include in *templ-specs* any arguments that the compiler can deduce from *arg-specs*. For example, in Listing 4.13 you can instantiate `Max<long>()` like this:

```
template long Max<>(long, long);
  // The compiler can tell from the arguments
  // that you are instantiating Max<long>()
```

Use explicit instantiation to make your program compile faster. Because the instantiations can be in one file with no other code, you can even put them in a separate library.

The compiler also supports the explicit instantiation of non-template members. Listing 4.15 shows an example.

Listing 4.14 Explicit Instantiation of Non-Template Members

```
template <class T> struct X {
  static T i;
};

template <class T> T X<T>::i = 1;
template char X<char>::i;
```

NOTE Explicit instantiation is not in the ARM but is part of the ISO C++ standard.

Better Template Conformance

Versions 2.5 and later of CodeWarrior C++ enforces the ISO C++ standard more closely when translating templates than previous versions of CodeWarrior C++. By default this new template translation is off. To ensure that template source code follows the ISO C++ standard more closely, turn on the **ISO C++ Template Parser** option in the C/C++ Language Panel.

The compiler provides pragmas to help update your source code to the more conformant template features. The `parse_func_templ` pragma controls the new template features. The `parse_mfunc_templ` pragma controls the new template features for class member functions only. The `warn_no_typename` pragma warns for the missing use of the `typename` keyword required by the ISO C++ standard. See "parse_func_templ" on page 187, "parse_mfunc_templ" on page 188, and "warn_no_typename" on page 219 for more information.

When using the new template parsing features, the compiler enforces more careful use of the `typename` and template keywords, and follows different rules for resolving names during declaration and instantiation than before.

A qualified name that refers to a type and that depends on a template parameter must begin with `typename` (ISO C++, §14.6). Listing 4.15 shows an example.

Listing 4.15 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 4.16 shows an example.

Listing 4.16 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 4.17 shows an example.

Listing 4.17 Binding Non-dependent Identifiers

```
void f(char);

template <typename T> void tmpl_func()
{
  f(1); // Uses f(char); f(int) 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 C++, §14.6.2). See Listing 4.18.

Listing 4.18 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 C++, §14.6.2.2) and the context of its instantiation (ISO C++, §14.6.4.2). Listing 4.19 shows an example.

Listing 4.19 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 4.20.

Listing 4.20 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 4.21.

Listing 4.21 Assembly Statements Cannot Depend on Template Arguments

```
template <typename T> void asm_tmpl()
{
  asm { move #sizeof(T), D0 ); // ERROR: Not yet supported.
}
```

The compiler also supports the address of template-id rules. See Listing 4.22.

Listing 4.22 Address of Template-id Supported

```
template <typename T> void foo(T) {}
template <typename T> void bar(T) {}
...
foo{ &bar<int> );     // now accepted
```

5

C++ and Embedded Systems

This chapter describes how to develop software for embedded systems using CodeWarrior C++ compilers.

Overview

Embedded systems topics include:

- Activating EC++
- Differences Between ISO C++ and EC++
- EC++ Specifications
- Obtaining Smaller Code Size in C++

NOTE This chapter discusses program design strategies for embedded systems and is not meant to be a definitive solution. Currently, you can use the CodeWarrior C++ compiler to develop embedded systems that are compatible with Embedded C++ (EC++).

Activating EC++

To compile EC++ source code, enable the EC++ Compatibility Mode setting in the C/C++ Language Panel.

To test for EC++ compatibility mode at compile time, use the `__embedded_cplusplus` predefined symbol. For more information, see "Predefined Symbols" on page 111.

Differences Between ISO C++ and EC++

The EC++ proposal does not support the following ISO C++ (ANSI C++) features:

- Templates

Templates

ANSI C++ supports 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 ANSI 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 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.

You can disable certain C++ features, such as RTTI and exceptions, using the compiler settings in the C++ Language panel, described in "C/C++ Compiler Settings" on page 23.

Library-Related Issues

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

Obtaining Smaller Code Size in C++

Consider the following C++ programming strategies to ensure optimal code size:

- Compiler-related strategies
- Language-related strategies
- Library-related strategies

NOTE In all strategies, reducing object code size can affect program performance.

The EC++ proposal uses some of these strategies as part of its specification. Other strategies apply to C++ programming in general. Any C++ program can use these strategies, regardless of whether it follows the EC++ proposal or not.

Compiler-related strategies

Compiler-related strategies rely on compiler features to reduce object code size.

- Size Optimizations—use the compiler size optimization settings
- Inlining—control and limit effectiveness of `inline` directive

Size Optimizations

Metrowerks compilers include optimization settings for size or speed and various levels of optimization. Choose size as your desired outcome and the level of optimization to apply.

Optimization settings for your target are controlled by settings in the Processor panel.

When debugging, compile your code without any optimizations. Some optimizations disrupt the relationship between the source and object code required by the debugger. Optimize your code after you have finished debugging.

See also "C/C++ Compiler Settings" on page 23.

Inlining

With CodeWarrior, you can disable inlining, allow normal inlining, auto-inline, or set the maximum depth of inlining.

Inlining can reduce or increase code size. There is no definite answer for this question. Inlining small functions can make a program smaller, especially if you have a class library with a lot of getter/setter member functions.

However, MSL C++ defines many functions as inline, which is not good if you want minimal code size. For optimal code size when using MSL C++, disable inlining when building the library. If you are not using MSL C++, normal inlining and a common-sense use of the keyword `inline` might improve your code size.

In CodeWarrior, you control inlining as a language setting in the C/C++ Language Panel.

When debugging your code, disable inlining to maintain a clear correspondence between source and object code. After debugging, set the inlining level that has the best effect on your object code.

See also "Inlining" on page 42.

Language-related strategies

Language-related strategies limit or avoid the use of ISO C++ features. While these features can make software design and maintenance easier, they can also increase code size.

- Virtual Functions—Not using virtual functions reduces code size.

- Runtime Type Identification—Extra data is not generated if program does not use Runtime Type Identification (RTTI).
- Exception Handling—While the CodeWarrior C++ compiler provides zero-overhead exception handling to provide optimum execution speed, it still generates extra object code for exception support.
- Operator New—Do not throw an exception within the `new` operator.
- Multiple and Virtual Inheritance

Virtual Functions

For optimal code size, do not use virtual functions unless absolutely necessary. A virtual function is never dead-stripped, even if it is never called.

Runtime Type Identification

If code size is an issue, do not use RTTI because it generates a data table for every class. Disabling RTTI decreases the size of the data section.

The EC++ proposal does not allow runtime type identification. Use the C/C++ Language Panel to disable RTTI.

See also "Controlling RTTI" on page 66.

Exception Handling

If you must handle exceptions, be careful when using C++ exception handling routines. CodeWarrior has a zero runtime overhead error handling mechanism. However, using exceptions does increase code size, particularly the exception tables data.

The EC++ proposal does not allow exception handling. Use the C/C++ Language Panel to disable exception handling.

NOTE The proposed ISO standard libraries and the use of the `new` operator require exception handling.

Operator New

The C++ `new` operator might throw an exception, depending on how the runtime library implements the `new` operator. To make the `new` operator throw exceptions, set `__throws_bad_alloc` to `1` in the prefix file for your target and rebuild your library. To prevent the `new` operator from throwing exceptions, set `__throws_bad_alloc` to `0` in the prefix file for your target and rebuild your library.

See your release notes or *Targeting* manual for more information.

Multiple and Virtual Inheritance

Implementing multiple inheritance requires a modest amount of code and data overhead. The EC++ proposal does not allow multiple inheritance.

For optimal code size, do not use virtual inheritance. Virtual base classes are often complex and add a lot of code to the constructor and destructor functions.

The EC++ proposal does not allow virtual inheritance.

Library-related strategies

- Stream-Based Classes—MSL classes comprise a lot of object code.
- Alternative Class Libraries—Non-standard class libraries can provide a subset of the standard library's functionality with less overhead.

Stream-Based Classes

MSL C++ stream-based classes initialize several instances of direct and indirect objects. When code size is critical, do not use stream-based classes, which include standard input (`cin`), standard output (`cout`), and standard error (`cerr`). There are also wide-character equivalents for the normal input and output routines. Use only standard C input and output functions unless stream-based classes are absolutely necessary.

In addition to the standard C++ stream classes, avoid using string streams for in-core formatting because they generate heavy overhead. If size is critical, use C's `sprintf` or `sscanf` functions instead.

The EC++ proposal does not support templatized classes or functions. MSL adheres to the ISO proposed standards that are template-based.

Alternative Class Libraries

MSL C++ is based on the ISO proposed C++ standard, which is implemented using templates that have a large initial overhead for specialization.

To avoid this overhead, consider devising your own commonly-used vector, string, or utility classes. You can also use other class libraries, such as the NIH's (National Institute of Health) Class Library. If you do use an alternative library, beware of potential problems with virtual inheritance, RTTI, or other causes of larger code size as described above.

6

Improving Compiler Performance

This chapter describes compiler features that decrease the amount of time to translate source code.

When to Use Precompiled Files

Source code files in a project typically use many header files. Typically, the same header files are included by each source code file in a project, forcing the compiler to read these same header files repeatedly during compilation. To shorten the time spent compiling and recompiling the same header files, CodeWarrior C/C++ can precompile a header file, allowing it to be subsequently preprocessed much faster than a regular text source code file.

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

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

This header file can be precompiled so that, instead of preprocessing multiple duplications, the compiler needs to load just one precompiled header file.

What Can be Precompiled

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

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

  You cannot precompile libraries or other binary files.
- A C source code file that will be automatically precompiled must have `.pch` file name extension.
- A C++ source code file that will be automatically precompiled must have a `.pch++` file name extension.
- Precompiled files must have a `.mch` file name extension.
- The file to be precompiled does not have to be in a CodeWarrior IDE project, although a project must be open to precompile the file.

  The CodeWarrior IDE uses the build target settings to precompile a file.
- The file must not contain any statements that generate data or executable code.

  However, the file may define static data. C++ source code can contain inline functions and constant variable declarations (`const`).
- Precompiled header files for different build targets are not interchangeable.

  For example, to generate a precompiled header for use with Windows® compilers, you must use a Windows® compiler.
- C source code may not include precompiled C++ header files and C++ source code may not include precompiled C header files.
- A source file may include only one precompiled file.
- A file may not define any items before including a precompiled file.

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

Using a Precompiled Header File

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

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

TIP Instead of explicitly including a precompiled file in each source code file with the `#include` directive, put the precompiled file name in the **Prefix File** field of the C/C++ Language settings panel. If the Prefix File field already specifies a file name, include the precompiled file in the prefix file with the `#include` directive.

Listing 6.1 and Listing 6.2 show an example.

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

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

#pragma precompile_target "sock_precomp.mch"

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

Listing 6.2 Using a Precompiled File

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

#include "sock_precomp.mch"

int main(void)
{
  // ...
  return 0;
}
```

Preprocessing and Precompiling

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

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

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

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

Pragma Scope in Precompiled Files

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

For example, the source code in Listing 6.3 specifies that the variable `xxx` is a `far` variable.

Listing 6.3 Pragma Settings in a Precompiled Header

```
// my_pch.pch

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

#pragma far_data on
extern int xxx;
```

The source code in Listing 6.4 includes the precompiled version of Listing 6.3.

Listing 6.4 Pragma Settings in an Included Precompiled File

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

#include "my_pch.mch"   // this precompiled file sets far_data on

// far_data is still off but xxx is still a far variable
```

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

Precompiling a File in the CodeWarrior IDE

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

1. Start the CodeWarrior IDE.
2. Open or create a project.
3. Choose or create a build target in the project.

   The settings in the project's active build target will be used when preprocessing and precompiling the file you want to precompile.
4. Open the source code file to precompile.

   See "What Can be Precompiled" on page 88 for information on what a precompiled file may contain.
5. From the Project menu, choose Precompile.

   A save dialog box appears.
6. Choose a location and type a name for the new precompiled file.

   The IDE precompiles the file and saves it.
7. Click Save.

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

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

Updating a Precompiled File Automatically

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

- The text file name ends with .pch (for C header files) or .pch++ (for C++ header files)
- The file is in a project's build target.
- The file uses the `precompile_target` pragma ("precompile_target" on page 190).
- The file, or files it depends on, have been modified.

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

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

Use Instance Manager

The **Use Instance Manager** option reduces compile time by telling the compiler to generate any instance of a template (or non-inlined inline) function only once. Object code size and associated debug information size is reduced. This does not affect the output file size, though, since the compiler is effectively doing the same task as the linker in this mode.

You can control where the instance database is stored using the `#pragma instmgr_file`.

7

Preventing Errors & Bugs

CodeWarrior C/C++ compilers have features for catching bugs, inconsistencies, ambiguities, and redundancies in your source code, much like the `lint` programming utility. Most of these features come in the form of warnings, which the compiler emits when it translates suspicious source code.

CodeWarrior C/C++ Errors and Warnings

The C/C++ compiler generates errors when it cannot translate your source code or generate object code due to improper syntax. For descriptions of errors related to the CodeWarrior C/C++ compiler, see the *CodeWarrior Error Reference*.

Like the `lint` programming utility, the CodeWarrior C/C++ compiler can generate warnings that alert you to source code that is syntactically correct but logically incorrect or ambiguous. Because these warnings are not fatal, the compiler still translates your source code. However, your program might not run as you intended.

This section describes these warnings:

- Bad Conversions of Pointer Values

Warnings as Errors

If you enable the **Treat All Warnings as Errors** setting, the compiler treats all warnings as though they were errors. It does not compile a file successfully until you resolve all warnings.

The **Treat All Warnings as Errors** setting corresponds to the pragma `warning_errors`, described at"warning_errors" on page 206. To check this setting, use `__option (warning_errors)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Illegal Pragmas

If you enable the **Illegal Pragmas** setting, the compiler issues a warning when it encounters a pragma it does not recognize. For example, the pragma statements in Listing 7.1 generate warnings with the **Illegal Pragmas** setting enabled.

Listing 7.1 Illegal Pragmas

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

The **Illegal Pragmas** setting corresponds to the pragma `warn_illpragma`, described at "warn_illpragma" on page 211. To check this setting, use `__option (warn_illpragma)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Empty Declarations

If you enable the **Empty Declarations** setting, the compiler issues a warning when it encounters a declaration with no variable name.

For example:

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

The **Empty Declarations** setting corresponds to the pragma `warn_emptydecl`, described at "warn_emptydecl" on page 207. To check this setting, use `__option (warn_emptydecl)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Common Errors

If you enable the **Possible Errors** setting, the compiler generates a warning if it encounters common errors such as the following:

- An assignment in either a logical expression or the conditional portion of an `if`, `while`, or `for` expression. This warning is useful if you use = when you mean to use ==. Listing 7.2 shows an example.

Listing 7.2 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, no warning

if (a==b) f();         // OK: (a==b) is a comparison, no warning
```

- An equal comparison in a statement that contains a single expression. This check is useful if you use == when you meant to use =. Listing 7.3 shows an example.

Listing 7.3 Confusing = and == Operators in Assignments

```
a == 0;          // WARNING: This is a comparison.
a = 0;           // OK: This is an assignment, no warning
```

- A semicolon (`;`) directly after a `while`, `if`, or `for` statement. For example, the following statement generates a warning and is probably an unintended infinite loop:

```
while (i++);  // WARNING: Unintended infinite loop
```

  If you intended to create an infinite loop, put white space or a comment between the `while` statement and the semicolon. These statements suppress the above errors or warnings.

```
while (i++) ; // OK: White space separation, no warning
while (i++)   /*: Comment separation, no warning */ ;
```

The **Possible Errors** setting corresponds to the pragma `warn_possunwant`, described at "warn_possunwant" on page 221. To check this setting, use `__option (warn_possunwant)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Unused Variables

If you enable the **Unused Variables** setting, the compiler generates a warning when it encounters a local 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 7.4 shows an example.

Listing 7.4 Unused Local Variables Example

```
int error;
void foo(void)
{
  int temp, errer;           // ERROR: errer is misspelled
  error = do_something()
                             // WARNING: temp and error 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 7.5.

Listing 7.5 Suppressing Unused Variable Warnings

```
void foo(void)
{
  int i, temp, error;

  #pragma unused (i, temp) /* Do not warn that i and temp */
  error=do_something();    /* are not used */

}
```

The **Unused Variables** setting corresponds to the pragma `warn_unusedvar`, described at "warn_unusedvar" on page 226. To check this setting, use `__option (warn_unusedvar)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Unused Arguments

If you enable the **Unused Arguments** setting, the compiler generates a warning 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.

```
void foo(int temp,int errer); // ERROR: errer is misspelled
{
  error = do_something();     // WARNING: temp and error are
```

```
                                    //          unused.
}
```

You can declare an argument that you do not use in two ways without receiving this warning:

- Use the pragma `unused`, as in this example:

```
void foo(int temp, int error)
{
  #pragma unused (temp)
  /* Compiler does not warn that temp is not used */

  error=do_something();
}
```

- Disable the **ANSI Strict** setting and do not give the unused argument a name. (See "Unnamed Arguments in Function Definitions" on page 38.) Listing 7.6 shows an example.

Listing 7.6 Unused, Unnamed Arguments

```
void foo(int /* temp */, int error)
{
  /* Compiler does not warn that "temp" is not used.

  error=do_something(); */
}
```

The **Unused Arguments** setting corresponds to the pragma `warn_unusedarg`, described at "warn_unusedarg" on page 225. To check this setting, use `__option (warn_unusedarg)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Extra Commas

If you enable the **Extra Commas** setting, the compiler generates a warning when it encounters an extra comma. For example, this statement is legal in C but generates a warning when you enable this setting:

```
int a[] = { 1, 2, 3, 4, }; // ^ WARNING: Extra comma after 4
```

The **Extra Commas** setting corresponds to the pragma `warn_extracomma`, described at "warn_extracomma" on page 208. To check this setting, use `__option (warn_extracomma)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Suspicious Assignments and Incorrect Function Returns

If you enable the **Extended Error Checking** setting, the C compiler generates a warning if it encounters one of the following potential problems:

- A non-`void` function that does not contain a `return` statement. For example, the source code in Listing 7.7 generates a warning.

Listing 7.7 Non-void Function with no return Statement

```
main()        /* assumed to return int */
{
  printf ("hello world\n");
} /* WARNING: no return statement */
```

Listing 7.8 does not generate a warning.

Listing 7.8 Explicitly Specifying a Function's void Return Type

```
void main() /* function declared to return void */
{
  printf ("hello world\n");
}
```

- An integer or floating-point value assigned to an `enum` type. Listing 7.9 shows an example.

Listing 7.9 Assigning to an Enumerated Type

```
enum Day { Sunday, Monday, Tuesday, Wednesday,
           Thursday, Friday, Saturday } d;

d = 5;             /* WARNING */
d = Monday;        /* OK */
d = (Day)3  ;      /* OK */
```

- An empty `return` statement in a function that is not declared `void`. For example, the following code results in a warning:

```
int MyInit(void)
{
```

```
  int err = GetMyResources();
  if (err!=0) return;  /* ERROR: Empty return statement */

  /* ... */
```

This is OK:

```
int MyInit(void)
{
  int err = GetMyResources();
  if (err!=0) return -1; /* OK */

  /* ... */
```

The **Extended Error Checking** setting corresponds to the pragma `extended_errorcheck`, described at "extended_errorcheck" on page 150. To check this setting, use `__option (extended_errorcheck)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Hidden Virtual Functions

If you enable the **Hidden virtual functions** setting, the compiler generates a warning if you declare a non-virtual member function in a subclass that hides an inherited virtual function in a superclass. One function hides another if it has the same name but a different argument type. Listing 7.10 shows an example.

Listing 7.10 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 **Hidden virtual functions** setting corresponds to the pragma `warn_hidevirtual`, described at "warn_hidevirtual" on page 211. To check this setting, use `__option (warn_hidevirtual)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Implicit Arithmetic Conversions

The compiler converts values automatically from one type to another to perform some operations (ISO C, §3.2 and ISO C++, §4). These kinds of conversions are called "implicit conversions" because they are not explicitly stated in the source code.

The rules the compiler follows for deciding when to apply implicit conversions sometimes gives results you do not expect. If you enable the **Implicit Arithmetic Conversions** setting, the compiler issues a warning when it applies implicit conversions:

- the destination of an operation is not large enough to hold all possible results
- a signed value is implicitly converted to an unsigned value
- an integer value is implicitly converted to a floating-point value
- a floating-point value is implicitly converted to an integer value

For example, assigning the value of a variable of type `long` to a variable of type `char` results in a warning if you enable this setting.

The compiler also has pragmas that control specific of implicit conversions the compiler warns about (Table 7.1).

Table 7.1 Implicit Arithmetic Conversion Pragmas

This pragma…	Warns about this kind of conversion
`warn_illunionmembers`	a floating point value to an integer value
`warn_impl_i2f_conv`	an integer value to a floating-point value
`warn_impl_s2u_conv`	a signed value to an unsigned value
`warn_implicitconv`	all; this pragma is equivalent to the **Implicit Arithmetic Conversions** setting.

`inline` Functions That Are Not Inlined

If you enable the **Non-Inlined Functions** setting, the compiler issues a warning when it cannot inline a function.

This setting corresponds to pragma `warn_notinlined`. To check this setting, use `__option (warn_notinlined)`.

See "Checking Settings" on page 115 for information on how to use this directive.

Mixed Use of ‘class’ and ‘struct’ Keywords

If you enable the **Inconsistent ‘class’ / ‘struct’ Usage** setting, the compiler issues a warning if you use the `class` and `struct` keywords in the definition and declaration of the same identifier (Listing 7.11).

Listing 7.11 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 setting corresponds to pragma `warn_structclass`. To check this setting, use `__option (warn_structclass)`.

See “Checking Settings” on page 115 for information on how to use this directive.

Redundant Statements

If you enable the pragma `warn_no_side_effect`, the compiler issues a warning when it encounters a statement that produces no side effect. To prevent a statement with no side effects from signalling this warning, cast the statement with `(void)`. See Listing 7.12 for an example.

Listing 7.12 Example of Pragma warn_no_side_effect

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

Realigned Data Structures

If you enable the pragma `warn_padding`, the compiler warns about any bytes it adds to data structures to improve their memory alignment. Refer to the appropriate *Targeting* manual for more information on how the compiler pads data structures for a particular processor or operating system.

This pragma reports warnings for C source code only. It does not report warnings for C++ source code.

Ignored Function Results

If you enable the pragma warn_resultnotused, the compiler issues a warning when it encounters a statement that calls a function without using its result. To prevent this warning, cast the statement with (void). See Listing 7.13 for an example.

Listing 7.13 Example of Pragma warn_resultnotused

```
#pragma warn_resultnotused on
void foo(int a,int b)
{
   bar();       // warning: result of bar() is not used
   (void)bar(); // void cast suppresses warning
}
```

Bad Conversions of Pointer Values

Use either of the following pragmas to detect bad pointer value conversions:

- Enable the pragma warn_ptr_int_conv to have the compiler issue a warning when an expression converts a pointer value to an integral value that is not large enough to hold a pointer value.
- Enable the pragma warn_any_ptr_int_conv to have the compiler issue a warning when an expression converts a pointer value to an integral value that is not large enough to hold a pointer value or vice versa.

8

C Implementation-Defined Behavior

The ISO standard for C leaves many details about the form and translation of C programs up to the implementation of the C compiler. Section J.3 of the ISO C Standard lists the unique implementation-defined behaviors. Numbers in parentheses that begin with "§" indicate the ISO C standard section to which an implementation-defined behavior refers.

This chapter refers to implementation-defined behaviors of the compiler itself. For information on implementation-defined behaviors of the Standard C Library, consult the *MSL C Library Reference*.

How to Identify Diagnostic Messages

Diagnostics are error and warning messages the C compiler issues when it encounters improper program syntax (ISO C, §5.1.1.3).

Within the CodeWarrior IDE, the CodeWarrior C compiler issues diagnostic messages in the **Errors & Warnings** window. For more information, see the *IDE User Guide*.

From the command line, CodeWarrior C issues diagnostic messages to the standard error file.

For more information on the error and warning messages themselves, see the *CodeWarrior Error Reference*.

Arguments to main()

The `main()` function can accept two or more arguments (ISO C, §5.1.2.2.1) of the form:

```
int main(int argc, char *argv[]) { /*...*/ }
```

The values stored in the `argc` and `argv` arguments depend on CodeWarrior C's target platform.

For example, on Mac OS, these values are 0 and `NULL`, respectively, unless you call the `ccommand()` function in `main()` before any other function. See the appropriate *Targeting* manual and the *MSL C Reference* for more information.

Interactive Device

An *interactive device* is that part of a computer that accepts input from and provides output to a human operator (ISO C, §5.1.2.3). Traditionally, the conventional interactive devices are consoles, keyboards, and character display terminals.

Some versions of CodeWarrior C, usually for desktop platforms, provide features that emulate a character display device in a graphical window. For example, on Microsoft Windows, CodeWarrior C uses the Windows console window. On Mac OS, CodeWarrior C provides the SIOUX library.

Other versions of CodeWarrior C, usually for embedded systems that do not have a keyboard or display, provide console interaction through a serial or Ethernet connection between the target and host computers.

Refer to the *Targeting* manual for more information on the kind of consoles CodeWarrior C provides.

Identifiers

(ISO C, §6.2.4.1) CodeWarrior C recognizes the first 255 characters of identifiers, whether or not the identifiers have external linkage. In identifiers with external linkage, uppercase and lowercase characters are distinct.

Character Sets

CodeWarrior generally supports the 8-bit character set of the host OS.

Enumerations

See "Enumerated Types" on page 34.

Implementation Quantities

The C/C++ compiler has the implementation quantities listed in Table 8.1, based on the ISO C++ Standard. Although the values in the right-side column are the recommended minimums for each quantity, they do not determine the compliance of the quantity.

NOTE The right-side column value "unlimited" means unlimited only up to and including memory and time limitations.

Table 8.1 Implementation Quantities for the C/C++ Compiler

Quantity	Minimum
Nesting levels of compound statements, iteration control structures, and selection control structures [256]	Unlimited
Nesting levels of conditional inclusion [256]	32
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 (255 significant in identifiers)
Number of initial characters in an external identifier [1024]	Unlimited (255 significant in identifiers)
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]	128
Arguments in one macro invocation [256]	128
Characters in one logical source line [65536]	Unlimited
Characters in a character string literal or wide string literal (after concatenation) [65536]	Unlimited
Size of an object [262144]	2 GB

Table 8.1 Implementation Quantities for the C/C++ Compiler

Quantity	Minimum
Nesting levels for `#include` files [256]	32
`Case` labels for a `switch` statement (excluding those for any nested `switch` statements) [16384]	Unlimited
Data members in a single class, structure, or union [16384]	Unlimited
Enumeration constants in a single enumeration [4096]	Unlimited
Levels of nested class, structure, or union definitions in a single struct-declaration-list [256]	Unlimited
Functions registered by `atexit()` [32]	64
Direct and indirect base classes [16384]	Unlimited
Direct base classes for a single class [1024]	Unlimited
Members declared in a single class [4096]	Unlimited
Final overriding virtual functions in a class, accessible or not [16384]	Unlimited
Direct and indirect virtual bases of a class [1024]	Unlimited
Static members of a class [1024]	Unlimited
Friend declarations in a class [4096]	Unlimited
Access control declarations in a class [4096]	Unlimited
Member initializers in a constructor definition [6144]	Unlimited
Scope qualifications of one identifier [256]	Unlimited
Nested external specifications [1024]	Unlimited
Template arguments in a template declaration [1024]	Unlimited

Table 8.1 Implementation Quantities for the C/C++ Compiler

Quantity	Minimum
Recursively nested template instantiations [17]	Unlimited
Handlers per try block [256]	Unlimited
Throw specifications on a single function declaration [256]	Unlimited

Library Behaviors

This reference does not cover implementation-defined behaviors in the Metrowerks Standard Library for C (MSL C).

9

C++ Implementation-Defined Behavior

The ISO standard for C++ leaves many details about the form and translation of C++ programs up to the implementation of the C++ compiler. This chapter lists the parts of the of the C++ standard that are left to the implementation to define and how CodeWarrior C++ behaves in these situations. Numbers in parentheses that begin with "§" denote the section of the ISO C++ standard that an implementation-defined behavior refers to.

This chapter refers to implementation-defined behaviors of the compiler itself. For information of implementation-defined behaviors of the Standard C++ Library, consult the *MSL C++ Library Reference*.

Size of Bytes

(ISO C++, §1.7) The standard specifies that the size of a byte is implementation-defined. The size of a byte in C++ is specified in the *Targeting* manual of the platform you are compiling for. Refer to the "C and C++" chapter in your *Targeting* manual for the sizes of data types.

Interactive Devices

(ISO C++, §1.9) The standard specifies that an *interactive device*, the part of a computer that accepts input from and provides output to a human operator, is implementation-defined. The most common instance of an interactive device is a console; a keyboard and character display terminal.

Some versions of CodeWarrior C++, typically for desktop platforms, provide libraries to emulate a character display device in a graphical window. For example, on Microsoft Windows CodeWarrior C++ uses the Windows console window. On Mac OS, CodeWarrior C++ provide the SIOUX library to emulate a console.

Other versions of CodeWarrior C++, typically for embedded systems that do not have a keyboard or display, provide console interaction through a serial or Ethernet connection between the target and host computers.

Refer to the *Targeting* manual for more information on the kind of console CodeWarrior C++ provides.

Source File Handling

(ISO C++, §2.1) The standard specifies how source files are prepared for translation.

If trigraph expansion is turned on, CodeWarrior C++ converts trigraph sequences with their single-character representations. Trigraph expansion is controlled by the **Expand Trigraphs** option in the **C/C++ Language Settings** panel and `#pragma trigraphs`.

At preprocessing-time, a sequence of two or more white-space characters, except new-line characters, is converted to a single space.

New-line characters are left untouched, unless preceded by a backslash (“\”), in which case the proceeding line is appended.

Header File Access

(ISO C++, §2.8) The standard requires implementations to specify how header names are mapped to actual files.

The CodeWarrior IDE manages the correspondence of header names to header files. See the *IDE User Guide* for information on using its access path and source tree features.

If you use CodeWarrior C++ on the command line, see “Command-Line Tools” on page 301 for information on specifying how CodeWarrior C++ should search for header files.

Character Literals

(ISO C++, §2.13.2) The standard specifies that a multicharacter literal, two or more characters surrounded by single quotes, has an implementation-defined value. See “Character Constants as Integer Values” on page 41 for information on how CodeWarrior C++ translates such values.

A character literal that begins with the letter “L” (without the quotes) is a wide-character literal. Each target translates wide-character literals in its own way; most targets use either two or four-byte wide characters. An escape sequence that uses an octal or hexadecimal value outside the range of `char` or `wchar_t` results in an error.

See your target documentation for more information on how your target handles wide-character literals.

10

Predefined Symbols

CodeWarrior C/C++ compilers define several preprocessor and compiler symbols that give you information about the compile-time environment. The compiler evaluates these symbols at compile time, not runtime. The topics in this section are:

- ANSI Predefined Symbols
- Metrowerks Predefined Symbols

ANSI Predefined Symbols

Table 10.1 lists the symbols required by the ANSI/ISO C standard.

Table 10.1 ANSI Predefined Symbols

This symbol…	is…
`__DATE__`	The date the file is compiled; for example, `"Jul 14, 1995"`. This symbol is a predefined macro.
`__FILE__`	The name of the file being compiled; for example `"prog.c"`. This symbol is a predefined macro.
`__func__`	The name of the function currently being compiled. This predefined identifier is only available under the emerging ANSI/ISO C99 standard.
`__LINE__`	The number of the line being compiled (before including any header files). This symbol is a predefined macro.

Table 10.1 ANSI Predefined Symbols

This symbol…	is…
`__TIME__`	The time the file is compiled in 24-hour format; for example, `"13:01:45"`. This symbol is a predefined macro.
`__STDC__`	Defined as `1` if compiling C source code; undefined when compiling C++ source code. This macro lets you know that Metrowerks C implements the ANSI C standard.

Listing 10.1 shows a small program that uses the ANSI predefined symbols.

Listing 10.1 Using ANSI Predefined Symbols

```
#include <stdio.h>

void main(void)
{
  printf("Hello World!\n");

  printf("%s, %s\n", __DATE__, __TIME__);
  printf("%s, line: %d\n", __FILE__, __LINE__);
}

/* The program prints something like the following:
Hello World!
Oct 31 1995, 18:23:50
main.ANSI.c, line: 10
*/
```

Metrowerks Predefined Symbols

Table 10.2 lists additional symbols provided by Metrowerks C/C++ but not defined as part of the ANSI/ISO C/C++ standards.

Table 10.2 Predefined Symbols for CodeWarrior C/C++ compilers

This symbol…	is…
`__A5__`	Defined as `1` if data is A5-relative, `0` if data is A4-relative. This symbol is defined for 68K compilers and undefined for other target platforms.
`__ALTIVEC__`	Currently defined as `100000000` when you enable PowerPC AltiVec™ features using `pragma altivec_model`.
`__cplusplus`	Defined as `199711L` if you are compiling this file as a C++ file; undefined if you are compiling this file as a C file. The value `199711L` indicates conformance with the ANSI/ISO C++ specification.
`__embedded_ cplusplus`	Defined as 1 if EC++ is activated; undefined if EC++ is not activated.
`__embedded__`	Defined as `1` if you are compiling code for an embedded target.
`__FUNCTION__`	Defined by the compiler as the name of the function currently being compiled. This symbol is a predefined identifier for GCC compatibility.
`__fourbyteints__`	Defined as `1` if you enable the **4-byte Ints** setting in the 68K Processor panel; `0` if you disable that setting. This symbol is defined for 68K compilers and undefined for other platforms.
__ide_target("*target_name*")	Returns `1` if *target_name* is the same as the active build target in the CodeWarrior IDE's active project. Returns `0` otherwise.
`__IEEEdoubles__`	Defined as `1` if you enable the **8-Byte Doubles** setting in the 68K Processor panel; `0` if you disable that setting. This symbol is defined for 68K compilers and undefined for all other target platforms.
`__INTEL__`	Defined as `1` if you are compiling this code with the x86 compiler; undefined for all other target platforms.

Table 10.2 Predefined Symbols for CodeWarrior C/C++ compilers

This symbol…	is…
`__MC68K__`	Defined as `1` if you are compiling this code with the 68K compiler; undefined for all other target platforms.
`__MC68020__`	Defined as `1` if you enable the **68020 Codegen** setting in the **Processor** panel; `0` if you disable it. This symbol is defined for 68K compilers and undefined for all other target platforms.
`__MC68881__`	defined as `1` if you enable the **68881 Codegen** setting in the **68K Processor** panel; `0` if you disable it. This symbol is defined for 68K compilers and undefined for all other target platforms.
`__MIPS__`	Defined as `1` for MIPS compilers; undefined for other target platforms.
`__MIPS_ISA2__`	Defined as `1` if the compiler's target platform is MIPS and you select the **ISA II** checkbox in the **MIPS Processor** panel. Undefined if you deselect the **ISA II** checkbox. It is always undefined for other target platforms.
`__MIPS_ISA3__`	Defined as `1` if the compiler's target platform is MIPS and you select the **ISA III** checkbox in the **MIPS Processor** panel. Undefined if you deselect the **ISA III** checkbox. It is always undefined for other target platforms.
`__MIPS_ISA4__`	Defined as `1` if the compiler's target platform is MIPS and you select the **ISA IV** checkbox in the **MIPS Processor** panel. Undefined if you deselect the **ISA IV** checkbox. It is always undefined for other target platforms
`__MWBROWSER__`	Defined as `1` if the CodeWarrior browser is parsing your code; `0` if not.

Table 10.2 Predefined Symbols for CodeWarrior C/C++ compilers

This symbol…	is…
`__MWERKS__`	Defined as the version number of the Metrowerks C/C++ compiler if you are using the CodeWarrior CW7 that was released in 1995. For example, with the Metrowerks C/C++ compiler version 2.2, the value of `__MWERKS__` is `0x2200`. This macro is defined as `1` if the compiler was issued before the CodeWarrior CW7 that was released in 1995.
`__PRETTY_FUNCTION__`	Defined by the compiler as the name of the qualified (“unmangled”) C++ function currently being compiled. This predefined identifier is only defined when the C++ compiler is active. See “C++ Compiler” on page 59 for related information.
`__profile__`	Defined as `1` if you enable the **Generate Profiler Calls** setting in the **Processor** panel; `0` if you disable it.
`__POWERPC__`	Defined as `1` if you are compiling this code with the PowerPC compiler; `0` if not.
`__VEC__`	Defined as the version of Motorola’s *AltiVec ™ Technology Programming Interface Manual* to which the compiler conforms. This value takes the form `vrrnn` which corresponds to the version number (*v.rr.nn*) of the *Programming Interface Manual*. Otherwise, this macro is undefined. See “altivec_model” on page 231 for related information.
`macintosh`	Defined as `1` if you are compiling this code with the 68K or PowerPC compilers for Mac OS; `0` if not.

Checking Settings

The preprocessor function `__option()` lets you check pragmas and other settings that control the C/C++ compiler and code generation. You typically modify these settings using various panels in the **Project Settings** dialog box.

The syntax for this preprocessor function is as follows:

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

If the specified setting is enabled, `__option()` returns `1`; otherwise it returns `0`. If *setting-name* is unrecognized, `__option()` returns false.

Use this function when you want one source file to contain code that uses different settings. The example below shows how to compile one series of lines if you are compiling for machines with the MC68881 floating-point unit and another series if you are compiling for machines without it:

```
#if __option (code68881)  // Code for 68K chip with FPU
#else
                          // Code for any 68K processor
#endif
```

Table 10.3 lists all the setting names you can use in the preprocessor function `__option()`.

Table 10.3 Preprocessor Setting Names for `__option()`

This argument...	Corresponds to the…
`a6frames`	Pragma `a6frames`.
`align_array_members`	Pragma `align_array_members`.
`altivec_codegen`	Pragma `altivec_codegen`.
`altivec_model`	Pragma `altivec_model`.
`altivec_vrsave`	Pragma `altivec_vrsave`.
`always_inline`	Pragma `always_inline`.
`ANSI_strict`	**ANSI Strict** setting in the C/C++ Language Panel and pragma `ANSI_strict`.
`arg_dep_lookup`	Pragma `arg_dep_lookup`.
`ARM_conform`	Pragma `ARM_conform`.
`array_new_delete`	Pragma `array_new_delete`
`auto_inline`	**Auto-Inline** setting of the **Inlining** menu in the C/C++ Language Panel and pragma `auto_inline`.
`bool`	**Enable bool Support** setting in the C/C++ Language Panel and pragma `bool`.

Table 10.3 Preprocessor Setting Names for `__option()`

This argument...	Corresponds to the...
`check_header_flags`	Pragma `check_header_flags`.
`code68020`	Pragma `code68020`.
`code68881`	Pragma `code68881`.
`const_multiply`	Pragma `const_multiply`.
`const_strings`	Pragma `const_strings`.
`cplusplus`	**Force C++ Compilation** setting in the C/C++ Language Panel, the pragma `cplusplus`, and the macro `cplusplus`. Indicates whether the compiler is compiling this file as a C++ file.
`cpp_extensions`	Pragma `cpp_extensions`.
`d0_pointers`	Pragmas `pointers_in_D0` and `pointers_in_A0`.
`def_inherited`	Pragma `def_inherited`.
`defer_codegen`	Pragma `defer_codegen`.
defer_defarg_parsing	Pragma `defer_defarg_parsing`
`direct_destruction`	No longer available
`direct_to_SOM`	Pragma `direct_to_SOM`.
`disable_registers`	Pragma `disable_registers`.
`dollar_identifiers`	Pragma `dollar_identifiers`.
`dont_inline`	**Don't Inline** setting in the C/C++ Language Panel and pragma `dont_inline`.
`dont_reuse_strings`	**Reuse Strings** setting in the C/C++ Language Panel and pragma `dont_reuse_strings`.
`ecplusplus`	Pragma `ecplusplus`.
`EIPC_EIPSW`	Pragma `EIPC_EIPSW`.

Table 10.3 Preprocessor Setting Names for `__option()`

This argument...	Corresponds to the...
`enumsalwaysint`	**Enums Always Int** setting in the C/C++ Language Panel and pragma `enumsalwaysint`.
`exceptions`	**Enable C++ Exceptions** setting in the C/C++ Language Panel and pragma `exceptions`.
`export`	Pragma `export`.
`extended_errorcheck`	**Extended Error Checking** setting in the C/C++ Warnings Panel and pragma `extended_errorcheck`.
`far_data`	Pragma `far_data`.
`far_strings`	Pragma `far_strings`.
`far_vtables`	Pragma `far_vtables`.
`faster_pch_gen`	Pragma `faster_pch_gen`.
`float_constants`	Pragma `float_constants`.
`force_active`	Pragma `force_active`.
`fourbyteints`	Pragma `fourbyteints`.
`fp_contract`	Pragma `fp_contract`.
`fullpath_prepdump`	Pragma `fullpath_prepdump`.
`function_align`	Pragma `function_align`.
`gcc_extensions`	Pragma `gcc_extensions`.
`IEEEdoubles`	Pragma `IEEEdoubles`.
`ignore_oldstyle`	Pragma `ignore_oldstyle`.
`import`	Pragma `import`.
`inline_bottom_up`	Pragma `inline_bottom_up`.
inline_bottom_up_once	Pragma `inline_bottom_up_once`
`inline_intrinsics`	Pragma `inline_intrinsics`.

Table 10.3 Preprocessor Setting Names for `__option()`

This argument...	Corresponds to the…
`inline_max_auto_size`	Pragma `inline_max_auto_size`
inline_max_size	Pragma `inline_max_size`
`inline_max_total_size`	Pragma `inline_max_total_size`
`internal`	Pragma `internal`.
`interrupt`	Pragma `interrupt`.
`k63d`	**K6 3D Favored** setting in the **Extended Instruction Set** menu of the **x86 CodeGen** panel and pragma `k63d`.
`k63d_calls`	**MMX + K6 3D** setting in the **Extended Instruction Set** menu of the **x86 CodeGen** panel and pragma `k63d_calls`.
`lib_export`	Pragma `lib_export`.
`line_prepdump`	Pragma `line_prepdump`.
`little_endian`	No option. Defined as `1` if you are compiling for a little endian target (such as x86); `0` if you are compiling for a big endian target (such as PowerPC).
`longlong`	Pragma `longlong`.
`longlong_enums`	Pragma `longlong_enums`.
`longlong_prepeval`	Pragma `longlong_prepeval`.
`macsbug`	Pragma `macsbug`.
max_errors	Pragma `max_errors`
`microsoft_exceptions`	Pragma `microsoft_exceptions`.
`microsoft_RTTI`	Pragma `microsoft_RTTI`.
`mmx`	**MMX** setting in the **Extended Instruction Set** menu of the **x86 CodeGen** panel and pragma `mmx`.
`mmx_call`	Pragma `mmx_call`.

Table 10.3 Preprocessor Setting Names for `__option()`

This argument...	Corresponds to the...
`mpwc`	Pragma `mpwc`.
`mpwc_newline`	Pragma `mpwc_newline`.
`mpwc_relax`	Pragma `mpwc_relax`.
msg_show_lineref	Pragma `msg_show_lineref`
msg_show_realref	Pragma `msg_show_realref`
multibyteaware_ preserve_literals	Pragma `multibyteaware_ preserve_literals`
no_conststringconv	Pragma `no_conststringconv`
`no_register_coloring`	Pragma `no_register_coloring`.
`no_static_dtors`	Pragma `no_static_dtors`.
no_conststringconv	Pragma `no_conststringconv`
`oldstyle_symbols`	Pragma `oldstyle_symbols`.
`only_std_keywords`	**ANSI Keywords Only** setting in the C/C++ Language Panel and pragma `only_std_keywords`.
`opt_classresults`	Pragma `opt_classresults`
`opt_common_subs`	Pragma `opt_common_subs`.
`opt_dead_assignments`	Pragma `opt_dead_assignments`.
`opt_dead_code`	Pragma `opt_dead_code`.
`opt_lifetimes`	Pragma `opt_lifetimes`.
`opt_loop_invariants`	Pragma `opt_loop_invariants`.
`opt_propagation`	Pragma `opt_propagation`.
`opt_strength_ reduction`	Pragma `opt_strength_reduction`.
`opt_strength_ reduction_strict`	Pragma `opt_strength_reduction_strict`.

Table 10.3 Preprocessor Setting Names for `__option()`

This argument...	Corresponds to the...
`opt_unroll_loops`	Pragma `opt_unroll_loops`.
`opt_vectorize_loops`	Pragma `opt_vectorize_loops`.
`optimize_for_size`	Pragma `optimize_for_size`.
`optimizewithasm`	Pragma `optimizewithasm`.
`pool_data`	Pragma `pool_data`.
`pool_strings`	**Pool Strings** setting in the C/C++ Language Panel and pragma `pool_strings`.
`ppc_unroll_ speculative`	Pragma `ppc_unroll_speculative`.
`precompile`	Whether or not the file is precompiled.
`preprocess`	Whether or not the file is preprocessed.
`profile`	Pragma `profile`.
`readonly_strings`	Pragma `readonly_strings`.
`register_coloring`	Pragma `register_coloring`.
`require_prototypes`	**Require Function Prototypes** setting in the C/C++ Language Panel and pragma `require_prototypes`.
`RTTI`	**Enable RTTI** setting in the C/C++ Language Panel and pragma `RTTI`.
showmessagenumber	Pragma `showmessagenumber`
`side_effects`	Pragma `side_effects`.
`simple_prepdump`	Pragma `simple_prepdump`.
`SOMCallOptimization`	Pragma `SOMCallOptimization`.
`SOMCheckEnvironment`	Pragma `SOMCheckEnvironment`.
`stack_cleanup`	Pragma `stack_cleanup`.
`suppress_init_code`	Pragma `suppress_init_code`.

Table 10.3 Preprocessor Setting Names for `__option()`

This argument...	Corresponds to the...
`suppress_warnings`	Pragma `suppress_warnings`.
`sym`	Marker in the project window debug column and pragma `sym`.
`syspath_once`	Pragma `syspath_once`.
text_encoding	Pragma `text_encoding`
`toc_data`	Pragma `toc_data`.
`traceback`	Pragma `traceback`.
`trigraphs`	**Expand Trigraphs** setting in the C/C++ Language Panel and pragma `trigraphs`.
`unsigned_char`	**Use Unsigned Chars** setting in the C/C++ Language Panel and pragma `unsigned_char`.
`use_fp_instructions`	Pragma `use_fp_instructions`.
`use_frame`	Pragma `use_frame`.
`use_mask_registers`	Pragma `use_mask_registers`.
`warning`	Pragma `warning`
warn_any_ptr_int_conv	Pragma `warn_any_ptr_int_conv`
`warn_emptydecl`	**Empty Declarations** setting in the C/C++ Warnings Panel and pragma `warn_emptydecl`.
`warn_extracomma`	**Extra Commas** setting in the C/C++ Warnings Panel and pragma `warn_extracomma`.
warn_hiddenlocals	Pragma `warn_hiddenlocals`
`warn_hidevirtual`	**Hidden virtual functions** setting in the C/C++ Warnings Panel and pragma `warn_hidevirtual`.

Table 10.3 Preprocessor Setting Names for `__option()`

This argument...	Corresponds to the...
`warn_illegal_ instructions`	Pragma `warn_illegal_instructions`.
`warn_illpragma`	**Illegal Pragmas** setting in the C/C++ Warnings Panel and pragma `warn_illpragma`.
warn_illtokenpasting	Pragma `warn_illtokenpasting`
warn_illunionmembers	Pragma `warn_illunionmembers`
`warn_impl_f2i_conv`	Pragma `warn_impl_f2i_conv`.
`warn_impl_i2f_conv`	Pragma `warn_impl_i2f_conv`.
`warn_impl_s2u_conv`	Pragma `warn_impl_s2u_conv`.
`warn_implicitconv`	**Implicit Arithmetic Conversions** setting in the C/C++ Warnings Panel and pragma `warn_implicitconv`.
`warn_missingreturn`	Pragma `warn_missingreturn`
`warn_no_explicit_ virtual`	Pragma `warn_no_explicit_virtual`
`warn_no_side_effect`	pragma `warn_no_side_effect`.
`warn_notinlined`	**Non-Inlined Functions** setting in the C/C++ Warnings Panel and pragma `warn_notinlined`.
`warn_padding`	pragma `warn_padding`.
`warn_possunwant`	**Possible Errors** setting in the C/C++ Warnings Panel and pragma `warn_possunwant`.
`warn_ptr_int_conv`	pragma `warn_ptr_int_conv`.
`warn_resultnotused`	pragma `warn_resultnotused`.

Table 10.3 Preprocessor Setting Names for `__option()`

This argument...	Corresponds to the…
`warn_structclass`	**Inconsistent 'class' / 'struct' Usage** setting in the C/C++ Warnings Panel and pragma `warn_structclass`.
warn_undefmacro	pragma `warn_undefmacro`
`warn_unusedarg`	**Unused Arguments** setting in the C/C++ Warnings Panel and pragma `warn_unusedarg`.
`warn_unusedvar`	**Unused Variables** setting in the C/C++ Warnings Panel and pragma `warn_unusedvar`.
`warning_errors`	**Treat Warnings As Errors** setting in the C/C++ Warnings Panel and pragma `warning_errors`.
`wchar_type`	**Enable wchar_t Support** setting in the C/C++ Warnings Panel and pragma `wchar_type`.

11

Common Pragmas

You configure the compiler for a project by changing the settings in the C/C++ Preprocessor Panel, C/C++ Language Panel, or the C/C++ Warnings Panel. You can also control compiler behavior in your code by including the appropriate pragmas.

Many of the pragmas correspond to settings in the C/C++ Language Panel and the settings panels for processors and operating systems.

Typically, you use these panels to select the settings for most of your code and use pragmas to change settings for special cases. For example, within the C/C++ Language Panel, you can disable a time-consuming optimization and then use a pragma to re-enable the optimization only for the code that benefits the most.

TIP If you use Metrowerks command-line tools, such as those for OS X or UNIX, see "Command-Line Tools" on page 301 for information on how to duplicate the effect of `#pragma` statements using command-line tool options.

The sections in this chapter are:

Pragma Syntax

Most pragmas have this syntax:

```
#pragma setting-name on | off | reset
```

Generally, use `on` or `off` to change the setting, then use `reset` to restore the original setting, as shown below:

```
#pragma profile off
  // If the Generate Profiler Calls setting is on,
  // turns it off for these functions.

#include <smallfuncs.h>

#pragma profile reset
  // If the Generate Profiler Calls setting was originally on,
```

```
// turns it back on. Otherwise, the setting remains off
```

Suppose that you use `#pragma profile on` instead of `#pragma profile reset`. If you later disable **Generate Profiler Calls** from the **Preference** dialog box, that pragma turns it on. Using `reset` ensures that you do not inadvertently change the settings in the **Project Settings** dialog box.

TIP To catch pragmas that the CodeWarrior C/C++ compiler does not recognize, use the `warn_illpragma` pragma. See also "Illegal Pragmas" on page 94.

Pragma Scope

The scope of a pragma setting is usually limited to a single compilation unit.

As discussed in "Pragma Syntax" on page 125, you should use `on` or `off` after the name of the pragma to change its setting to the desired condition. All code after that point is compiled with that setting until either:

- You change the setting with `on`, `off`, or (preferred) `reset`.
- You reach the end of the compilation unit.

At the beginning of each file, the compiler reverts to the project or default settings.

Common Pragma Reference

NOTE See your target documentation for information on any pragmas you use in your programs. If your target documentation covers any of the pragmas listed in this section, the information provided by your target documentation always takes precedence.

access_errors

Controls whether or not to change illegal access errors to warnings.

Syntax

```
#pragma access_errors on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues an error instead of a warning when it detects illegal access to protected or private members. This has the effect of including the file once (like `#pragma once`). This is not a portable usage.

This pragma does not correspond to any panel setting. To check this setting, use `__option (access_errors)`, described in "Checking Settings" on page 115. By default, this pragma is enabled.

align

See options.

always_import

Controls whether or not `include` statements are treated as `import` statements.

Syntax

```
#pragma always_import on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler treats all `include` statements as `import` statements.

This pragma does not correspond to any panel setting. To check this setting, use `__option (always_import)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

always_inline

Controls the use of inlined functions.

Syntax

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

Targets

All platforms.

Remarks

This pragma is strongly deprecated. Use the `inline_depth()` pragma instead.

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

This pragma does not correspond to any panel setting. To check this setting, use `__option (always_inline)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

ANSI_strict

Controls the use of non-standard language features.

Syntax

```
#pragma ANSI_strict on | off | reset
```

Targets

All platforms.

Remarks

If you enable the pragma `ANSI_strict`, the compiler generates an error if it encounters any of the following common ANSI extensions:

– C++-style comments. Listing 11.1 shows an example.

Listing 11.1 C++ Comments

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

– Unnamed arguments in function definitions. Listing 11.2 shows an example.

Listing 11.2 Unnamed Arguments

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

– A # token that does not appear before an argument in a macro definition. Listing 11.3 shows an example.

Listing 11.3 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"  */
```

– An identifier after `#endif`. Listing 11.4 shows an example.

Listing 11.4 Identifiers After #endif

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

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

This pragma corresponds to the **ANSI Strict** setting in the C/C++ Language Panel. To check this setting, use `__option (ANSI_strict)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

arg_dep_lookup

Controls C++ argument-dependent name lookup.

Syntax

```
#pragma arg_dep_lookup on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the C++ compiler uses argument-dependent name lookup.

This pragma does not correspond to any panel setting. To check this setting, use `__option (arg_dep_lookup)`, described in "Checking Settings" on page 115. By default, this setting is `on`.

ARM_conform

This pragma is no longer available.

ARM_scoping

Controls the scope of variables declared in the expression parts of if, while, and for statements.

Syntax

```
#pragma ARM_scoping on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, any variable you declare in any of the above conditional expressions remains valid until the end of the block that contains the statement. Otherwise, the variables only remains valid until the end of that statement. Listing 11.5 shows an example.

Listing 11.5 Example of Using Variables Declared in `for` Statement

```
for(int i=1; i<1000; i++) { /* . . . */ }
return i;  // OK if ARM_conform is enabled.
```

This pragma corresponds to the **ARM Conformance** setting in the C/C++ Language Panel. To check this setting, use `__option (ARM_scoping)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

array_new_delete

Use to enable the operator `new[]/delete[]` in array allocation/deallocation operations.

Syntax

```
#pragma array_new_delete on | off | reset
```

Targets

All platforms.

Remarks

To check this setting, use `__option (array_new_delete)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

asmpoundcomment

Controls whether the "#" symbol is treated as a comment character in inline assembly.

Syntax

```
#pragma asmpoundcomment on | off | reset
```

Targets

All platforms.

Remarks

NOTE Some targets may have additional comment characters, and may treat these characters as comments even when `#pragma asmpoundcomment off` is used.

WARNING! Using this pragma may interfere with the assembly language of a specific target.

This pragma does not correspond to any panel setting. To check this setting, use `__option (asmpoundcomment)`, described in "Checking Settings" on page 115. 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
```

Targets

All platforms.

Remarks

NOTE Some targets may have additional comment characters, and may treat these characters as comments even when `#pragma asmsemicolcomment off` is used.

WARNING! Using this pragma may interfere with the assembly language of a specific target.

This pragma does not correspond to any panel setting. To check this setting, use `__option (asmsemicolcomment)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

auto_inline

Controls which functions to inline.

Syntax

```
#pragma auto_inline on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler automatically chooses functions to inline for you, in addition to functions declared "inline".

Note that if you enable either the **Don't Inline** setting ("Inlining" on page 42) or the `dont_inline` pragma ("dont_inline" on page 145), 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 C/C++ Language Panel. To check this setting, use `__option (auto_inline)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

See also "Inlining" on page 42.

bool

Determines whether or not `bool`, `true`, and `false` are treated as keywords.

Syntax

```
#pragma bool on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, you can use the standard C++ `bool` type to represent `true` and `false`. Disable this pragma if recognizing `bool`, `true`, or `false` as keywords causes problems in your program.

NOTE This only applies when C++ is enabled using either the **Force C++ Compilation** option or `#pragma cplusplus on`.

This pragma corresponds to the **Enable bool Support** setting in the C/C++ Language Panel, described in "Using the bool Type" on page 67. To check this setting, use `__option(bool)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

c99

Controls the use of a subset of C99 language features.

Syntax

```
#pragma c99 on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler lets you use the following C99 language features that are supported by CodeWarrior:

– Trailing commas in enumerations

– GCC/C9x style compound literals. Listing 11.6 shows an example.

Listing 11.6 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. Listing 11.7 shows an example.

Listing 11.7 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
```

– `__func__` predefine

– Implicit `return 0;` in `main()`

– Non-`const` static data initializations

– Variable argument macros (`__VA_ARGS__`)

– `bool` / `_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. Listing 11.8 shows an example.

Listing 11.8 Example of an Empty Array as the Last struct Member

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

- Designated initializers
- 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 syntax is shown in Table 11.1.

Table 11.1 C99 Hexadecimal Floating-Point Constants

Constant Type	Format
hex-fp-constant	hex-string ['.' hex-string] hex-exp [size]
hex-string	hex-digit { hex-digit }
hex-exp	'p' decimal-string
size	'f' | 'F' | 'l' | 'L'
decimal-string	decimal-digit { decimal-digit }
decimal-digit	<any decimal digit>
hex-digit	<any hexadecimal digit>

The compiler generates a warning when the mantissa is more precise than the host floating point format, and warnings are enabled. It generates an error if the exponent is too wide for the host float format.

Examples include:

```
0x2f.3a2p3
```

```
0xEp1f
0x1.8p0L
```

The standard library supports printing floats in this format using the "`%a`" and "`%A`" specifiers.

– Variable length arrays are supported within local or function prototype scope (as required by the C99 standard), as shown in Listing 11.9.

Listing 11.9 Example of C99 Variable Length Array usage

```
#pragma c99 on

void f(int n) {
   int arr[n];
   ...
}
```

While the example shown in Listing 11.10 generates an error.

Listing 11.10 Bad Example of C99 Variable Length Array usage

```
#pragma c99 on

int n;
int arr[n];  // generates an 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 11.11.

Listing 11.11 Bad Example of C99 usage in Function Prototype

```
#pragma c99 on

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

– Unsuffixed decimal constant rules supported. Example shown in Listing 11.12.

Listing 11.12 Examples of C99 Unsuffixed Constants

```
#pragma c99 on  // and 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)
```

The following C99 features are not currently supported:

- long long bitfields on some targets
- `++bool--` expressions
- `(T) (int-list)` are handled/parsed as cast-expressions and as literals
- `__STDC_HOSTED__` is `0`
- `<bool> op= <expr>` does not always work correctly

This pragma corresponds to the **Enable C99 Extensions** setting in the C/C++ Language Panel. To check this setting, use `__option (c99)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

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
```

Targets

All platforms.

Remarks

This pragma affects precompiled headers only.

If you enable this pragma, the compiler ensures that the precompiled header's preferences for `double` size (8-byte or 12-byte), `int` size (2-byte or 4-byte) and floating point math correspond to the build target's settings. If they do not match, the compiler generates an error.

If your precompiled header file depends on these settings, enable this pragma. Otherwise, disable it.

This pragma does not correspond to any panel setting. To check this setting, use `__option (check_header_flags)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

const_strings

Controls the `const`-ness of string literals.

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the type of string literals is an array `const char[`*n*`]`, or `const wchar_t[`*n*`]` for wide strings, where *n* is the length of the string literal plus 1 for a terminating `NUL` character. Otherwise, the type `char[`*n*`]` or `wchar_t[`*n*`]` is used.

This pragma does not correspond to any setting in the C/C++ Language Panel. To check this setting, use `__option (const_strings)`, described in "Checking Settings" on page 115. By default, this pragma is enabled when compiling C++ source code; disabled when compiling C source code.

cplusplus

Controls whether or not to translate subsequent source code as C or C++ source code.

Syntax

```
#pragma cplusplus on | off | reset
```

Targets

All platforms.

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 C/C++ Language Panel. To check this setting, use `__option (cplusplus)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

cpp_extensions

Controls language extensions to ISO C++.

Syntax

```
#pragma cpp_extensions on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, you can use the following extensions to the ANSI C++ standard that would otherwise be illegal:

- Anonymous `struct` & `union` objects. Listing 11.13 shows an example.

Listing 11.13 Example of Anonymous `struct` & `union` Objects

```
#pragma cpp_extensions on
void foo()
{
  union {
    long  hilo;
    struct { short hi, lo; }; //  anonymous struct
  };
  hi=0x1234;
  lo=0x5678;  //  hilo==0x12345678
```

```
}
```

- Unqualified pointer to a member function. Listing 11.14 shows an example.

Listing 11.14 Example of an Unqualified Pointer to a Member Function

```
#pragma cpp_extensions on
struct Foo { void f(); }
void Foo::f()
{
  void (Foo::*ptmf1)() = &Foo::f; // ALWAYS OK

  void (Foo::*ptmf2)() = f; // OK if you enable cpp_extensions.
}
```

- Inclusion of `const` data in precompiled headers.

  This pragma does not correspond to any setting in the C/C++ Language Panel. To check this setting, use the `__option (cpp_extensions)`, described in "Checking Settings" on page 115. By default, this pragma is enabled if generating Win32/x86-compatible object code; disabled if not.

debuginline

Controls whether the compiler emits debugging information for expanded inline function calls.

Syntax

```
#pragma debuginline on | off | reset
```

Targets

All platforms.

Remarks

If the compiler emits debugging information for inline function calls, then the debugger can step to the body of the inlined function. This 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. To check this setting, use `__option (debuginline)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

def_inherited

Controls the use of `inherited`.

Syntax

```
#pragma def_inherited on | off | reset
```

Targets

All platforms.

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 C++ standard does not support the `inherited` symbol. Only the CodeWarrior C++ language implements the `inherited` symbol for single inheritance.

NOTE `#pragmas` may no longer correspond to their textual position in the file when this `#pragma` is enabled.

This pragma does not correspond to any setting in the C/C++ Language Panel. To check this setting, use the `__option (def_inherited)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

defer_codegen

Controls the inlining of functions that are not yet compiled.

Syntax

```
#pragma defer_codegen on | off | reset
```

Targets

All platforms.

Remarks

This setting lets you use inline and auto-inline functions that are called before their definition:

```
#pragma defer_codegen on
#pragma auto_inline on

extern void f();
extern void g();

main()
{
   f(); // will be inlined
   g(); // will be inlined
}

inline void f() {}
void g() {}
```

NOTE The compiler requires more memory at compile time if you enable this pragma.

This pragma corresponds to the **Deferred Inlining** setting of the **Inline Depth** menu in the C/C++ Language Panel. To check this setting, use the `__option(defer_codegen)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

defer_defarg_parsing

Defers the parsing of default arguments in member functions.

Syntax

```
#pragma defer_defarg_parsing on | off
```

Targets

All platforms.

Remarks

To be accepted as valid, some default expressions with template arguments will require additional parenthesis. For example, the following source code generates an error:

```
template<typename T,typename U> struct X { T t; U u; };

struct Y {
   // the following line is not accepted, and generates
   // an error with defer_defarg_parsing on
   void f(X<int,int> = X<int,int>());
};
```

While this version will not:

```
template<typename T,typename U> struct X { T t; U u; };

struct Y {
   // following line is OK,
   // if default argument is parenthesized
   void f(X<int,int> = (X<int,int>()) );
};
```

This pragma does not correspond to any panel setting. To check this setting, use `__option (defer_defar_parsing)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

direct_destruction

This pragma is no longer available.

direct_to_som

Controls the generation of SOM object code.

Syntax

```
#pragma direct_to_som on | off | reset
```

Targets

All platforms.

Remarks

This pragma is available for C++ only and Mac OS.

This pragma lets you create SOM code directly in the CodeWarrior IDE. For more information, see *Targeting Mac OS*.

If you enable this pragma, the compiler automatically enables the **Enums Always Int** setting in the C/C++ Language Panel, described in "Enumerated Types" on page 34.

This pragma does not correspond to any panel setting. Selecting **On** from that menu is like setting this pragma to `on` and setting the `SOMCheckEnvironment` pragma to `off`. Selecting **On with Environment Checks** from that menu is like enabling both this pragma and `SOMCheckEnvironment`. Selecting **off** from that menu is like disabling both this pragma and `SOMCheckEnvironment`.

To check this setting, use the `__option (direct_to_SOM)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

dollar_identifiers

Controls use of dollar signs ($) in identifiers.

Syntax

```
#pragma dollar_identifiers on | off | reset
```

Targets

All platforms.

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. To check this setting, use the `__option (dollar_identifiers)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

dont_inline

Controls the generation of inline functions.

Syntax

```
#pragma dont_inline on | off | reset
```

Targets

All platforms.

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 132. If you disable this pragma, the compiler expands all inline function calls, within the limits you set through other inlining-related pragmas.

This pragma corresponds to the **Don't Inline** setting in the C/C++ Language Panel. To check this setting, use `__option (dont_inline)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

dont_reuse_strings

Controls whether or not to store each string literal separately in the string pool.

Syntax

```
#pragma dont_reuse_strings on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler stores each string literal separately. Otherwise, the compiler stores only one copy of identical string literals. This pragma helps you save memory if your program contains a lot of identical string literals that you do not modify.

For example, take this code segment:

```
char *str1="Hello";
char *str2="Hello"
*str2 = 'Y';
```

If you enable this pragma, `str1` is `"Hello"`, and `str2` is `"Yello"`. Otherwise, both `str1` and `str2` are `"Yello"`.

This pragma corresponds to the **Reuse Strings** setting in the C/C++ Language Panel. To check this setting, use `__option (dont_reuse_strings)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

ecplusplus

Controls the use of embedded C++ features.

Syntax

```
#pragma ecplusplus on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the C++ compiler disables the non-EC++ features of ANSI C++ such as templates, multiple inheritance, and so on. See "C++ and Embedded Systems" on page 81 for more information on Embedded C++ support in CodeWarrior C/C++ compilers.

This pragma corresponds to the **EC++ Compatibility Mode** setting in the C/C++ Language Panel. To check this setting, use `__option (ecplusplus)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

enumsalwaysint

Specifies the size of enumerated types.

Syntax

```
#pragma enumsalwaysint on | off | reset
```

Targets

All platforms.

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. 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 11.15 shows an example.

Listing 11.15 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. Otherwise, this type is
     the same size as a long long. */
```

For more information on how the compiler handles enumerated types, see "Enumerated Types" on page 34.

This pragma corresponds to the **Enums Always Int** setting in the C/C++ Language Panel. To check this setting, use `__option (enumsalwaysint)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

errno_name

Tells the optimizer how to find the `errno` identifier.

Syntax

```
#pragma errno_name id | ...
```

Targets

All platforms.

Remarks

When this pragma is used, the optimizer can use the identifier `errno` (either a macro or a function call) 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 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 globals, use this form:

#pragma errno_name ...

Otherwise, do not use this pragma as it can result in incorrect optimizations in your code.

This pragma does not correspond to any panel setting. By default, this pragma is unspecified (worst case assumption).

exceptions

Controls the support of C++ exception handling.

Syntax

```
#pragma exceptions on | off | reset
```

Targets

All platforms.

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 the following:

– Libraries compiled with `#pragma exceptions off`

If you throw an exception across one of these, the code calls `terminate()` and exits.

This pragma does not correspond to an option in any panel. To check this setting, use `__option (exceptions)`, described in "Checking Settings" on page 115. By default, this pragma is enabled.

explicit_zero_data

Controls the placement of zero-initialized data.

Syntax

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

Targets

All platforms.

Remarks

Places zero-initialized data into the initialized data section instead of the BSS section when `on`.

To check this setting, use `__option (explicit_zero_data)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

export

Controls the exporting of data or functions.

Syntax

```
#pragma export on | off | reset
#pragma export list name1 [, name2 ]*
```

Targets

All platforms.

Remarks

When using the `#pragma export on` format, all functions are automatically exported.

When using the `#pragma export list` format, use it to tag data or functions for exporting. 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 11.16 shows an example:

Listing 11.16 Example of an Exported List

```
extern int f(),g;
#pragma export list f,g
```

To check this setting, use `__option (export)`, described in "Checking Settings" on page 115.

extended_errorcheck

Controls the issuing of warnings for possible unintended logical errors.

Syntax

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

Targets

All platforms.

Remarks

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

Incorrect Function Returns" on page 98 for descriptions of the errors that result in this warning.

This pragma corresponds to the **Extended Error Checking** setting in the C/C++ Warnings Panel. To check this setting, use `__option (extended_errorcheck)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

faster_pch_gen

Controls the performance of precompiled header generation.

Syntax

```
#pragma faster_pch_gen on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, generating a precompiled header can be much faster, depending on the header structure. However, it can also be slightly larger.

This pragma does not correspond to any panel setting. To check this setting, use the `__option (faster_pch_gen)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

flat_include

Searches an `#include` using a relative path using only the base filename.

Syntax

```
#pragma flat_include on | off | reset
```

Targets

All platforms.

Remarks

When on, searches are performed where, for example "`#include <sys/stat.h>`" is treated the same as "`#include <stat.h>`".

This pragma is useful when porting code form a different operating system, or when a project's access paths cannot reach a given file.

To check this setting, use the `__option (flat_include)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

float_constants

Controls how floating pointing constants are treated.

Syntax

```
#pragma float_constants on | off | reset
```

Targets

All platforms.

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. For related information, see "The "D" Constant Suffix" on page 49.

This pragma does not correspond to any panel setting. To check this setting, use the `__option (float_constants)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

force_active

Controls how "dead" functions are linked.

Syntax

```
#pragma force_active on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the linker leaves functions within the scope of the pragma in the finished application, even if the functions are never called in the program.

When compiling for Mac OS, this setting is disabled by default.

This pragma does not correspond to any panel setting. To check this setting, use the `__option (force_active)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

fullpath_file

Controls if `__FILE__` macro returns a full path or the base filename.

Syntax

```
#pragma fullpath_file on | off | reset
```

Targets

All platforms.

Remarks

When on, the `__FILE__` macro returns a full path to the current file, otherwise it returns the base filename.

fullpath_prepdump

Shows the full path of included files in preprocessor output.

Syntax

```
#pragma fullpath_prepdump on | off | reset
```

Targets

All platforms.

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 "C/C++ Preprocessor Panel" on page 23. To check this setting, use the `__option (fullpath_prepdump)`, described in "Checking Settings" on page 115. See also "line_prepdump" on page 165. By default, this pragma is disabled.

gcc_extensions

Controls the acceptance of GNU C language extensions.

Syntax

```
#pragma gcc_extensions on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler accepts GNU C extensions in C source code. This includes the following non-ANSI C extensions:

– Initialization of automatic `struct` or `array` variables with non-`const` values. Listing 11.17 provides an example.

Listing 11.17 Example of Array Initialization with a Non-`const` Value

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

– `sizeof( void ) == 1`

– `sizeof(` *function-type* `) == 1`

– Limited support for GCC statements and declarations within expressions. Listing 11.18 provides an example.

This feature only works for expressions in function bodies and does not support code that requires any form of C++ exception handling (for example, throwing or catching exceptions or creating local or temporary class objects that require a destructor call).

Listing 11.18 Example of GCC Statements and Declarations Within Expressions

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

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

- Macro redefinitions without a previous `#undef`.
- The GCC keyword `typeof`. See "The __typeof__() and typeof() operators" on page 50, "Initialization of Local Arrays and Structures" on page 50 for a description of these extensions.
- Function pointer arithmetic supported
- `void*` arithmetic supported
- `__builtin_constant_p (<expr>)` supported
- Forward declarations of arrays of incomplete type supported
- Pre-C99 designated initializer syntax (deprecated) supported. Example shown in Listing 11.19.

Listing 11.19 Example of GCC Support for Pre-C99 Initializer Syntax

```
#pragma gcc_extensions on
struct S { int a, b, b; } s = { c:3, b:2, a:1 };
```

- Conditionals with Omitted Operands supported. Example shown in Listing 11.20.

Listing 11.20 Example of GCC Support for Omitted Operands

```
#pragma gcc_extensions on
int x, y, z;
x = y ?: z;
```

Outputs:

```
x = y ? y ? z;
```

- `long __builtin_expect (long exp, long c)` now accepted

This pragma corresponds to the **Enable GCC Extensions** setting in the C/C++ Language Panel. To check the global optimizer, use `__option (gcc_extensions)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

global_optimizer

Controls whether the Global Optimizer is invoked by the compiler.

Syntax

```
#pragma global_optimizer on | off | reset
```

Targets

All platforms.

Remarks

In most compilers, this #pragma determines whether the Global Optimizer is invoked (configured by options in the panel of the same name). If disabled, only simple optimizations and back-end optimizations are performed.

NOTE This is not the same as `#pragma optimization_level`. The Global Optimizer is invoked even at `optimization_level 0` if `#pragma global_optimizer` is enabled.

This pragma corresponds to to the settings in the **Global Optimizations** panel. To check this setting, use `__option (ignore_oldstyle)`, described in "Checking Settings" on page 115. By default, this setting is `on`.

ignore_oldstyle

Controls the recognition of function declarations that follow the conventions in place before ANSI/ISO C (i.e., "K&R" style).

Syntax

```
#pragma ignore_oldstyle on | off | reset
```

Targets

All platforms.

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 do not specify the types of the arguments in the argument list but on separate lines.

For example, the code in Listing 11.21 defines a prototype for a function with an old-style declaration.

Listing 11.21 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. To check this setting, use `__option (ignore_oldstyle)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

import

Controls the importing of data or functions.

Syntax

`#pragma import on | off | reset`

`#pragma import list` *name1* `[,` *name2* `]*`

Targets

All platforms.

Remarks

When using the `#pragma import on` format, all functions are automatically imported.

When using the `#pragma import list` format, use to tag data or functions for importing. 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 11.22 shows an example:

Listing 11.22 Example of an Imported List

```
extern int f(),g;
#pragma import list f,g
```

This pragma does not correspond to any panel setting. To check this setting, use `__option (import)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

inline_bottom_up

Controls the bottom-up function inlining method.

Syntax

```
#pragma inline_bottom_up on | off | reset
```

Targets

All platforms.

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 11.23 and Listing 11.24.

Listing 11.23 Maximum Complexity of an Inlined Function

```
//  maximum complexity of an inlined function
#pragma inline_max_size( max )          // default max == 256
```

Listing 11.24 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 C/C++ Language Panel.To check this setting, use `__option (inline_bottom_up)`, described in "Checking Settings" on page 115. 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
```

Targets

All platforms.

Remarks

To check this setting, use `__option (inline_bottom_up_once)`, described in "Checking Settings" on page 115. 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 `(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.

Targets

All platforms.

Remarks

The pragmas `dont_inline` and `always_inline` override this pragma. This pragma corresponds to the **Inline Depth** setting in the C/C++ Language Panel. By default, this pragma is disabled.

inline_max_auto_size

Determines the maximum complexity for an auto-inlined function.

Syntax

```
#pragma inline_max_auto_size ( complex )
```

Parameters

`complex`

The `complex` value is an approximation of the number of statements in a function, the current default value is 15. Selecting a higher value will inline more functions, but can lead to excessive code bloat.

Targets

All platforms.

Remarks

This pragma does not correspond to any panel setting. To check this setting, use `__option (inline_max_auto_size)`, described in "Checking Settings" on page 115.

---

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.

Targets

All platforms.

Remarks

This pragma does not correspond to any panel setting. To check this setting, use `__option (inline_max_size)`, described in "Checking Settings" on page 115.

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.

Targets

All platforms.

Remarks

This pragma does not correspond to any panel setting. To check this setting, use `__option (inline_max_total_size)`, described in "Checking Settings" on page 115.

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"
```

Targets

All platforms.

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 writen 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.

internal

Controls the internalization of data or functions.

Syntax

```
#pragma internal on | off | reset
#pragma internal list name1 [, name2 ]*
```

Targets

All platforms.

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 11.25 shows an example:

Listing 11.25 Example of an Internalized List

```
extern int f(),g;
#pragma internal list f,g
```

This pragma does not correspond to any panel setting. To check this setting, use `__option (internal)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

keepcomments

Controls whether comments are emitted in the preprocessor output.

Syntax

```
#pragma keepcomments on | off | reset
```

Targets

All platforms.

Remarks

This pragma corresponds to the **Keep comments** option in the C/C++ Preprocessor Panel. By default, this pragma is disabled.

lib_export

Controls the exporting of data or functions.

Syntax

```
#pragma lib_export on | off | reset
#pragma lib_export list name1 [, name2 ]*
```

Targets

All platforms.

Remarks

When using the `#pragma lib_export on` format, all data and functions are automatically exported.

Use the `#pragma lib_export list` format to to tag specific data or functions for exporting. 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 11.26 shows an example:

Listing 11.26 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. To check this setting, use `__option (lib_export)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

line_prepdump

Shows `#line` directives in preprocessor output.

Syntax

```
#pragma line_prepdump on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, `#line` directives appear in preprocessor output, and line spacing is preserved through the insertion of empty lines.

Use this pragma with the command-line compiler's `-E` option to make sure that `#line` directives are inserted in the compiler's output.

This pragma corresponds to the **Use #line** option in the "C/C++ Preprocessor Panel" on page 23. To check this setting, use the `__option (line_prepdump)`, described in "Checking Settings" on page 115. See also "fullpath_prepdump" on page 153. By default, this pragma is disabled.

longlong

Controls the availability of the `long long` type.

Syntax

```
#pragma longlong on | off | reset
```

Targets

All platforms.

Remarks

When the `longlong` pragma is enabled, the C or C++ compiler lets you define a 64-bit integer with the type specifier `long long`. This type is twice as large as a `long int`, which is a 32-bit integer. A variable of type `long long` can hold values from

```
-9,223,372,036,854,775,808
```

to

`9,223,372,036,854,775,807.`

An `unsigned long long` can hold values from `0` to `18,446,744,073,709,551,615.`

This pragma does not correspond to any panel setting. To check this setting, use `__option (longlong)`, described in "Checking Settings" on page 115.

By default, this pragma is `on` in compiles that support this type. It is `off` in compiles 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
```

Targets

All platforms.

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 147).

For more information on how the compiler handles enumerated types, see "Enumerated Types" on page 34.

This pragma does not correspond to any panel setting. To check this setting, use `__option (longlong_enums)`, described in "Checking Settings" on page 115. By default, this setting is enabled.

longlong_prepeval

Controls whether or not the preprocessor treats integral constant expressions as `long long`.

Syntax

```
#pragma longlong_prepeval on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the C/C++ preprocessor treats integral constant expressions in `#if`, `#elif` as type `long long`.

This pragma does not correspond to any panel setting. To check this setting, use `__option (longlong_prepval)`, described in "Checking Settings" on page 115. By default, this setting is enabled.

macro_prepdump

Controls whether macro `#defines` and `#undefs` are emitted in the output when parsed.

Syntax

```
#pragma macro_prepdump on | off | reset
```

Targets

All platforms.

Remarks

TIP Use this pragma to help unravel confusing problems like macros that are aliasing indentifiers or where headers are redefining macros unexpectedly.

mark

Adds an item to the **Function** pop-up menu in the IDE editor.

Syntax

`#pragma mark` *itemName*

Targets

All platforms.

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.

maxerrorcount

Limits the number of errors 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`

Effectively unbounds the number of issued error messages.

Targets

All platforms.

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. The message appears in the **Errors & Warnings** window.

Syntax

```
#pragma message( msg )
```

Parameter

`msg`

Actual message to issue. Does not have to be a string literal.

Targets

All platforms.

Remarks

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
```

Targets

All platforms.

Remarks

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

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

If you enable this pragma, use ANSI C/C++ libraries that were compiled when this pragma was enabled. The file names of the 68K versions of these libraries include the letters NL (for example, `MSL C.68K (NL_2i).Lib`). The PowerPC versions of these libraries are marked with NL; for example, `MSL C.PPC (NL).Lib`.

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

This pragma does not correspond to any panel setting. To check this setting, use `__option (mpwc_newline)`, described in "Checking Settings" on page 115. 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
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler treats `char*` and `unsigned char*` as the same type. This setting is especially useful if you are using code written before the ANSI C standard. This old source code frequently used these types interchangeably.

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

You can use this pragma to relax function pointer checking:

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

This pragma does not correspond to any panel setting. To check this setting, `__option (mpwc_relax)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

msg_show_lineref

Controls diagnostic output when `#line` directives involved by showing errors in source pointed to by `#line`.

Syntax

```
#pragma msg_show_lineref on | off | reset
```

Targets

All platforms.

Remarks

This pragma does not correspond to any panel setting. To check this setting, use `__option (msg_show_lineref)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

msg_show_realref

Controls diagnostic output when `#line` directives are encountered by showing error in actual source where `#line` reference exists.

Syntax

```
#pragma msg_show_realref on | off | reset
```

Targets

All platforms.

Remarks

This pragma does not correspond to any panel setting. To check this setting, use `__option (msg_show_realref)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

multibyteaware

Controls how the Source encoding option is treated

Syntax

```
#pragma multibyteaware on | off | reset
```

Targets

All platforms.

Remarks

This #pragma has been deprecated. See the section "Multibyte and Unicode Support" on page 54 and the #pragma text_encoding for more details.

When on, this pragma treats the Source encoding of "ASCII" like "Autodetect". When off, this pragma treats a Source encoding of "Autodetect" like "ASCII" setting.

NOTE Previously, this pragma told the compiler to "look twice" at the "\" character to avoid misinterpreting the character in a multibyte character sequence.

This pragma does not correspond to any panel setting, but the replacement option **Source encoding** appears in the "C/C++ Preprocessor Panel" on page 23. To check this setting, use `__option (multibyteaware)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

multibyteaware_preserve_literals

Controls the treatment of multibyte character sequences in narrow string literals.

Syntax

```
#pragma multibyteaware_preserve_literals on | off | reset
```

Targets

All platforms.

Remarks

This pragma does not correspond to any panel setting. To check this setting, use `__option (multibyteaware_preserve_literals)`, described in "Checking Settings" on page 115. 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
```

Targets

All platforms.

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. To check this setting, use `__option (new_mangler)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

no_conststringconv

Disables the deprecated implicit const string literal conversion (ISO C++, §4.2).

Syntax

```
#pragma no_conststringconv on | off | reset
```

Targets

All platforms.

Remarks

When enabled, the compiler generates an error when it encounters an implicit const string conversion. See example in Listing 11.27.

Listing 11.27 Example of const string conversion

```
#pragma no_conststringconv on

char *cp = "Hello World";  // generates error: illegal
```

```
                        // implicit conversion from
                        // 'const char[12]' to 'char *'
```

This pragma does not correspond to any panel setting. To check this setting, use `__option (no_conststringconv)`, described in "Checking Settings" on page 115. 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
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler does not generate destructor calls for static data objects. Use this pragma for smaller object code for C++ programs that never exit.

This pragma does not correspond to any panel setting. To check this setting, use `__option (no_static_dtors)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

nosyminline

Controls whether debug information is gathered for inline/template functions.

Syntax

```
#pragma nosyminline on | off | reset
```

Targets

All platforms.

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.

notonce

Controls whether or not the compiler lets included files be repeatedly included, even with `#pragma once on`.

Syntax

```
#pragma notonce
```

Targets

All platforms.

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 176.

This pragma does not correspond to any panel setting.

objective_c

Controls the use of Objective-C keywords.

Syntax

```
#pragma objective_c on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler lets you use the following additional objective C keywords:

Table 11.2 Objective-C keywords supported

`@class`	`@def`	`@encode`
`@end`	`@implementation`	`@interface`
`@private`	`@protocol`	`@protected`
`@public`	`@selector`	`bycopy`
`byref`	`in`	`inout`
`oneway`	`out`	

– `@protocol` forward declarations are supported

This pragma does not correspond to any panel setting. To check this setting, use `__option (objective_c)`, described in "Checking Settings" on page 115.

By default, this setting is automatically enabled when a file's extension is ".m" or ".pchm". This setting and `#pragma cplusplus on` are automatically enabled when a file's extension is ".mm", ".M", or ".pchmm".

old_pragma_once

This pragma is no longer available.

old_vtable

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 ]
```

Targets

All platforms.

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 insure that *any* file is included only once in a source file.

Beware that when using `#pragma once on`, precompiled headers might not necessarily transfer from machine to machine and provide the same results. This is because the full paths of included files are stored to distinguish between two distinct files that have identical filenames but different paths. Use the `warn_pch_portability` pragma to issue a warning when `#pragma once on` is used in a precompiled header. For more information, see "warn_pch_portability" on page 221.

Also, if you enable the `old_pragma_once` on pragma, the `once` pragma completely ignores path names. For more information, see "old_pragma_once" on page 176.

This pragma does not correspond to any panel setting. By default, this pragma is disabled.

only_std_keywords

Controls the use of ISO keywords.

Syntax

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

Targets

All platforms.

Remarks

The C/C++ compiler recognizes additional reserved keywords. If you are writing code that must follow the ANSI standard strictly, enable the pragma `only_std_keywords`. For more information, see "ANSI Keywords Only" on page 40.

This pragma corresponds to the **ANSI Keywords Only** setting in the C/C++ Language Panel. To check this setting, use `__option (only_std_keywords)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

options

Specifies how to align struct and class data.

Syntax

`#pragma options align=` *alignment*

Parameter

`alignment`

Specifies the boundary on which struct and class data is aligned in memory. Values for *alignment* range from 1 to 16, or use one of the following preset values:

Table 11.3 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 Macintoshes.
`mac68k4byte`	Aligns every field on 4-byte boundaries.
`power`	Aligns every field on its natural boundary. This is the standard alignment for Power Macintoshes. 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.

Table 11.3 Structs and Classes Alignment

If *alignment* is …	The compiler …
`native`	Aligns every field using the standard alignment. It is equivalent to using `mac68k` for 68K Macintoshes and `power` for Power Macintoshes.
`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`.

Targets

All platforms.

Remarks

This pragma corresponds to the **Struct Alignment** setting in the **<*Target*> Processor** or **<*Target*> CodeGen** panel.

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
```

Targets

All platforms.

Remarks

Listing 11.28 shows an example.

Listing 11.28 Example #pragma opt_classresults

```
#pragma opt_classresults on

struct X {
  X();
  X(const X&);
  // ...
};

X f() {
  X x;      // directly constructed in function result buffer
  // ...
  return x; // no copy-ctor call
}
```

This pragma does not correspond to any panel setting. To check this setting, use `__option (opt_classresults)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

opt_common_subs

Controls the use of common subexpression optimization.

Syntax

```
#pragma opt_common_subs on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (opt_common_subs)`, described in "Checking Settings" on page 115. By default, this settings is related to the "global optimizer" on page 156 level.

opt_dead_assignments

Controls the use of dead store optimization.

Syntax

```
#pragma opt_dead_assignments on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (opt_dead_assignments)`, described in "Checking Settings" on page 115. By default, this settings is related to the "global optimizer" on page 156 level.

opt_dead_code

Controls the use of dead code optimization.

Syntax

```
#pragma opt_dead_code on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (opt_dead_code)`, described in "Checking Settings" on page 115. By default, this settings is related to the "global_optimizer" on page 156 level.

opt_lifetimes

Controls the use of lifetime analysis optimization.

Syntax

```
#pragma opt_lifetimes on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (opt_lifetimes)`, described in "Checking Settings" on page 115. By default, this settings is related to the "global_optimizer" on page 156 level.

opt_loop_invariants

Controls the use of loop invariant optimization.

Syntax

```
#pragma opt_loop_invariants on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (opt_loop_invariants)`, described in "Checking Settings" on page 115. By default, this settings is related to the "global_optimizer" on page 156 level.

opt_propagation

Controls the use of copy and constant propagation optimization.

Syntax

```
#pragma opt_propagation on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (opt_propagation)`, described in "Checking Settings" on page 115. By default, this settings is related to the "global_optimizer" on page 156 level.

opt_strength_reduction

Controls the use of strength reduction optimization.

Syntax

```
#pragma opt_strength_reduction on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (opt_strength_reduction)`, described in "Checking Settings" on page 115. By default, this settings is related to the "global optimizer" on page 156 level.

opt_strength_reduction_strict

Uses a safer variation of strength reduction optimization.

Syntax

```
#pragma opt_strength_reduction_strict on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (opt_strength_reduction_strict)`, described in "Checking Settings" on page 115. 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
```

Targets

All platforms.

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. To check this setting, use `__option (opt_unroll_loops)`, described in "Checking Settings" on page 115. By default, this settings is related to the "global optimizer" on page 156 level.

opt_vectorize_loops

Controls the use of loop vectorizing optimization.

Syntax

```
#pragma opt_vectorize_loops on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler improves loop performance.

NOTE Do not confuse loop vectorizing with the vector instructions supported by some chips, like the PowerPC with AltiVec™. Loop vectorizing is the rearrangement of instructions in loops to improve performance. PowerPC AltiVec™ instructions are specialized instructions that manipulate vectors and available only on specific PowerPC processors.

Only the **x86 CodeGen** settings panel has the **Autovectorize logs** option, it is not available as an option setting in any other panel. To check this setting, use `__option (opt_vectorize_loops)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

optimization_level

Controls global optimization.

Syntax

```
#pragma optimization_level 0 | 1 | 2 | 3 | 4 | reset
```

Targets

All platforms.

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 sets the optimization level to its previous state. Example:

```
#pragma optimization_level 2
#pragma optimization_level 1
//reset optimization_level to previous state.
#pragma optimization_level reset
```

For more information on the optimization the compiler performs for each optimization level, refer to the *Targeting* manual for your target platform.

These pragmas correspond to the settings in the **Global Optimizations** panel. By default, this pragma is disabled.

optimize_for_size

Controls optimization to reduce the size of object code.

Syntax

```
#pragma optimize_for_size on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (optimize_for_size)`, described in "Checking Settings" on page 115.

optimizewithasm

Controls optimization of assembly language.

Syntax

```
#pragma optimizewithasm on | off | reset
```

Targets

All platforms.

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. To check this setting, use `__option (optimizewithasm)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

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
```

Targets

All platforms.

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 corresposnds to the **ISO C++ Template Parser** option (together with pragmas parse_func_templ and warn_no_typename). To check this setting, use `__option (parse_func_templ)`, described in "Checking Settings" on page 115. 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
```

Targets

All platforms.

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. To check this setting, use `__option (parse_mfunc_templ)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

pool_strings

Controls how string literals are stored.

Syntax

```
#pragma pool_strings on | off | reset
```

Targets

All platforms.

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 and TOC entry for each string constant. While this decreases the number of data sections in your program, on some processors, like the PowerPC, 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 Metrowerks 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 C/C++ Language Panel.

NOTE Some compilers, like the **Pool Strings** option in the **PPC Linker/Compiler** panel will override this panel setting.

To check this setting, use `__option (pool_strings)`, described in "Checking Settings" on page 115.

pop, push

Save and restore pragma settings.

Syntax

```
#pragma push
#pragma pop
```

Targets

All platforms.

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 11.29.

Listing 11.29 push and pop Example

```
#pragma far_data on
#pragma pointers_in_A0
#pragma push      // push all compiler settings
```

```
#pragma far_data off
#pragma pointers_in_D0
                    // pop restores "far_data" and "pointers_in_A0"
#pragma pop
```

NOTE This example uses some platform-specific pragmas for illustrative purposes only. See the *Targeting* manual for your platform to determine which pragmas are supported.

This pragma does not correspond to any panel setting. By default, this pragma is `off`.

TIP `#pragmas` that allow `on|off|reset` already form a stack of previous option values. It is not necessary to use `#pragma pop/push` with such `#pragmas`.

pragma_prepdump

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

Syntax

```
#pragma pragma_prepdump on | off | reset
```

Targets

All platforms.

Remarks

This pragma corresponds to the **Emit #pragmas** option in the C/C++ Preprocessor Panel. By default, this pragma is disabled.

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`

Filename can be a simple filename or an absolute pathname. If *filename* is a simple filename, 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.

Targets

All platforms.

Remarks

This pragma specifies the filename for a precompiled header file. If you do not specify the filename, the compiler gives the precompiled header file the same name as its source file.

Listing 11.30 shows sample source code from the MacHeaders precompiled header source file. By using the predefined symbols `__cplusplus` and `powerc` and the pragma `precompile_target`, the compiler can use the same source code to create different precompiled header files for C/C++, 680x0 and PowerPC.

Listing 11.30 Using #pragma precompile_target

```
#ifdef __cplusplus
 #ifdef powerc
  #pragma precompile_target "MacHeadersPPC++"
 #else
  #pragma precompile_target "MacHeaders68K++"
 #endif
#else
 #ifdef powerc
  #pragma precompile_target "MacHeadersPPC"
 #else
  #pragma precompile_target "MacHeaders68K"
 #endif
#endif
```

This pragma does not correspond to any panel setting.

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
```

Targets

All platforms.

Remarks

If you enable this pragma, C strings used in your source code (for example, "hello") are output to the read-only data section instead of the global data section. In effect, these strings act like `const char *`, even though their type is really `char *`.

This pragma does not correspond to any panel setting. To check this setting, use `__option (readonly_strings)`, described in "Checking Settings" on page 115..

require_prototypes

Controls whether or not the compiler should expect function prototypes.

Syntax

```
#pragma require_prototypes on | off | reset
```

Targets

All platforms.

Remarks

This pragma only affects non-static functions.

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

This pragma corresponds to the **Require Function Prototypes** setting in the C/C++ Language Panel. To check this setting, use `__option (require_prototypes)`, described in "Checking Settings" on page 115.

reverse_bitfields

Controls whether or not the compiler reverses the bitfield allocation.

Syntax

```
#pragma reverse_bitfields on | off | reset
```

Targets

All platforms.

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 rightmost position.

This pragma does not correspond to any panel setting. To check this setting, use `__option (reverse_bitfields)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

RTTI

Controls the availability of runtime type information.

Syntax

```
#pragma RTTI on | off | reset
```

Targets

All platforms.

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 yet implemented.

This pragma corresponds to the **Enable RTTI** setting in the C/C++ Language Panel. To check this setting, use `__option (RTTI)`, described in "Checking Settings" on page 115.

showmessagenumber

Controls the appearance of warning or error numbers in displayed messages.

Syntax

```
#pragma showmessagenumber on | off | reset
```

Targets

All targets.

Remarks

When enabled, this pragma causes messages to appear with their numbers visible. You can then use the "warning" on page 205 pragma with a warning number to suppress the appearance of specific warning messages.

This pragma does not correspond to any panel setting. To check whether this pragma is enabled, use `__option (showmessagenumber)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

show_error_filestack

Controls the appearance of the current `#includes` file stack within error messages occuring inside deeply-included files.

Syntax

```
#pragma show_error_filestack on | off | reset
```

Targets

All targets.

Remarks

This pragma does not correspond to any panel setting. By default, this pragma is `on`.

simple_prepdump

Controls the suppression of comments in preprocessor dumps.

Syntax

```
#pragma simple_prepdump on | off | reset
```

Targets

All platforms.

Remarks

By default, the preprocessor adds comments about the current include file being processed in its output. Enabling this pragma disables these comments.

This pragma corresponds to the **Emit file changes** option in the "C/C++ Preprocessor Panel" on page 23. To check this setting, use `__option(simple_prepdump)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

space_prepdump

Controls whether whitespace is stripped out or copied into the output.

Syntax

```
#pragma space_prepdump on | off | reset
```

Targets

All platforms.

Remarks

This pragma is useful for keeping the starting column aligned with the original source, though the compiler attempts to preserve space within the line. This doesn't apply when macros are expanded.

This pragma corresponds to the **Keep whitespace** option in the C/C++ Preprocessor Panel. By default, this pragma is disabled.

srcrelincludes

Controls the lookup of `#include` files.

Syntax

```
#pragma srcrelincludes on | off | reset
```

Targets

All platforms.

Remarks

When on, this IDE looks for `#include` files relative to the previously `#include` file (not just the source file). This is useful when multiple files use the same filename and are intended to be picked up by another header in that directory. This is a common occurance in UNIX.

This pragma corresponds to the **Source-relative includes** option in the **Access Paths** panel. By default, this pragma is `off`.

store_object_files

Controls the storage location of object data, either in the target data driectory or as a separate file.

Syntax

```
#pragma store_object_files on | off | reset
```

Targets

All platforms.

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`.

strictheaderchecking

Controls how strict the compiler checks headers for standard C library functions.

Syntax

```
#pragma strictheaderchecking on | off | reset
```

Targets

All platforms.

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`.

suppress_init_code

Controls the suppression of static initialization object code.

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the compiler does not generate any code for static data initialization such as C++ constructors.

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

This pragma does not correspond to any panel setting. To check this setting, use `__option (suppress_init_code)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

suppress_warnings

Controls the issuing of warnings.

Syntax

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

Targets

All platforms.

Remarks

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

This pragma does not correspond to any panel setting. To check this setting, use `__option (suppress_warnings)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

sym

Controls the generation of debugger symbol information.

Syntax

```
#pragma sym on | off | reset
```

Targets

All platforms.

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 project window. This pragma changes only which functions have information in that symbol file.

This pragma does not correspond to any panel setting. To check this setting, use `__option (sym)`, described in "Checking Settings" on page 115. By default, this pragma is enabled.

syspath_once

Controls how include files are treated when `#pragma once` is enabled.

Syntax

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

Targets

All platforms.

Remarks

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

This pragma does not correspond to any panel setting. To check this setting, use `__option (syspath_once)`, described in "Checking Settings" on page 115. By default, this setting is enabled.

template_depth

Controls how many nested or recursive class templates you can instantiate.

Syntax

```
#pragma template_depth(n)
```

Targets

All platforms.

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.

---

text_encoding

Identifies the files source text encoding format to the compiler.

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.

Targets

All platforms.

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 C/C++ Preprocessor Panel. To check this setting, use `__option (text_encoding)`, described in "Checking Settings" on page 115. By default, this setting is `ASCII`.

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
```

Targets

All platforms.

Remarks

When C++ programs use static local initializations, like this:

```
int func() {
  static int countdown = 20;
  return countdown--;
}
```

The static locals are initialized the first time the function is executed. As such, if multiple threads are running at the same time, and two of them happen to enter the function at the same time, there will be contention over which one initializes the variable.

When the #pragma is `on`, the compiler inserts a mutex around the initialization to avoid this problem.

NOTE This requires runtime support which may not be implemented on all platforms, due to the possible need for operating system support.

Listing 11.31 shows another example.

Listing 11.31 Example thread_safe_init

```
#pragma thread_safe_init on

void thread_heavy_func()
{
  // multiple routines cannot enter at the same time
  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`.

trigraphs

Controls the use ISO trigraph sequences.

Syntax

```
#pragma trigraphs on | off | reset
```

Targets

All platforms.

Remarks

If you are writing code that must strictly adhere to the ANSI standard, enable this pragma.

Table 11.4 Trigraph table

Trigraph	Character
??=	#
??/	\
??'	^
??(	[

Table 11.4 Trigraph table

Trigraph	Character
??)	]
??!	|
??<	{
??>	}
??-	~

NOTE Use of this pragma may cause a portability issue for some targets.

Many common Macintosh character constants look like trigraph sequences, and this pragma lets you use them without including escape characters. Be careful when initializing strings or multi-character constants that contain question marks.

Listing 11.32 Example of Pragma trigraphs

```
char c = '????';      // ERROR: Trigraph sequence expands to '??^
char d = '\?\?\?\?'; // OK
```

This pragma corresponds to the **Expand Trigraphs** setting in the C/C++ Language Panel. To check this setting, use `__option (trigraphs)`, described in "Checking Settings" on page 115. 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
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler treats a `char` declaration as if it were an `unsigned char` declaration.

NOTE If you enable this pragma, your code might not be compatible with libraries that were compiled when the pragma was disabled. In particular, your code might not work with the ANSI libraries included with CodeWarrior.

This pragma corresponds to the **Use unsigned chars** setting in the C/C++ Language Panel. To check this setting, use `__option(unsigned_char)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

unused

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

Syntax

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

Targets

All platforms.

Remarks

This pragma suppresses the compile time warnings for the unused variables and parameters specified in its argument list. You can use this pragma only within a function body, and the listed variables must be within the scope of the function. You cannot use this pragma with functions defined within a class definition or with template functions.

Listing 11.33 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 11.34 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 not used.
                   // . . .
}
```

This pragma does not correspond to any 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 don't match):

`#pragma warning(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:

- `once`
- `default`
- `1`
- `2`
- `3`
- `4`

- `disable`
- `error`

`warning_num_list`

The `warning_num_list` is a list of warning numbers separated by spaces.

Targets

All targets.

Remarks

Use the pragma "showmessagenumber" on page 193 to display warning messages with their warning numbers.

The alternative syntax applies to x86 programming only. Included for compatibility with Microsoft.

This pragma does not correspond to any panel setting. To check this setting, use `__option (warning)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

---

warning_errors

Controls whether or not warnings are treated as errors.

Syntax

```
#pragma warning_errors on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler treats all warnings as though they were errors and does not translate your file until you resolve them.

This pragma corresponds to the **Treat All Warnings as Errors** setting in the **C/C++ Warnings** panel. To check this setting, use `__option (warning_errors)`, described in "Checking Settings" on page 115.

warn_any_ptr_int_conv

Controls if the compiler generates a warning when an integral type is explicitly converted to a pointer type or vice versa.

Syntax

```
#pragma warn_any_ptr_int_conv on | off | reset
```

Targets

All targets.

Remarks

This pragma is useful to identify potential 64-bit pointer portability issues. An example is shown in.

Listing 11.35 Example of warn_any_ptr_int_conv

```
#pragma warn_ptr_int_conv on

short i, *ip

void foo() {
   i = (short)ip;     // WARNING: integral type is not large
                      // large enough to hold pointer
}

#pragma warn_any_ptr_int_conv on

void bar() {
   i  = (int)ip;      // WARNING: pointer to integral
                      // conversion
   ip = (short *)i;   // WARNING: integral to pointer
                      // conversion
}
```

Remarks

See also "warn_ptr_int_conv" on page 222.

This pragma corresponds to the **Pointer/Integral Conversions** setting in the C/C++ Warnings Panel. To check this setting, use `__option(warn_any_ptr_int_conv)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

warn_emptydecl

Controls the recognition of declarations without variables.

Syntax

```
#pragma warn_emptydecl on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler displays a warning when it encounters a declaration with no variables.

Listing 11.36 Examples of Pragma warn_emptydecl

```
int ;      // WARNING
int i;     // OK
  ...
long j;;   // WARNING
long j;    // OK
  ...
extern "C" {
  ...
};         // WARNING
```

This pragma corresponds to the **Empty Declarations** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_emptydecl)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

warn_extracomma

Controls the recognition of superfluous commas.

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning when it encounters an extra comma. For more information about this warning, see "Extra Commas" on page 97.

This pragma corresponds to the **Extra Commas** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_extracomma)`, described in "Checking Settings" on page 115. 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
```

Targets

All platforms.

Remarks

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

By default, this pragma only checks the spelling of user includes such as the following:

```
#include "file"
```

For more information on checking system includes, see warn_filenamecaps_system.

This pragma corresponds to the **Include File Capitalization** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_filenamecaps)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

warn_filenamecaps_system

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

Syntax

```
#pragma warn_filenamecaps_system on | off | reset
```

Targets

All platforms.

Remarks

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

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 C/C++ Warnings Panel. To check this setting, use `__option (warn_filenamecaps_system)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

NOTE Some SDKs 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
```

Targets

All platforms.

Remarks

When `on`, the compiler issues a warning when it encounters a local variable that hides another local variable. An example appears in Listing 11.37.

Listing 11.37 Example of hidden local variables warning

```
#pragma warn_hiddenlocals on

void foo(int a)
{
   {
      int a;
   }
}          // generates a warning: object 'a' hidden by
           // local declaration
```

This pragma does not correspond to any panel setting. To check this setting, use `__option (warn_hiddenlocals)`, described in "Checking Settings" on page 115. By default, this setting 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
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning if you declare a non-virtual member function that hides a virtual function in a superclass. For more information about this warning, see "Hidden Virtual Functions" on page 99. The ISO C++ Standard does not require this pragma.

NOTE A warning 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 C/C++ Warnings Panel. To check this setting, use `__option (warn_hidevirtual)`.

warn_illpragma

Controls the recognition of illegal pragma directives.

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the compiler displays a warning when it encounters a pragma it does not recognize. For more information about this warning, see "Illegal Pragmas" on page 94.

This pragma corresponds to the **Illegal Pragmas** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_illpragma)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

warn_illtokenpasting

Controls whether a warning is issued with illegal token pasting.

Syntax

```
#pragma warn_illtokenpasting on | off | reset
```

Targets

All platforms.

Remarks

An example of this is shown below:

```
#define PTR(x)  x##* / PTR(foo)
```

Token pasting is used to create a single token. In this example, `foo` and `x` cannot be combined. Often the warning indicates the macros uses "##" unnecessarilly.

This pragma does not correspond to any panel setting. To check this setting, use `__option (warn_illtokenpasting)`, described in "Checking Settings" on page 115. By default, this pragma is enabled.

warn_illunionmembers

Controls whether a warning is issued when illegal union members are made, such as unions with reference or non-trivial class members.

Syntax

```
#pragma warn_illunionmembers on | off | reset
```

Targets

All platforms.

Remarks

This pragma does not correspond to any panel setting. To check this setting, use `__option (warn_illunionmembers)`, described in "Checking Settings" on page 115. By default, this pragma is `on`.

warn_impl_f2i_conv

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

Syntax

```
#pragma warn_impl_f2i_conv on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning for implicitly converting floating-point values to integral values. Listing 11.38 provides an example.

Listing 11.38 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 C/C++ Warnings Panel. To check this setting, use `__option (warn_impl_f2i_conv)`, described in "Checking Settings" on page 115. By default, this pragma is enabled.

warn_impl_i2f_conv

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

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning for implicitly converting integral values to floating-point values. Listing 11.39 provides an example.

Listing 11.39 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 C/C++ Warnings Panel. To check this setting, use `__option (warn_impl_i2f_conv)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

warn_impl_s2u_conv

Controls the issuing of warnings for implicit conversions between the `signed int` and `unsigned int` data types.

Syntax

```
#pragma warn_impl_s2u_conv on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning for implicitly converting either from `signed int` to `unsigned int` or vice versa. Listing 11.40 provides an example.

Listing 11.40 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 C/C++ Warnings Panel. To check this setting, use `__option (warn_impl_s2u_conv)`, described in "Checking Settings" on page 115. By default, this pragma is enabled.

warn_implicitconv

Controls the issuing of warnings for all implicit arithmetic conversions.

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning for all implicit arithmetic conversions when the destination type might not represent the source value. Listing 11.41 provides an example.

Listing 11.41 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
}
```

For more information about this warning, see "Implicit Arithmetic Conversions" on page 100.

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 C/C++ Warnings Panel. To check this setting, use `__option (warn_implicitconv)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

warn_largeargs

Controls the issuing of warnings for passing non-"int" numeric values to unprototyped functions.

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning if you attempt to pass a non-integer numeric value, such as a float or long long, to an unprototyped function when the require_prototypes pragma is disabled.

This pragma does not correspond to any panel setting. To check this setting, use `__option (warn_largeargs)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

warn_missingreturn

Issues a warning when a function that returns a value is missing a `return` statement.

Syntax

```
#pragma warn_missingreturn on | off | reset
```

Targets

All platforms.

Remarks

An example is shown in Listing 11.42.

Listing 11.42 Example of warn_missingreturn pragma

```
#pragma warn_missingreturn on

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

This pragma corresponds to the **Missing 'return' Statements** option in the C/C++ Warnings Panel. To check this setting, use `__option (warn_missingreturn)`, described in "Checking Settings" on page 115.

By default, this pragma is set to "Extended error checking".

warn_no_explicit_virtual

Controls the issuing of warnings if an overriding function is not declared with a virtual keyword.

Syntax

```
#pragma warn_no_explicit_virtual on | off | reset
```

Targets

All platforms.

Remarks

Listing 11.43 shows an example.

Listing 11.43 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 is not required by the ISO C++ standard, but can help you track down unwanted overrides.

This pragma does not correspond to any panel setting. To check this setting, use `__option (warn_no_explicit_virtual)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

warn_no_side_effect

Controls the issuing of warnings for redundant statements.

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning when it encounters a statement that produces no side effect. To suppress this warning, cast the statement with `(void)`. Listing 11.44 provides an example.

Listing 11.44 Example of Pragma warn_no_side_effect

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

For more information about this warning, see "Redundant Statements" on page 101.

This pragma corresponds to the **Expression Has No Side Effect** panel setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_no_side_effect)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

warn_no_typename

Controls the issuing of warnings for missing `typenames`.

Syntax

```
#pragma warn_no_typename on | off | reset
```

Targets

All platforms.

Remarks

The compiler issues a warning 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. To check this setting, use `__option (warn_no_typename)`, described in "Checking Settings" on page 115. This pragma is enabled by the ISO C++ Template Parser.

warn_notinlined

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

Syntax

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

Targets

All platforms.

Remarks

The compiler issues a warning for non-inlined inline (i.e., on those indicated by the `inline` keyword or in line in a class declaration) function calls. For more

information about this warning, see "inline Functions That Are Not Inlined" on page 100.

This pragma corresponds to the **Non-Inlined Functions** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_notinlined)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

warn_padding

Controls the issuing of warnings for data structure padding.

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the compiler warns about any bytes that were implicitly added after an ANSI C `struct` member to improve memory alignment. Refer to the appropriate *Targeting* manual for more information on how the compiler pads data structures for a particular processor or operating system. For more information about this warning, see "Realigned Data Structures" on page 101.

This pragma corresponds to the **Pad Bytes Added** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_padding)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

warn_pch_portability

Controls whether or not to issue a warning when `#pragma once on` is used in a precompiled header.

Syntax

```
#pragma warn_pch_portability on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning 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 "once" on page 176.

This pragma does not correspond to any panel setting. To check this setting, use `__option (warn_pch_portability)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

warn_possunwant

Controls the recognition of possible unintentional logical errors.

Syntax

```
#pragma warn_possunwant on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler checks for common errors that are legal C/C++ but might produce unexpected results, such as putting in unintended semicolons or confusing = and ==. For more information about this warning, see "Common Errors" on page 95.

This pragma corresponds to the **Possible Errors** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_possunwant)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

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
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning if an expression attempts to convert a pointer value to an integral type that is not large enough to hold the pointer value.

Listing 11.45 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 206.

For more information about this warning, see "Common Errors" on page 95.

This pragma corresponds to the **Pointer / Integral Conversions** setting in the C/C++ Warnings Panel. To check this setting, use `__option(warn_ptr_int_conv)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

warn_resultnotused

Controls the issuing of warnings when function results are ignored.

Syntax

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

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning when it encounters a statement that calls a function without using its result. To prevent this, cast the statement with `(void)`. Listing 11.46 provides an example.

Listing 11.46 Example of Function Calls with Unused Results

```
#pragma warn_resultnotused on

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

For more information about this warning, see "Ignored Function Results" on page 102.

This pragma does not correspond to any panel setting. To check this setting, use `__option (warn_resultnotused)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

warn_structclass

Controls the issuing of warnings for the inconsistent use of the `class` and `struct` keywords.

Syntax

```
#pragma warn_structclass on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning if you use the `class` and `struct` keywords in the definition and declaration of the same identifier. For more information about this warning, see "Mixed Use of 'class' and 'struct' Keywords" on page 101.

This pragma corresponds to the **Inconsistent 'class' / 'struct' Usage** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_structclass)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

warn_undefmacro

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

Syntax

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

Target

All targets.

Remarks

Listing 11.47 provides an example.

Listing 11.47 Example of Undefined Macro

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

Use this pragma to detect the use of undefined macros (especially expressions) where the default value 0 is used. To suppress this warning, check if defined first.

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

This pragma corresponds to the **Undefined Macro in #if** setting in the C/C++ Warnings Panel. To check this setting, use `__option(warn_undefmacro)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

warn_uninitializedvar

Controls the compiler to perform some dataflow analysis and emits warnings whenever local variables are initialized before being used.

Syntax

```
#pragma warn_uninitializedvar on | off | reset
```

Targets

All platforms.

Remarks

This pragma corresponds to the **Unused Variables** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_uninitializedvar)`, described in "Checking Settings" on page 115. By default, this pragma is `off`.

---

warn_unusedarg

Controls the recognition of unreferenced arguments.

Syntax

```
#pragma warn_unusedarg on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning when it encounters an argument you declare but do not use. For more information about this warning, see "Unused Arguments" on page 96. To suppress this warning in C++ source code, leave an argument identifier out of the function parameter list. Listing 11.34 shows an example.

This pragma corresponds to the **Unused Arguments** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_unusedarg)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

---

warn_unusedvar

Controls the recognition of unreferenced variables.

Syntax

```
#pragma warn_unusedvar on | off | reset
```

Targets

All platforms.

Remarks

If you enable this pragma, the compiler issues a warning when it encounters a variable you declare but do not use. For more information about this warning, see "Unused Variables" on page 96.

This pragma corresponds to the **Unused Variables** setting in the C/C++ Warnings Panel. To check this setting, use `__option (warn_unusedvar)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

wchar_type

Controls the size and format of the `wchar_t` type.

Targets

All platforms.

Syntax

```
#pragma wchar_type on | off | reset
```

Remarks

If you enable this pragma, `wchar_t` is treated as a built-in type.

This pragma corresponds to the **Enable wchar_t Support** setting in the "C/C++ Language Panel" on page 25. To check this setting, use `__option (wchar_type)`, described in "Checking Settings" on page 115. By default, this pragma is enabled.

12

PowerPC Pragmas

This chapter provides information on PowerPC and PowerPC with Altivec specific pragmas. You configure the compiler for a project by changing the settings in the ***<Target>* Processor** panel. You can also control compiler behavior in your code by including the appropriate pragmas.

Many of the pragmas correspond to option settings in the settings panels for processors and operating systems.

PowerPC Pragma Reference

This section describes all the pragmas available for PowerPC development. It is divided into the following sections:

NOTE See your target documentation for information on any pragmas you use in your programs. If your target documentation covers any of the pragmas listed in this section, the information provided by your target documentation always takes precedence.

align

Specifies how to align struct and class data.

Syntax

```
#pragma options align= alignment
```

Parameter

`alignment`

alignment is one of the following values:

Table 12.1 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 Macintoshes.
`mac68k4byte`	Aligns every field on 4-byte boundaries.
`power`	Aligns every field on its natural boundary. This is the standard alignment for Power Macintoshes. 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. It is equivalent to using `mac68k` for 68K Macintoshes and `power` for Power Macintoshes.
`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, if there is one, or to the value in the **68K** or **PPC Processor** panel.

NOTE There is a space between `options` and `align`.

Targets

68K, PowerPC, MIPS

Remarks

When using the C compiler, it is possible to use “char/short” bitfields to align fields on 1-byte boundaries. However, this is not possible if **ANSI Strict** is enabled.

The `#pragma align` / `#pragma pack` options only apply to the layout of fields that are larger than a byte. For example, a “short” field will appear on a 2-byte boundary, but using a packed alignment may allow it to be placed on a 1-byte boundary. Bitfields may be aligned to the maximum size used as the base of the bitfield (the “char” in `char foo:1` or the “int” in `int foo:3`), not necessarily the number of bits allocated.

This pragma corresponds to the **Struct Alignment** setting in the **68K Processor** panel.

---

altivec_codegen

Controls the use PowerPC AltiVec™ instructions during optimization.

Syntax

```
#pragma altivec_codegen on | off | reset
```

Targets

PowerPC processors with AltiVec technology.

Remarks

If you enable this pragma, the compiler uses PowerPC AltiVec instructions, if possible, during optimization. When this pragma is disabled, the `pragma altivec_model` is also set to `off`.

To check this setting, use `__option (altivec_codegen)`, described in “Checking Settings” on page 115.

---

altivec_model

Controls the use PowerPC AltiVec™ language extensions.

Syntax

```
#pragma altivec_model on | off | reset
```

Targets

PowerPC processors with AltiVec technology.

Remarks

If you enable this pragma, the compiler allows language extensions to take advantage of the AltiVec instructions available on some PowerPC processors. The `#pragma altivec_codegen` and `__ALTIVEC__` are also defined when you enable this pragma.

To check this setting, use `__option (altivec_model)`, described in "Checking Settings" on page 115. See also `__ALTIVEC__` in "Metrowerks Predefined Symbols" on page 112

altivec_pim_warnings

Enables GCC-style AltiVec declarations..

Syntax

```
#pragma altivec_pim_warnings on | off | reset
```

Targets

PowerPC processors with AltiVec technology.

Remarks

Use this pragma to allow you to define AltiVec vectors that differ from the AltiVec PIM and be more consistant with the C syntax. Listing 12.1 shows and example.

Listing 12.1 Example if using altivec_pim_warnings

```
vector signed int vsi = {1};       // (vector signed int) (1)
vector signed int vsi2 = {1, 2, 3, 4}; // (vector signed int) (1,2,3,4)

vsi = vec_add((vector signed int){1}, vsi2);
vector signed int loadvector(int a, b, c, d)
{
  vector signed int local_vsi = {a, b, c, d};
  return local_vsi;
}
```

altivec_vrsave

Controls whether or not AltiVec™ registers are saved between function calls.

Syntax

```
#pragma altivec_vrsave on | off | reset | allon
```

Parameter

`allon`

Tells the compiler to set all bits in the `VRSAVE` register.

Targets

PowerPC processors with AltiVec technology.

Remarks

If you enable this pragma, the compiler generates, at the beginning and end of functions that, extra instructions that tell the VRSave register to specify which AltiVec registers to save. Use `allon` to ensure that all AltiVec registers are saved.

To check this setting, use `__option (altivec_vrsave)`, described in "Checking Settings" on page 115.

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`).

Targets

Embedded PowerPC

bc_range

Insures that all branch conditional instructions branch no further than *value*.

Syntax

```
#pragma bc_range value | off | default
```

Parameter

`value`

Branch value. Default value is (`0x00010000 - 1`) if `prepare_compress` is `off` and (`x00001000 - 1`) if `prepare_compress` is `on`.

Targets

Embedded PowerPC

CALL_ON_MODULE_BIND, CALL_ON_LOAD

Specifies the *init_function* that should be called when the dynamic library or bundle is loaded.

Syntax

```
#pragma CALL_ON_MODULE_BIND init_function
#pragma CALL_ON_LOAD init_function
```

Targets

PowerPC

Remarks

This pragma applies to Mac OS X Mach-O only.

CALL_ON_MODULE_TERM, CALL_ON_UNLOAD

Specifies the *term_function* that should be called just before a dynamic library or bundle is unloaded.

Syntax

```
#pragma CALL_ON_MODULE_TERM term_function
#pragma CALL_ON_UNLOAD term_function
```

Targets

PowerPC

Remarks

This pragma applies to Mac OS X Mach-O only.

disable_registers

Controls compatibility for the ANSI/ISO function `setjmp()`.

Syntax

```
#pragma disable_registers on | off | reset
```

Targets

PowerPC

Remarks

If you enable this pragma, 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 updated values.

NOTE This setting disables register optimizations in functions that use PowerPlant's `TRY` and `CATCH` macros but *not* in functions that use the ANSI-standard `try` and `catch` statements. The `TRY` and `CATCH` macros use `setjmp()`, while the `try` and `catch` statements are implemented at a lower level and do not use `setjmp()`.

For Classic Mac OS, this pragma mimics a feature that is available in THINK C and Symantec C++. 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.

To check this setting, use the `__option (disable_registers)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

dynamic

Controls the generation of code for the dynamic link editor (`usr/lib/dyld`) on Mac OS X Mach-O.

Syntax

```
#pragma dynamic on | off
```

Targets

PowerPC

Remarks

This pragma applies to Mac OS X Mach-O only.

When this pragma is `off`, the `#pragma pic` is also set to `off`. Default is `on`.

fp_constants

Controls floating point constants on Classic Mac OS.

Syntax

```
#pragma fp_constants default | pool | merge
```

Targets

PowerPC

Parameters

`default`

Place each floating point constant in it's own static variable, when used with small data in TOC on it provides the fastest access to the constant. (Small data in TOC on and merge on has the same preformance behavior).

`pool`

For each source file using floating point constants pool the fp constants (by size) so that there is only one TOC pointer used for floats and one for doubles. (If no constants of a given size exist no TOC pointer is used). This method uses the least TOC pointers if a lot of unique float contants are used in a single file (and these values do not appear in other source files).

`merge`

Place each floating point constant 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 accessable by the user). This option works with either small data in TOC on or off. This option minimized TOC entry usage for programs which frequently use the same floating point constant in many different source files. (Like for example 0.0).

fp_contract

Controls the use of special floating point instructions to improve performance.

Syntax

```
#pragma fp_contract on | off | reset
```

Targets

PowerPC

Remarks

This pragma applies to PowerPC programming only.

If you enable this pragma, the compiler uses such PowerPC instructions as `FMADD`, `FMSUB`, and `FNMAD` to speed up floating-point computations. However, certain computations might produce unexpected results.

Listing 12.2 shows an example.

Listing 12.2 Example of #pragma fp_contract

```
register double A, B, C, D, Y, Z;
register double T1, T2;

A = C = 2.0e23;
B = D = 3.0e23;

Y = (A * B) - (C * D);
printf("Y = %f\n", Y);
/* prints 21267700587560961875633692999968.000000 */

T1 = (A * B);
T2 = (C * D);
Z = T1 - T2;
```

```
printf("Z = %f\n", Z);  /* prints 0.000000 */
```

If you disable this pragma, `Y` and `Z` have the same value.

This pragma corresponds to the **Use FMADD & FMSUB** setting in the **PowerPC Processor** panel. To check this setting, use `__option(fp_contract)`, described in "Checking Settings" on page 115.

function_align

Aligns the executable object code of functions on specified boundaries.

Syntax

```
#pragma function_align 4 | 8 | 16 | 32 | 64 | 128 | reset
```

Targets

PowerPC

Remarks

If you enable this pragma, the compiler aligns functions so that the start at the specified byte boundary.

To check whether the global optimizer is enabled, use `__option(function_align)`, described in "Checking Settings" on page 115.

gen_fsel

Controls generation of `fsel` instructions by limiting the number of conditional instructions that can be executed before an `fsel` instruction..

Syntax

```
#pragma gen_fsel on | off | reset | value
```

Targets

PowerPC

interrupt

Controls the compilation of object code for interrupt routines.

Syntax

For Embedded PowerPC:

```
#pragma interrupt [SRR DAR DSISR enable] on | off | reset
```

Targets

PowerPC, 68K, NEC V800, MIPS, Embedded 68K

Remarks

If you enable this pragma, the compiler generates a special prologue and epilogue for functions so that they can handle interrupts.

For convenience, the compiler also marks interrupt functions so that the linker does not dead-strip them. See "force_active" on page 152 for related information.

You can also use `__declspec(interrupt)` to mark functions as interrupt routines.

Listing 12.3 Example of `__declspec()`

```
__declspec(interrupt) void foo()
{
// enter code here
}
```

To check this setting, use `__option (interrupt)`, described in "Checking Settings" on page 115.

min_struct_align

Use to increase aggragate alignments for better memory access.

Syntax

```
#pragma min_struct_align 4 | 8 | 16 | 32 | 64 | 128 | on | off
     | reset
```

Targets

PowerPC

Remarks

When this pragma is `off`, the default alignment for the type is used. Default alignment is 4.

NOTE Only applies if optimization level is > 0.

misaligned_mem_access

Controls whether the compiler copies `structs` without regard ro alignment.

Syntax

```
#pragma misaligned_mem_access on | off | reset
```

Targets

PowerPC

Remarks

When this pragma is `on`, the default is for Desktop, when `off`, the default is for Embedded. It is recommended that all libraries be rebuilt.

no_register_save_helpers

Controls the save and restore registers without calling helper functions

Syntax

```
#pragma no_register_save_helpers on | off | reset
```

Targets

PowerPC

optimizewithasm

Enables the optimizer to function on inline assembler blocks.

Syntax

```
#pragma optimizewithasm on | off | reset
```

Targets

PowerPC

Remarks

When this pragma is `off` by default below optimization level 3 and `on` by default for optimization level 3 and above.

optimize_exported_references

Controls whether exported symbols are referenced directly.

Syntax

```
#pragma optimize_exported_references on | off | reset
```

Targets

PowerPC

Remarks

This pragma applies to Mac OS X Mach-O only and the Mac OS X PowerPC Mach-O linker.

When this pragma is `on`, exported symbols are referenced directly. The default setting for this pragma is `on`.

optimize_multidef_references

Controls whether coalescable/multidef/overloaded symbols are referenced directly.

Syntax

```
#pragma optimize_multidef_references on | off | reset
```

Targets

PowerPC

Remarks

This pragma applies to Mac OS X Mach-O only and the Mac OS X PowerPC Mach-O linker.

When this pragma is `on`, coalescable/multidef/overloaded symbols are referenced directly. The default setting for this pragma is on.

---

overload

Controls the generation of coalesced sections for merging C++ object that may get linked multiple times.

Syntax

```
#pragma overload ...
```

Targets

PowerPC

Remarks

This pragma applies to Mac OS X Mach-O only.

The Mach-O compiler now generates coalesced sections for merging C++ objects which may get defined multiple times. This mostly just works, but it can cause linker errors about duplicate symbols with different sections. Generally these are caused by templates being compiled with different versions of the compiler. Rebuilding the object which defined the symbol in the non-coalesced section will fix the problem. The one exception to this is trying to overload `new`.

To overload `new` you must declare the `new` function as "`overload`" using the overload pragma. This differs from the PEF compiler where only the version which was allowed to be overloaded needed this declaration. Since Mach-O requires that the names be defined in the same way, both must be defined as overload, AND the link order must be such that the overloading implementation must come before the implemenation to be overloaded.

The pragma looks like this:

```
#pragma overload extern void *operator
new(_CSTD::size_t) throw(_STD::bad_alloc);
```

More examples are in the Runtime sources in the file `New.cp`. Generally it is a good idea to make sure that the runtime libraries come after the source file as Mach-O is much more picky about link order.

peephole

Controls the use peephole optimization.

Syntax

```
#pragma peephole on | off | reset
```

Targets

PowerPC, Intel x86, MIPS

Remarks

If you enable this pragma, the compiler performs *peephole optimizations*, which are small, local optimizations that eliminate some compare instructions and improve branch sequences.

This pragma corresponds to the **Peephole Optimizer** setting in the **PPC Processor** panel. To check this setting, use `__option (peephole)`, described in "Checking Settings" on page 115.

pic

Controls the generation of position independent code on Mac OS X Mach-O.

Syntax

```
#pragma pic on | off
```

Targets

PowerPC

Remarks

This pragma applies to Mac OS X Mach-O only.

This pragma is `on`, the compiler generates position-independent code. This pragma requires that `#pragma dynamic` also be `on`. Default settings is `on`.

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
```

Targets

PowerPC

Remarks

This pragma is available for embedded PowerPC programming only.

If you enable this pragma, 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 **Pool Data** setting in the PPC Processor panel. To check this setting, use `__option (pool_data)`, described in "Checking Settings" on page 115.

pool_fp_consts

Controls pooling of floating point constants.

Syntax

```
#pragma pool_fp_consts on | off | reset
```

Targets

PowerPC

Remarks

Applies only to Classic Mac OS PEF only.

---

ppc_lvxl_stvxl_errata

Controls the instruction encoding for the `lvxl` and `stvxl` instructions on the 7450 to correct a bug in the processor.

Syntax

```
#pragma ppc_lvxl_stvxl_errata on | off | reset
```

Targets

PowerPC

---

ppc_unroll_factor_limit

Limits number of loop iterations in an unrolled loop *value*.

Syntax

```
#pragma ppc_unroll_factor_limit value
```

Parameter

`value`

Count limit of loop iterations.

Targets

PowerPC

Remarks

Use this pragma to specify the maximum number of copies of the loop body to place in an “unrolled” loop. The `opt_unroll_loops` pragma controls loop unrolling optimization.

The default value of *number* is 10.

ppc_unroll_instructions_limit

Limits number of instructions in an unrolled loop to *value*.

Syntax

```
#pragma ppc_unroll_instructions_limit value
```

Parameter

`value`

Count limit of instructions.

Targets

PowerPC

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.

The default value of *number* is 100.

ppc_unroll_speculative

Controls speculative unrolling of counting loops (which are not fixed count loops).

Syntax

```
#pragma ppc_unroll_speculative on | off
```

Targets

PowerPC

Remarks

If you enable this pragma, the compiler guesses how many times to unroll a loop when the number of loop iterations is a runtime calculation instead of a constant value calculated at compile time.

This optimization is only applied when:

– loop unrolling is turned on

– the loop iterator is a 32-bit value (`int`, `long`, `unsigned int`, `unsigned long`)

– no conditional statements exist in the loop body

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 loop unrolling optimization. To check this setting, use `__option (ppc_unroll_speculative)`, described in "Checking Settings" on page 115. By default, this pragma is enabled when loop unrolling is enabled.

prepare_compress

Insures that generated code is suitable for compression by post-link tool.

Syntax

```
#pragma prepare_compress on | off | reset
```

Targets

Embedded PowerPC

processor

Specifies the scheduling model used for instruction scheduling optimization.

Syntax

```
#pragma processor 601 | 603 | 604 |  750 | 7400 | 7450 |
     generic  | PPC603e | PPC604e
#if TARGETOS==PPC_EABI
#pragma processor 401 | 403 | 505 | 509 | 555 | 602 | 740 |
801 | 821 | 823 | 850 | 860 | 5100 | 8240 | 8260 | PPC56X
#endif
#if GECKO // true spelling is "gekko"
#pragma processor gekko | gecko
#endif
#if ELF_PROCESSOR
```

```
#pragma processor e500
#endif
```

Targets

PowerPC

profile

Controls the generation of extra object code for use with the CodeWarrior profiler.

Syntax

```
#pragma profile on | off | reset
```

Targets

68K, PowerPC

Remarks

This pragma applies to Mac OS programming only.

If you enable this pragma, the compiler generates code for each function that lets the Metrowerks Profiler collect information on it. For more information, see the *Metrowerks Profiler Manual*.

This pragma corresponds to the **Generate Profiler Calls** setting in the **68K Processor** panel and the **Emit Profiler Calls** setting in the **PPC Processor** panel. To check this setting, use `__option (profile)` described in "Checking Settings" on page 115.

schedule

Specifies the use of instruction scheduling optimization.

Syntax

```
#pragma schedule once | twice | altivec
```

Targets

PowerPC

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 601 | 603 | 604 |  750 | 7400 | 7450 | vger
     | altivec | PPC603e | PPC604e | on | off | once |twice |
     reset

#if TARGETOS==PPC_EABI

#pragma scheduling 401 | 403 | 505 | 509 | 555 | 602 | 740 |
801 | 821 | 823 | 850 | 860 | 5100 | 8240 | 8260 | PPC403GA |
PPC403GB | PPC403GC | PPC403GCX | PPC56X| on | off | once
|twice |  reset

#endif

#if ELF_PROCESSOR

#pragma scheduling e500 | on | off | once |twice |  reset

#endif
```

Targets

PowerPC, Intel x86

Remarks

This pragma lets you choose how the compiler rearranges instructions to increase speed. Some instructions, such as a memory load, take more than one processor cycle. By moving an unrelated instruction between the load and the instruction that uses the loaded item, the compiler saves a cycle when executing the program.

However, if you are debugging your code, disable this pragma. Otherwise, the debugger rearranges the instructions produced from your code and cannot match your source code statements to the rearranged instructions.

This pragma does not correspond to any panel setting.

section

A sophisticated and powerful pragma that lets you arrange compiled object code into predefined sections and sections you define.

Syntax

For PowerPC:

```
#pragma section [ objecttype | permission ] [iname] [uname]
[data_mode=datamode] [code_mode=codemode]
```

Parameters (PowerPC)

The optional *objecttype* parameter specifies where types of object data are stored. It can be one or more of the following values:

- `code_type`—executable object code
- `data_type`—non-constant data of a size greater than the size specified in the small data threshold setting in the **PowerPC EABI Project** panel
- `sdata_type`—non-constant data of a size less than or equal to the size specified in the small data threshold setting in the **PowerPC EABI Project** panel
- `const_type`—constant data of a size greater than the size specified in the small `const` data threshold setting in the **PowerPC EABI Project** panel
- `sconst_type`—constant data of a size less than or equal to the size specified in the small const data threshold setting in the **PowerPC EABI Project** panel
- `all_types`—all data

Specify one or more of these object types without quotes and separated by spaces.

CodeWarrior C/C++ compilers generate their own data, such as exception and static initializer objects, which the `#pragma section` statement does not affect.

NOTE CodeWarrior C/C++ compilers use the initial setting of the **Make Strings ReadOnly** setting in the **PowerPC EABI Processor** panel to classify character strings. If you enable this pragma, character strings are stored in the same section as data of type `const_type`. Otherwise, strings are stored in the same section as data for `data_type`.

The optional *permission* parameter specifies access permission. It can be one or more of these values:

- `R`—read only permission
- `W`—write permission
- `X`—execute permission

For information on access permission, see "Section access permissions for PowerPC" on page 252. Specify one or more of these permissions in any order, without quotes, and no spaces.

The optional *iname* parameter is a quoted name that specifies the name of the section where the compiler stores initialized objects. Examples of initialized objects include functions, character strings, and variables that are initialized at the time they are defined. The *iname* parameter can be of the form "`.abs.xxxxxxxx`" where *xxxxxxxx* is an 8-digit hexadecimal number specifying the address of the section.

The optional *uname* parameter is a quoted name that 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 can be either a unique name or 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 is stored in the common section. The linker puts all common section data into the "`.bss`" section. When the **Use Common Section** setting is enabled in the **PowerPC EABI Processor** panel, `COMM` is the default *uname* for the "`.data`" section. When the **Use Common Section** setting is disabled, `COMM` is the default *uname* for the "`.bss`" section.

You can change the *uname* parameter. For example, you might want most uninitialized data to go into the "`.bss`" section while specific variables are stored in the `COMM` section. Listing 12.4 shows how to specify that certain uninitialized variables be stored in the `COMM` section.

Listing 12.4 Storing Uninitialized Data in the COMM Section

```
// the Use Common Section setting is disabled
#pragma push // save the current state
#pragma section ".data" "COMM"
int foo;
int bar;
#pragma pop // restore the previous state
```

You cannot use any of the object types, data modes, or code modes as the names of sections. Also, you cannot use predefined section names in the PowerPC EABI for your own section names.

The optional `data_mode=`*datamode* parameter tells the compiler what kind of addressing mode to use for referring to data objects for a section.

The permissible addressing modes for *datamode* are:

- `near_abs`—objects must be within the first 16 bits of RAM.
- `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.

You can only use the `sda_rel` addressing mode with the "`.sdata`", "`.sbss`", "`.sdata2`", "`.sbss2`", "`.EMB.PPC.sdata0`", and "`.EMB.PPC.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 these addressing modes without quotes.

The optional `code_mode=codemode` parameter tells the compiler what kind of addressing mode to use for referring to executable routines for a section.

The permissible addressing modes for *codemode* are:

- `pc_rel`—routines must be within 24 bits of where it is called.
- `near_abs`—routines must be within the first 24 bits of RAM.

The default addressing mode for executable code sections is `pc_rel`.

Specify one 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 lets you store executable code in data sections and data in executable code sections, this practice is not encouraged.

Targets

PowerPC, Embedded 68K

Remarks

Section access permissions for PowerPC

When you define a section using `#pragma section`, its default access permission is read-only. If you change the current section for a particular object

type, the compiler adjusts the access permission to allow the storage of objects of that type while continuing to allow objects of previously allowed object types. Associating `code_type` to a section adds execute permission to that section. Associating `data_type`, `sdata_type`, or `sconst_type` to a section adds write permission to that section.

Occasionally, you might create a section without making it the current section for 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 so that the object can be stored in the section. The compiler then issues a warning. To avoid this warning, give the section the proper access permissions before storing object code or data there. As with associating an object type to a section, passing a specific permission adds to the permissions that a section already has.

NOTE Associating an object type with a section sets the appropriate access permissions for you.

Predefined sections and default sections for PowerPC

The predefined sections set with an object type become the default section for that type. After assigning a non-standard section to an object type, you can refer to the default section with one of the forms in "Forms for #pragma section for PowerPC" on page 254.

The compiler predefines the sections in Listing 12.5.

Listing 12.5 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 ".EMB.PPC.sdata0" ".EMB.PPC.sbss0" \
 data_mode=sda_rel code_mode=pc_rel
```

NOTE The "`.EMB.PPC.sdata0`" and "`.EMB.PPC.sbss0`" sections are predefined as an alternative to the `sdata_type` object type.

Forms for #pragma section for PowerPC

This pragma has these principal forms:

```
#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 can also specify the uninitialized object section, *uname*. If you know that the section is should 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 can also specify the uninitialized object section, *uname*. This feature is useful for temporarily circumventing the small data threshold.

```
#pragma section objecttype
```

When there is no *iname* parameter, the compiler resets the section for the object types specified to the default section. For information on predefined sections, see "Predefined sections and default sections for PowerPC" on page 253. Resetting an object type's section does not reset its addressing modes. You must do so explicitly.

When declaring or setting sections, you can add an uninitialized section to a section that did not originally have one by specifying a *uname* parameter. However, once you associate an uninitialized section with an initialized section, you cannot change the uninitialized section. Remember that an initialized section's corresponding uninitialized section might be the same.

Forcing individual objects into specific sections for PowerPC

You can 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 12.6 shows examples.

Listing 12.6 Using __declspec to Force Objects Into Specific Sections

```
#pragma force_active on
```

```
#pragma section ".burnInfo" __declspec(section ".burnInfo") extern
const int myVar = 0x12345678;

#pragma force_active reset
```

NOTE When using C++, only the `const` variables actually used are written to the .o file.

Using #pragma section with #pragma push and #pragma pop for PowerPC

You can use this pragma with `#pragma push` and `#pragma pop` to ease complex or frequent changes to sections settings. See Listing 12.4 for an example. Note that `#pragma pop` does not restore any changes to the access permissions of sections that exist before or after the corresponding `#pragma push`.

segment

Controls the code segment where subsequent object code is stored.

Syntax

`#pragma segment` *name*

Targets

68K, PowerPC, Embedded 68K

Remarks

This pragma applies to Classic Mac OS programming only.

This pragma places all the functions that follow into the code segment named *name*. For more on function-level segmentation, consult the *Targeting* manual for your target platform.

Generally, the PowerPC compilers ignore this directive because PowerPC applications do not have code segments. However, if you choose **by #pragma segment** from the **Code Sorting** pop-up menu in the **PPC PEF** panel, the PowerPC compilers group functions in the same segment together. For more information, consult the *Targeting* manual for your target platform.

This pragma does not correspond to any panel setting.

SOMCallOptimization

Controls the error checking used for making calls to SOM objects.

Syntax

```
#pragma SOMCallOptimization on | off | reset
```

Targets

PowerPC

Remarks

This pragma is only available for Mac OS using C++ code.

The PowerPC compiler uses an optimized error check that is smaller but slightly slower.

This pragma is ignored if the `direct_to_SOM` pragma, described in "direct_to_som" on page 143, is disabled.

This pragma does not correspond to any panel setting. To check this setting, use `__option (SOMCallOptimization)`. See on "Checking Settings" on page 115. By default, this pragma is disabled.

SOMCallStyle

Specifies the convention used to call SOM objects.

Syntax

```
#pragma SOMCallStyle OIDL | IDL
```

Targets

PowerPC

Remarks

This pragma is only available for Mac OS using C++ code.

The `SOMCallStyle` pragma chooses between two SOM call styles:

- OIDL, an older style that does not support DSOM
- IDL, a newer style that does support DSOM.

If a class uses the IDL style, its methods must have an Environment pointer as the first parameter. Note that the SOMClass and SOMObject classes use OIDL, so if you override a method from one of them, you should not include the Environment pointer.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS*, is disabled.

This pragma does not correspond to any panel setting. To check this setting, use `__option (SOMCheckEnvironment)`. See "Checking Settings" on page 115. By default, this pragma is set to `IDL`.

SOMCheckEnvironment

Controls whether or not to perform SOM environment checking.

Syntax

```
#pragma SOMCheckEnvironment on | off | reset
```

Targets

PowerPC

Remarks

This pragma is only available for Mac OS using C++ code.

If you enable this pragma, the compiler performs automatic SOM environment checking. It transforms every IDL method call and new allocation into an expression which also calls an error-checking function. You must define separate error-checking functions for method calls and allocations. For more information on how to write these functions, see *Targeting Mac OS*.

For example, the compiler transforms this IDL method call:

```
SOMobj->func(&env, arg1, arg2);
```

into something that is equivalent to this:

```
( temp=SOMobj->func(&env, arg1, arg2),
  __som_check_ev(&env), temp ) ;
```

First, the compiler calls the method and stores the result in a temporary variable. Then it checks the environment pointer. Finally, it returns the result of the method.

The compiler then transforms this new allocation:

```
new SOMclass;
```

into something like this:

```
( temp=new SOMclass, __som_check_new(temp),
  temp);
```

First, the compiler creates the object and stores it in a temporary variable. Then it checks the object and returns it.

The PowerPC compiler uses an optimized error check that is smaller but slightly slower than the one given above. To use the error check shown above in PowerPC code, use the pragma `SOMCallOptimization`, described in "SOMCallOptimization" on page 256.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS* is disabled.

This pragma does not correspond to an option in an panel. Selecting **On with Environment Checks** from that menu is like setting this pragma to `on`. Selecting anything else from that menu is like setting this pragma to `off`. To check this setting, use `__option (SOMCheckEnvironment)`, described in "Checking Settings" on page 115. By default, this pragma is enabled.

SOMClassVersion

Specifies the version of an SOM class.

Syntax

`#pragma SOMClassVersion(`*class*`,`*majorVer*`,`*minorVer*`)`

Targets

PowerPC

Remarks

This pragma is only available for Mac OS using C++ code.

SOM uses the version number of the class to insure its compatibility with other software you are using. If you do not declare the version number, SOM assumes `0`. The version number must be positive or `0`.

When you define the class, the program passes its version number to the SOM kernel in the class metadata. When you instantiate an object of the class, the program passes the version to the runtime kernel, which checks to make sure the class is compatible with the running software.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS*, is disabled.

This pragma does not correspond to any panel setting.

SOMMetaClass

Specifies the metaclass of a SOM class.

Syntax

`#pragma SOMMetaClass (`*class*`, `*metaclass*`)`

Targets

PowerPC

Remarks

This pragma is only available for Mac OS using C++ code.

A metaclass is a special kind of SOM class that defines the implementation of other SOM classes. All SOM classes have a metaclass, including metaclasses themselves. By default, the metaclass for a SOM class is SOMClass. If you want to use another metaclass, use the `SOMMetaClass` pragma:

The metaclass must be a descendant of SOMClass. Also, a class cannot be its own metaclass. That is, *class* and *metaclass* must name different classes.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS*, is disabled.

This pragma does not correspond to any panel setting.

SOMReleaseOrder

Specifies the order in which the member functions of an SOM class are released.

Syntax

`#pragma SOMRelaseOrder(`*func1*`, `*func2*`,... `*funcN*`)`

Targets

PowerPC

Remarks

This pragma is only available for Mac OS using C++ code.

A SOM class must specify the release order of its member functions. As a convenience for when you are first developing the class, the CodeWarrior C++ language lets you leave out the `SOMReleaseOrder` pragma and assumes the release order is the same as the order in which the functions appear in the class declaration. However, when you release a version of the class, use the pragma because you'll need to modify its list in later versions of the class.

You must specify every SOM method that the class introduces. Do not specify virtual inline member functions because they are not considered SOM methods. Do not specify overridden functions.

If you remove a function from a later version of the class, leave its name in the release order list. If you add a function, place it at the end of the list. If you move a function up in the class hierarchy, leave it in the original list and add it to the list for the new class.

This pragma is ignored if the `direct_to_SOM` pragma, described in *Targeting Mac OS*, is disabled.

This pragma does not correspond to any panel setting.

---

strict_ieee_fp

Controls generation of strict IEEE conformant floating-point code.

Syntax

```
#pragma strict_ieee_fp on | off | reset
```

Targets

PowerPC

Remarks

Disabling this option may improve preformance 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

fsel 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
```

Targets

PowerPC

toc_data

Controls how static variables are stored.

Syntax

```
#pragma toc_data on | off | reset
```

Targets

68K, PowerPC

Remarks

This pragma applies to Mac OS CFM programming only.

If you enable this pragma, the compiler stores static variables that are 4 bytes or smaller directly in the TOC instead of allocating space for the variables elsewhere and storing pointers to them in the TOC. This makes your code smaller and faster. Disable this pragma only if your code expects the TOC to contain pointers to data.

This pragma corresponds to the **Store Static Data in TOC** setting in the PPC Processor panel. To check this setting, use `__option (toc_data)`, described in "Checking Settings" on page 115.

traceback

Controls the generation of AIX-format traceback tables for debugging.

Syntax

```
#pragma traceback on | off | reset
```

Targets

PowerPC

Remarks

This pragma helps other people debug your application or shared library if you do not distribute the source code. If you enable this pragma, the compiler generates an AIX-format traceback table for each function in the executable code. Both the CodeWarrior and Apple debuggers can use traceback tables.

This pragma corresponds to the **Emit Traceback Tables** setting in the **PPC Linker** panel. To check this setting, use the `__option (traceback)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

use_lmw_stmw

Controls the use of `lmw` and `stmw` instructions.

Syntax

```
#pragma use_lmw_stmw on | off | reset
```

Targets

PowerPC

Remarks

In theory, 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
```

Targets

PowerPC

Remarks

This pragma applies to Mac OS X Mach-O only.

When this pragma is `on`, `wchar_t` changes from 4-bytes to 2-bytes.

13

Intel x86 Pragmas

This chapter provides information on Intel x86 specific pragmas. You configure the compiler for a project by changing the settings in the ***<Target>* CodeGen** panel. You can also control compiler behavior in your code by including the appropriate pragmas.

Many of the pragmas correspond to option settings in the settings panels for processors and operating systems.

x86 Pragma Reference

NOTE See your target documentation for information on any pragmas you use in your programs. If your target documentation covers any of the pragmas listed in this section, the information provided by your target documentation always takes precedence.

code_seg

Specifies the segment into which code is placed.

Syntax

`#pragma code_seg(`*name*`)`

Targets

Intel x86

Remarks

This pragma designates the segment into which compiled code is placed. The *name* is a string specifying the name of the code segment. For example, the pragma

```
#pragma code_seg(".code")
```

places all subsequent code into a segment named `.code`.

function

Ignored but included for compatibility with Microsoft compilers.

Syntax

`#pragma function(` *funcname1*`,` *funcname2*`, ... )`

Targets

Intel x86

Remarks

Ignored. Included for compatibility with Microsoft compilers.

init_seg

Controls the order in which initialization code is executed.

Syntax

`pragma init_seg( compiler | lib | user | "`*name*`")`

Targets

Intel x86

Remarks

This pragma controls the order in which initialization code is executed. The initialization code for a C++ compiled module calls constructors for any statically declared objects. For C, no initialization code is generated.

The order of initialization is:

– compiler

– lib

– user

If you specify the name of a segment, a pointer to the initialization code is placed in the designated segment. In this case, the initialization code is not called automatically; you must call it explicitly.

k63d

Controls special code generation for AMD K6 3D extensions.

Syntax

```
#pragma k63d on | off | reset
```

Targets

Intel x86

Remarks

This pragma tells the x86 compiler to generate code specifically for processors that have the circuitry needed to execute specialized 3D instructions, such as AMD K6 3D extensions.

This pragma corresponds to the **K6 3D Favored** setting in the **Extended Instruction Set** menu of the **x86 CodeGen** panel.

NOTE This pragma generates code that is not compatible with the Intel Pentium class of microprocessors.

To learn more about this pragma, read the *Targeting Windows*® manual. To check this setting, use `__option (k63d)`, described in "Checking Settings" on page 115.

k63d_calls

Controls use of AMD K6 3D calling conventions.

Syntax

```
#pragma k63d_calls on | off | reset
```

Targets

Intel x86

Remarks

This pragma tells the x86 compiler to generate code that is requires fewer register operations at mode switching time and especially suited for AMD K6 3D and Intel MMX extensions.

This pragma corresponds to the **MMX + K6 3D** setting in the **Extended Instruction Set** menu of the **x86 CodeGen** panel.

To learn more about this pragma, read the *Targeting Windows*® manual. To check this setting, use `__option (k63d_calls)`, described in "Checking Settings" on page 115.

microsoft_exceptions

Controls the use of Microsoft C++ exception handling.

Syntax

```
#pragma microsoft_exceptions on | off | reset
```

Targets

Intel x86

Remarks

This pragma tells the x86 compiler to generate exception handling code that is compatible with Microsoft C++ exception handling code.

To check this setting, use `__option (microsoft_exceptions)`, described in "Checking Settings" on page 115.

microsoft_RTTI

Controls the use of Microsoft C++ runtime type information.

Syntax

```
#pragma microsoft_RTTI on | off | reset
```

Targets

Intel x86

Remarks

This pragma tells the x86 compiler to generate runtime type information that is compatible with Microsoft C++.

To check this setting, use `__option (microsoft_RTTI)`, described in "Checking Settings" on page 115.

mmx

Controls special code generation Intel MMX extensions.

Syntax

```
#pragma mmx on | off | reset
```

Targets

Intel x86

Remarks

This pragma tells the x86 compiler to generate Intel MMX extension code that only runs on processors that have with the circuitry needed to execute the more than 50 specialized MMX instructions.

This pragma corresponds to the **MMX** setting in the **Extended Instruction Set** menu of the **x86 CodeGen** panel. To learn more about this pragma, read the *Targeting Windows*® manual. To check this setting, use `__option (mmx)`, described in "Checking Settings" on page 115.

mmx_call

Controls the use of MMX calling conventions.

Syntax

```
#pragma mmx_call on | off | reset
```

Targets

Intel x86

Remarks

If you enable this pragma, the compiler favors the use of MMX calling conventions.

To learn more about this pragma, read the *Targeting Windows®* manual. To check this setting, use `__option (mmx_call)`, described in "Checking Settings" on page 115.

pack

Controls the alignment of data structures.

Syntax

```
#pragma pack( [n | push, n | pop] )
```

Targets

Intel x86, MIPS

Remarks

Sets the packing alignment for data structures. It affects all data structures declared after this pragma until you change it again with another `pack` pragma.

Table 13.1 Pack pragma

This pragma…	Does this…
`#pragma pack(n)`	Sets the alignment modulus to *n*, where *n* can be `1`, `2`, `4`, `8`, or `16`. For MIPS compilers, if n is 0, structure alignment is reset to the default setting.
`#pragma pack(push, n)`	Pushes the current alignment modulus on a stack, then sets it to *n*, where *n* can be `1`, `2`, `4`, `8`, or `16`. Use `push` and `pop` when you need a specific modulus for some declaration or set of declarations, but do not want to disturb the default setting. MIPS compilers do not support this form.

Table 13.1 Pack pragma

This pragma…	Does this…
`#pragma pack(pop)`	Pops a previously pushed alignment modulus from the stack. MIPS compilers do not support this form.
`#pragma pack()`	For x86 compilers, resets alignment modulus to the value specified in the **x86 CodeGen** panel. For MIPS compilers, resets structure alignment to the default setting.

This pragma corresponds to the **Byte Alignment** setting in the **x86 CodeGen** panel.

peephole

Controls the use peephole optimization.

Syntax

```
#pragma peephole on | off | reset
```

Targets

PowerPC, Intel x86, MIPS

Remarks

If you enable this pragma, the compiler performs *peephole optimizations*, which are small, local optimizations that eliminate some compare instructions and improve branch sequences.

This pragma corresponds to the **Peephole Optimizer** setting in the **PPC Processor** panel. To check this setting, use `__option (peephole)`, described in "Checking Settings" on page 115.

register_coloring

Controls the use of register coloring.

Syntax

```
#pragma register_coloring on | off | reset
```

Targets

Intel x86

Remarks

If you enable this pragma, the compiler uses a single register to hold the values of multiple variables that are never used in the same statement. This improves program performance.

TIP Disable this setting when debugging a program.

This pragma corresponds to the **Register Coloring** setting in the **x86 Codegen** panel. To check this setting, use `__option (register_coloring)`, described in "Checking Settings" on page 115.

scheduling

Specifies the use of instruction scheduling optimization.

Syntax

```
#pragma scheduling 401 | 403 | 505 | 555 | 601 | 602 | 603 |
     604 | 740 | 750 | 801 | 821 | 823 | 850 | 860 | 8240 |
     8260 | altivec | PPC603e | PPC604e | PPC403GA | PPC403GB
     | PPC403GC | PPC403GCX | on | off | twice| once | reset
```

Targets

PowerPC, Intel x86

Remarks

This pragma lets you choose how the compiler rearranges instructions to increase speed. Some instructions, such as a memory load, take more than one processor cycle. By moving an unrelated instruction between the load and the instruction that uses the loaded item, the compiler saves a cycle when executing the program.

For PowerPC, you can use the `401`, `403`, `505`, `555`, `601`, `602`, `603`, `604`, `740`, `750`, `801`, `821`, `823`, `850`, `860`, `8240`, `8260`, `altivec`, `PPC603e`, `PPC604e`, `PPC403GA`, `PPC403GB`, `PPC403GC`, or `PPC403GCX`.

However, if you are debugging your code, disable this pragma. Otherwise, the debugger rearranges the instructions produced from your code and cannot match your source code statements to the rearranged instructions.

This pragma does not correspond to any panel setting.

use_frame

Controls the use of the BP register for stack frames.

Syntax

```
#pragma use_frame on | off | reset
```

Targets

Intel x86

Remarks

If you enable this pragma, the compiler uses the BP register to point to the start of the stack frame.

To check this setting, use `__option (use_frame)`, described in "Checking Settings" on page 115.

warn_illegal_instructions

Controls the recognition of assembly instructions not available to an Intel x86 processor.

Syntax

```
#pragma warn_illegal_instructions on | off | reset
```

Targets

Intel x86

Remarks

If you enable this pragma, the compiler displays a warning when it encounters an assembly language instruction that is not available on the Intel x86 processor for which the compiler is generating object code.

To check this setting, use `__option (warn_illegal_instructions)`, described in "Checking Settings" on page 115.

14

68K Pragmas

This chapter provides information on Embedded 68K and 68K specific pragmas. You configure the compiler for a project by changing the settings in the ***<Target>* Processor** panel. You can also control compiler behavior in your code by including the appropriate pragmas.

Many of the pragmas correspond to option settings in the settings panels for processors and operating systems.

68K Pragma Reference

NOTE See your target documentation for information on any pragmas you use in your programs. If your target documentation covers any of the pragmas listed in this section, the information provided by your target documentation always takes precedence.

a6frames

Controls the generation of stack frames based on the A6 register.

Syntax

```
#pragma a6frames on | off | reset
```

Targets

68K, Embedded 68K

Remarks

This pragma applies to Mac OS on 68K programming only.

If you enable this pragma, the compiler generates `A6` stack frames that let debuggers trace through the call stack and find each routine. Many debuggers, including the Metrowerks debugger and Jasik's The Debugger, require these frames. If you disable this pragma, the compiler does not generate the `A6` stack frames, which results in smaller and faster code.

This is the code that the compiler generates for each function if you enable this pragma:

```
LINK #nn,A6
UNLK A6
```

This pragma corresponds to the **Generate A6 Stack Frames** setting in the **68K Linker** panel. To check this setting, use `__option (a6frames)`, described in "Checking Settings" on page 115.

align

Specifies how to align struct and class data.

Syntax

```
#pragma options align= alignment
```

Parameter

`alignment`

alignment is one of the following values:

Table 14.1 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 Macintoshes.
`mac68k4byte`	Aligns every field on 4-byte boundaries.
`power`	Aligns every field on its natural boundary. This is the standard alignment for Power Macintoshes. 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.

Table 14.1 Structs and Classes Alignment

If *alignment* is …	The compiler …
`native`	Aligns every field using the standard alignment. It is equivalent to using `mac68k` for 68K Macintoshes and `power` for Power Macintoshes.
`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, if there is one, or to the value in the **68K** or **PPC Processor** panel.

NOTE There is a space between `options` and `align`.

Targets

68K, PowerPC, MIPS

Remarks

When using the C compiler, it is possible to use “char/short” bitfields to align fields on 1-byte boundaries. However, this is not possible if **ANSI Strict** is enabled.

The `#pragma align` / `#pragma pack` options only apply to the layout of fields that are larger than a byte. For example, a “short” field will appear on a 2-byte boundary, but using a packed alignment may allow it to be placed on a 1-byte boundary. Bitfields may be aligned to the maximum size used as the base of the bitfield (the “char” in `char foo:1` or the “int” in `int foo:3`), not necessarily the number of bits allocated.

This pragma corresponds to the **Struct Alignment** setting in the **68K Processor** panel.

altivec_codegen

Controls the use PowerPC AltiVec™ instructions during optimization.

Syntax

```
#pragma altivec_codegen on | off | reset
```

Targets

PowerPC processors with AltiVec technology.

Remarks

If you enable this pragma, the compiler uses PowerPC AltiVec instructions, if possible, during optimization.

To check this setting, use `__option (altivec_codegen)`, described in "Checking Settings" on page 115.

code68020

Controls object code generation for Motorola 680x0 (and higher) processors.

Syntax

```
#pragma code68020 on | off | reset
```

Targets

68K, Embedded 68K

Remarks

This pragma applies to 68K programming only.

If you enable this pragma, the compiler generates code that is optimized for the MC68020. The resulting object code runs on Power Macintoshes and Macintoshes with MC68020 and MC68040 processors. However, it might crash on Macintoshes with MC68000 processors. If you disable this pragma, the compiler generates code that runs on any Macintosh or embedded 68K processor.

WARNING! Do not change this setting within a function definition.

On Mac OS, before your program runs code optimized for the MC68020, use the `gestalt()` function to make sure the chip is available. For more information on `gestalt()`, see Chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities*.

In the Mac OS compiler, this setting is disabled by default.

This pragma corresponds to the **68020 Codegen** setting in the 68K Processor panel. To check this setting, use `__option (code68020)`, described in "Checking Settings" on page 115.

code68881

Controls object code generation for Motorola 68881 (and higher) math coprocessors.

Syntax

```
#pragma code68881 on | off | reset
```

Targets

68K

Remarks

This pragma applies to 68K programming only.

If you enable this pragma, the compiler generates code that is optimized for the MC68881 floating-point unit (FPU). This code runs on Macintoshes with MC68881 FPU, MC68882 FPU, and MC68040 processors. (The MC68040 has a built-in MC68881 FPU.) The code does not run on Power Macintoshes or on Macintoshes with MC68LC040 processors or other processors that do not have FPUs. If you disable this pragma, the compiler generates code that runs on any Macintosh.

WARNING! If you enable this pragma, place it at the beginning of your file before including any files and declaring any variables and functions.

Before your program runs code optimized for the MC68881, use the `gestalt()` function to make sure an FPU is available. For more information on `gestalt()`, see Chapter "Gestalt Manager" in *Inside Macintosh: Operating System Utilities.*

This pragma corresponds to the **68881 Codegen** setting in the 68K Processor panel. To check this setting, use `__option (code68881)`, described in "Checking Settings" on page 115.

codeColdFire

Controls the organization of object code.

Syntax

```
#pragma codeColdFire [ on | off | reset | MCF206e | MCF5307 ]
```

Targets

Embedded 68K

Remarks

This pragma controls the generation of ColdFire object code.

The default value for `on` assumes a MCF5037, but you can use `#pragma codeColdFire MCF5206e` or `MCF5307` to specify and control the exact core.

const_multiply

Enables support for constant multiplies using shifts and add/subtracts.

Syntax

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

Targets

Embedded 68K

Remarks

To check this setting, use `__option (const_multiply)`, described in "Checking Settings" on page 115. By default, this value is enabled.

d0_pointers

Controls which register to use for holding function result pointers.

Syntax

```
#pragma d0_pointers
```

Targets

68K

Remarks

This pragma applies to 68K programming only.

This pragma lets you choose between two calling conventions: the convention for MPW and Macintosh Toolbox routines and the convention for Metrowerks C/C++ routines. In the MPW and Macintosh Toolbox calling convention, functions return pointers in the register `D0`. In the Metrowerks C/C++ convention, functions return pointers in the register `A0`.

When you declare functions from the Macintosh Toolbox or a library compiled with MPW, enable the `d0_pointers` pragma. After you declare those functions, disable the pragma to start declaring or defining Metrowerks C/C++ functions.

In Listing 14.1, the Toolbox functions in `Sound.h` return pointers in `D0` and the user-defined functions in `Myheader.h` use `A0`.

Listing 14.1 Using #pragma pointers_in_A0 and #pragma pointers_in_D0

```
#pragma d0_pointers on     // set for Toolbox calls
#include <Sound.h>
#pragma d0_pointers reset // set for my routines
#include "Myheader.h"
```

The pragmas `pointers_in_A0` and `pointers_in_D0` have much the same meaning as `d0_pointers` and are available for backward compatibility. The pragma `pointers_in_A0` corresponds to `#pragma d0_pointers off` and the pragma `pointers_in_D0` corresponds to `#pragma d0_pointers on`. The pragma `d0_pointers` is recommended for new code since it supports the `reset` argument. For more information, see "pointers_in_A0, pointers_in_D0" on page 294.

This pragma does not correspond to any setting in the **68K Processor** panel. To check this setting, use the `__option (d0_pointers)`, described in "Checking Settings" on page 115.

define_section

Arranges object code into sections.

Syntax

`#pragma define_section` *sname istr* `[`*ustr*`] [`*addrmode*`] [`*accmode*`]`

Parameters

`sname`

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

For example,

`#pragma section` *sname* `begin`

or

`__declspec(`*sname*`)`

`istr`

The section name string for initialized data assigned to sname, such as `.data` (applies to uninitialized data if `ustr` is omitted).

`ustr`

The elf section name for uninitialized data assigned to `sname`. The default value for `ustr` is the same as `istr`.

`addmode`

Indicates how the section is addressed. It can be one of the following:

- `standard`—32-bit absolute address (default)
- `near_absolute`—16-bit absolute address
- `far_absolute`—32-bit absolute address
- `near_code`—16-bit offset from TP
- `far_code`—32-bit offset from TP
- `near_data`—16-bit offset from GP
- `far_data`—32-bit offset from GP

`accmode`

Indicates the attributes of the section. It can be one of the following:

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

TIP Refer to the appropriate *Targeting* manual for information on the sections that the CodeWarrior C/C++ compiler predefines for your build target.

Targets

Embedded 68K, MIPS

Remarks

This pragma lets you arrange compiled object code into predefined sections and sections that you define.

You can also use `#pragma define_section` to redefine the attributes of existing sections:

You can force all data to be addressed using 16-bit absolute addresses with the following code:

```
#pragma define_section data ".data" near_absolute
```

You can force exception tables to be addressed using 32-bit TP-relative with the following code:

```
#pragma define_section exceptlist ".exceptlist" far_code
#pragma define_section exception ".exception" far_code
```

If you do this, put these pragmas in a prefix file or some other header that all source files in your program reference.

far_code, near_code, smart_code

Specify the kind of addressing to use for executable code.

Syntax

```
#pragma far_code
#pragma near_code
#pragma smart_code
```

Targets

68K, Embedded 68K

Remarks

This pragma applies to Mac OS on 68K and embedded 68K programming only.

These pragmas determine what kind of addressing the compiler uses to refer to functions:

- `#pragma far_code` always generates 32-bit addressing, even if 16-bit addressing can be used.
- `#pragma near_code` always generates 16-bit addressing, even if data or instructions are out of range.
- `#pragma smart_code` generates 16-bit addressing whenever possible and uses 32-bit addressing only when necessary.

For more information on these code models, see the *CodeWarrior Development Tools IDE User Guide*.

These pragmas correspond to the **Code Model** setting in the **68K Processor** panel. The default is `#pragma smart_code`.

far_data

Controls the use of 32-bit addressing to refer to global data.

Syntax

```
#pragma far_data on | off | reset
```

Targets

68K, Embedded 68K

Remarks

If you enable this pragma, global data is called using 32-bit addressing instead of 16-bit addressing. While this allows more global data, it also makes your program slightly bigger and slower. If you disable this pragma, your global data is stored as near data and adds to the 64K limit on near data.

This pragma corresponds to the **Far Data** setting in the **68K Processor** panel. To check this setting, use `__option (far_data)`, described in "Checking Settings" on page 115.

far_strings

Controls the use of 32-bit addressing to refer to string literals.

Syntax

```
#pragma far_strings on | off | reset
```

Targets

68K, Embedded 68K

Remarks

If you enable this pragma, you have a much higher number of string literals because the compiler uses 32-bit addressing to refer to string literals instead of 16-bit addressing. This also makes your program slightly bigger and slower. If you disable this pragma, your string literals are stored as near data and add to the 64K limit on near data.

This pragma corresponds to the **Far String Constants** setting in the **68K Processor** panel. To check this setting, use `__option (far_strings)`, described in "Checking Settings" on page 115.

far_vtables

Controls the use of 32-bit addressing for C++ virtual function tables.

Syntax

```
#pragma far_vtables on | off | reset
```

Targets

68K, Embedded 68K

Remarks

This pragma applies to Mac OS on 68K and embedded 68K programming only.

A class with virtual function members has to create a virtual function dispatch table in a data segment. If you enable this pragma, this table can be any size because the compiler uses 32-bit addressing to refer to the table instead of 16-bit addressing. However, this also makes your program slightly bigger and slower. If you disable this pragma, the table is stored as near data and adds to the 64K limit on near data.

This pragma corresponds to the **Far Method Tables** setting in the **68K Processor** panel. To check this setting, use `__option (far_vtables)`, described in "Checking Settings" on page 115.

fourbyteints

Controls the size of the `int` data type.

Syntax

```
#pragma fourbyteints on | off | reset
```

Targets

68K

Remarks

If you enable this pragma, the size of an `int` is 4 bytes. Otherwise, the size of an `int` is 2 bytes.

This pragma corresponds to the **4-Byte Ints** setting in the **68K Processor** panel. To check this setting, use `__option (fourbyteints)`, described in "Checking Settings" on page 115.

NOTE Whenever possible, select this setting from the panel, not a pragma. If you must use this pragma, place it at the beginning of your program before including files or declaring functions or variables.

fp_pilot_traps

Controls floating point code generation for Palm OS.

Syntax

```
#pragma fp_pilot_traps on | off | reset
```

Targets

68K

Remarks

This pragma controls floating point code generation. If you enable this pragma, the compiler references Palm OS library routines to perform floating point operations.

IEEEdoubles

Specifies the size of the `double` type.

Syntax

```
#pragma IEEEdoubles on | off | reset
```

Targets

68K, Embedded 68K

Remarks

This setting, along with the **68881 Codegen** setting in the **68K Processor** panel, specifies the length of a `double`. Table 14.2 shows how these settings work.

Table 14.2 Using the `IEEEdoubles` Pragma

If the `IEEEdoubles` pragma is…	and 68881 Codegen is…	Then a `double` is this size…
on	on or off	64 bits
off	off	80 bits
off	on	96 bits

This pragma corresponds to the **8-Byte Doubles** setting in the **68K Processor** panel. To check this setting, use `__option (IEEEdoubles)`, described in "Checking Settings" on page 115.

NOTE Whenever possible, select this setting from the panel, not a pragma. If you must use the pragma, place it at the beginning of your program before including files or declaring functions or variables.

interrupt

Controls the compilation of object code for interrupt routines.

Syntax

```
#pragma interrupt on|off|reset
```

For Embedded PowerPC:

```
#pragma interrupt [SRR DAR DSISR enable] on | off | reset
```

Targets

68K, Embedded 68K, PowerPC, NEC V800, MIPS

Remarks

If you enable this pragma, the compiler generates a special prologue and epilogue for functions so that they can handle interrupts.

For convenience, the compiler also marks interrupt functions so that the linker does not dead-strip them. See "force_active" on page 152 for related information.

Embedded 68K

If you enable this pragma, the compiler generates a special prologue and epilogue for functions encapsulated by this pragma. All modified registers (both nonvolatile and scratch registers) are saved or restored, and functions return via `RETI` instead of `JMP [LP]`.

You can also use `__declspec(interrupt)` to mark functions as interrupt routines.

Listing 14.2 Example of `__declspec()`

```
__declspec(interrupt) void foo()
{
// enter code here
}
```

To check this setting, use `__option (interrupt)`, described in "Checking Settings" on page 115.

interrupt_fast

Controls the compilation of object code for interrupt routines.

Syntax

```
#pragma interrupt_fast on|off|reset
```

Targets

68K

Remarks

If you enable this pragma, the compiler generates a special prologue and epilogue for functions so that they can handle interrupts.

For convenience, the compiler also marks interrupt functions so that the linker does not dead-strip them. See "force_active" on page 152 for related information.

macsbug

Control the generation of debugger data for MacsBug.

Syntax

```
#pragma macsbug on | off | reset
  #pragma oldstyle_symbols on | off | reset
```

Targets

68K, Embedded 68K

Remarks

These pragmas apply to Mac OS on 68K programming only.

They let you choose how the compiler generates Macsbug symbols. Many debuggers, including Metrowerks debugger, use Macsbug symbols to display the names of functions and variables. The pragma `macsbug` lets you enable and off Macsbug generation. The pragma `oldstyle_symbols` lets you choose which type of symbols to generate. The table below shows how these pragmas work:

Table 14.3 Macsbug pragma

To do this…	Use these pragmas…
Do not generate Macsbug symbols	`#pragma macsbug on`
Generate old style Macsbug symbols	`#pragma macsbug on` `#pragma oldstyle_symbols on`
Generate new style Macsbug symbols	`#pragma macsbug on` `#pragma oldstyle_symbols off`

These pragmas corresponds to **MacsBug Symbols** setting in the **68K Linker** panel. To check this pragma, use `__option (macsbug)` described in "Checking Settings" on page 115. To check the old style pragma, use `__option (oldstyle_symbols)` described in "Checking Settings" on page 115.

mpwc

Controls the use of the Apple MPW C calling conventions.

Syntax

```
#pragma mpwc on | off | reset
```

Targets

68K

Remarks

This pragma applies to Mac OS on 68K processors only.

If you enable this pragma, the compiler does the following to ensure compatibility with the MPW C calling conventions:

– Passes any integral argument that is smaller than 2 bytes as a sign-extended `long integer`. For example, the compiler converts this declaration:

```
int MPWfunc ( char a, short b, int c, long d,
char *e );
```

to this:

```
long MPWfunc( long a, long b, long c, long d,
char *e );
```

– Passes any floating-point arguments as a `long double`. For example, the compiler converts this declaration:

```
void MPWfunc( float a, double b, long double c );
```

to this:

```
void MPWfunc( long double a, long double b,
long double c );
```

– Returns any pointer value in `D0` (even if the pragma `pointers_in_D0` is disabled).

– Returns any 1-byte, 2-byte, or 4-byte structure in `D0`.

– If the **68881 Codegen** setting is enabled, returns any floating-point value in `FP0`.

This pragma corresponds to the **MPW C Calling Convention** setting in the 68K Processor panel. To check this setting, use `__option (mpwc)`, described in "Checking Settings" on page 115.

no_register_coloring

Controls the use of a register to hold the values of more than one variable.

Syntax

```
#pragma no_register_coloring on | off | reset
```

Targets

68K

Remarks

If you disable this pragma, the compiler performs register coloring. In this optimization, the compiler lets two or more variables share a register: it assigns different variables or parameters to the same register if you do not use the variables at the same time. In this example, the compilers could place `i` and `j` in the same register:

```
short i;
int j;

for (i=0; i<100;  i++) {   MyFunc(i);  }
for (j=0; j<1000; j++) {   OurFunc(j); }
```

However, if a line like the one below appears anywhere in the function, the compiler would realize that you are using `i` and `j` at the same time and place them in different registers:

```
int k = i + j;
```

If register coloring is enabled while you debug your project, it might look like something is wrong with the variables sharing a register. In the example above, `i` and `j` would always have the same value. When `i` changes, `j` changes in the same way. When `j` changes, `i` changes in the same way. To avoid this confusion while debugging, disable register coloring or declare the variables you want to watch as volatile.

The pragma corresponds to the **Global Register Allocation** setting in the 68K Processor panel. To check this setting, use `__option`

`(no_register_coloring)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

See also "SDS_debug_support" on page 295.

oldstyle_symbols

See "macsbug" on page 289 for information about this pragma.

parameter

Specifies the use of registers to pass parameters.

Syntax

`#pragma parameter` *return-reg* *func-name*`(`*param-regs*`)`

Targets

68K, Embedded 68K

Remarks

This pragma applies to 68K programming only.

The compiler passes the parameters for the function *func-name* in the registers specified in *param-regs* instead of the stack. The compiler then returns any value in the register *return-reg*. Both *return-reg* and *param-regs* are optional.

Here are some samples:

```
#pragma parameter __D0 Gestalt(__D0, __A1)
#pragma parameter __A0 GetZone
#pragma parameter HLock(__A0)
```

When you define the function, you need to specify the registers right in the parameter list, as described in the appropriate *Targeting* manual.

This pragma does not correspond to any panel setting.

pcrelstrings

Controls the storage and reference of string literals from the program counter.

Syntax

```
#pragma pcrelstrings on | off | reset
```

Targets

68K, Embedded 68K

Remarks

If you enable this pragma, the compiler stores the string constants used locally scope in the code segment and addresses these strings with PC-relative instructions. Otherwise, the compiler stores all string constants in the global data segment. Either way, the compiler stores string constants used in the global scope in the global data segment.

Listing 14.3 Example of pragma pcrelstrings

```
#pragma pcrelstrings on
int foo(char *);

int x = f("Hello");  // "Hello" is allocated in
                     // the global data segment
int bar()
{
  return f("World"); //"World" is allocated in the code segment
}                    // (pc-relative)
```

Strings in C++ initialization code are always allocated in the global data segment.

NOTE If you enable the `pool_strings` pragma, the compiler ignores the setting of the `pcrelstrings` pragma.

This pragma corresponds to the **PC-Relative Strings** setting in the 68K Processor panel. To check this setting, use `__option (pcrelstrings)`, described in "Checking Settings" on page 115. By default, this setting is disabled.

pointers_in_A0, pointers_in_D0

Controls which calling convention to use.

Syntax

```
#pragma pointers_in_A0
#pragma pointers_in_D0
```

Targets

68K, Embedded 68K

Remarks

These pragmas are available for Mac OS on 68K processors only.

They let you choose between two calling conventions: the convention for MPW and Macintosh Toolbox routines and the convention for Metrowerks C/C++ routines. In the MPW and Macintosh Toolbox calling convention, functions return pointers in the register `D0`. In the Metrowerks C/C++ convention, functions return pointers in the register `A0`.

When you declare functions from the Macintosh Toolbox or a library compiled with MPW, use the pragma `pointers_in_D0`. After you declare those functions, use the pragma `pointers_in_A0` to start declaring or defining Metrowerks C/C++ functions.

In Listing 14.4, the Toolbox functions in `Sound.h` return pointers in `D0` and the user-defined functions in `Myheader.h` use `A0`.

Listing 14.4 Using #pragma pointers_in_A0 and #pragma pointers_in_D0

```
#pragma pointers_in_D0 // set for Toolbox calls
#include <Sound.h>
#pragma pointers_in_A0 // set for my own routines
#include "Myheader.h"
```

The pragmas `pointers_in_A0` and `pointers_in_D0` have much the same meaning as `d0_pointers` and are available for backwards compatibility. The pragma `pointers_in_A0` corresponds to `#pragma d0_pointers off` and the pragma `pointers_in_D0` corresponds to `#pragma d0_pointers on`. The pragma `d0_pointers` is recommended for new code since it supports the `reset` argument. For more information, see "d0_pointers" on page 280.

This pragma does not correspond to any panel setting. To check this setting, use the `__option (d0_pointers)`, described in "Checking Settings" on page 115.

profile

Controls the generation of extra object code for use with the CodeWarrior profiler.

Syntax

```
#pragma profile on | off | reset
```

Targets

68K, PowerPC

Remarks

This pragma applies to Mac OS programming only.

If you enable this pragma, the compiler generates code for each function that lets the Metrowerks Profiler collect information on it. For more information, see the *Metrowerks Profiler Manual*.

This pragma corresponds to the **Generate Profiler Calls** setting in the **68K Processor** panel and the **Emit Profiler Calls** setting in the **PPC Processor** panel. To check this setting, use `__option (profile)` described in "Checking Settings" on page 115.

SDS_debug_support

Enables SDS support in DWARF.

Syntax

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

Targets

Embedded 68K

Remarks

This pragma enables limited-implementation SDS support in the generated DWARF. We are working on making the Metrowerks compiler output compatible with the SDS debugger.

This pragma does not correspond to any panel setting. The default value is disabled.

---

section

A sophisticated and powerful pragma that lets you arrange compiled object code into predefined sections and sections you define.

Syntax

For Embedded 68K:

```
#pragma section sname [begin | end]
```

Parameters for Embedded 68K

`sname`

Specifies the name of the section where the compiler stores initialized objects.

`begin, end`

Specifies the start and the end of a `#pragma section block`. The code and data within the block is placed in the named section.

Targets

PowerPC, Embedded 68K

Remarks

Topics for Embedded 68K are organized into these parts:

Using #pragma section with #pragma push and #pragma pop for Embedded 68K

You can use this pragma with `#pragma push` and `#pragma pop` to ease complex or frequent changes to sections settings. However, `#pragma section` blocks do not nest. You must end the previous section block before you can switch to a new one.

segment

Controls the code segment where subsequent object code is stored.

Syntax

```
#pragma segment name
```

Targets

68K, Embedded 68K, PowerPC

Remarks

This pragma applies to Mac OS programming only.

This pragma places all the functions that follow into the code segment named *name*. For more on function-level segmentation, consult the *Targeting* manual for your target platform.

Generally, the PowerPC compilers ignore this directive because PowerPC applications do not have code segments. However, if you choose **by #pragma segment** from the **Code Sorting** pop-up menu in the **PPC PEF** panel, the PowerPC compilers group functions in the same segment together. For more information, consult the *Targeting* manual for your target platform.

This pragma does not correspond to any panel setting.

side_effects

Controls the use of pointer aliases.

Syntax

```
#pragma side_effects on | off | reset
```

Targets

68K, Embedded 68K

Remarks

If your program does not use pointer aliases, disable this pragma to make your program smaller and faster. Otherwise, enable this pragma to avoid incorrect code. A pointer alias looks like this:

```
int a, *p;
p = &a;       // *p is an alias for a.
```

Pointer aliases are important because the compiler must load a variable into a register before performing arithmetic on it. So, in the example below, the compiler loads `a` into a register before the first addition. If `*p` is an alias for `a`, the compiler must load `a` into a register again before the second addition, since changing `*p` also changes `a`. If `*p` is not an alias for `a`, the compiler does not need to load `a` into a register again because changing `*p` does not change `a`.

```
x = a + 1;
*p = 0;       // If *p is an alias for a,
y = a + 2;    // this changes a.
```

This pragma does not correspond to any panel setting. To check whether this pragma is enabled, use `__option (side_effects)`, described in "Checking Settings" on page 115. By default, this pragma is enabled.

stack_cleanup

Controls when the compiler generates code to clean up the stack.

Syntax

```
#pragma stack_cleanup on | off | reset
```

Targets

68K

Remarks

Enabling this pragma disables the deferred stack cleanup after function calls, forcing the compiler to remove arguments from the stack after every function call. Although this setting slows down execution, it reduces stack usage so that the stack does not intrude on other parts of the program.

This pragma does not correspond to any panel setting. To check this setting, use `__option (stack_cleanup)`, described in "Checking Settings" on page 115. By default, this pragma is disabled.

toc_data

Controls how static variables are stored.

Syntax

```
#pragma toc_data on | off | reset
```

Targets

68K, PowerPC

Remarks

This pragma applies to Mac OS CFM programming only.

If you enable this pragma, the compiler stores static variables that are 4 bytes or smaller directly in the TOC instead of allocating space for the variables elsewhere and storing pointers to them in the TOC. This makes your code smaller and faster. Disable this pragma only if your code expects the TOC to contain pointers to data.

This pragma corresponds to the **Store Static Data in TOC** setting in the PPC Processor panel. To check this setting, use `__option (toc_data)`, described in "Checking Settings" on page 115.

15

Command-Line Tools

This chapter describes how to configure and use the command-line tools. It contains the following sections:

Overview

The CodeWarrior™ IDE uses compilers and linkers to generate object code for x86, PowerPC desktop, and embedded platforms. The IDE also provides *command-line* versions of these tools that also generate and combine object code files to produce executable files such as applications, dynamic link libraries (DLLs), code resources, or static libraries.

You configure each command-line tool by specifying various options when you invoke the tool. Many of these options correspond to settings in the IDE's **Target Settings** window.

TIP A command-line user interface interacts with you through a text-based console instead of GUI items such as windows, menus, and buttons.

Tool Naming Conventions

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

`mw`<*tool*><*arch/OS*>

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

<*arch/OS*> is the target platform for which the tool generates object code, unless a target platform has multiple tool versions. For example, for Embedded PowerPC, <*arch/OS*> is `eppc` (`mwcceppc`, `mwldeppc`, `mwasmeppc`). For Windows®/x86, <*arch/OS*> is empty (`mwcc`, `mwld`, `mwasm`).

Working with Environment Variables

To use the command-line tools, you must change several environment variables. If you are using CodeWarrior command-line tools with Microsoft® Windows®, you can assign environment variables through the **Environment** tab under the **System** control panel in Windows XP/2000/NT or the `autoexec.bat` file in Windows 95/98.

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

- CWFolder Environment Variable
- Setting the PATH Environment Variable
- Getting Environmental Variables
- Search Path Environment Variables

CWFolder Environment Variable

Use the following syntax when defining variables in batch files or on the command line (Listing 15.1).

Listing 15.1 Example of Setting CWFolder

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

In this example, `CWFolder` refers to the path where you installed CodeWarrior for Embedded PowerPC. It is not necessary to include quotation marks when defining environment variables that include spaces. Because Windows does not strip out the quotes, this leads to unknown directory warnings.

Setting the PATH Environment Variable

The `PATH` variable should include the paths for the Embedded PowerPC tools, shown in Listing 15.2. For other tools, the paths can vary.

Listing 15.2 Example of Setting PATH

```
%CWFolder%\Bin
%CWFolder%\EPPC_Tools\Command_Line_Tools
```

The first path in Listing 15.2 contains the FlexLM license manager DLL, and the second path contains the tools. To run FlexLM, copy the following file into the directory containing the command-line tools:

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

Or, you can define the variable `LM_LICENSE_FILE` as:

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

which points to the license information. It might point to alternate versions of this file, as needed.

Getting Environmental Variables

Use the predefined macro `__env_var()` to return host-specific environmental variables. Listing 15.3 shows an example.

Listing 15.3 Example of Getting an Environmental Variable

```
// returns "username" environmental variable
// (host-specific)
char* username = __env_var(username);
```

Search Path Environment Variables

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

For example, in Embedded PowerPC, unless you pass `-nodefaults` to the command line, the compiler searches first for an environment variable called `MWCEABIPPCIncludes`, then `MWCIncludes`. These variables contain a list of system access paths to be searched after the user-specified system access paths. The assembler uses the variables `MWAsmEABIPPCIncludes` and `MWAsmIncludes` to perform a similar search.

Similarly, unless you specify `-nodefaults` or `-disassemble`, the linker searches the environment for a list of system access paths and library files to be added to the end of the search and link orders. For example, with Embedded PowerPC, the linkers searches for files, libraries, and command files, using the system library paths found within the variables `MWEABIPPCLibraries` and `MWLibraries`. Associated with these lists are `MWEABIPPCLibraryFiles` and `MWLibraryFiles`, which contain lists of libraries (or object files or command files) to add to the end of the link order. These files can be located in any of the cumulative access paths at runtime.

If you are only building for one target, you can use `MWCIncludes`, `MWAsmIncludes`, `MWLibraries`, and `MWLibraryFiles`. Because the target-specific versions of these variables override the generic variables, they are useful when working with multiple targets. If the target-specific variable exists, then the generic variable is not used because you cannot combine the contents of the two variables.

Invoking Command-Line Tools

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

The tool performs the operation on the files you specify. If the tool successfully finishes its operation, a new prompt appears on the command line. Otherwise, it reports any problems as text messages on the command line before a new prompt appears.

You can also write *scripts* that automate the process to build your software. Scripts contain a list of command-line tools to invoke, one after another. 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.

Command follow this convention:

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 tell the tool what operation it should perform and how to perform it, and *files* is a list of zero or more files on which the tool should operate. Which options and files you use depends on what operation you want the tool to perform.

File Extensions

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

Although the command-line version of the CodeWarrior C/C++ compiler accepts non-standard file extensions as source, it also emit a warning when this happens. By default, the compiler assumes that a file with any extensions other than `.c`, `.h`, or `.pch` is a C++ source file. The linker must be able to identify all files as object code, libraries, or command files. It ignores all other files.

Linker command files must end in `.lcf`. You can add them to the link line. Listing 15.4 provides an example for Embedded PowerPC.

Listing 15.4 Example of Using Linker Command files

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

For more information on linker command files, see your target-specific *Targeting* manual.

Help and Administrative Options

This section provides examples of how to retrieve general and help information from the command-line tools.

For example, to obtain help information from a tool that has some compatibility options with Visual C++, type the following command:

```
mwcc -?
```

To get more specific information from the same tool, type the following command:

```
mwcc -help [argument,...]
```

where *argument* is a valid keyword such as `usage`, `all`, or `this`. For example, with the Windows® x86 C/C++ compiler, typing:

```
mwcc -help usage
```

or

```
mwcc -help opt=help
```

provides information about the help options available with the `mwcc` Windows® x86 tool.

Command-Line Settings Conventions

In all cases, text in brackets (`[]`) is optional, although the brackets themselves never appear in the actual command. For example, the command `-str[ings] pool` can mean either:

```
-strings pool
```

or

```
-str pool
```

Where an option has several possible permutations, the possibilities are separated by the pipe (`|`) character. For example:

```
-sym on|off|full|fullpath
```

means the `-sym` command can be followed by one or more of the following options: `on`, `off`, `full`, or `fullpath`. If you have more than one option, separate each option with a comma. So you might have `-sym on`, `-sym off`, `-sym full`, or `-sym on, fullpath`.

The plus sign (+) means that the parameter to an option must not be separated from the option name by a space. For example,

```
-D+name[=value]
```

means that you can have `-DVAR` or `-DVAR=3`, but not `-D VAR`.

In cases where you provide a variable parameter such as a file name, that item is in italic text. For example, `-precompile` *filename* means you must provide a file name. The help text that corresponds to the compiler option explains what you must provide.

CodeWarrior Command Line Tools for Mac OS X

CodeWarrior for Mac OS contains Mac OS X command-line compilers and linkers. For instructions on how to use CodeWarrior command-line compilers with Mac OS X, see the release notes in the `OS X Development Items` folder on the CodeWarrior Tools CD for Mac OS.

Index

Symbols

Numerics

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

U

V

W

X